2017 46th International Conference on Parallel Processing

High-performance and Memory-saving Sparse General Matrix-Matrix Multiplication for NVIDIA Pascal GPU

Yusuke Nagasaka, Akira Nukada, Satoshi Matsuoka
Tokyo Institute of Technology
Tokyo, Japan
nagasaka.y.aa@m.titech.ac.jp

***Abstract*—Sparse general matrix-matrix multiplication (SpGEMM) is one of the key kernels of preconditioners such as algebraic multigrid method or graph algorithms. However, the performance of SpGEMM is quite low on modern processors due to random memory access to both input and output matrices. As well as the number and the pattern of non-zero elements in the output matrix, important for achieving locality, are unknown before the execution. Moreover, the state-of-the-art GPU implementations of SpGEMM requires large amounts of memory for temporary results, limiting the matrix size computable on fast GPU device memory. We propose a new fast SpGEMM algorithm requiring small amount of memory and achieving high performance. Calculation of the pattern and value in output matrix is optimized by using GPU ' s on-chip shared memory and a hash table. Additionally, our algorithm launches multiple kernels running concurrently to improve the utilization of GPU resources. The kernels for the calculation of each row of output matrix are chosen based on the number of non-zero elements. Performance evaluation using matrices from the Sparse Matrix Collection of University Florida on NVIDIA ' s Pascal generation GPU shows that our approach achieves speedups of up to x4.3 in single precision and x4.4 in double precision compared to existing SpGEMM libraries. Furthermore, the memory usage is reduced by 14.7% in single precision and 10.9% in double precision on average, allowing larger matrices to be computed.**

***Keywords*-Sparse matrix, SpGEMM, GPU**

I. Introduction

Numerical applications such as scientific simulations and graph processing often require sparse matrix computation. Algebraic multigrid (AMG) method [1] for preconditioner of iterative method in solving equations and some graph algorithms such as graph clustering [2] and breadth-first search (BFS) [3] compute matrix multiplication of sparse matrices. Generally, sparse matrix data is compressed by holding only non-zero elements to reduce memory usage and amount of computation. In sparse general matrix-matrix multiplication (SpGEMM), input and output matrices should be compressed. However, the number and pattern of non-zero elements in the output matrix depend on the input matrices and are unknown before execution. This property makes it hard to allocate memory of the output matrix. Furthermore, unlike dense matrix computations, SpGEMM causes frequent cache misses because of indirect memory access to the input matrix [4].

Recently, Graphics Processing Unit (GPU) is utilized for general purpose computations. The computational kernels are accelerated by GPU's high computing power. Sparse matrix computations are also ported to GPUs [5]–[7], especially, sparse matrix vector multiplication (SpMV) is highly accelerated due to high parallelism and memory bandwidth of GPU. Although SpGEMM is also accelerated by GPU [1], [8], existing approaches require much larger memory compared to final usage of output matrix because it is not easy to manage the memory allocation in massive thread executing. As a result, the applicable matrix data is limited by the capacity of GPU's device memory. Furthermore, since high parallelism and various types of memory on GPU are not utilized efficiently, SpGEMM is not fully accelerated on GPU devices.

We propose a state-of-the-art algorithm which not only accelerates SpGEMM computation on GPU but also reduces memory usage to compute various matrix data. Our approach utilizes GPU's on-chip shared memory and assigns GPU resources in a step-by-step manner by grouping the rows based on the number of non-zero elements or computational complexity. We evaluate the performance of our SpGEMM algorithm for various kinds of 12 matrix datasets taken from University of Florida Sparse Matrix Collection [9]. Compared to existing sparse libraries: CUSP, cuSPARSE and BHSPARSE, we achieve the maximum speedups of x4.3 in single precision and x4.4 in double precision on brand new NVIDIA's Pascal generation GPU. The memory usage of our approach is reduced from other sparse matrix libraries for all matrix datasets, and we achieve 14.7% reduction on average in single precision and 10.9% in double precision. We also examine the performance of SpGEMM for large size matrix data generated in graph analysis. BHSPARSE and CUSP cannot handle some of these large size matrix data since they consume much memory space. For these matrices, our proposal shows significant speedups up to x11.6 compared to cuSPARSE with low memory usage.

2332-5690/17 $31.00 © 2017 IEEE
DOI 10.1109/ICPP.2017.19

II. Background

We first introduce sparse matrix format which reduces both memory usage and computations. Then we describe the sequential SpGEMM method and discuss the problems of SpGEMM including GPU case.

A. Sparse Matrix Format

When a matrix has many zero elements, the matrix is compressed by removing zero elements and the sparse matrix format holds only non-zero elements and pointers/indices. Coordinated (COO) and Compressed Sparse Row (CSR) are widely-used sparse matrix formats. COO format simply contains the array of tuples of row, column and value for each non-zero element. CSR format has an array of pointers to the beginning of each row, called row pointer (rpt), instead of holding row indices of each non-zero element. CSR format requires less memory usage compared to COO format.

In order to accelerate SpMV, many of sparse matrix formats have been proposed [10], [11], and recently some sparse matrix formats show the outstanding performance of SpMV on GPUs [12], [13]. These sparse matrix formats usually require format conversion from standard format such as CSR or COO format. In iterative methods which require many times of SpMV, the acceleration of SpMV with converting matrix to other sparse matrix format is an useful way since the cost of format conversion can be negligible. On the other hand, when the execution time of sparse matrix computation is relatively small, the computation should be executed without format conversion since the cost of format conversion becomes relatively large.

B. Sparse General Matrix-Matrix Multiplication

We first describe the sequential basic SpGEMM. Let A, B be input matrices, and SpGEMM computes a matrix C such that $C = AB$. The input and output matrices are sparse and they are stored in sparse matrix format. When the matrices are stored in CSR format, SpGEMM computation is executed for each row of output matrix. Algorithm 1 shows the pseudo code of SpGEMM in CSR format. Let a_{ij} be the element in i-th row and j-th column of matrix A and a_{i*} be the i-th row of matrix A. SpGEMM is executed for each row of matrix A, that is, matrix C. The row of matrix B corresponding to each non-zero element of matrix A is read, and each non-zero element of output matrix C is calculated.

SpGEMM computation has three critical issues unlike dense matrix multiplication. First issue is indirect memory access to matrix data. According to Algorithm 1, the memory access to matrix B depends on the non-zero elements of matrix A. Therefore, the memory access to each non-zero element of matrix B is indirect, and the cache miss may occur frequently. Second is that the pattern and the number of non-zero elements of output matrix are not known beforehand. For this reason, the memory allocation of output matrix becomes hard, and we need to select from two strategies. One is a two-pass strategy, which counts the number of non-zero elements of output matrix first, and then allocates memory and computes. The other is that we allocate enough large memory space for output matrix and compute. The former requires more computation cost, and the latter uses much more memory space. Finally, third issue is about combining the intermediate products ($value$ in Algorithm 1) to non-zero elements of output matrix. Since the output matrix is also sparse, it is hard to efficiently combine intermediate products. This procedure becomes a performance bottleneck of SpGEMM computation.

Algorithm 1 Pseudo code of SpGEMM

```
set matrix C to ∅
for all a_{i*} in matrix A do
  for all a_{ik} in row a_{i*} do
    for all b_{kj} in row b_{k*} do
      value ← a_{ik} b_{kj}
      if c_{ij} ∈ c_{i*} then
        insert (c_{ij} ← value) to c_{i*}
      else
        c_{ij} ← c_{ij} + value
      end if
    end for
  end for
end for
```

SpGEMM has been accelerated on GPU from CPUs by exploiting its high parallelism and computing power [17]. In GPU case, we should additionally consider how to improve the load balance between threads. Since the computation cost of each row largely varies, the threads should be carefully assigned to rows. Bell et al. proposed ESC algorithm, which decomposes SpGEMM computation to GPU friendly procedures in order to improve load balance [1]. The ESC algorithm consists of three phases; expansion, sorting and contraction. First, the algorithm generates the list of tuple of row index, column index and value for all intermediate products of matrix-matrix multiplication. Next, the entries in the list are sorted by the row and column indices. Finally, the intermediate products with same row and column indices are accumulated, and the algorithm outputs the sets of row index, column index and value of each non-zero element. Each phase of ESC algorithm exploits high parallelism of GPU. However, since ESC algorithm handles extremely large amount of intermediate data, the performance of SpGEMM is low and the large memory usage is required. For these problems, Dalton et al. improved ESC algorithm by

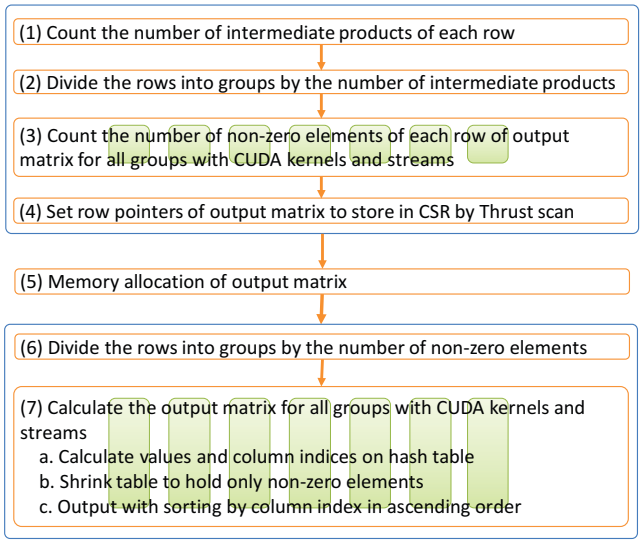


Figure 1: Flow of our SpGEMM algorithm

utilizing on-chip shared memory [14]. Nevertheless, the performance is still low since the base of the algorithm is ESC and the problem is not fundamentally solved.

III. Proposal

Many of researchers have proposed new SpGEMM algorithms for GPUs. However, the acceleration comes from requiring large amount of working memory. We propose the state of the art algorithm which accelerates SpGEMM on GPU and reduces memory usage. Our algorithm allocates hash table on fast shared memory and appropriately divides the rows into groups in order to exploit GPU resource. Figure 1 shows the flow of our proposed SpGEMM algorithm. Our proposal consists of two phases not to require redundant working memory. First phase for counting the number of non-zero elements of output matrix is from (1) to (4), and second phase for calculating the output matrix is (6) and (7) in Figure 1. In order to exploit GPU resource more efficiently including better load balancing, the rows are divided into groups based on the number of intermediate products in (2) and on the number of non-zero elements in (6).

We describe three key phases: grouping, counting the number of non-zero elements and calculation of output matrix. Finally, we describe the parameters setting for each group. In the following of this paper, let A, B be input matrices and C be output matrix, and we compute $C = AB$. All input and output matrices are stored in CSR format.

A. Grouping

Since the computational cost of each row varies in SpGEMM computation, simple parallelization may cause load imbalance. In order to improve the load balance and to exploit GPU resources such as shared memory, our algorithm divides the rows into groups by the number of intermediate products or non-zero elements of output matrix. The number of non-zero elements of each row in output matrix is unknown at grouping phase (2). Therefore, after counting the intermediate products of each row in (1), the rows are divided into groups by the number of intermediate products. In grouping phase (6), the rows are divided into groups based on the number of non-zero elements, which are counted in (3). Since the execution cost of reordering whole input matrix is too large, the grouping is done by generating the array of gathered row indices corresponding to each group. This array size is additional memory usage on GPU in our algorithm.

Counting the number of intermediate products: Algorithm 2 shows how to calculate the number of intermediate products of each row in (1). The number of intermediate products is an upper limit of the number of non-zero elements of output matrix. Since this procedure requires only the row pointers and column indices of matrix A and the row pointers of matrix B, the execution cost is relatively small compared to whole SpGEMM execution.

Algorithm 2 Count the number of intermediate products of i-th row

$n_{prod} \leftarrow 0$
for $j = rpt_A[i]$ to $rpt_A[i+1]$ **do**
  $n_{prod} \leftarrow n_{prod} + (rpt_B[col_A[j]+1] - rpt_B[col_A[j]])$
end for

B. Counting the number of non-zero elements

The number of intermediate products is typically much larger than the number of non-zero elements in output matrix. This is because same column may be counted for multiple times. We count the number of non-zero elements of each row in output matrix with considering overlapping of the column indices. We improve memory access to input matrices on GPU, and employ hash tables to manage the column indices with considering overlapping. In order to exploit GPU resources and improve the load balance, we set different hash table size, thread assignment and thread block size for each group. Our approach launches multiple CUDA kernels with different CUDA streams for each group to execute the kernels in parallel.

1) Thread Assignments and Memory Access Optimization: We combine two-way thread assignments; PWARP/ROW and TB/ROW, where partial warp is bundle of 4 threads, and TB means thread block. Our algorithm sets better one for each group to improve the load balance. The PWARP/ROW assigns 4 threads for each row in matrix A, one thread for each non-zero element of matrix

Algorithm 3 Count the number of non-zero elements of i-th row by PWARP/ROW

```
tid ← threadIdx%4
for j ← rpt_A[i] to rpt_A[i + 1] stride 4 do
  d ← col_A[j + tid]
  for k ← rpt_B[d] to rpt_B[d + 1] stride 1 do
    //hash operation
  end for
end for
```

Algorithm 4 Count the number of non-zero elements of i-th row by TB/ROW

```
tid ← threadIdx%warpsize
wid ← threadIdx/warpsize
wnum ← blockDim/warpsize
for j ← rpt_A[i] + wid to rpt_A[i + 1] stride wnum do
  d ← col_A[j]
  for k ← rpt_B[d] + tid to rpt_B[d + 1] stride 32 do
    //hash operation
  end for
end for
```

A and corresponding row of matrix B. Algorithm 3 shows how to count the number of non-zero elements by PWARP/ROW. We did preliminary evaluation with changing the number of threads per row as 1, 2, 4, 8 and 16. In the result, 4 threads per row stably shows best performance. Thus, we select 4 threads as PWARP. Since the PWARP/ROW executes the procedure of each row by only 4 threads, the PWARP/ROW is selected for the groups with small numbers of intermediate products or non-zero elements. The warp shuffle function is utilized for accumulating the numbers of non-zero elements of each thread in same PWARP, and the numbers of non-zero elements in each row are output.

The TB/ROW assigns one thread block for each row of matrix A, one warp for each non-zero element of matrix A and one thread for each non-zero element of matrix B. Algorithm 4 shows how to count the number of non-zero elements by TB/ROW. Since the procedure of each row is executed by many threads, the TB/ROW is selected for the groups which have large numbers of intermediate products or non-zero elements. To calculate the number of non-zero elements of each row, counted numbers of non-zero elements by each thread in same warp are accumulated by the warp shuffle function as well as PWARP/ROW, and then the TB/ROW accumulates the sum of each warp by utilizing shared memory.

2) Hash Table: It is most time-consuming part in SpGEMM computation to combine the intermediate products with same row and column indices into one non-zero element. Our proposal utilizes hash table to accelerate SpGEMM computation. The hash table is utilized not only for counting the numbers of non-zero elements of each row in (3), but also for calculation of values of output matrix in (7). Algorithm 5 shows hash algorithm of our proposal. The column indices are inserted into hash table as keys. The hash table is initialized by storing -1 since the column indices are no less than 0. The column index is multiplied by constant number, $HASH\_SCAL$, and divided by hash table size to compute the remainder. The $hash$ in Algorithm 5 is the result of this calculation. The hashing algorithm is based on linear probing. When the hash is collided, the hashing algorithm tries next entry until occurring no hash collision. Since the entry in hash table would be accessed by multiple threads simultaneously, the operation should be executed exclusively by compare-and-swap (atomicCAS).

Algorithm 5 Hash Algorithm

```
(Hash table is initialized: table[] ← −1)
(nz is initialized: nz ← 0)
(k comes from Algorithm 3, 4)
key ← col_B[k]
hash ← (key ∗ HASH_SCAL)%t_size
while true do
  if table[hash] = key then
    break
  else if table[hash] = −1 then
    old ← atomicCAS(table + hash, −1, key)
    if old = −1 then
      nz ← nz + 1
      break
    end if
  else
    hash ← (hash + 1)%t_size
  end if
end while
```

The size of hash table (t_{size} in Algorithm 5) is set based on the number of intermediate products or non-zero elements. The hash table can be allocated on shared memory except the group for large number of non-zero elements. For the group with large number of intermediate products, the process of counting the number of non-zero elements is executed in two phases. First phase tries to count the number of non-zero elements of each row, using the maximum hash table size for shared memory. When a row finds larger number of non-zero elements than the hash table size, the thread block records the row index, and immediately terminates its execution. After this first phase, our algorithm counts the numbers of non-zero elements for recorded rows by allocating enough large hash table on global memory. The size of hash table on global memory is set based on the number of intermediate

Table I: Parameter setting for each group on Tesla P100

Group ID	(3) Num of intermediate products	(6) Num of non-zero elements	Assignment	Thread Block size	#TB
0	8193-	4097-	TB/ROW	1024	2
1	4097-8192	2049-4096	TB/ROW	1024	2
2	2049-4096	1025-2048	TB/ROW	512	4
3	1025-2048	513-1024	TB/ROW	256	8
4	513-1024	257-512	TB/ROW	128	16
5	33-512	17-256	TB/ROW	64	32
6	0-32	0-16	PWARP/ROW	512	4

products. Since the number of non-zero elements is usually much smaller than the number of intermediate products, most of rows complete in the first phase.

C. Calculation of Output Matrix

Calculation of output matrix C consists of three phases. First phase computes the values and column indices of output matrix by utilizing hash table as well as counting the number of non-zero elements of each row in C. Our algorithm employs another table for the calculation of values. After the calculation of each row finishes, the hash table contains the computed non-zero elements. Second phase gathers non-zero elements in the hash table. Finally, third phase sorts the non-zero elements in ascending order of column indices. The thread is assigned to each non-zero element in hash table, and computes the order of the column index of the non-zero element by comparing the other non-zero elements in same hash table. According to this order, the non-zero elements are output to global memory and SpGEMM computation finishes. When the hash table can be allocated on shared memory, all phases of calculating output matrix are executed on shared memory.

D. Parameter Setting for Each Group

In order to improve the load balance and exploit shared memory, it is important to set appropriate parameters for our proposal. We describe how to set the hash table size and thread block size for each group. Table I shows the parameter setting on Tesla P100 GPU. NVIDIA GPUs have few dozen streaming multiprocessors (SMs). Each SM implements 64 CUDA cores and on-chip shared memory. The size of shared memory per SM is 64 KB and maximum shared memory size per thread block is 48 KB on P100.

First, we determine the group of the largest hash table that can be allocated on shared memory. Algorithm 5 requires modulus operation of hash table size, t_{size}. Since the modulus operation is expensive, we utilize lightweight bit operations by setting t_{size} to powers of two. In double precision, the hash tables need 8 bytes for each value data, and 4 bytes for each column index. When computing output matrix by TB/ROW, each thread block requires $t_{size} \times 12[\mathrm{Byte}]$ for hash table. Therefore, the largest table size is set as $t_{size} = 48[\mathrm{KB}]/12[\mathrm{Byte}] = 4096$, respected to Group 1. When counting the number of non-zero elements of each row in step (3), the hash table size is set larger than that of step (7) since the table for value data is not necessary. The thread block size is set as upper limit, 1024. The Group 0 handles the rows, which require larger hash tables than the size of shared memory, and the hash tables are allocated on global memory.

After the upper limit size of hash table is set, we set the one-step smaller hash table size. The hash table size and thread block size are halved. This enables to increase the number of concurrently executing thread blocks on each SM, and improves the GPU resource usage and occupancy. When the number of thread blocks executing in each SM is larger than fixed max thread blocks / multiprocessor (= 32), the algorithm finishes making row group with TB/ROW. Finally, the PWARP/ROW is assigned to the rows with few non-zero elements. Each borderline between TB/ROW and PWARP/ROW is set as 32 for (3) and 16 for (7).

IV. Performance Evaluation

We compare the performance of our proposal and major existing sparse libraries. Our evaluation is based on FLOPS performance of squaring matrix, which is twice the number of intermediate products divided by execution time.

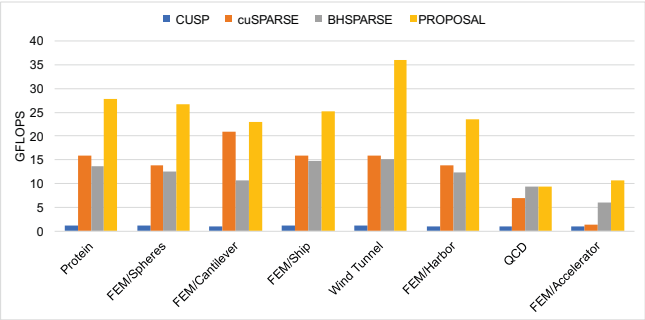
We evaluate the performance of SpGEMM on NVIDIA Tesla P100 PCI-e GPU, which is Pascal generation. The GPU has 16 GBytes of device memory and its bandwidth is up to 732 GB/sec. The machine has Intel Xeon CPU E5-2650 v3. GPU codes are implemented in CUDA 8.0 and are executed on CentOS Linux release 7.2.1511.

We use three sparse matrix libraries, cuSPARSE [15], CUSP [16] and BHSPARSE [17]. The cuSPARSE employs two-phases SpGEMM algorithm. First phase counts the number of non-zero elements and second phase computes the output matrix. The CUSP adopts ESC algorithm. The SpGEMM kernel of BHSPARSE is optimized for irregular matrices, which likely cause load imbalance.

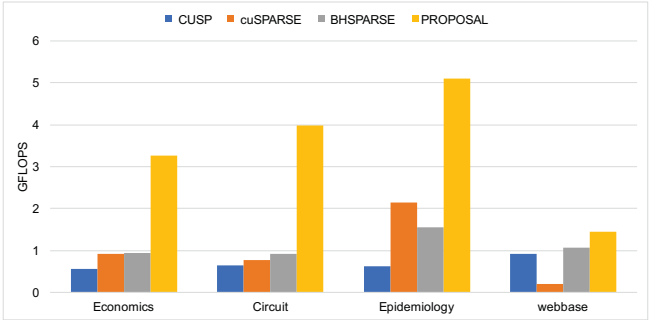
For our performance evaluations, we selected various kinds of 12 square matrices which are often used for the evaluation of sparse matrix computation on GPUs [5] from the Sparse Matrix Collection at University of Florida [9]. The name and size of each matrix data are summarized in Table II. The matrix size, the number of non-zero elements and the pattern of non-zero elements are largely different by matrix. Based on the average number of non-zero

Table II: Matrix Data

Name	Row	Non-zero	Nnz/row	Max nnz/row	Intermediate product of A^2	Nnz of A^2
Protein	36,417	4,344,765	119.3	204	555,322,659	19,594,581
FEM/Spheres	83,334	6,010,480	72.1	81	463,845,030	26,539,736
FEM/Cantilever	62,451	4,007,383	64.2	78	269,486,473	17,440,029
FEM/Ship	140,874	7,813,404	55.5	102	450,639,288	24,086,412
Wind Tunnel	217,918	11,634,424	53.4	180	626,054,402	32,772,236
FEM/Harbor	46,835	2,374,001	50.7	145	156,480,259	7,900,917
QCD	49,152	1,916,928	39.0	39	74,760,192	10,911,744
FEM/Accelerator	121,192	2,624,331	21.7	81	79,883,385	18,705,069
Economics	206,500	1,273,389	6.2	44	7,556,897	6,704,899
Circuit	170,998	958,936	5.6	353	8,676,313	5,222,525
Epidemiology	525,825	2,100,225	4.0	4	8,391,680	5,245,952
webbase	1,000,005	3,105,536	3.1	4700	69,524,195	51,111,996
cage15	5,154,859	99,199,551	19.2	47	2,078,631,615	929,023,247
wb-edu	9,845,725	57,156,537	5.8	3841	1,559,579,990	630,077,764
cit-Patents	3,774,768	16,518,948	4.4	770	82,152,992	68,848,721



(a) High-Throughput Matrices



(b) Low-Throughput Matrices

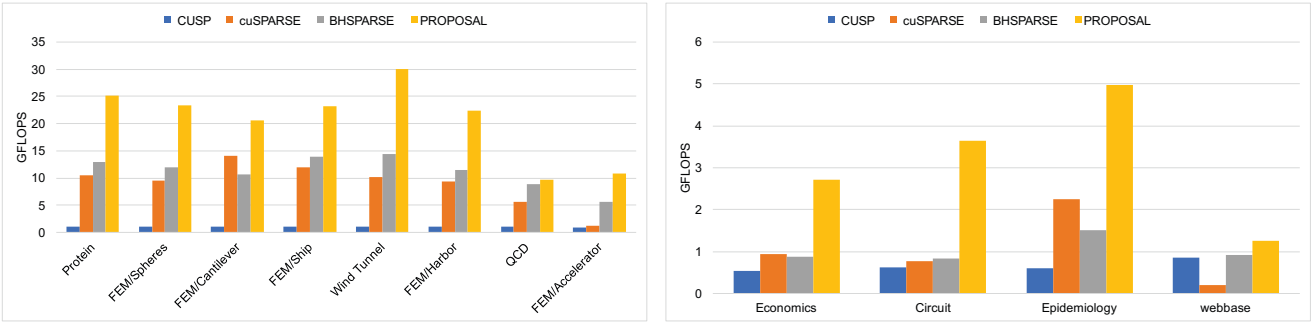
Figure 2: Performance of SpGEMM computation in single precision

elements per row (Nnz/row), top eight matrices are called as High-Throughput Matrices and the other four matrices as Low-Throughput Matrices in this paper. Since the number of intermediate products is much larger than the number of non-zero elements of output matrix, some existing SpGEMM algorithms, which increase the memory usages depending on the numbers of intermediate products, may require much memory usage. The dataset 'webbase' is distinctive matrix; maximum number of non-zero elements per row (Max nnz/row) is much larger than Nnz/row, that is, only some rows have many non-zero elements and most rows have very few non-zero elements. We additionally selected large size matrices, which are generated in graph analysis. These selected three matrices have highly irregularity and SpGEMM computation on GPU for these matrices often causes terrible load imbalance.

A. *Performance of SpGEMM*

Figure 2 shows the performance of SpGEMM computation in single precision. The CUSP achieves constant performance for all matrices. The performance is independent to the matrix size and the pattern of non-zero elements. The BHSPARSE shows superior performance compared to cuSPARSE for low Nnz/row matrices. On the other hand, cuSPARSE achieves higher performance for denser or regular matrices. Our proposal shows best performance in the evaluation for all evaluated matrices. The appropriate thread assignments and the construction of hash table with utilizing shared memory make large performance gains for both High- and Low-Throughput Matrices. Our proposal achieves significant speedups of x32.3, x8.1 and x4.3 on maximum compared to CUSP, cuSPARSE and BHSPARSE respectively, and x15.7, x3.2 and x2.3 on average respectively. Performance evaluation shows the speedup is up to x4.3 compared to best of existing sparse matrix libraries.

Figure 3 shows the performance of SpGEMM computation in double precision. Performance trend is similar to single precision, and our approach also shows good speedups compared to existing sparse matrix libraries. Our proposal achieves speedups of x28.7, x8.7 and x4.4 on maximum compared to CUSP, cuSPARSE and BHSPARSE respectively, and x15.1, x3.3 and x2.2 on average respectively. Performance evaluation shows the speedup is up to x4.4 compared to best of existing sparse



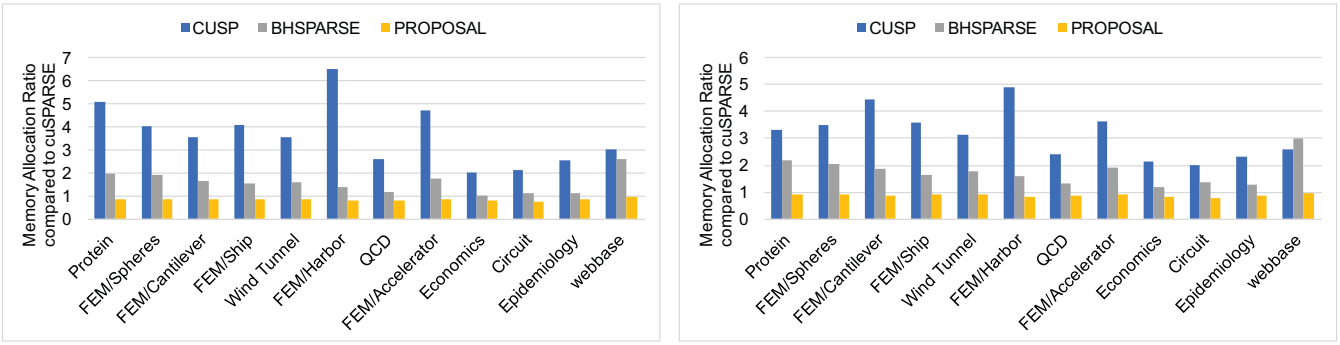
(a) High-Throughput Matrices

(b) Low-Throughput Matrices

Figure 3: Performance of SpGEMM computation in double precision

Table III: Performance of SpGEMM computation for Large size Graph Data [GFLOPS]

	Matrix	CUSP	cuSPARSE	BHSPARSE	PROPOSAL	Speedup
single	cage15	-	0.519	-	5.955	x11.5
	wb-edu	-	2.348	-	5.403	x2.3
	cit-Patents	0.837	0.028	0.880	3.351	x3.8
double	cage15	-	0.491	-	5.684	x11.6
	wb-edu	-	2.145	-	4.618	x2.2
	cit-Patents	0.780	0.028	0.813	2.980	x3.7



(a) Single precision

(b) Double precision

Figure 4: Maximum memory usage in SpGEMM computation

matrix libraries.

Table III shows the performance of SpGEMM computation for large size matrix data. The CUSP and BHSPARSE cannot execute SpGEMM of 'cage15' and 'wb-edu' because of large amounts of memory use for temporary results. On the other hand, the cuSPARSE executes SpGEMM of these matrices. However, the performance of cuSPARSE for such irregular matrices is really poor. Our proposal executes SpGEMM with low memory usage and achieves significant speedups up to x11.6.

B. Memory Usage

Our proposal aims to both accelerate SpGEMM computation and reduce memory usage on GPU. We evaluate the maximum memory usage during SpGEMM computation with existing SpGEMM libraries and our proposal. Figure 4 shows the ratio of maximum memory usage to cuSPARSE in single and double precision. Memory usage of our approach is much lower than other existing libraries for any matrix data in any precisions. The memory usages are reduced by 14.7% in single precision and 10.9% in double precision on average. This result indicates that our proposal accelerates SpGEMM computation of large matrix data on GPUs' limited device memory. Although BHSPARSE achieves superior performance to cuSPARSE for irregular matrices, BHSPARSE requires much larger memory. In contrast, our proposal shows higher performance compared to

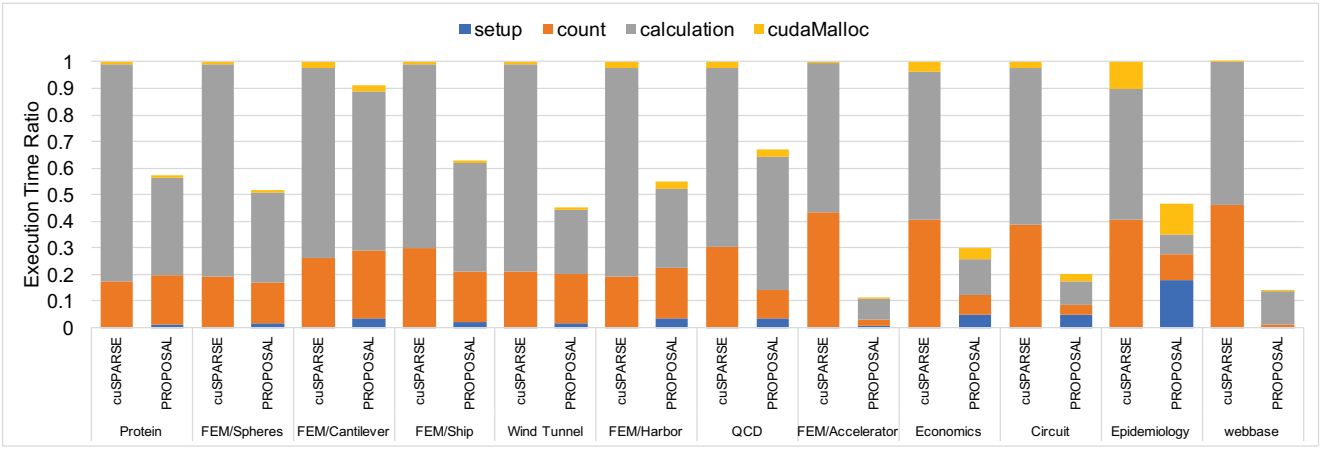


Figure 5: Performance Breakdown comparing with cuSPARSE in single precision

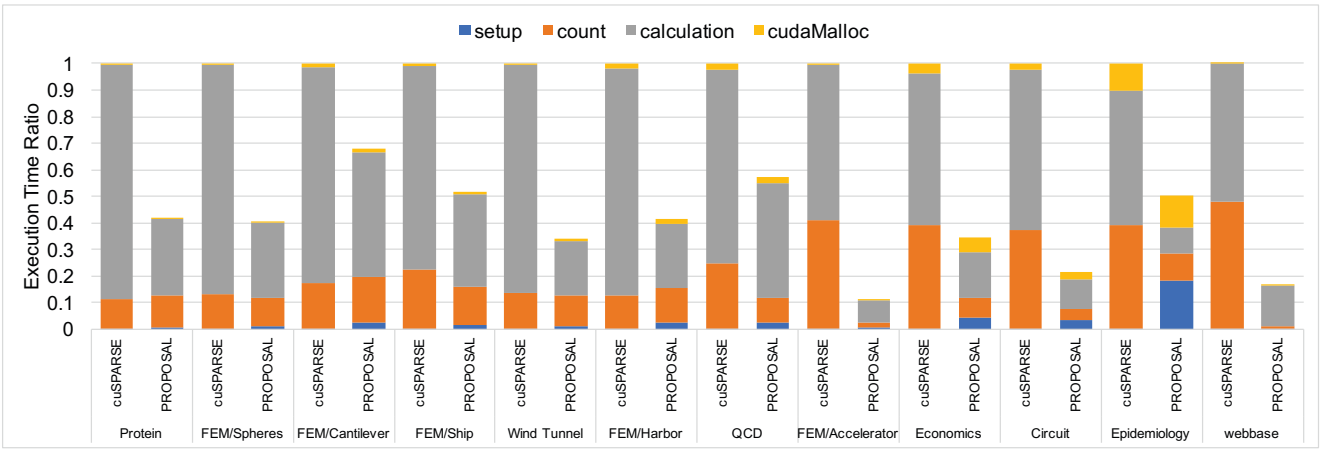


Figure 6: Performance Breakdown comparing with cuSPARSE in double precision

BHSPARSE for irregular matrices and also reduces memory usage by 67.7% on maximum.

C. Performance Analysis

We show the detailed performance analysis of our SpGEMM algorithm on P100.

First, we describe the effectiveness of CUDA stream. Our proposal launches multiple CUDA kernels with CUDA streams for each group to execute concurrently. Since some groups handle few rows less than 10, the CUDA stream enables to utilize GPU resource. For the matrix 'Cuircuit', one of the groups handles only 8 rows for counting and 9 rows for calculation. We confirmed that our proposal with CUDA stream achieves x1.3 speedups compared to the proposal without CUDA stream.

Next, we show the effectiveness of PWARP/ROW. The PWARP/ROW efficiently processes the rows with few non-zero elements. Especially, a matrix with low Nnz/row has many of rows with few non-zero elements. For the matrix 'Epidemiology', whose Nnz/row is 4.0, the PWARP/ROW significantly improves the performance, and we confirmed that the speedup is x3.1 compared to the proposal without PWARP/ROW.

Finally, we show the execution time breakdown of cuSPARSE and our proposal. Figure 5 and 6 shows the ratio of execution time of each phase, where the total execution time of cuSPARSE is set as 1. The result is divided into 4 phases; setup phase for grouping in our proposal, count phase, calculation phase and cudaMalloc time of output matrix. For High-Throughput Matrix, our proposal largely reduces the 'calculation' time from cuSPARSE though the cost of 'count' is roughly same in both cuSPARSE and our proposal. The 'setup' phase, which is an overhead in our proposal, is negligible for almost matrices. The cost of cudaMalloc on Pascal GPU becomes larger compared to previous generation GPUs, and the cudaMalloc phase becomes considerable for

sparser matrices. The GPU memory is also allocated even in setup phase. For the matrix 'Epidemiology', which is highly sparse and regular, the execution time of 'count' and 'calculation' are largely reduced from cuSPARSE. However, the large cost of cudaMalloc in setup phase and of cudaMalloc for output matrix prevents the performance improvement.

V. Related work

There has been several past work of SpGEMM computation on GPUs.

Demouth proposed two-phases SpGEMM algorithm implemented in cuSPARSE library [18]. The SpGEMM kernel of cuSPARSE allocates hash table on shared memory and global memory. If the insertion to the hash table on shared memory does not success, the algorithm tries for global memory. This algorithm causes many random global memory access and do not efficiently utilize fast shared memory. On the other hand, our proposal appropriately divides the rows into groups, and the execution of most of rows can be finished on shared memory.

Gremse et al. proposed Iterative Row Merge method, which efficiently combines intermediate products among warp [19]. Part of threads in warp is assigned to each row of input matrix A, and each thread generates the table of non-zero elements in the row of input matrix B corresponding to the non-zero element of A. By merging tables among threads, the values and column indices of non-zero elements in output matrix are computed. The merge operation itself is fast since it's executed among warp. However, the operation can be applicable for the rows which have few non-zero elements. The Iterative Row Merge method often requires decomposition of given matrix in order to execute, and this decomposition causes additional cost of memory usage and memory access.

Liu et al. proposed the SpGEMM algorithm for irregular matrix data like graph data on GPUs [17], and this algorithm is implemented in BHSPARSE library. The execution time of SpGEMM in each row largely varies, and depends on the number of intermediate products or non-zero elements. In order to improve the load balance of combining intermediate products, this algorithm assigns the rows to multiple bins based on the number of intermediate products, and selects appropriate algorithms for each bin. They use heap method, bionic ESC method and mergepath method for combining intermediate products. They evaluate the performance of GPUs and CPU, and show the performance advantage of GPUs. The aims of grouping rows of our proposal are the improvement of parallelism and exploiting GPU resource by appropriate thread assignments and the coordination of hash table size.

Anh et al. proposed Balanced Hash, which improves both the load balance and the process of combining intermediate products [20]. They improve the load balance by generating the worklist, which is the list of pairs of indices of non-zero elements required in the computation of intermediate products. Multiple rows are bundled based on the number of intermediate products, and are assigned one thread block to. Balanced Hash utilizes hash table to combine intermediate products. The hash table is on shared memory and the size is fixed. When the hash collisions occur, the intermediate products are stored into queue to execute later. After the calculation finishes once, the non-zero elements in hash table are stored to global memory. Then, the table is refreshed and the calculation is done again if there are still any intermediate products in the queue. The process continues until the queue becomes empty. The Balanced Hash algorithm accelerates SpGEMM computation since most of process can be executed on shared memory. On the other hand, the worklist requires much memory usage since it is stored in memory once. The queue for hash collision also requires additional memory usage and memory access. When the number of non-zero elements is small, the Balanced Hash algorithm uses more shared memory than necessary and cannot exploit the GPU resource. By constructing hash table with different size for each group, our algorithm assigns more thread blocks to each SM and fully utilizes GPU resource. Furthermore, since our algorithm does not require memory access when the hash collision occurs, the execution of most of rows can finish on shared memory and our algorithm achieves high SpGEMM performance.

VI. Conclusion

Accelerating SpGEMM computation, which is a bottleneck of the performance in the preconditioner of iterative method such as AMG method and graph algorithms, is one of the critical issues for scientific and engineering applications. Although SpGEMM is accelerated on many-core processors such as GPUs, which are widely installed on supercomputers, the memory access and utilization of shared memory are not well optimized. Furthermore, since existing work requires much memory usage to accelerate SpGEMM, the applicable matrix is limited on GPUs with small device memory. We propose the state of the art SpGEMM algorithm in order to both accelerate SpGEMM and reduce memory usage by appropriate grouping and efficient utilization of shared memory. Our proposal achieves superior performance to competing sparse matrix libraries on NVIDIA Pascal generation GPU, demonstrating up to x4.3 speedup in single precision and x4.4 speedup in double precision over CUSP, cuSPARSE and BHSPARSE libraries. Our proposal successfully reduces memory usage from existing sparse matrix libraries. Maximum memory usage is reduced by 14.7% in single precision and 10.9% in double precision on average.

For future work, we plan to evaluate our SpGEMM algorithm on other many-core processors such as AMD Radeon GPU and Intel Xeon Phi. Our algorithm should work well on AMD Radeon GPU since the architecture is similar to NVIDIA GPUs. Although Intel Xeon Phi does not support vector atomic operations required in our algorithm, the conflict detection instruction of AVX512 makes it possible to perform similar operations efficiently. We also plan to evaluate our SpGEMM algorithm for solvers and real world applications.

ACKNOWLEDGMENT

This work was partially supported by JST CREST Grant Number JPMJCR1303 and JPMJCR1687, and NVIDIA GPU Center of Excellence.

APPENDIX

Our implementation of the SpGEMM algorithm on GPU is available at https://github.com/EBD-CREST/nsparse.

REFERENCES

[1] 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.

[2] S. Van Dongen, "Graph Clustering Via a Discrete Uncoupling Process," *SIAM Journal on Matrix Analysis and Applications*, vol. 30, no. 1, pp. 121–141, 2008.

[3] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design, implementation, and applications," *International Journal of High Performance Computing Applications*, 2011.

[4] P. D. Sulatycke and K. Ghose, "Caching-efficient multithreaded fast multiplication of sparse matrices," in *12th International Parallel Processing Symposium*. IEEE Computer Society, 1998, pp. 117–123.

[5] N. Bell and M. Garland, "Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, no. 18. ACM, 2009.

[6] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarath, and P. Sadayappan, "Fast Sparse Matrix-Vector Multiplication on GPUs for Graph Applications," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2014, pp. 781–792.

[7] J. Greathouse and M. Daga, "Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2014, pp. 769–780.

[8] K. Matam, S. R. K. B. Indarapu, and K. Kothapalli, "Sparse Matrix-Matrix Multiplication on Modern Architectures," in *2012 19th International Conference on High Performance Computing (HiPC)*. IEEE, 2012, pp. 1–10.

[9] T. Davis, "The University of Florida Sparse Matrix Collection." [Online]. Available: http://www.cise.ufl.edu/research/sparse/matrices

[10] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick, "When cache blocking of sparse matrix vector multiply works and why," *Applicable Algebra in Engineering, Communication and Computing*, vol. 18, no. 3, pp. 297–311, 2007.

[11] V. Karakasis, G. Goumas, and N. Koziris, "Perfomance Models for Blocked Sparse Matrix-Vector Multiplication kernels," in *ICPP'09: International Conference on Parallel Processing*. IEEE, 2009, pp. 356–364.

[12] S. Yan, C. Li, Y. Zhang, and H. Zhou, "yaSpMV: Yet Another SpMV Framework on GPUs," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. ACM, 2014, pp. 107–118.

[13] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C401–C423, 2014.

[14] S. Dalton, L. Olson, and N. Bell, "Optimizing sparse matrix—matrix multiplication for the gpu," *ACM Transactions on Mathematical Software (TOMS)*, vol. 41, no. 4, p. 25, 2015.

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

[16] S. Dalton, N. Bell, L. Olson, and M. Garland, "Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations Ver.0.5.1," 2014. [Online]. Available: http://cusplibrary.github.io/

[17] W. Liu and B. Vinter, "An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 370–381.

[18] J. Demouth, "Sparse Matrix-Matrix Multiplication on the GPU," in *GPU Technology Conference*, 2012.

[19] F. Gremse, A. Hofter, 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.

[20] P. N. Q. Anh, R. Fan, and Y. Wen, "Balanced Hashing and Efficient GPU Sparse General Matrix-Matrix Multiplication," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16, vol. 1. New York, NY, USA: ACM, 2016, pp. 36:1–36:12.