Noname manuscript No.
(will be inserted by the editor)

Register-Aware Optimizations for Parallel Sparse Matrix-Matrix Multiplication

Junhong Liu · Xin He · Weifeng Liu · Guangming Tan

Received: date / Accepted: date

Abstract General sparse matrix-matrix multiplication (SpGEMM) is a fundamental building block of a number of high-level algorithms and real-world applications. In recent years, several efficient SpGEMM algorithms have been proposed for many-core processors such as GPUs. However, their implementations of sparse accumulators, the core component of SpGEMM, mostly use low speed on-chip shared memory and global memory, and high speed registers are seriously underutilised. In this paper, we propose three novel register-aware SpGEMM algorithms for three representative sparse accumulators, i.e., sort, merge and hash, respectively. We fully utilise the GPU registers to fetch data, finish computations and store results out. In the experiments, our algorithms deliver excellent performance on a benchmark suite including 205 sparse matrices from the SuiteSparse Matrix Collection. Specifically, on an Nvidia Pascal P100 GPU, our three register-aware sparse accumulators achieve on average 2.0x (up to 5.4x), 2.6x (up to 10.5x) and 1.7x (up to 5.2x) speedups over their original implementations in libraries bhSPARSE, RMerge and NSPARSE, respectively.

Keywords Sparse matrix · Sparse matrix-matrix multiplication · GPU · Register

Junhong Liu · Xin He · Guangming Tan
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
University of Chinese Academy of Sciences
E-mail: liujunhong@ncic.ac.cn,hexin2016@ict.ac.cn,tgm@ict.ac.cn

Weifeng Liu
Department of Computer Science, Norwegian University of Science and Technology
E-mail: weifeng.liu@ntnu.no

1 Introduction

General sparse matrix-matrix multiplication (SpGEMM) operation multiplies a sparse matrix A with a sparse matrix B and generates a resulting sparse matrix C. This is an essential building block in a number of applications such as algebraic multigrid methods [1], shortest path algorithms [2], breadth first search algorithms [3], and Markov cluster algorithms [4]. It is also an important kernel in the GraphBLAS standard [5,6]. As a result, fast algorithms for parallel SpGEMM received much more attention in recent years [7–23].

The most basic way to calculate SpGEMM is the row-by-row method proposed by Gustavson [24] that multiplies each row of A with the whole matrix B for the corresponding row of C. So the SpGEMM computation becomes a combination of a number of sparse vector-matrix multiplications, i.e., so-called sparse accumulators, which is different with the sparse matrix-vector multiplication [25–28]. Because the rows are independent of each other, they can be easily parallelized on modern many-core processors. Since GPUs provide higher computational power in terms of the theoretical peak floating-point operations and memory bandwidth, several SpGEMM algorithms, in particular various sparse accumulators, have been proposed for GPUs [7,8,11,29, 30]. Moreover, to achieve better load balancing, several studies [11,29] create a couple of tens of bins and then group rows requiring the similar number of floating point operations into the same bin.

The compute for each row includes traversing nonzeros a_{ij} of the ith row of A and accumulating all nonzeros in the jth rows of B into the ith row of C. To efficiently accumulate the nonzeros from different rows of B, different sparse accumulators have been developed. The dense-based [31], sort-based [7, 32], heap-based [29], merge-based [30] and hash-based algorithms [8,11,33] are representative methods. All methods can be applied to bins of different sizes. For bins accumulating a small number of nonzeros (in general no more than 512), the compute can be completed in on-chip shared memory for fast processing. On the other hand, the longer bins have to be finished in global memory due to the limited size of the shared memory.

Besides shared memory and global memory, thread registers have proven to be another very efficient alternative memory level to build fast algorithms. Li et al. [34,35] pointed out that register reuse and communication are crucial for more fine-grained GPU execution model. Rawat et al. [36] used GPU registers for improving various stencil computations. However, for SpGEMM computation, although the overhead associated with the global memory accesses can be reduced to a certain extent by performing the shared memory-based sorting or hashing operation on the sub-matrices [9,30], an efficient utilisation of the GPU registers for SpGEMM is not foreseeable and the best performance is not achieved yet.

Motivated by the underuse of registers in recent SpGEMM algorithms, we in this paper propose three register-aware algorithms, i.e., sort-, merge- and hash-based, to improve the performance of SpGEMM, in particular for matrices including bins of moderate sizes (i.e., the number of intermediate products

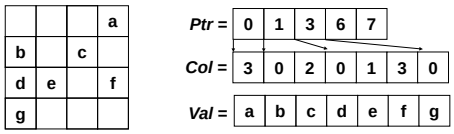


Fig. 1: A sparse matrix and its CSR format.

is smaller than 512). Specifically, we use register to load entries from global memory and store intermediate products calculated into shared memory, then load them back to registers to work in a load balanced way and to implement new primitives for fully using various in-register communication schemes.

The main contributions of this paper include proposing three register-aware SpGEMM algorithms, relying on the sort-, merge- and hash-based sparse accumulators, respectively, and achieving significantly improved performance over state-of-the-art libraries. Specifically, on an Nvidia Pascal P100 GPU, our three register-aware sparse accumulators achieve on average 2.0x (up to 5.4x), 2.6x (up to 10.5x) and 1.7x (up to 5.2x) speedups over their original implementations in libraries bhSPARSE [18], RMerge [30] and NSPARSE [11], respectively.

2 Background

2.1 Sparse Matrix

To avoid storing and calculating zeros, some matrices can be stored in their sparse form. The most widely used storage scheme is the so-called Compressed Sparse Row (CSR) format. Figure 1 shows a sparse matrix and its CSR storage. The CSR format consists of three arrays, namely Val, Col and Ptr. The Val array stores values, and the Col array stores column indexes of the nonzero entries. The third array Ptr records the starting storage position (or offset) of each row in the arrays Val and Col, thus the number of the nonzero entries in the ith row of matrix can be calculated by $Ptr[i+1] - Ptr[i]$.

2.2 SpGEMM and Sparse Accumulator

SpGEMM operation $C = AB$ multiplies two sparse matrices A and B, and obtains a sparse matrix C. Figure 2 gives an example. It can be seen that SpGEMM can be calculated row-by-row, and can be parallelized easily since rows are independent of each other. As a result, the SpGEMM operation becomes a group of sparse vector-matrix multiplication[1] $c = aB$, where c and a are sparse vectors of C and A, respectively.

[1] This should not be confused with sparse matrix-vector multiplication (SpMV), which multiplies a sparse matrix with a dense vector and obtains a dense vector.

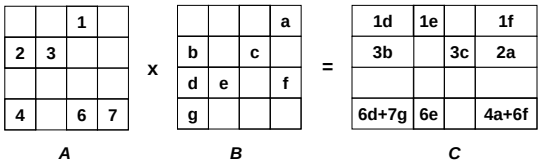


Fig. 2: An illustration of the SpGEMM operation.

This $c = aB$ operation is also known as sparse accumulator. Based on the expression, the computation of each output row c converts to the sum of the intermediate products, namely the rows of B that are selected and scaled by the nonzero elements of a. For example, when we compute the last row of C in Figure 2, we multiply the first element of the last row of A, i.e. $a_{30} = 4$, by the $0th$ row of B, i.e., $b_0 = \{a\}$, obtaining one intermediate product $c_{33} = 4a$. Then we multiply the second element of the last row of A, i.e. $a_{32} = 6$, by the $2nd$ row of B, i.e., $b_2 = \{d, e, f\}$, and obtain three intermediate products $c_{30} = 6d$, $c_{31} = 6e$, and $c_{33} = 6f$. Finally, we multiply the last element of the last row of A, i.e. $a_{33} = 7$, by the $3rd$ row of B, i.e., $b_3 = \{g\}$, to obtain $c_{30} = 7g$. For getting the last row of C, these intermediate products with the same column indexes need to be summed up. Specifically, two intermediate products $6d$ and $7g$ are summed up into c_{30}, and $4a$ and $6f$ are summed up into c_{33}. Otherwise the intermediate products without the same column indexes only need to insert themselves to the final results of c. In this case, c_{31} only needs an insertion.

2.3 Three Implementations of Sparse Accumulator

There are three typical algorithms, i.e., sort-based [7,29], merge-based [30], and hash-based [8,33,11], for implementing sparse accumulator. Also using the last row in Figure 2 as an example, sort-based sparse accumulator first sorts the five intermediate products $\{4a\}$, $\{6d, 6g, 6f\}$ and $\{7g\}$ according to their column indexes to obtain a sequence $\{6d, 7g, 6e, 4a, 6f\}$ and calls a segmented sum primitive to sum up values in the same segment (with the same column index) to obtain the final results $\{6d + 7g, 6e, 4a + 6f\}$.

The merge-based sparse accumulator runs in multiple iterations, and each iteration vertically merges one element of the resulting sparse vector. In our case, the first round merges $\{4a\}$, $\{6d\}$ and $\{7g\}$, and calculates out $\{6d+7g\}$. In the second round, $\{4a\}$ and $\{6e\}$ are merged and $\{6e\}$ is computed out. In the final round, $\{4a\}$ and $\{6f\}$ are merged into $\{4a + 6f\}$.

The hash-based sparse accumulator takes advantage of the hash functions to fast locate the positions of $\{4a\}$, $\{6d, 6g, 6f\}$ and $\{7g\}$ and to accumulate the values with the same column indexes. If conflict occurs, linear probing can be used for finding out an empty position for inserting the entry and the procedure will ensure to finish if memory space is enough large.

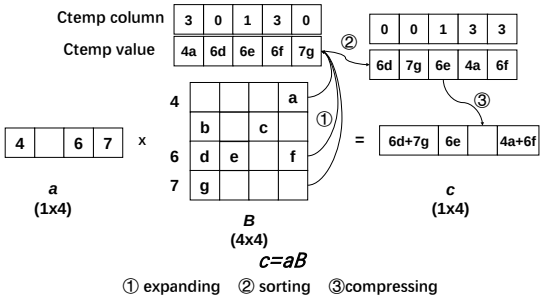


Fig. 3: Original sort implementation.

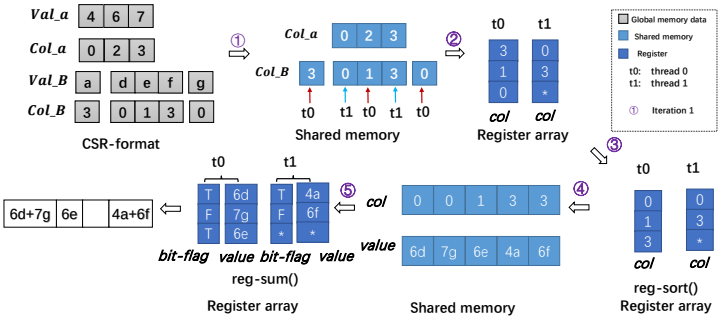


Fig. 4: Our `reg-sort` method.

The three methods are plotted in Figures 3, 5 and 7, respectively, and will be further explained along with our register-aware optimizations later on.

3 Methodology

3.1 Reg-sort: Register-Aware Sort-based Sparse Accumulator

Sort-based sparse accumulator [7,29] shown in Figure 3 has three phases: expanding, sorting and compressing. The expanding phase first generates the intermediate products, namely the rows of B that are selected and scaled by the nonzero elements of a, stored in two arrays $Ctemp\,column$ and $Ctemp\,value$ in shared memory. The second sorting phase sorts the intermediate products in the two arrays according to the $Ctemp\,column$ index in shared memory. The last compressing phase sums the values with the same column index. It can be seen that in the original implementations, all operations should be completed in shared memory, since entries stored in random locations need to compare, move and sum up.

Fortunately, there are still some patterns can be exploited. Hou et al. [32] pointed out that sorting network can be implemented more efficiently by using an N-to-M pattern that uses N threads to sort M entries inside registers. Also,

Blelloch et al. [37] developed a vectorized method for parallel segmented sum which is also used in SpMV computation [38]. We utilise the two methods in GPU registers for our sparse accumulator. To our knowledge, this is for the first time that the two methods are implemented for GPU registers.

Figure 4 shows five steps of our `reg-sort` algorithm for completing the same work in Figure 3, i.e., calculating $c = aB$ for the last row of the matrix in the Figure 2. We here use two threads to compute, and each thread has three registers allocated. We in the first step fetch and compute the corresponding intermediate products $\{4a\}$, $\{6d, 6g, 6f\}$ and $\{7g\}$ into the shared memory. In the second step, each thread fetches data from the shared memory to its own register vectors. Specifically, thread 0 gets column indexes $\{3, 1, 0\}$, and thread 1 gets $\{0, 3\}$. Then in the third step we use the N-to-M sorting network pattern for sorting the data in the register. Here the thread shuffle instructions are heavily used for comparing and moving entries between threads. After this step, the two threads store $\{0, 1, 3\}$ and $\{0, 3\}$ in their registers, respectively. Then in the fourth step, the column indexes are stored back to shared memory. In the fifth step, we load the values to register in a transposed pattern, meaning that the two threads now have $\{6d, 7g, 6e\}$ and $\{4a, 6f\}$, respectively. To label which values are to be summed up, we allocate an extra bit-flag register vector for each thread to support the parallel segmented sum. In this example, the bit-flag array has a TRUE flag at the beginning of each same column index segment, and has a FALSE otherwise. So the two threads now have $\{T, F, T\}$ and $\{T, F\}$, respectively. In the procedure of segmented sum, each thread first sums up values in the local segments, and communicate with the other thread to get values across different threads. Finally, the values summed up will stored in the positions with bit-flag TRUE. After the five steps, those values are stored to the global memory directly since their positions are already known.

Compared with the original method, the time consuming operations including sorting and compression are now all calculated inside registers. Though shared memory is still needed to use for exchanging data between the above steps, there is no need to use it for heavy computes and data movement.

3.2 Reg-merge: Register-Aware Merge-based Sparse Accumulator

Figure 5 shows the merge-based sparse accumulator. This method has two stages. The first stage is dividing the matrix A into small sub-matrices, if the maximum row length of the first matrix is smaller than the warp size 32, this row will be performed in the shared memory. The second stage is using one warp to multiply the sub-matrices by the second input matrix B.

We still take the last row of the matrix in Figure 2 as an example. Suppose we have four threads in one warp. Since the warp size is larger than the row length of a, the first stage of dividing the matrix into sub-matrices is not required. In the second stage, the rows of B corresponding to $\{4, 6, 7\}$, i.e., $\{a\}$, $\{d, e, f\}$ and $\{g\}$, are merged at first. The process of merging is sequential. Each thread needs to compute the minimum column index of the output row

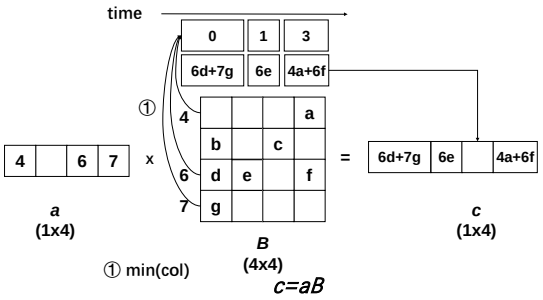


Fig. 5: Original merge implementation.

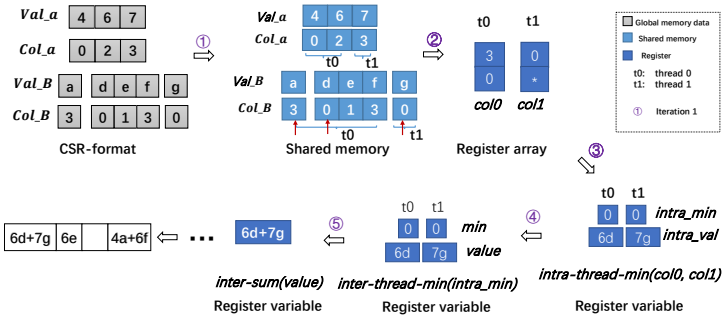


Fig. 6: Our `reg-merge` method.

c, namely $\{0\}$ in this case. Then the value of the corresponding column index $\{0\}$ of each thread is summed up using the register shuffle instructions, and the first entry $\{6d+7g\}$ is obtained. Then in the second iteration, the second entry of c is acquired. After the third iteration, all of the entries of c is obtained. This method has a disadvantage that the first stage, i.e., dividing the first input matrix into sub-matrices, consumes relatively high cost in the whole process of SpGEMM. For reducing the cost of the first stage, we devise `reg-merge` method to use more registers to make more work done in each thread. As follows we illustrate how the register merge-based algorithm computes the output row $c = aB$ using the threads within a warp.

Figure 6 shows our `reg-merge` method by using the last row of the case in Figure 2 of Section 2.2. At first the data is fetched from global memory to shared memory. The first two elements of a and the corresponding two rows of B is assigned to thread 0, and thread 1 processes the last element of a and the corresponding last row of B. In the second step, each thread fetch two column indexes of its corresponding data from shared memory to its own register vectors, which lengths are 2 in this example. For thread 1, it only has one element, so its second value of the register vector is INTMAX. Then, each thread performs an $intra - thread - min(col0, col1)$ function to obtain the private minimum column index $intra\_min$ of each private register

vector, i.e., 0 for thread 0 and 0 for thread 1 in this case. At the same time, each thread gets the scaled corresponding intermediate products for each $intra\_min$ column index, i.e., $6d$ for thread 0 and $7g$ for thread 1. Next, each thread performs $inter{-}thread{-}min(intra\_min)$ function to get the minimum column index across the threads at the same warp. The minimum column index across the two threads are stored to the register variable min of each thread and the corresponding scaled values are stored to the register variable $value$. The $inter - thread - min()$ function is implemented by the GPU register shuffle instructions and each thread can get the minimum value of each thread at the same time in the same variable names. After that, in the fifth step, using the $inter - sum(value)$ function, each thread can obtain the reduction sum of the two threads, i.e., $6d + 7g$, the first element of the output c. The $inter - sum()$ function is also implemented by the GPU register shuffle instructions and each thread can acquire the same reduction sum of the two threads. Using the same process, we can obtain the other two elements of c, i.e., $6e$ and $4a + 6f$. Then computing the row of C is completed.

Compared with the original method, the time consuming operations including dividing the matrix into sub-matrices in the global memory is now completely eliminated and all of the intermediate products are calculated inside registers.

3.3 Reg-hash: Register-Aware Hash-based Sparse Accumulator

Hash-based sparse accumulator [8,33,11] shown in Figure 7 also has three phases. The first phase is the hashing operations. It allocates a memory space of the size of the number of intermediate products, namely the rows of B that are selected and scaled by the nonzero elements of a, as the hash table. It uses the column index of these intermediate products as the key, and is able to leverage different hash methods to implement SpGEMM. For the values with the same key, the scaled values need to be summed together to get one entry of the output c. All of these operations are completed in the shared memory using the atomic function $atomicCAS()$ to sequentially access the same position of the hash table for each thread. After the hash operations, it needs to shrink the hash table to a dense state, i.e., putting the three effective values $\{4a + 6f, 6e, 6d + 7g\}$ to the first three positions of the hash table as shown in Figure 7. At last, sort the values of these elements of c according to their column indexes to obtain the final compressed storage format. In this case, we take two threads to accomplish the computing of the row. The hash operations need four iterations: 1) thread 0 puts $\{4a\}$ and thread 1 puts nothing to the hash table; 2) thread 0 puts $\{6d\}$ and thread 1 puts $\{6f\}$ to the hash table; 3) thread 0 puts $\{6f\}$ and thread 1 puts nothing to the hash table; 4) thread 0 puts $\{7g\}$ and thread 1 put nothing to the hash table.

In our register-based hash `reg-hash` algorithm, we optimize the data allocations of each thread, rather than implementing the hash table in registers. Also, we take the example of $c = aB$ in Section 2.2 to illustrate the process

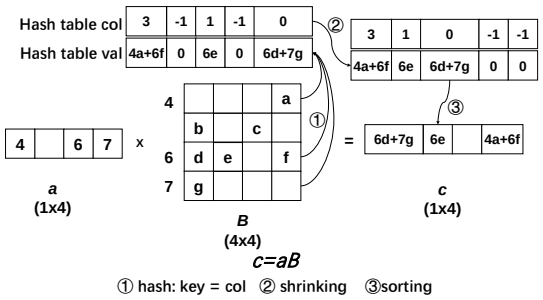


Fig. 7: Original hash implementation.

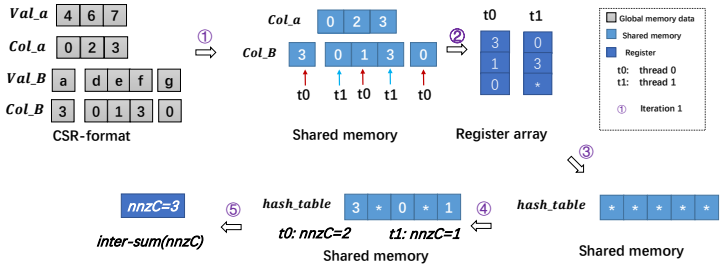


Fig. 8: Our `reg-hash` method.

of our `reg-hash` algorithm, as shown in Figure 8. In this case, we use two threads to compute $c = aB$. Each thread has a register vector with the size of three. At first, we load data from the input matrices to the shared memory as shown in the first step of Figure 8. After that, each thread fetches data from the shared memory to its own register vectors alternately. Thread 0 gets the column index data {3}, {1}, and {0} to its register vectors, and thread 1 gets the column index data {0} and {3} to its register vectors. The computing of the indexes is a bit more complex. We need to compute both the index of the shared memory tables and the register arrays precisely. In the third steps, the hash table shared by the two threads in the shared memory is initialized. Then, each thread fetches the data from its register arrays to perform the hash operations of the hash table in the shared memory. The hash operations need three iterations: 1) thread 0 puts{3} and thread 1 puts {0} to the hash table; 2) thread 0 puts {1} and thread 1 puts {3} to the hash table; 3) thread 0 puts {0} and thread 1 puts nothing to the hash table. The number of times of accessing shared memory is less than that of the original hash method, leading to the performance improvement. Each thread has its private register variable $nnzC$ to record the partial number of nonzeros of the output c for each thread. For the hash operations, the $nnzC$ with the same key of each thread, namely the same column index, is only accumulated once for each threads. We use the atomic function $atomicCAS()$ to guarantee accuracy of parallel hash operations. At last, in the fifth step, the same function $inter - sum(nnzC)$

of `reg-merge` is performed by each thread to obtain the total sum of partial number of nonzeros of the output c, i.e., $nnzC$ of each threads. Finally, the real size of c is acquired. We use the `reg-hash` to compute the size of the output matrix.

Compared with the original hash method, though shared memory is still needed to use for hash operations, the total number of shared memory hash operation can decrease significantly, since now the intermediate products are well organized and accumulated into their final positions in a load balanced way.

3.4 Implementation Details

For the computation of each row of C, the numbers of the floating point operations, twice the number of intermediate products, can vary greatly because of the various sparsity structures of the two input matrices A and B. To achieve load balanced calculation on GPUs, we create a couple of bins and then groups rows requiring the similar number of floating point operations into the same bin. Here we focus on the rows having intermediate products between 0 to 512. We divide those rows into nine bins according to their intermediate products. The number of intermediate products is $0-2$ for bin0, $3-4$ for bin1, $5-8$ for bin2, $9-16$ for bin3, $17-32$ for bin4, $33-64$ for bin5, $65-128$ for bin6, $129-256$ for bin7, and $257-512$ for bin8.

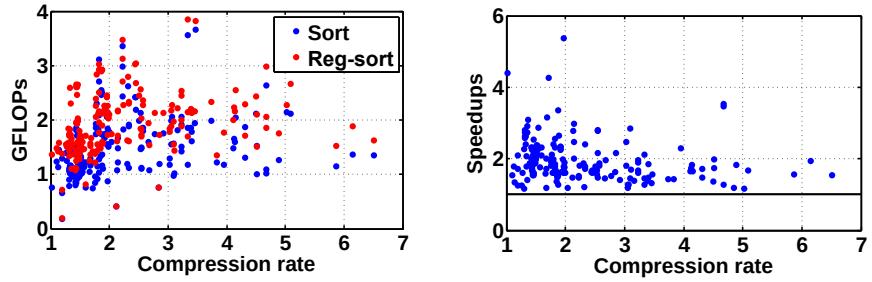
The register number for each method used in each thread is different. For `reg-sort` and `reg-hash`, each thread uses the register vector with the size of 8. For `reg-merge`, each thread can use the register vector with the size of 2, 4, 8, 16 or 32, depending on the length of the rows of the first input matrix of SpGEMM.

4 Evaluation

The experimental evaluation is conducted on an Nvidia Pascal P100 GPU (3584 CUDA cores and 16GB HBM2 memory) hosted by an Intel Xeon server. The programs are all compiled by CUDA v8.0 and Intel C/C++ compiler v18.

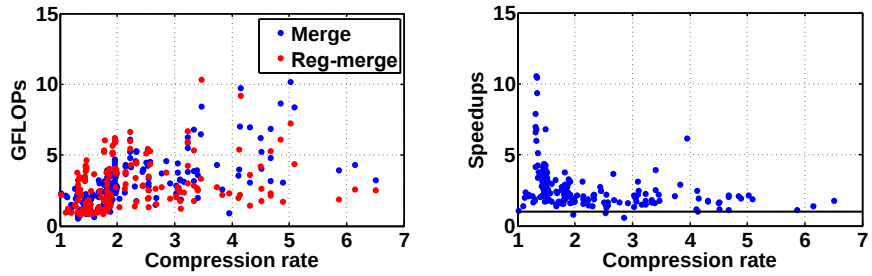
As for the benchmark suite, we first select 956 enough large matrices used in our previous work [16] (with the number of nonzeros no less than 100K and no more than 200M) from in total 2757 sparse matrices in the SuiteSparse Matrix Collection [39]. For SpGEMM operation, we also calculate $C = A^2$ to align with the existing work [7,11,29,30]. Furthermore, to clearly show the effectiveness of the participating algorithms, we select 205 matrices which with over 95% of their rows are small enough (calculating no more than 512 intermediate products in the rows of C) to use on-chip memories.

Our algorithms are compared with three state-of-the-art SpGEMM methods in bhSPARSE [29], RMerge [30] and NSPARSE [11] on GPU. Figures 9, 10 and 11 show the overall performance (their left side) and kernel performance (their right side) of the three methods and our register-aware optimization



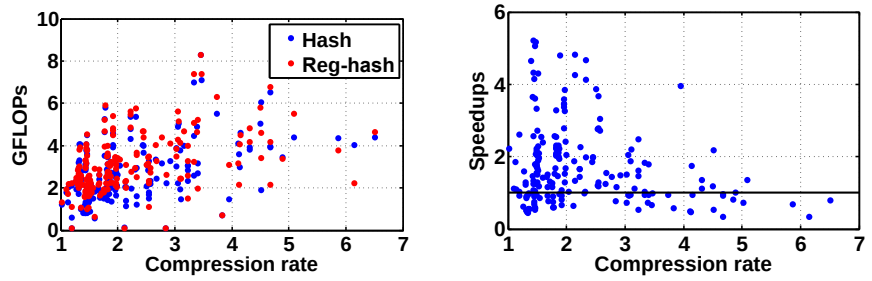
(a) Overall performance. (b) Kernel speedups.

Fig. 9: The performance and speedups of the `reg-sort` method.



(a) Overall performance. (b) Kernel speedups.

Fig. 10: The performance and speedups of the `reg-merge` method.



(a) Overall performance. (b) Kernel speedups.

Fig. 11: The performance and speedups of the `reg-hash` method.

methods, respectively. The overall performance is evaluated in terms of GFLOPs (floating point operations per second) in double-precision, and the kernel performance is shown in speedups of our sparse accumulators over their original implementations. The x-axis is the factor *compression rate* denoting the ratio of the number of intermediate products and the number of nonzeros of the output matrix C.

In Figures 9a, 10a and 11a, it can be seen that the performance of our register-aware optimization in general brings better performance over bhSPARSE, RMerge and NSPARSE. We obtain average 1.3x (up to 2.0x), 1.1x (up to 5.4x), and 1.2x (up to 2.7x) speedups for the overall performance over the three methods, respectively. In the `reg-sort` method, the speedups mostly come from the fast in-register implementations of sorting network and segmented sum. Matrices *Baumann*, *ncvxqp3*, and *wathen120* obtain the speedups of 2.0x, 1.9x and 1.9x, respectively. For `reg-merge`, the improvements of the performance mainly comes from assigning more work to each thread, and reducing the expensive global and shared memory accesses. Matrices *netherlands_osm*, *road_central* and *great-britain_osm* obtain 5.4x, 5.2x, and 4.7x speedups, respectively. As for `reg-hash`, the improvement of the performance is most from more balanced workload when performing the hash operations. Matrices *ASIC_100ks*, *al2010* and *id2010* obtain the speedups of 2.7x, 2.3x, and 2.3x, respectively.

Figures 9b, 10b, and 11b compare the pure performance of sparse accumulators including load and store cost between global memory and on-chip memory and compute overhead. In other words, it is the execution time of all CUDA kernels to finish an SpGEMM operation. Overall, our three implementations obtain average speedups 2.0x (up to 5.4x), 2.6x (up to 10.5x), and 1.7x (up to 5.2x) over bhSPARSE, RMerge and NSPARSE, respectively. Those kernel speedups are higher than the full SpGEMM, since the complete SpGEMM algorithm also includes binning, memory allocation and possible data copy overhead. In detail, for `reg-sort`, matrices *Baumann*, *ch7-8-b5*, *ncvxqp3* obtain the highest speedups 5.4x, 4.4x, and 4.3x, respectively. For `reg-merge`, matrices *europe_osm*, *road_usa*, *road_central* obtain the highest speedups 10.5x, 10.4x, and 9.3x, respectively. For `reg-hash`, matrices *id2010*, *al2010*, *co2010* obtain the highest speedups 5.2x, 5.2x, and 5.1x, respectively. It is also worth to note that kernel performance of `reg-sort` and `reg-merge` is almost constantly better than the original method, while `reg-hash` is not always better than the original one. The reason is that some matrices already bring rather good load balancing and adequate floating point operations. Thus the load balanced computations may lead to lower performance.

5 Related Work

There has been much work focusing on **parallel SpGEMM algorithms**. Bell et al. [1] developed the ESC (expanding, sorting and compressing) method, and Dalton et al. further improved it for better locality [7] and better load balancing [40]. Liu and Vinter [29] grouped rows into 37 bins and utilized various sorting algorithms, i.e., heap sort, bitonic sort and mergepath sort, for different bins. Gremse et al. [30] merged rows of B in a vertical way. Demouth [8] for the first time used hash table for sparse accumulator, and both Pham et al. [33] and Nagasaka et al. [11] took load balancing into consideration in hash methods. Deveci et al. [23] used hash methods on KNL and GPUs.

Moreover, Buluç et al. [12], Azad et al. [13], Ballard et al. [14], Akbudak et al. [10] and Nagasaka et al. [20] proposed various novel SpGEMM algorithms for many-core x86 processors and distributed memory machines. Besides, for the sparse matrix problem, the memory bandwidth is also an important factor to analyse [41].

Additionally, **GPU registers** have been proven to be an effective tool for implementing faster execution models and algorithms. For example, Li et al. [34,35] pointed out that register reuse and communication are crucial for more fine-grained GPU execution model. Rawat et al. [36] demonstrated that various stencil computations can be accelerated through GPU registers. Xie et al. [42] designs a framework to allocate the registers by analyzing the lifetime of variables.

In spite of the previous efforts, register-aware optimization is still not well studied for SpGEMM algorithms. Compared to the existing literature, our work presented in this paper exploits GPU registers to improve three representative sparse accumulators, i.e., sort, merge and hash, in various SpGEMM algorithms and achieves significant speedups on modern GPUs.

6 Conclusion

We in this paper have proposed three register-aware optimization techniques to improve performance of SpGEMM. The three newly implemented sparse accumulators covered representative parallel primitives, i.e., sort, merge and hash, and demonstrated significant speedups over their original implementations. To our knowledge, this is for the first time that registers are fully utilised for accelerating SpGEMM on massively parallel architectures. In the future, we would like to explore the relationship between the features of sparse matrices and the three different spare accumulators, and exploit the data reuse of SpGEMM [43]. Furthermore, we are interested in performing our algorithms to the real world applications using existing autotuning technologies [44,45].

Acknowledgements We would like to express our gratitude to all reviewers constructive comments for helping us polishing this paper. This work is supported by the National Key Research and Development Program of China (2017YFB0202105, 2016YFB0201305, 2016YFB0200803, 2016YFB0200300), National Natural Science Foundation of China, under grant no. (61521092, 91430218, 31327901, 61472395, 61432018), and the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie project (Grant No. 752321).

References

1. Bell, N., Dalton, S., Olson, L.: Exposing fine-grained parallelism in algebraic multigrid methods. SIAM Journal on Scientific Computing **34**(4) (2012) C123–C152
2. D'Alberto, P., Nicolau, A.: R-kleene: A high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks. Algorithmica **47**(2) (2007) 203–213

3. Zhang, F., Lin, H., Zhai, J., Cheng, J., Xiang, D., Li, J., Chai, Y., Du, X.: An adaptive breadth-first search algorithm on integrated architectures. The Journal of Supercomputing (Aug 2018)
4. Azad, A., Pavlopoulos, G.A., Ouzounis, C.A., Kyrpides, N.C., Buluç, A.: Hipmcl: a high-performance parallel implementation of the markov clustering algorithm for large-scale networks. Nucleic Acids Research **46**(6) (2018) e33
5. Mattson, T.G., Yang, C., McMillan, S., Buluç, A., Moreira, J.E.: GraphBLAS C API: Ideas for future versions of the specification. In: IEEE High Performance Extreme Computing Conference (HPEC). (2017)
6. Davis, T.A.: Graph algorithms via SuiteSparse: GraphBLAS: triangle counting and k-truss. In: IEEE High Performance Extreme Computing Conference (HPEC). (2018)
7. Dalton, S., Olson, L., Bell, N.: Optimizing sparse matrix-matrix multiplication for the gpu. ACM Trans. Math. Softw. **41**(4) (October 2015) 25:1–25:20
8. Demouth, J.: Sparse matrix-matrix multiplication on the gpu. GTC '12 (2012)
9. Kunchum, R., Chaudhry, A., Sukumaran-Rajam, A., Niu, Q., Nisa, I., Sadayappan, P.: On improving performance of sparse matrix-matrix multiplication on gpus. In: Proceedings of the International Conference on Supercomputing. ICS '17 (2017) 14:1–14:11
10. Akbudak, K., Aykanat, C.: Exploiting locality in sparse matrix-matrix multiplication on many-core architectures. IEEE Transactions on Parallel and Distributed Systems **28**(8) (Aug 2017) 2258–2271
11. 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). (Aug 2017) 101–110
12. Buluç, A., Gilbert, J.R.: Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. SIAM Journal on Scientific Computing **34**(4) (2012) C170–C191
13. Azad, A., Ballard, G., Buluç, A., Demmel, J., Grigori, L., Schwartz, O., Toledo, S., Williams, S.: Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. SIAM Journal on Scientific Computing **38**(6) (2016) C624–C651
14. Ballard, G., Druinsky, A., Knight, N., Schwartz, O.: Hypergraph partitioning for sparse matrix-matrix multiplication. ACM Trans. Parallel Comput. **3**(3) (December 2016) 18:1–18:34
15. Yuster, R., Zwick, U.: Fast sparse matrix multiplication. ACM Trans. Algorithms **1**(1) (July 2005) 2–13
16. 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. PPoPP '18 (2018) 407–408
17. Liu, W.: Parallel and Scalable Sparse Basic Linear Algebra Subprograms. PhD thesis, University of Copenhagen (2015)
18. Liu, W., Vinter, B.: A framework for general sparse matrix-matrix multiplication on gpus and heterogeneous processors. Journal of Parallel and Distributed Computing **85**(C) (2015) 47–61
19. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. Journal of Parallel and Distributed Computing **74**(12) (2014) 3202 – 3216
20. Nagasaka, Y., Matsuoka, S., Azad, A., Buluç, A.: High-performance sparse matrix-matrix products on intel knl and multicore architectures. In: Proceedings of the 47th International Conference on Parallel Processing Companion. ICPP '18 Workshop (2018) 34:1–34:10
21. Gremse, F., Kpper, K., Naumann, U.: Memory-efficient sparse matrix-matrix multiplication by row merging on many-core architectures. SIAM Journal on Scientific Computing **40**(4) (2018) C429–C449
22. Buluç, A., Gilbert, J.R.: Challenges and advances in parallel sparse matrix-matrix multiplication. In: 2008 37th International Conference on Parallel Processing. (Sept 2008) 503–510

23. Deveci, M., Trott, C., Rajamanickam, S.: Multithreaded sparse matrix-matrix multiplication for many-core and gpu architectures. Parallel Computing **78** (2018) 33 – 46
24. Gustavson, F.G.: Two fast algorithms for sparse matrices: Multiplication and permuted transposition. ACM Trans. Math. Softw. **4**(3) (September 1978) 250–269
25. Zhang, F., Wu, B., Zhai, J., He, B., Chen, W.: FinePar: Irregularity-Aware Fine-Grained Workload Partitioning on Integrated Architectures. In: Proceedings of the 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). (2017) 27–38
26. Zhang, F., Zhai, J., He, B., Zhang, S., Chen, W.: Understanding co-running behaviors on integrated cpu/gpu architectures. IEEE Transactions on Parallel and Distributed Systems **28**(3) (March 2017) 905–918
27. Tan, G., Liu, J., Li, J.: Design and implementation of adaptive spmv library for multicore and many-core architecture. ACM Trans. Math. Softw. **44**(4) (August 2018) 46:1–46:25
28. 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. ICS '15 (2015) 339–350
29. Liu, W., Vinter, B.: An efficient gpu general sparse matrix-matrix multiplication for irregular data. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium. IPDPS '14 (May 2014) 370–381
30. Gremse, F., Höfter, A., Schwen, L.O., Kiessling, F., Naumann, U.: Gpu-accelerated sparse matrix-matrix multiplication by iterative row merging. SIAM Journal on Scientific Computing **37**(1) (2015) C54–C71
31. Gilbert, J., Moler, C., Schreiber, R.: Sparse matrices in matlab: Design and implementation. SIAM Journal on Matrix Analysis and Applications **13**(1) (1992) 333–356
32. 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) 12:1–12:10
33. Anh, P.N.Q., Fan, R., Wen, Y.: Balanced hashing and efficient gpu sparse general matrix-matrix multiplication. In: Proceedings of the 2016 International Conference on Supercomputing. ICS '16 (2016) 36:1–36:12
34. Li, A., Song, S.L., Liu, W., Liu, X., Kumar, A., Corporaal, H.: Locality-aware cta clustering for modern gpus. In: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '17 (2017) 297–311
35. Li, A., Liu, W., Wang, L., Barker, K., Song, S.L.: Warp-consolidation: A novel execution model for gpus. In: Proceedings of the 2018 International Conference on Supercomputing. ICS '18 (2018) 53–64
36. Rawat, P.S., Rastello, F., Sukumaran-Rajam, A., Pouchet, L.N., Rountev, A., Sadayappan, P.: Register optimizations for stencils on gpus. In: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '18 (2018) 168–182
37. Blelloch, G.E., Heroux, M.A., Zagha, M.: Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors. Technical report, CMU (1993)
38. Liu, W., Vinter, B.: Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors. Parallel Computing **49**(C) (2015) 179–193
39. Davis, T.A., Hu, Y.: The university of florida sparse matrix collection. ACM Trans. Math. Softw. **38**(1) (December 2011) 1:1–1:25
40. Dalton, S., Baxter, S., Merrill, D., Olson, L., Garland, M.: Optimizing sparse matrix operations on gpus using merge path. In: 2015 IEEE International Parallel and Distributed Processing Symposium. IPDPS '15 (May 2015) 407–416
41. Li, A., Liu, W., Kristensen, M.R.B., Vinter, B., Wang, H., Hou, K., Marquez, A., Song, S.L.: Exploring and analyzing the real impact of modern on-package memory on hpc scientific kernels. SC '17 (2017) 26:1–26:14
42. Xie, X., Liang, Y., Li, X., Wu, Y., Sun, G., Wang, T., Fan, D.: Enabling coordinated register allocation and thread-level parallelism optimization for gpus. In: 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). (Dec 2015) 395–406

43. Yuan, L., Liu, J., Luo, Y., Tan, G.: Locality of computation for stencil optimization. In Carretero, J., Garcia-Blas, J., Ko, R.K., Mueller, P., Nakano, K., eds.: Algorithms and Architectures for Parallel Processing, Cham, Springer International Publishing (2016) 449–456
44. Liu, J., Tan, G., Luo, Y., Li, J., Mo, Z., Sun, N.: An autotuning protocol to rapidly build autotuners. ACM Trans. Parallel Comput. (2018)
45. Li, J., Tan, G., Chen, M., Sun, N.: SMAT: An input adaptive auto-tuner for sparse matrix-vector multiplication. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '13 (2013) 117–126