On Improving Performance of Sparse Matrix-Matrix Multiplication on GPUs

Rakshith Kunchum, Ankur Chaudhry, Aravind Sukumaran-Rajam,
Qingpeng Niu, Israt Nisa, P. Sadayappan
The Ohio State University
Columbus, OH, USA

kunchum.1,chaudhry.54,sukumaranrajam.1,niu.1,nisa.1,sadayappan.1@osu.edu

ABSTRACT

Sparse matrix-matrix multiplication (SpGEMM) is an important primitive for many data analytics algorithms, such as Markov clustering. Unlike the dense case, where performance of matrix-matrix multiplication is considerably higher than matrix-vector multiplication, the opposite is true for the sparse case on GPUs. A significant challenge is that the sparsity structure of the output sparse matrix is not known a priori, and many additive contributions must be combined to generate its non-zero elements. We use synthetic matrices to characterize the effectiveness of alternate approaches and devise a hybrid approach that is demonstrated to be consistently superior to other available GPU SpGEMM implementations.

KEYWORDS

SpGEMM, Load Balancing, Sparse matrix;

ACM Reference format:

Rakshith Kunchum, Ankur Chaudhry, Aravind Sukumaran-Rajam, Qingpeng Niu, Israt Nisa, P. Sadayappan. 2017. On Improving Performance of Sparse Matrix-Matrix Multiplication on GPUs. In *Proceedings of ICS '17, Chicago, IL, USA, June 14-16, 2017*, 11 pages. DOI: <http://dx.doi.org/10.1145/3079079.3079106>

1 INTRODUCTION

Sparse matrix-matrix multiplication (SpGEMM) is a fundamental operation required in several domains. It is a primitive operation in many data-analytic and graph-analytic algorithms [6, 13, 19, 23] as well as in algebraic multi-grid methods [3]. There have been a number of recent efforts to develop GPU implementations for SpGEMM [2, 8, 9, 14, 16, 18, 20]. But achieving high performance for SpGEMM is very challenging. Unlike the dense case, where achieved performance with matrix-matrix multiplication (in GFLOPs) is considerably higher than sparse matrix-vector multiplication, performance of SpGEMM is generally much lower than SpMV for all known GPU implementations.

Table 1 shows the achieved SpGEMM performance with Nvidia's cuSPARSE library on a set of sparse matrices from the Florida sparse matrix collection [10, 22]. This set of sparse matrices has been used in recent work to evaluate sparse matrix multiplication

[2, 14, 16, 18], and we use the same set of sparse matrices in this paper. As has been done in prior work, in this paper too we report performance of SpGEMM implementations using the same matrix for both input operands, i.e., perform $C = A * A$. This enables better insights into the impact of matrix structure on SpGEMM performance, which would be more difficult if two sparse matrices with different sparsity structure were used.

It can be seen that the performance achieved in computing the product of each matrix with itself ($A * A$) does not exceed even 5 GFLOPs (single-precision), and is less than 0.5 GFLOPs for more than half the matrices in the test set. In contrast, on the same GPU, the Nvidia cuBLAS routine for dense MM achieves over 2500 GFLOPs for a $4k \times 4k$ matrix.

There are a number of reasons why achieving high GPU performance with SpGEMM is very challenging:

- **Low Arithmetic Intensity:** If we define the arithmetic intensity of a computation as the ratio of the number of arithmetic operations to the number of distinct data elements accessed, dense MM for computing $C = A * A$ has an arithmetic intensity of $0.5 \times N$ FMAs (Fused Multiply-Adds), since N^3 FMAs are executed and $N^2 + N^2$ distinct data elements are accessed for the input and output matrices. The compression ratio (CR) for an SpGEMM computation is defined as the ratio of the number of FMA operations to the number of non-zeros in the result matrix. Thus, CR serves as an upper bound on the arithmetic intensity for SpGEMM. It can be observed from Table 1 that CR is no more than 30 for any of the test matrices, and is less than 5 for more than half of the matrices.

The arithmetic intensity can be used to upper bound the operational intensity [24], defined as the ratio of the number of executed computational (here floating-point) operations over the number of bytes of data moved between main memory and the GPU cores. Unless the operational intensity is sufficiently high, a computation will necessarily be memory-bandwidth limited. The very high arithmetic intensity of dense MM makes it not memory-bandwidth limited on GPUs, enabling it to achieve a significant fraction of peak performance. In contrast, the low CR for SpGEMM means that it will be impossible to achieve anywhere close to peak performance on GPUs.

- **Index Matching** The achieved performance for SpGEMM in Table 1 is in all cases much lower than the roofline-based [24] bound using the CR and achievable bandwidth between global-memory and GPU cores. This is because of the index-matching problem for SpGEMM, explained in greater detail in the next section. The problem is that the structure of sparsity of the result matrix (i.e., which elements in each row are non-zero) is unknown, and further it is non-trivial to locate where a given contribution from

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICS '17, Chicago, IL, USA

© 2017 ACM. 978-1-4503-5020-4/17/06...\$15.00

DOI: <http://dx.doi.org/10.1145/3079079.3079106>

the product of a_{ik} with b_{kj} to result non-zero c_{ij} should be accumulated. As explained in the next section, several approaches have been proposed to address this index-matching problem, and each of these approaches entails additional data access and computational operations.

- **Load Balancing** Unlike dense MM, which features the same number of elements in every row (column) and the same number of arithmetic operations for each result matrix element, sparse matrices that arise in practice exhibit significant non-uniformity. Our test set of sparse matrices includes structural matrices, web connection graphs, protein models, macroeconomic models and circuit simulation graphs. The chosen matrices are diverse and inherently irregular in their sparsity structure. Table 1 shows the mean (μ) and standard deviation (σ) of the number of non-zeros per row. It can be seen that most matrices have significant non-uniformity across different rows. Table 1 shows a distribution of work (number of operations) per row, grouped into four bins: i) 1-256 operations, ii) 257-1024 ops., iii) 1025-4K ops., and iv) more than 4K ops. It can be seen that different matrices can have very different distribution of work.

In this paper, we develop a hybrid SpGEMM algorithm for GPUs, based on the following rationale:

- Many sparse matrices exhibit significant nonuniformity in the number of nonzero elements in different rows/columns, with some rows having very few nonzeros, while others have thousands or more nonzeros. Hence, it is very challenging to achieve both inter-thread-block load balance as well as intra-warp load balance and avoidance of thread divergence with any approach that uses a uniform row-wise division of work among thread/warps/thread-blocks.
- Achieving load balance is much more feasible with an element-wise division of work among threads, but all known SpGEMM approaches (elaborated later) that take this approach suffer from high data access costs.
- We therefore develop a hybrid SpGEMM algorithm that combines two approaches with complementary characteristics: i) For rows which have few operations and thus low concurrency potential, we use an element-wise parallel strategy, and ii) for rows with a large number of operations, we resort to a row-wise work partitioning strategy that prioritizes minimization of data movement.

In this paper, we make the following contributions:

- We characterize different approaches to SpGEMM using synthetic matrices that remove intra-row non-uniformity.
- We develop an efficient GPU implementation of SpGEMM based on use of a scatter-vector.
- Based on the performance characterization using synthetic matrices, we devise a hybrid SpGEMM implementation that uses different approaches for light versus heavy rows, based on the number of operations needed for them.
- We present an experimental evaluation using 23 sparse matrices from different domains that demonstrates higher performance with the new GPU-SpGEMM implementation than other SpGEMM implementations for GPUs.

Algorithm 1: SPGEMM loop structure

```

1 for each  $a_{i*}$  in matrix  $A$  do
2   set  $c_{i*} = 0$ 
3   for each non-zero entry  $a_{ij}$  in  $a_{i*}$  do
4     for each non-zero entry  $b_{jk}$  in  $b_{j*}$  do
5       value =  $a_{ij} * b_{jk}$ 
6       if  $c_{ik} \notin c_{i*}$  then
7         insert  $c_{ik}$  to  $c_{i*}$ 
8          $c_{ik} = \text{value}$ 
9       else
10         $c_{ik} = c_{ik} + \text{value}$ 

```

2 BACKGROUND AND RELATED WORK

SpGEMM: General Sparse Matrix Matrix Multiplication (SpGEMM) can be represented by operations on row vectors of the matrices, represented using the standard CSR (Compressed Sparse Row) format. When multiplying sparse matrices, the row vector operations result in data computations with a large disparity between the number of floating point operations (flops) across different rows. This is mainly due to the fact that rows are sparsely populated and the variance of the number of non-zeros per row is very high (as shown in Table 1). A number of previous efforts have developed parallel SpGEMM implementations [2, 5, 8, 9, 12, 14, 16, 18, 20, 21, 25]. In this section, we present background information about SpGEMM, discuss key challenges to implementation of parallel SpGEMM, and describe some of the main approaches that have been previously pursued for implementation of SpGEMM.

Compression Ratio (CR): Each row of the left input matrix A in the product AB contributes only to the corresponding row of the result matrix C . On processing a row in matrix A , many intermediate products are formed that contribute to elements in the corresponding row of the output matrix. Since several intermediate products could be added to the same non-zero element in the result matrix C , the final number of nonzeros formed in C is often much less than the number of intermediate products formed. The ratio of the number of intermediate products to the number of non zeros is called as compression ratio. Table 1 shows the compression ratio for different sparse matrices. The CR represents an upper bound on the operational intensity in terms of FMAs per data element moved to global memory. It has been shown to be positively correlated to achieved performance for different SpGEMM algorithms on GPUs [14].

2.1 Challenges to Parallel Formulation of SpGEMM

As pointed out by Buluc and Gilbert [4], dividing the work among different threads and memory allocation for the result matrix are two major challenges for parallel formulations of SpGEMM.

The number of non-zeros in the result matrix C is not known. One of the major concerns in SpGEMM is that the number of non-zero elements in the result matrix is not known in advance, complicating memory management for the result matrix.

Load-balanced work division among threads is non-trivial for SpGEMM. This is mainly because of the fact that the number of

Table 1: Non-uniformity of SpGEMM work distribution across rows

Matrix	Rows	Avg nnz(A)	Sigma nnz(A)	CR	cuSPARSE GFLOPs	Range of per-row operation count			
						1-256	257-1024	1025-4096	>4097
2cubes_sphere	101K	17	2.65	3	0.23	28	72	0	0
accelerator	121K	22	13.79	4	0.4	*	68	14	0
cage12	130K	16	4.71	2	0.24	55	45	0	0
cantilever	62K	65	14.06	15	4.05	*	*	40	60
economics	206K	6	4.44	1	0.23	100	0	0	0
epidemiology	525K	4	0.08	2	0.74	100	0	0	0
filter3D	106K	26	12.68	4	0.31	1	85	14	0
harbor	46K	51	27.78	20	2.62	1	19	47	32
hood	220K	50	12.82	16	1.65	0	0	100	0
m133-b3	200K	4	0.00	1	0.09	100	0	0	0
majorbasis	160K	11	0.42	2	0.67	100	0	0	0
mario002	389K	6	1.96	2	0.2	100	0	0	0
mono.500Hz	169K	30	17.90	5	0.32	*	65	34	*
offshore	259K	17	2.81	3	0.16	29	71	0	0
patents_main	240K	2	3.00	1	0.03	72	*	0	0
poisson3Da	13K	26	13.76	4	0.18	1	82	17	0
protein	36K	120	31.86	28	3.08	0	*	2	98
pwtk	217K	54	4.74	19	2.35	*	*	99	*
QCD	49K	39	0.00	7	1.48	0	0	100	0
scircuit	170K	6	4.39	2	0.31	98	2	*	0
ship	140K	56	11.07	19	2.78	0	*	83	17
spheres	83K	73	19.08	18	2.28	4	*	13	83
webbase	1M	3	25.35	1	0.12	96	1	2	*

All the values under the ops-range column are in %. * implies less than 1%

Avg-nnz(A): average number of non-zeros per row in Matrix A. CR: Compression Ratio.

Sigma-nnz(A): standard-deviation of non-zeros per row in Matrix A.

cuSPARSE column denotes GFlops achieved for SpGEMM using the Nvidia cuSPARSE library.

non-zeros in different rows of a sparse matrix can be very different, and the work per row is non uniform. One approach to divide the work among the threads in a uniform fashion is to derive work-lists and divide the work from work-lists among the threads in a cyclic fashion. But this approach inherently has a drawback that it will result in a large volume of data movement to/from GPU global memory, thereby limiting performance.

In the next rest of this section, we describe how the above two concerns have been addressed by previous SpGEMM implementations. In the later sections of the paper, we discuss how we address these issues and develop a more efficient implementation of SpGEMM.

A number of considerations are important in developing efficient GPU applications. While dividing work among different threads in a GPU, the approach to work distribution among threads affects a number of important metrics: warp occupancy, data movement volume and efficiency, and thread divergence. Consider the case of performing SpGEMM using work-lists. This approach has very high concurrency. But the drawback of this approach is that it involves high data movement. If we try to reduce the high data movement using shared memory, the potential problem faced is that of low occupancy. Thus, a high-performance SpGEMM implementation must address all these issues in order to achieve high performance.

As stated earlier, the size of the result matrix is not known a priori. In the next section, we briefly describe how this issue is addressed by different approaches.

2.2 Alternative Approaches to SpGEMM

2-Phase Approach In this approach, the number of elements in the result matrix is computed in the first phase and accordingly the necessary memory locations are allocated. During the second phase, the actual values of the result matrix are computed. There are a number of ways to determine the structure of the result matrix C and allocate space for it - precise, probability, upper bound and progressive [16]. The precise method pre computes the exact number of non-zero elements in the result matrix. A row-merging approach [14], as well as SpGEMM methods in cuSPARSE and MKL use the precise method in their implementations. Probabilistic methods use random sampling and probability analysis to estimate the number of non-zero elements in the result matrix[1, 7]. The upper bound method calculates an upper bound on the number of non-zero elements. The progressive method starts sparse matrix computation with an arbitrary initial amount of memory and re-allocates memory if more space is required. Gilbert et al. use the progressive method for their parallel implementation[12].

Expand-Sort-Compress(ESC) This algorithm was developed as part of Nvidia's CUSP library function [8] [9] for computing the product of two sparse matrices. It creates an intermediate set of key-value pairs $\hat{C}$ which contains all the elemental contributions generated due to arithmetic operations. After the computation step, all the elements in matrix $\hat{C}$ are sorted according to their row and column numbers. In the final step, the values belonging to the same row and column are accumulated to produce the result matrix C.

Hashing Hashing is a technique that has been used for SpGEMM [2, 11, 20]. In this method, during the computation step a hash table is created for elements of the result matrix, with row/column indices being the key and the computed result being the value of a computed sparse matrix element. A potential race condition is that multiple threads could attempt modification of the same element simultaneously. In order to avoid this, the update operation on the hash table must be atomic. The cuSPARSE SpGEMM implementation [20] uses a global memory based hash implementation. Balanced Hashing [2] is another efficient implementation of this approach, where shared-memory hashing is used as much as possible.

2.3 Scatter-Vector

Algorithm 2: Sequential Scatter Vector Implementation

```

1  sv[:] = -1
2  nnz_so_far = 0
3  C.row[0] = 0
4  for i = 0 to m - 1 do
5      for a_index = A.row[i] to A.row[i + 1] - 1 do
6          a = A.val[a_index]
7          j = A.col[a_index]
8          for b_index = B.row[j] to B.row[j + 1] - 1 do
9              k = B.col[b_index]
10             if sv[k] == -1 then
11                 C.col[nnz_so_far] = k
12                 C.val[nnz_so_far] = a * B.val[b_index]
13                 sv[k] = nnz_so_far
14                 nnz_so_far = nnz_so_far + 1
15             else
16                 p = sv[k]
17                 C.col[p] = k
18                 C.val[p] = a * B.val[b_index]
19             C.row[i + 1] = nnz_so_far
20             sv[:] = -1

```

The scatter-vector approach for SpGEMM was devised for CPUs [15]. A full vector of size n is allocated, where n is the number of columns in B. As the rows of 'A' are processed in serial fashion, this 'scatter vector' stores pointers to compacted elements of the corresponding resultant row in 'C'. When any new intermediate product is formed in the result row of 'C', the pointer to the value is stored in the scatter-vector at the corresponding column id. After each additive contribution is generated, the scatter vector is accessed at the pertinent column index first to determine if the corresponding element already has a valid location in the result row of C. If so, the new intermediate product is added at the same position. Otherwise, the intermediate product is written at a new location and the corresponding column id in the scatter-vector is updated with a pointer to this new location.

In this paper, we characterize performance of some alternative SpGEMM approaches for GPUs and develop a hybrid implementation that combines multiple strategies.

3 SCATTER VECTOR IMPLEMENTATION ON GPU

One of the earliest developed approaches for efficient SpGEMM [15] used a scatter-vector, but its use for GPU implementation of SpGEMM has not been reported. In this section, we describe a scatter-vector based SpGEMM on GPUs. Threads in a thread block (or virtual warp, as discussed later) collectively process a row in matrix A/C, and all the rows in A are distributed among the thread blocks in a cyclic fashion. An array of size $n * (blocks_num) * (warps_num_per_block)$ is created in the global memory, where n is the number of columns in B. Thus each warp gets its own private scatter vector. Threads in a warp sequentially process the elements of a row in A. The elements of the corresponding row in B are mapped in a cyclic manner among threads, to compute updates in parallel to different elements in the result row C.

Algorithm 3: Scatter Vector Implementation on GPU

```

1  sv[:, :] = -1
2  shared int count = 0
3  for q = block_id to bin_size stride grid_dim do
4      i = rowIds[q]
5      for a_index = A.row[i] to A.row[i + 1] - 1 do
6          a = A.val[a_index]
7          j = A.col[a_index]
8          for b_index = B.row[j] + thread_id to
              B.row[j + 1] - 1 stride block_dim do
9              k = B.col[b_index]
10             if sv[block_id][k] == -1 then
11                 p = atomicAdd(&count, 1)
12                 update_temp_c(p, a * B.val[b_index])
13                 sv[block_id][k] = p
14             else
15                 p = sv[block_id][k]
16                 update_temp_c(p, a * B.val[b_index])
17             C.row[i + 1] = count
18             sv[:, :] = -1
19             count = 0

```

In the implementation, memory allocation is based on the upper bound method. The number of multiplication operations for each row in the matrix is pre-computed by adding the number of non-zeros in the relevant rows of B. This provides an upper-bound on the maximum possible number of non-zeros in each row of C. The time taken for this pre-computation is negligible.

Initially none of the C elements is assigned a memory location. The first thread that tries to update its contribution to a given C element is responsible for assigning a unique memory location and updating the scatter-vector to point to the new location. Multiple threads may simultaneously try to obtain a unique memory location for different C elements. In order to prevent the competing threads from assigning the same memory location for different elements in C, a locking mechanism is required. Since each row in C has a memory pool, only the threads working on the same row of C will compete for the same lock. Locking is only required when creating

**Table 2: Performance comparison (GFlops)
ESC-Shared vs. ESC-Global vs. Scatter-Vector**

Ops per row	ESC-SM	ESC-GM	SV
9	1.05	0.20	0.09
16	1.74	0.31	0.14
25	1.83	0.37	0.21
64	2.50	0.65	0.46
100	2.63	0.74	0.55
256	2.03	0.75	1.09
400	1.02	0.69	1.81
484	1.33	0.67	1.96

a new element in C . Multiple threads can atomically update the same element if it already has a valid memory location assigned to it.

The above locking mechanism is implemented using an atomic add operation on a shared variable *count*. (Algorithm 3). The number of times this atomic operation is performed is equal to the number of non zeros created in the given row of C .

The update of intermediate products also requires atomic operations if concurrent updates could occur for an element. We avoid this by sequentially processing the elements of a given row in A . This guarantees that no two threads can simultaneously attempt to update the same non zero element in a row of C . The *update_temp_C* function updates the products formed for a given row of C in the temporary space allocated for it. Later an additional step is required to copy all the products belonging to all the rows from their temporary space to the result array $C.col$ and $C.val$. Also, a simple scan operation on $C.row$ array is performed to obtain row pointers in C , which along with $C.col$ and $C.val$ form the final result C matrix in unordered CSR format. The total time for these additional steps is very low relative to the time taken to form the result elements.

4 SPGEMM CHARACTERIZATION USING SYNTHETIC MATRICES

The performance of different SpGEMM algorithms is dependent on input matrix characteristics. The number of non-zeros per row in these matrices (Table 1) can vary considerably, making it challenging to effectively distribute the work among the threads. Significant variance in the number of operations per row (Table 1) impacts warp occupancy, data-movement and thread divergence. In order to systematically characterize different alternatives, we measure the performance of each algorithm on synthetic sparse matrices having distinct features. Details on the generation of these synthetic matrices is discussed later in section 7.1. Results from experimentation with synthetic matrices is presented in Table 2.

Synthetic sparse matrices were created with random column positions but a fixed total number of non-zeros in every row of the matrix. The purpose of this experiment was to remove one source of variability in real matrices - the variance in the number of non-zeros per row - and observe the impact of the number of non-zeros per row on performance. Different matrices were generated which differed in the number of non-zeros per row. When computing $C=A*A$, each row of C requires exactly the same number of operations since each row of the source matrices has a fixed number of non zeros per row.

Figure 1: ESC-continuous vs. ESC-row-wise

For example, if each row of A has 4 non zeros, 16 FMAs would be used to form each row of C .

Table 2 shows the performance achieved by ESC in global memory, ESC in shared memory and the scatter vector approach. As the work per row increases, initially the performance of ESC-shared increases. However, as work increases, the shared memory requirement also increases. This limits occupancy and negatively affects performance. ESC using global memory does not have this constraint. However, ESC-global involves sorting and compression in global memory, which leads to higher global memory traffic and this in turn limits performance. The scatter-vector approach has poor performance when the work per row is small. In this case the memory access to the scatter-vector in global memory is random and the probability of L2 cache hits is low. However, the scatter vector approach neither faces the problem of low occupancy (like ESC-shared) nor is it IO intensive (like ESC-global). As the work per row increases, the scatter vector approach performs significantly better.

The experiments indicate that use of ESC with shared memory is a better choice for rows having low work (total number of ops), and use of the scatter-vector approach is better suited for rows associated with a high operation count. However, the number of non zeros in a row impacts the performance of the ESC algorithm. In the next section we develop an alternative variant for ESC in shared memory, with a choice to be made between the two variants based on the number of non-zeros in a given row.

5 ESC VARIANTS

For the experiments in the previous section, although the work per row was varied, the number of elements per row in 'A' was not varied. The number of elements present in a row of 'A' can have a significant impact on performance. Consider the following two cases: i) Rows in 'A' have a large number of non-zero elements but they point to very sparsely populated rows in 'B' and ii) Rows in 'A' have very few non-zero elements, but each points to rows in 'B' that are heavily populated. In both the cases the work per result row could be similar, but the amount of concurrency is quite different. In this section we show how the classical approach is better suited for the first scenario and a slightly modified approach called ESC-row-wise suits the second scenario.

Algorithm 4: ESC Continuous

```

1 shared  $s\_key[maxops]$ ,  $s\_scan[maxops + 1]$ ,  $s\_val[maxops]$ ,
   $s\_nnz\_cnt[maxops]$ 
2 for  $q = block\_id$  to  $bin\_size - 1$  stride  $grid\_dim$  do
3    $i = rowIds[q]$ 
4    $s\_nnz\_cnt[:] = no\_of\_nnz(B, i)$ 
5    $s\_scan[:] = prefix\_sum(s\_nnz\_cnt)$ 
6    $ops\_cnt = s\_scan[max\_index]$ 
7   for  $p = thread\_id$  to  $ops\_cnt$  stride  $block\_dim$  do
8      $a\_index = comp\_index(A, s\_nnz\_cnt, s\_scan)$ 
9      $b\_index = comp\_index(B, s\_nnz\_cnt, s\_scan)$ 
10     $s\_key[p] = B.col[b\_index]$ 
11     $s\_val[p] = A.val[a\_index] * B.val[b\_index]$ 
12   $key\_value\_sort(s\_key, s\_val)$ 
13   $s\_scan = indices\_to\_be\_compressed(s\_key)$ 
14   $compress(s\_key, s\_val, s\_scan)$ 
15   $syncthreads()$ 
16   $flush\_to\_temp\_c(s\_key, s\_val)$ 
17   $C.row[i] = count(s\_key)$ 

```

Algorithm 5: ESC Row-wise

```

1 shared  $s\_key[maxops]$ ,  $s\_scan[maxops + 1]$ ,  $s\_val[maxops]$ 
2 for  $q = block\_id$  to  $bin\_size - 1$  stride  $grid\_dim$  do
3    $i = rowIds[q]$ 
4    $ops = 0$ 
5   for  $a\_index = A.row[i]$  to  $A.row[i + 1]$  stride  $block\_dim$ 
6     do
7        $j = A.col[a\_index]$ 
8        $r = 0$ 
9       for  $b\_index = B.row[j] + thread\_id$  to
10         $B.row[j + 1] - 1$  stride  $block\_dim$  do
11         $s = ops + r * block\_dim + thread\_id$ 
12         $s\_key[s] = B.col[b\_index]$ 
13         $s\_val[s] = A.val[a\_index] * B.val[b\_index]$ 
14         $r++$ 
15       $ops += B.row[j + 1] - B.row[j]$ 
16   $key\_value\_sort(s\_key, s\_val)$ 
17   $s\_scan = indices\_to\_be\_compressed(s\_key)$ 
18   $compress(s\_key, s\_val, s\_scan)$ 
19   $syncthreads()$ 
20   $flush\_to\_temp\_c(s\_key, s\_val)$ 
21   $C.row[i] = count(s\_key)$ 

```

The classical parallel ESC algorithm [9] slices the rows in matrix A and for each row slice, generates the key-value pairs in the expand phase, sorts the generated pairs according to the keys and compresses all the values with the same key. This approach is quite efficient in terms of dividing the work among different threads, since no synchronization or locks on update locations is necessary during the expand phase. At the beginning of the expand phase each thread has to identify the element a_{ij} and b_{jk} to be multiplied.

The cost of identifying the multiplicands is well amortized if the number of non-zeros in the rows in 'A' are quite large. The classical ESC approach is called ESC-continuous.

If the number of elements in a given row of 'A' is small, and each of these elements point to sufficiently populated rows in 'B', we can avoid the overhead of identifying the multiplicands. This can be done by sequentially iterating over the elements in a row in 'A' and exploiting parallelism by cyclic mapping of threads across the elements in the row of matrix 'B' to compute the intermediate products. Thus, by modifying the expand phase of classical ESC approach, we can achieve better performance. Since the rows in 'A' are binned according to total operation count, a low number of non-zeros in a row of 'A' implies higher number of operations per element (since the total op-count remains the same). We call this approach ESC-row-wise.

We compared the two ESC variants on synthetic matrices. Figure 1 compares ESC-continuous with ESC-row-wise. We observe that for the rows that have very few non-zeros, ESC-row-wise fares better than ESC-continuous. But when the number of non-zeros in a row of 'A' is high, the ESC-continuous performs well.

From the characterization experiments in section 4, it is clear that ESC is preferable for rows with low work, while the scatter-vector approach is preferable for rows with a lot of work. In the next section, we seek to further improve the performance of the scatter-vector approach for such heavy rows.

6 SCATTER VECTOR WITH MULTIPLE LOCKS

Figure 2: 2-Lock based memory acquisition

Figure 3: Multi-lock based memory acquisition

As discussed in Section 2.3, the scatter-vector approach requires locking to acquire new locations. The drawback of this approach is that thread contention is high when they all compete for a single lock. One way to reduce thread contention is to introduce more locks. However, it should be guaranteed that different threads that update the same 'C' element should go through the same lock. Otherwise, different threads may assign different memory locations to the same 'C' element. A two-lock scheme can be used where threads trying to acquire a new location choose one of two

locks based on whether the column id is even or odd. This ensures that different threads trying write to the same column index will necessarily go through the same lock, guaranteeing that only one of them will create a new location (the one which fails to create the new location, will add its contribution to the location generated by the other). The staging area/memory pool is also divided into two. The first lock allocates new locations from the left of the staging area and the second one allocates from the right, ensuring maximum space utilization. Once the entire row is processed, the two contiguous regions are written back to global memory.

Figure 4: Memory Management in Multi-Lock Approach

We further extend our two lock approach to multiple locks. The main challenge with this approach is that although the total number of operations is known, the number of new locations required per thread is not known a priori. One way to overcome this problem is to use a staging area of $N * \text{max ops}$, which will ensure the space requirements. However, if the staging area is in shared memory, this will reduce occupancy and thus limit performance. In the multi-lock approach, we divide the entire staging area into N chunks and each lock owns/guards a chunk. An extra indirection array called `lock_selector` is used to indicate which lock should be used to acquire a new location. Each lock selector is initialized with values from 0 to $N - 1$. For example, let us consider the use of eight locks. In this case we divide the entire scratch space into 8 chunks. `lock_selector[0]` will be initialized with the value 0, `lock_selector[1]` will be initialized with value 1 and so on. This is shown in the Figure 3. When a thread attempts to acquire a new location, it looks at the `lock_selector` array based on the modulo N of its column id to get the lock id. Then it performs a lock operation on the corresponding lock. In our example (where $N = 8$), if a thread attempts to acquire a new location for column 42 it would refer to `lock_selector[2]` ($42 \% 8 == 2$), which initially will contain the value 2. Hence the thread will try to lock on lock 2 and if successful will get a new location in the chunk owned by lock 2. If a thread acquiring lock observes that it obtained the last location in the given chunk, it updates the corresponding `lock_selector` value with the id of the next sequential chunk which has a free location. This is shown in the 4. In our example, assume that a thread locking for column 42 observes that it received the last location owned by lock 2. It will then update `lock_selector[2]` with value 3. Now assume that another thread requires a new location for column 50, it will check the `lock_selector[2]` which will contain 3, hence the thread will try

Table 3: Multi-Lock Performance Comparison(GFlops)

size of row in A	GFlops			
	no locks	2	4	8
22	5.6	7.3	8.0	8.4
10	1.8	2.3	3.5	3.2
5	2.0	3.2	4.5	5.4
2	2.2	4.0	5.8	6.2

to get a new location by locking on lock 3. Any other thread trying to obtain a new location for column 50 will try to get a location on lock 3 and will fail (the failed threads will update the existing value by adding its contribution). Once the entire row is processed, each occupied entry in the staging area is written back to the global memory. By using this strategy, we reduce thread contention with out any extra space overhead.

Algorithm 6: 2-lock SV with staging in shared.

```

1   $sv[:, :] = -1$ 
2  shared  $s\_key[\text{maxops}], s\_val[\text{maxops}]$ 
3  shared  $count[2] = 0$ 
4  for  $q = \text{block\_id}$  to  $\text{bin\_size} - 1$  stride  $\text{grid\_dim}$  do
5       $i = \text{rowIds}[q]$ 
6      for  $a\_index = A.\text{row}[i]$  to  $A.\text{row}[i + 1] - 1$  do
7           $a = A.\text{val}[a\_index]$ 
8           $j = A.\text{col}[a\_index]$ 
9          for  $b\_index = B.\text{row}[j] + \text{thread\_id}$  to
             $B.\text{row}[j + 1] - 1$  stride  $\text{block\_dim}$  do
10              $k = B.\text{col}[b\_index]$ 
11             if  $sv[\text{block\_id}][k] == -1$  then
12                  $p = \text{get\_index}(j, \text{count})$ 
13                  $s\_key[p] = k$ 
14                  $s\_val[p] = a * B.\text{val}[b\_index]$ 
15                  $sv[\text{block\_id}][k] = p$ 
16             else
17                  $p = sv[\text{block\_id}][k]$ 
18                  $s\_val[p] += a * B.\text{val}[b\_index]$ 
19              $\text{syncthreads}()$ 
20          $\text{flush\_to\_temp\_c}(s\_key, s\_val)$ 
21          $C.\text{row}[i + 1] = \text{sum}(count[:])$ 
22          $sv[:, :] = -1$ 
23          $count[:] = 0$ 

```

6.1 Scatter vector with managed memory

The scatter-vector approach discussed in Section 3 uses an upper bound method based on the number of operations corresponding to that row for allocating memory to the result row in matrix C. The rows in matrix 'A' are binned based on the operation count (more information about binning is provided in section 7.2). Each bin contains a set of rows whose operation counts fall in a particular range. For all the rows in a given bin, memory space corresponding to the maximum op-count for that bin is allocated. But this strategy

Table 4: Machines used for Benchmarking

Resource	Machine 1	Machine 2
CPU	Intel E3-1275 V2(4 cores, 3.5 GHz, 8MB L3 cache), 16GB (DDR3-1600)	Intel core i7-4770K (4 cores, 3.5 GHz, 8MB L3 cache), 16GB (DDR3-1600)
GPU	GeForce K20c (13 Kepler SMs, 192 cores/MP, 5 GB Global Memory, 706MHz, 1.5 MB L2 cache, ECC off)	GeForce GTX TITAN (14 Kepler SMs, 192 cores/MP, 6 GB Global Memory, 876MHz, 1.5 MB L2 cache, ECC off)
Software	Red Hat Enterprise Linux Server release 6.7 x86_64, CUDA 7.5, GCC 4.9.2, ICC 13.1.3, CUSP 0.4.0, CUSPARSE v8.0, Nvidia driver 352.79	Red Hat Enterprise Linux Server release 6.7 x86_64, CUDA 7.5, GCC 4.9.2, ICC 13.1.3, CUSP 0.4.0, CUSPARSE v8.0, Nvidia driver 352.79

(a) Matrices with low throughput: Single Precision **(b) Matrices with high throughput: Single Precision** **(c) Matrices with low throughput: Double Precision** **(d) Matrices with high throughput: Double Precision****Figure 5: Throughput comparison for CUSP, cuSPARSE, bhSPARSE and HybridSparse on Nvidia Kepler GTX Titan GPU**

of memory allocation becomes very inefficient at high operation counts, especially for matrices with high compression ratio. Hence we use a single bin for all rows having op-count greater than 1024 and adopt a dynamic memory allocation strategy for this bin. A chunk of predefined size is initially allocated to each block. As

blocks process the rows, they sequentially write the elements to the chunk space owned by them. If a block runs out of chunk space a new chunk of memory from the memory bank is allocated.

7 IMPLEMENTATION

7.1 Creating Synthetic Matrices

For characterization experiments, two sets of synthetic matrices were generated. The first set of experiments were designed to study the performance of different SpGEMM approaches with increasing work per row. For this purpose, matrices were generated with a fixed number of non zeros in every row of a matrix, but differing across the set of synthetic matrices. Consider the case where the work per row is 64; for this we created a matrix 'A' such that each row has 8 elements with each element having a random column id. With such a matrix, when computing $C = A * A$ exactly 64 FMAs are required for every row of C. Since the column ids are chosen randomly, we avoid any structure in the access pattern. The matrices were created with sufficiently large number of rows (of the order of 200k) so that there is enough work for load-balanced execution across the SMs.

The second set of experiments were designed to study the impact on performance of different algorithms when the structure of rows in A changes but the work per row remains constant. Consider a case where the work per row remains constant at 220 FMAs per row. We created a matrix 'A' with 10 non-zeros per row with each element having a random column id and correspondingly we created matrix B with each row having 22 non-zero elements per row with each element having a random column id. Thus when we perform $A * B$, we are guaranteed that the work per row is 220. Similarly we created other synthetic matrix-pairs A and B having same work per row but different number of non-zeros in 'A'.

7.2 Binning

In order to have good concurrency among all the threads, the rows of A are initially grouped into different bins based on the total number of ops each row is involved in. We are creating bins with range of ops being powers of 2; i.e. the range of ops for different bins is 2-4, 4-8, 4-16, 16-32 and so on. For all the rows in a given bin the memory in matrix C is allocated in upper bound fashion; i.e. for all the rows in the bin having 32-64 ops are preallocated with 64 memory locations. Note that the maximum number of non-zeros that are possible in the corresponding result row of C is 64. As the number of ops per row increases beyond a certain point the upper bound method for memory allocation becomes were inefficient. Hence we put all the rows having ops greater than 1024 in the last bin and allocate memory in a progressive fashion. After binning the rows based on ops, we are sub-binning them based on the number of non-zeros in each row of A. We are employing different SpGEMM algorithm for each sub-bin. Other kinds of binning have been tried by different papers.[16]

7.3 Thread block configuration and kernel selection

The main goal of binning is to choose the thread block size according to the bins, such that sufficient work and load balancing can be ensured. Also, the shared memory used can be tuned based on the bin to achieve higher occupancy. In addition to the latter, for different bins we choose different SpGEMM strategies/kernels. For the bins having small number of ops (i.e. upto 8 ops per row) we use 2-phase mechanism described in Section 2.2. In the first phase we determine the number of non-zeros in the result-row and we

will reserve space for these rows in C accordingly. In the second phase we perform the actual multiplication.

For the bins having ops from 8 to 256, we use ESC approach using shared memory with virtual warps. Virtual warps are used to ensure load balancing among all the threads in a virtual warp. Hence we choose the size of a virtual warp for a given bin in accordance with the maximum number of ops for the bin. Different kernels are launched for different sub-bins based on the number of non zero elements in row A.

As we observed in the Section 4, for rows with higher number of ops, scatter vector performs better than ESC. Hence for all the higher bins we use scatter vector approach. For the rows in bins having ops from 256-1024 we allocate memory corresponding to maximum ops for that bin and for rows having ops beyond 1024, we allocate memory dynamically.

8 EXPERIMENTAL EVALUATION

The new HybridSparse SpGEMM implementation was evaluated by comparing performance against three publicly available SpGEMM implementations: from Nvidia's CUSP library, from Nvidia's cuSPARSE library, and bhSPARSE [16, 18]. Tests were performed on two GPUs - details are provided in Table 4. Both single and double precision performance were measured. The same set of sparse matrices from the Florida dataset collection [22] were used that were also used by Liu et al. [16, 18].

A recent SpGEMM implementation called *balanced hashing* has reported better performance than bhSPARSE [2]. Since source code was unavailable, we present an indirect comparison against it. Anh et al. [2] present a comparison of their balanced hashing approach against bhSPARSE for single precision data on an Nvidia Tesla K20Xm GPU with 2688 cores in 14 SMs. In Fig. 6a and Fig. 6b, for each test matrix, GFLOPs performance data is presented for four scenarios: i) measured performance with our hybrid SpGEMM implementation on a K20c GPU, ii) measured performance of bhSPARSE on the K20c GPU, iii) performance reported by Anh et al. for their balanced hashing approach on a K20Xm GPU, and iv) performance reported by Anh et al. for bhSPARSE on a K20Xm GPU. From Fig. 6 and 5, we can see that except for the case of the scircuit dataset, the new hybrid SpGEMM outperforms the balanced hashing approach, as well as the bhSPARSE approach.

Figure 6 and 5 shows that that our hybridSparse is faster than the other SpGEMM algorithms both in the case of single precision and double precision on all of the matrices. Especially for the case of web-networks(webbase-1M) and biological-networks(protein), we achieved speedups upto 3.9x and 4.1x over bhSPARSE respectively. The hybrid approach achieves high GFlops for the matrices such as hood, cant, harbor, pwt, ship, spheres and protein. These matrices have rows which are high in ops and are processed through scatter vector approach. Hence high GFlops is a consequence of our implementation of scatter vector approach on GPU for these matrices. We achieved on average 2X speedup over bhSPARSE on all the matrices in both single and double precision multiplications and on both the machines.

Even for the matrices that don't have rows with high number of ops, the new hybrid approach outperforms bhSPARSE with very good speedup.

(a) Matrices with low throughput: Single Precision

(b) Matrices with high throughput: Single Precision

(c) Matrices with low throughput: Double Precision

(d) Matrices with high throughput: Double Precision

Figure 6: Throughput comparison for CUSP, CUSPARSE, BHSPARSE and HybridSparse on Machine 1

9 DISCUSSION AND CONCLUSION

In this paper, we have attempted a systematic analysis of alternate approaches to performing SpGEMM on a GPU. By use of synthetic matrices whose characteristics could be controlled, the benefits of alternate approaches were characterized, and a hybrid scheme was implemented.

Although the new hybrid SpGEMM has improved performance over prior GPU SpGEMM implementations, there seems to be much further room for improvement. An interesting open question is whether good upper bounds on SpGEMM can be established. For sparse matrix-vector multiplication, a practically useful bound can be established using the roofline model. Given a sparse $n \times n$ matrix with nnz nonzero elements, SpMV requires a minimum data movement from/to global memory of $12 * nnz + 8 * n$ bytes of data for double-precision computation using the CSR representation. This allows an upper bound estimation of SpMV time as

$(12 * nnz + 8 * n) / \text{Peak-memory-BW}$. A similar upper limit could be attempted for SpMM, based on the total data volume for the input and output matrices. Fig. 7a shows the results of such an exercise for the tested matrices, with Fig. 7b showing the actual GFLOPs achieved by SpMV and SpGEMM. For SpMV, we used the state-of-the-art CSR5 implementation from Liu and Vinter [17], and for SpGEMM we used the hybrid-sparse implementation presented here. It can be seen that whereas the achieved performance for SpMV is often within a factor of 2 of the roofline upper bound based on global-memory bandwidth of the GPU, the achieved performance with SpGEMM is very far from such a bound. It may also be seen that the absolute performance achieved with SpGEMM is often far lower than SpMV performance, in stark contrast to dense matrix multiplication where MM achieves orders of magnitude higher performance than MV.

(a) Ratio of bandwidth-based performance upper bound to actual performance

(b) SpGEMM GFLOPs vs SpMV GFLOPs

Figure 7: Comparing achieved performance with upper bounds: SpGEMM versus SpMV

ACKNOWLEDGMENT

We thank the reviewers for their valuable feedback and suggestions that helped improve the paper. This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Contract D16PC00183, and the National Science Foundation (NSF) through awards 1440749, 1404995, and 1629548.

REFERENCES

- [1] Rasmus Resen Amossen, Andrea Campagna, and Rasmus Pagh. 2010. Better size estimation for sparse matrix products. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*. Springer, 406–419.
- [2] Pham Nguyen Quang Anh, Rui Fan, and Yonggang Wen. 2016. Balanced Hashing and Efficient GPU Sparse General Matrix-Matrix Multiplication. In *2016 International Conference on Supercomputing (ICS '16)*. ACM, New York, NY, USA, Article 36, 12 pages.
- [3] Nathan Bell, Steven Dalton, and Luke N Olson. 2012. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing* 34, 4 (2012), C123–C152.
- [4] Aydin Buluc and John R Gilbert. 2008. Challenges and advances in parallel sparse matrix-matrix multiplication. In *2008 37th International Conference on Parallel Processing*. IEEE, 503–510.
- [5] Aydin Buluc and John R Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *IEEE International Symposium on Parallel and Distributed Processing, (IPDPS '08)*. IEEE, 1–11.
- [6] Timothy M Chan. 2010. More algorithms for all-pairs shortest paths in weighted graphs. *SIAM J. Comput.* 39, 5 (2010), 2075–2089.
- [7] Edith Cohen. 1996. On optimizing multiplications of sparse matrices. In *International Conference on Integer Programming and Combinatorial Optimization*. Springer, 219–233.
- [8] S. Dalton and N. Bell. 2014. A c++ templated sparse matrix library. (2014). <http://cusplibrary.github.com>
- [9] Steven Dalton, Luke Olson, and Nathan Bell. 2015. Optimizing sparse matrix-matrix multiplication for the gpu. *ACM Transactions on Mathematical Software (TOMS)* 41, 4 (2015), 25.
- [10] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1.
- [11] Julien Demouth. 2012. Sparse matrix-matrix multiplication on the gpu. In *Proceedings of the GPU Technology Conference*.
- [12] John R Gilbert, Cleve Moler, and Robert Schreiber. 1992. Sparse matrices in MATLAB: Design and implementation. *SIAM J. Matrix Anal. Appl.* 13, 1 (1992), 333–356.
- [13] John R Gilbert, Steve Reinhardt, and Viral B Shah. 2006. High-performance graph algorithms from parallel sparse matrices. In *International Workshop on Applied Parallel Computing*. Springer, 260–269.
- [14] Felix Gremse, Andreas Hoyer, Lars Ole Schwen, Fabian Kiessling, and Uwe Naumann. 2015. GPU-accelerated sparse matrix-matrix multiplication by iterative row merging. *SIAM Journal on Scientific Computing* 37, 1 (2015), C54–C71.
- [15] Fred G. Gustavson. 1978. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.* 4, 3 (Sept. 1978), 250–269.
- [16] Weifeng Liu and Brian Vinter. 2014. An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS '14)*. 370–381.
- [17] Weifeng Liu and Brian Vinter. 2015. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 339–350.
- [18] Weifeng Liu and Brian Vinter. 2015. A framework for general sparse matrix-matrix multiplication on GPUs and heterogeneous processors. *J. Parallel and Distrib. Comput.* 85 (2015), 47–61.
- [19] Qingpeng Niu, Pai-Wei Lai, SM Faisal, Srinivasan Parthasarathy, and P Sadayappan. 2014. A fast implementation of MLR-MCL algorithm on multi-core processors. In *High Performance Computing (HPC), 2014 21st International Conference on*. IEEE, 1–10.
- [20] NVIDIA. 2017. Nvidia cusparse library. (2017). <http://developer.nvidia.com/cusparse>
- [21] Peter D Sulatycke and Kanad Ghose. 1998. Caching-efficient multithreaded fast multiplication of sparse matrices. In *Parallel Processing Symposium, 1998. IPPS/SPDP*. IEEE, 117–123.
- [22] UFL. 2011. Sparse Matrix Collection. (2011). <https://www.cise.ufl.edu/research/sparse/matrices/>
- [23] Virginia Vassilevska, Ryan Williams, and Raphael Yuster. 2010. Finding heaviest H-subgraphs in real weighted graphs, with applications. *ACM Transactions on Algorithms (TALG)* 6, 3 (2010), 44.
- [24] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.
- [25] Raphael Yuster and Uri Zwick. 2005. Fast sparse matrix multiplication. *ACM Transactions on Algorithms (TALG)* 1, 1 (2005), 2–13.