2014 IEEE 28th International Parallel & Distributed Processing Symposium

An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data

Weifeng Liu, Brian Vinter
Niels Bohr Institute
University of Copenhagen
Copenhagen, Denmark
{*weifeng, vinter*}*@nbi.dk*

***Abstract*—General sparse matrix-matrix multiplication (SpGEMM) is a fundamental building block for numerous applications such as algebraic multigrid method, breadth first search and shortest path problem. Compared to other sparse BLAS routines, an efficient parallel SpGEMM algorithm has to handle extra irregularity from three aspects: (1) the number of the nonzero entries in the result sparse matrix is unknown in advance, (2) very expensive parallel insert operations at random positions in the result sparse matrix dominate the execution time, and (3) load balancing must account for sparse data in both input matrices. Recent work on GPU SpGEMM has demonstrated rather good both time and space complexity, but works best for fairly regular matrices.**

In this work we present a GPU SpGEMM algorithm that particularly focuses on the above three problems. Memory pre-allocation for the result matrix is organized by a hybrid method that saves a large amount of global memory space and efficiently utilizes the very limited on-chip scratchpad memory. Parallel insert operations of the nonzero entries are implemented through the GPU merge path algorithm that is experimentally found to be the fastest GPU merge approach. Load balancing builds on the number of the necessary arithmetic operations on the nonzero entries and is guaranteed in all stages.

Compared with the state-of-the-art GPU SpGEMM methods in the CUSPARSE library and the CUSP library and the latest CPU SpGEMM method in the Intel Math Kernel Library, our approach delivers excellent absolute performance and relative speedups on a benchmark suite composed of 23 matrices with diverse sparsity structures.

***Keywords*-sparse matrices; matrix multiplication; linear algebra; GPU; merging; parallel algorithms;**

I. Introduction

General matrix-matrix multiplication (GEMM) is one of the most crucial operations in computational science and modeling. The operation multiplies a matrix A of size $m \times k$ with a matrix B of size $k \times n$ and gives a result matrix C of size $m \times n$. In many linear solvers and graph problems such as algebraic multigrid method [1], breadth first search [2], finding shortest path [3], colored intersection [4] and sub-graphs [5], it is required to exploit sparsity of the two input matrices and the result matrix because their dense forms normally need huge storage space and computation cost for the zero entries. Therefore general sparse matrix-matrix multiplication (SpGEMM) becomes a common building block in these applications.

Compared to the CPUs, modern graphics processing units (GPUs) promise much higher peak floating-point performance and memory bandwidth. Thus a lot of research has concentrated on GPU accelerated sparse matrix-dense vector multiplication [6] and sparse matrix-dense matrix multiplication [7], [8] and achieved relatively attractive performance. However, despite the prior achievements on these GPU sparse BLAS routines, massive parallelism in the GPUs is still significantly underused for the SpGEMM algorithm, because it has to handle three more challenging problems: (1) the number of the nonzero entries in the result matrix is unknown in advance, (2) very expensive parallel insert operations at random positions in the result matrix dominate the execution time, and (3) load balancing must account for sparse data in both input matrices with diverse sparsity structures.

Previous GPU SpGEMM methods [9], [10], [1], [11], [12] have proposed a few solutions for the above problems and demonstrated relatively good time and space complexity. However, the experimental results showed that they either only work best for fairly regular sparse matrices (with most of the nonzero entries are on the diagonal) [9], [10], or bring extra high memory overhead for matrices with some specific sparsity structures [1], [11], [12]. Moreover, in the usual sense, none of these methods can outperform well optimized SpGEMM approach [13] for multicore CPUs.

Our work described in this paper particularly focuses on improving the GPU SpGEMM performance for matrices with arbitrary irregular sparsity structures by proposing more efficient methods to solve the above three problems on the GPUs. We make the following contributions:

- ***A Hybrid method for the result matrix pre-allocation***. We present a hybrid method that initially allocates memory of upper bound size for the short rows and progressively allocates memory for the long rows. The experimental results show that our method saves a large amount of global memory space and efficiently utilizes the very limited on-chip scratchpad memory.
- ***Parallel insert operations through fast merging***. We propose an efficient parallel insert method for the long rows of the result matrix by using the fastest merge algorithm available on the GPUs. We make an experimental evaluation and choose the GPU merge path

1530-2075/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPS.2014.47

algorithm from five candidate GPU merge approaches.
- ***Heuristic-based load balancing***. We develop a load balancing oriented heuristic method that assigns the rows of the result matrix to multiple bins with different subsequent computational methods. Our approach guarantees load balancing in all calculation stages.

Our algorithm delivers excellent performance in experiments that compute $C = A^2$ on a benchmark suite composed of 23 sparse matrices with diverse sparsity structures. First, compared with the state-of-the-art GPU SpGEMM methods in the CUSPARSE library and the CUSP library on an nVidia GeForce GTX Titan GPU, our approach delivers average 2.6x (up to 7.7x) and 4x (up to 8.9x) speedup on the single precision SpGEMM (SpSGEMM) and average 2.7x (up to 7.9x) and 2.4x (up to 5.4x) speedup on the double precision SpGEMM (SpDGEMM), respectively. Second, compared to the SpGEMM method in the state-of-the-art Intel Math Kernel Library (MKL) on a machine with one six-core Xeon E5-2630 CPU and quad-channel system memory, our method gives average 1.3x (up to 2.6x) and 2x (up to 4.5x) SpSGEMM speedup on the nVidia GPU and an AMD Radeon HD 7970 GPU, respectively, and average 1.1x (up to 1.9x) and 1.4x (up to 2.4x) SpDGEMM speedup on the above two GPUs, respectively. To the best of our knowledge, this is the first time the GPU SpGEMM algorithm outperforms CPU method in the Intel MKL on the latest CPU hardware.

II. SpGEMM Overview

For the sake of generality, the SpGEMM algorithm description starts from discussion of the GEMM and gradually takes sparsity of the matrices A, B and C into consideration. For the matrix A, we write a_{ij} to denote the entry in the ith row and the jth column of A and a_{i*} to denote the vector consisting of the ith row of A. Similarly, the notation a_{*j} denotes the jth column of A. In the GEMM, the ith row of the result matrix C can be defined by

$$c_{i*} = (a_{i*} \cdot b_{*1}, a_{i*} \cdot b_{*2}, \ldots, a_{i*} \cdot b_{*p}),$$

where the operation $\cdot$ is dot product of the two vectors.

We first give sparsity of the matrix A consideration. Without loss of generality, we assume that the ith row of A only consists of two nonzero entries in the kth and the lth column, respectively. Thus a_{i*} becomes (a_{ik}, a_{il}). Since all other entries are zeros, we do not record them explicitly and ignore their influence on the dot products in the calculation of the ith row of C. Then we obtain

$$c_{i*} = (a_{ik}b_{k1} + a_{il}b_{l1}, a_{ik}b_{k2} + a_{il}b_{l2}, \ldots, a_{ik}b_{kp} + a_{il}b_{lp}).$$

We can see in this case, only entries in the kth and the lth row of B have contribution to the ith row of C. Then row vector form instead of column vector form is used for the matrix B. So we obtain

$$c_{i*} = a_{ik}b_{k*} + a_{il}b_{l*}.$$

Since the matrix B is sparse as well, again without loss of generality, we assume that the kth row of B has only two nonzero entries in the rth and the tth column, and the lth row of B also has only two nonzero entries in the sth and the tth column. So the two rows are given by $b_{k*} = (b_{kr}, b_{kt})$ and $b_{l*} = (b_{ls}, b_{lt})$. Then

$$c_{i*} = a_{ik}(b_{kr}, b_{kt}) + a_{il}(b_{ls}, b_{lt}).$$

Because the matrix C is also sparse and the ith row of C only has three nonzero entries in the rth, the sth and the tth column, the row can be given by

$$c_{i*} = (c_{ir}, c_{is}, c_{it}),$$

where $c_{ir} = a_{ik}b_{kr}$, $c_{is} = a_{il}b_{ls}$ and $c_{it} = a_{ik}b_{kt} + a_{il}b_{lt}$.

In general there are more nonzero entries per rows of the matrices A, B and C. But from the above derivation we can see that the SpGEMM can be represented by operations on row vectors of the matrices. Therefore, in this work we store all sparse matrices in compressed sparse row (CSR) format. The CSR format of a matrix consists of three separate arrays: (1) row pointer array of size $n + 1$, where n is the number of the rows of the matrix, (2) column index array of size nnz, where nnz is the number of the nonzero entries of the matrix, and (3) value array of size nnz. Hence the overall space complexity of the CSR format is $O(n+nnz)$. Actually compressed sparse column (CSC) format is also widely used for sparse matrices stored in column-major order [14]. The SpGEMM in the CSC format is almost the same as in the CSR format except rows are changed to columns and vice versa.

The above CSR format-based SpGEMM algorithm can be performed by pseudocode in Algorithm 1. An early description of this algorithm was given by Gustavson [15].

Algorithm 1 Pseudocode for the SpGEMM.

```
1:  for each a_{i*} in the matrix A do
2:      set c_{i*} to ∅
3:      for each nonzero entry a_{ij} in a_{i*} do
4:          load b_{j*}
5:          for each nonzero entry b_{jk} in b_{j*} do
6:              value ← a_{ij} b_{jk}
7:              if c_{ik} ∉ c_{i*} then
8:                  insert c_{ik} to c_{i*}
9:                  c_{ik} ← value
10:             else
11:                 c_{ik} ← c_{ik} + value
12:             end if
13:         end for
14:     end for
15: end for
```

III. Related Work

A. Prior SpGEMM Algorithms

A classical CPU SpGEMM algorithm, also known as Matlab algorithm, was proposed by Gilbert et al. [14]. This approach uses a dense vector-based sparse accumulator (or SPA) and takes $O(flops + nnz(B) + n)$ time to complete the SpGEMM, where $flops$ is defined as the number of the necessary arithmetic operations on the nonzero entries, $nnz(B)$ is defined as the number of the nonzero entries in the matrix B, and n is the number of rows/columns of the input square matrices. Matam et al. [16] developed a similar Matlab algorithm implementation for GPUs. Sulatycke and Ghose [17] proposed a cache hits-oriented algorithm runs in relatively long $O(flops + n^2)$ time. A fast serial SpGEMM algorithm with time complexity $O(nnz^{0.7}n^{1.2} + n^{2+o(1)})$ was developed by Yuster and Zwick [18]. Buluç and Gilbert [19] presented a SpGEMM algorithm with time complexity independent to the size of the input matrices under assumptions that the algorithm is used as a sub-routine of 2D distributed memory SpGEMM and the input matrices are hypersparse ($nnz < n$).

Recent GPU-based SpGEMM algorithms showed better time complexity. The SpGEMM algorithm in the CUSPARSE library [9], [10] utilized GPU hash table for the insert operations (lines 7–11 in the Algorithm 1). So time complexity of this approach is $O(flops)$ on average and $O(flops\ nnzr(C))$ in the worst case, where $nnzr(C)$ is defined as the average number of the nonzero entries in the rows of the matrix C. Because the algorithm allocates one hash table of fixed size for each row of C, the space complexity is $O(nnz(A) + nnz(B) + n + nnz(C))$.

The CUSP library [1], [11] developed a SpGEMM method called expansion, sorting and compression (ESC) that expands all candidate nonzero entries generated by the necessary arithmetic operations (line 6 in the Algorithm 1) into an intermediate sparse matrix $\widehat{C}$, sorts the matrix by rows and columns and compresses it into the result matrix C by eliminating entries in duplicate positions. By using the GPU radix sort algorithm (with linear time complexity while size of the index data type of the matrices is fixed) and prefix-sum scan algorithm (with linear time complexity) as building blocks, time complexity of the ESC algorithm is $O(flops + nnz(\widehat{C}) + nnz(\widehat{C}))$. Since $nnz(\widehat{C})$ equals half of $flops$, the ESC algorithm takes the optimal $O(flops)$ time. Dalton et al. [12] improved the ESC algorithm by executing sorting and compression on the rows of $\widehat{C}$, but not on the entire matrix. Therefore fast on-chip memory has a chance to be utilized more efficiently. The improved method sorts the very short rows (of size no more than 32) by using sorting network algorithm (with time complexity $O(nnzr(\widehat{C}) \log^2(nnzr(\widehat{C})))$) instead of the radix sort algorithm which is mainly efficient for long lists. So the newer method is more efficient in practice, even though its time complexity is not lower than the original ESC algorithm. Because both of the ESC algorithms allocate an intermediate matrix $\widehat{C}$, they have the same space complexity $O(nnz(A) + nnz(B) + nnz(\widehat{C}) + nnz(C))$.

B. Terminology Definition for GPU Programming

Because CUDA and OpenCL are both widely used in GPU programming and they actually deliver comparable performance [20], our SpGEMM algorithm support both of them. We use CUDA implementation on nVidia GPU and OpenCL implementation on AMD GPU in our SpGEMM evaluation.

For simplicity, we define the following unified terminologies: (1) *thread* denotes *thread* in CUDA and *work item* in OpenCL, (2) *thread bunch* denotes *warp* in nVidia GPU and *wavefront* in AMD GPU, (3) *thread group* denotes *thread block* or *cooperative thread array (CTA)* in CUDA and *work group* in OpenCL, (4) *core* denotes *streaming multiprocessor (SMX)* in nVidia GPU and *compute unit* in AMD GPU, and (5) *scratchpad memory* denotes *shared memory* in CUDA and *local memory* in OpenCL.

IV. Benchmark Suite

To evaluate our SpGEMM algorithm, we choose 23 sparse matrices as our benchmark suite. 16 of them were widely used for performance evaluations in previous sparse matrix computation research [21], [22], [9], [12], [23], [13]. The other 7 new matrices are chosen since they bring more diverse irregular sparsity structures that challenge the SpGEMM algorithm design. The variety of sparsity structures are from many application fields, such as finite element mesh, macroeconomic model, protein data, circuit simulation, web connectivity, combinational problem. All of the 23 matrices are downloadable from the University of Florida Sparse Matrix Collection [24].

Without loss of generality, in this paper we only evaluate multiplication of sparse square matrix and itself (i.e. $C = A^2$) to avoid introducing another sparse matrix as a multiplier with different sparsity structure. Even though the operation cannot cover all real world problems, it offers a relatively fair platform for benchmarking the SpGEMM algorithms. Moreover, symmetry in the sparse matrices is not used in our SpGEMM algorithm, although some matrices in the benchmark suite are symmetric.

Besides the input matrix A, the work complexities of the different SpGEMM algorithms also depend on the intermediate matrix $\widehat{C}$ and the result matrix C. So we list characteristics of the three matrices in Table I. The set of characteristics includes n (matrix dimension), nnz (the number of the nonzero entries) and $nnzr$ (the average number of the nonzero entries in rows). The upper 9 matrices in the Table I have relatively regular nonzero entry distribution mostly on the diagonal. The other 14 matrices include various irregular sparsity structures.

Table I
OVERVIEW OF EVALUATED SPARSE MATRICES

Name	Plot	n	$nnz(A)$ $nnzr(A)$	$nnz(\widehat{C})$ $nnzr(\widehat{C})$	$nnz(C)$ $nnzr(C)$
cant		63 K	4 M 64	269.5 M 4315	17.4 M 279
economics		207 K	1.3 M 6	7.6 M 37	6.7 M 32
epidemiology		207 K	2.1 M 4	8.4 M 16	5.2 M 10
filter3D		526 K	2.7 M 25	86 M 808	20.2 M 189
pwtk		106 K	11.6 M 53	626.1 M 2873	32.8 M 150
ship		141 K	7.8 M 55	450.6 M 3199	24.1 M 171
harbor		47 K	2.4 M 51	156.5 M 3341	7.9 M 169
protein		36 K	4.3 M 119	555.3 M 15249	19.6 M 538
spheres		83 K	6 M 72	463.8 M 5566	26.5 M 318
2cubes_sphere		102 K	1.6 M 16	27.5 M 270	9 M 88
accelerator		121 K	2.6 M 22	79.9 M 659	18.7 M 154
cage12		130 K	2 M 16	34.6 M 266	15.2 M 117
hood		221 K	10.8 M 49	562 M 2548	34.2 M 155
m133-b3		200 K	0.8 M 4	3.2 M 16	3.2 M 16
majorbasis		160 K	1.8 M 11	19.2 M 120	8.2 M 52
mario002		390 K	2.1 M 5	12.8 M 33	6.4 M 17
mono_500Hz		169 K	5 M 30	204 M 1204	41.4 M 244
offshore		260 K	4.2 M 16	71.3 M 275	23.4 M 90
patents_main		241 K	0.6 M 2	2.6 M 11	2.3 M 9
poisson3Da		14 K	0.4 M 26	11.8 M 871	3 M 219
QCD		49 K	1.9 M 39	74.8 M 1521	10.9 M 222
scircuit		171 K	1 M 6	8.7 M 51	5.2 M 31
webbase-1M		1 M	3.1 M 3	69.5 M 70	51.1 M 51

V. PERFORMANCE CONSIDERATIONS

A. Memory Pre-allocation For the Result Matrix

Compared to the SpGEMM, other sparse matrix multiplication operations, such as multiplication of sparse matrix and dense matrix [7], [8], [23] and its special case sparse matrix-vector multiplication [21], [25], [6], pre-allocate a dense matrix or a dense vector of trivially predictable size and store entries to predictable memory addresses. However, because the number of the nonzero entries in the result sparse matrix C is unknown in advance, precise memory allocation of the SpGEMM is impossible before real computation. And physical address of each new entry is unknown either (consider line 7 in the Algorithm 1, the position k is only a column index that cannot trivially map to physical address on memory space).

To solve this problem, the previous SpGEMM algorithms proposed four different solutions: (1) precise method, (2) probabilistic method, (3) upper bound method, and (4) progressive method.

The first method, precise method, pre-computes a simplified SpGEMM by the same computational pattern. We can imagine that multiplication of sparse boolean matrices is more efficient than multiplication of sparse floating-point matrices. The SpGEMM methods in the CUSPARSE library and the Intel MKL are representatives of this method. Even though the pre-computation generates precise size of the result matrix C, this method is relatively expensive since the SpGEMM operation in the same pattern is executed twice.

The second method, probabilistic method, estimates an imprecise $nnz(C)$. This group of approaches [26], [27] are based on random sampling and probability analysis on the input matrices. Since they do not guarantee a safe lower bound for the result matrix C and extra memory has to be allocated while the estimation fails, they were mostly used for estimating the shortest execution time of multiplication of multiple sparse matrices.

The third method, upper bound method, computes an upper bound of the number of the nonzero entries in the result matrix C and allocates corresponding memory space. Numerically, the upper bound size equals $nnz(\widehat{C})$, or half of $flops$, the number of necessary arithmetic operations. The ESC algorithms use this method for memory pre-allocation. Even though this approach saves cost of the pre-computation in the precise method, it brings another problem that the intermediate matrix $\widehat{C}$ might be too large to fit in the device global memory. Since the SpGEMM algorithm does not take into consideration cancellation that eliminates zero entries generated by arithmetic operations, the result matrix is normally larger than the input matrices. The Table I shows that the $nnz(\widehat{C})$ is much larger than the $nnz(C)$ in some cases. For example, the sparse matrix "pwtk" generates 626.1 million nonzero entries (or 7.5 GB memory space for 32-bit index and 64-bit value) for the intermediate matrix $\widehat{C}$ while the real product C only contains 32.8 million nonzero entries. Although the upper bound method can partition the intermediate matrix $\widehat{C}$ into multiple sub-matrices, higher global memory pressure might reduce overall performance.

The last method, progressive method, first allocates memory of a proper size, starts sparse matrix computation and reallocates the buffer if larger space is required. Some CPU sparse matrix libraries use this method. For instance, sparse matrix computation in the Matlab [14] increases the buffer by a ratio of 50% if the current memory space is exhausted.

Since the upper bound method sacrifices space efficiency for the sake of improved performance and the progressive method is good at saving space, we use a hybrid method

composed of the both approaches. However, compared to the relatively convenient upper bound method, it is hard to directly implement a progressive method for the GPUs, because although modern GPU devices have ability of allocating global memory while kernels are running, they still cannot reallocate device memory on the fly. We will describe our hybrid method designed for the GPUs in the next section.

B. Parallel Insert Operations

As shown in the Algorithm 1, for each trivial arithmetic computation (line 6), one much more expensive insert operation (lines 7–11) is required. To the best of our knowledge, none of the previous GPU SpGEMM methods takes into account that the input sequence (line 4) is ordered because of the CSR format[1]. One of our algorithm design objectives is to efficiently utilize this property. Based on experiments by Kim et al. [28], as the SIMD units are getting wider and wider, merge sort methods will outperform hash table methods on the join-merge problem, which is a similar problem in the SpGEMM. Then our problem converts to finding a fast GPU method for merging sorted sequences. Later on we will describe our strategy in detail.

C. Load Balancing

Because distribution patterns of the nonzero entries in the both input sparse matrices are very diverse (consider plots of the matrices in the Table I), input space-based data decomposition [17], [9] normally does not bring efficient load balancing. One exception is that computing the SpGEMM for huge sparse matrices on large scale distributed memory systems, 2D and 3D decomposition on input space methods demonstrated good load balancing and scalability by utilizing efficient communication strategies [29], [30], [2]. However, in this paper we mainly consider load balancing for fine-grained parallelism in the GPU shared memory architectures.

Therefore we use the other group of load balancing methods based on output space decomposition. Dalton et al. [12] presented a method that sorts the rows of the intermediate matrix $\widehat{C}$, divides it into 3 sub-matrices that include the rows in different size ranges, and uses differentiated ESC methods for the sub-matrices. We have a similar consideration, but our implementation is completely different. We do not strictly sort the rows of the intermediate matrix $\widehat{C}$ but just assign rows to a fixed number of bins through a much faster linear time traverse on the CPU. And we decompose the output space in a more detailed way that guarantees much more efficient load balancing. We will demonstrate that our method is always load balanced in all stages for maximizing resource utilization of the GPUs.

[1]Actually according to the CSR format standard, the column indices in each row do not necessarily have to be sorted. But most implementations choose to do so, thus our method reasonably makes this assumption.

VI. Methodology

A. Algorithm Design

Our SpGEMM algorithm is composed of four stages: (1) calculating upper bound, (2) binning, (3) computing the result matrix, and (4) arranging data.

The first stage, calculating upper bound, generates the upper bound number of the nonzero entries in each row of the result matrix C. We create an array U of size m, where m is the number of rows of C, for the upper bound sizes of the rows. We use one GPU thread for computing each entry of the array U. Algorithm 2 describes this procedure.

Algorithm 2 Pseudocode for the first stage on the GPUs.

```
1: for each entry u_i in U in parallel do
2:     u_i ← 0
3:     for each nonzero entry a_ij in a_i* do
4:         u_i ← u_i + nnz(b_j*)
5:     end for
6: end for
```

The second stage, binning, deals with load balancing and memory pre-allocation. We first allocate 38 bins and put them into five bin groups. The bins contain the indices of the entries in the array U and present as one array of size m with 38 segments. Then all rows are assigned to corresponding bins according to the number of the nonzero entries. Finally, based on the sizes of the bins, we allocate a temporary matrix for the nonzero entries in the result matrix C.

The first bin group includes one bin that contains the indices of the rows of size 0. The second bin group also only has one bin that contains the indices of the rows of size 1. Because the rows in the first two bins only require trivial operations, they are excluded from subsequent more complex computation on the GPUs. Thus a better load balancing can be expected.

The third bin group is composed of 31 bins that contain the indices of the rows of size 2–32, respectively. Since the sizes of these rows are no more than the size of a single thread bunch (32 in nVidia GPU or 64 in AMD GPU) and these rows require non-trivial computation, using one thread bunch or one thread group for each row cannot bring efficient instruction throughput on the GPUs. Therefore, we use one thread for each row. And because each bin only contains the rows of the same upper bound size, the bins can be executed separately on the GPUs with different kernel programs for efficient load balancing. In other words, 31 GPU kernel programs are executed for the 31 bins.

The fourth bin group consists of 4 bins that contain the indices of the rows located in size ranges 33–64, 65–128, 129–256 and 257–512, respectively. The rows of these sizes are grouped because of three reasons: (1) each of them is large enough to be efficiently executed by a thread group, (2) each of them is small enough for scratchpad memory (48

KB per core in nVidia Kepler GPU and 64 KB per core in AMD Graphics Core Next, or GCN, GPU), and (3) the final sizes of these rows in the result matrix C are predictable in a reasonable small range (no less than the lower bound 1 and no more than the corresponding upper bound sizes). Even though the rows in each bin do not have exactly the same upper bound size, a good load balancing still can be expected because each row is executed by using one thread group and inter-thread group load balancing is naturally guaranteed by the GPU low-level scheduling sub-systems.

The fifth bin group includes the last bin that contains the indices of the rest of the rows of size larger than 512. These rows have two common features: (1) their sizes can be too large (recall $nnzr(\widehat{C})$ in the Table 1) to fit in the scratchpad memory, and (2) predicting the final sizes of these rows to a small range (scratchpad memory level) is not possible in advance. Therefore, we execute them in a unified progressive method described later. Again because we use one thread group for each row, load balancing is naturally guaranteed.

Since we do not use precise method for memory pre-allocation, a temporary memory space for the result matrix C is required. We design a hybrid method that allocates a CSR format sparse matrix $\widetilde{C}$ of the same size of the result matrix C as temporary matrix. We set $nnz(\widetilde{c}_{i*})$ to u_i while the row index i is located in the bin groups 1–4 because compared with modern GPU global memory capacity, the total space requirement of these rows is relatively small. For the rows in the bin group 5, we set $nnz(\widetilde{c}_{i*})$ to a fixed size 256 since normally this is an efficient working size for the scratchpad memory. Therefore, we can see that if all of the indices of the rows are in the bin groups 1–4, our hybrid method converts to the upper bound method, on the other extreme end, our method converts to the progressive method. But generally, we obtain benefits from the both individual methods. The stage 2 is executed on the CPU since it only requires a few simple linear time traverses, which are more efficient for the CPU cache sub-systems.

The third stage, computing the result matrix, generates the final nonzero entries stored in the temporary matrix $\widetilde{C}$ and $nnz(c_{i*})$, the numbers of the nonzero entries in the rows of the result matrix C.

For the rows in the bin groups 1–2, we simply update the numbers of the corresponding nonzero entries. For the rows in the bin groups 3–5, we use three totally different methods: (1) heap method, (2) bitonic ESC method, and (3) merge method, respectively.

The heap method first creates an empty implicit index-value pair heap (or priority queue) of the upper bound size for each row in the bin group 3. The heaps are located in the scratchpad memory and collect all candidate nonzero entries for corresponding rows. Then each heap executes a heapsort-like operation to generate an ordered sequence located in the tail part of the heap. The difference between this operation and the classical heapsort operation is that

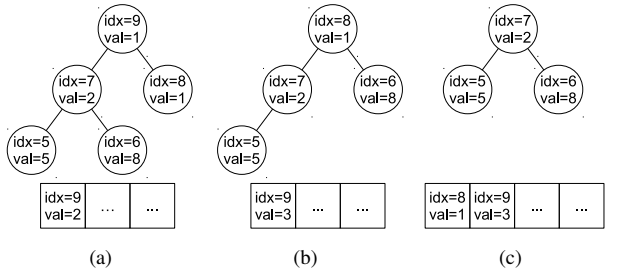


Figure 1. Two steps of an example of the heap method. From (a) to (b), the root entry is fused to the first entry in result sequence since they share the same index. From (b) to (c), the root entry is inserted to the sequence since they have different indices. After each step, the heap property is reconstructed.

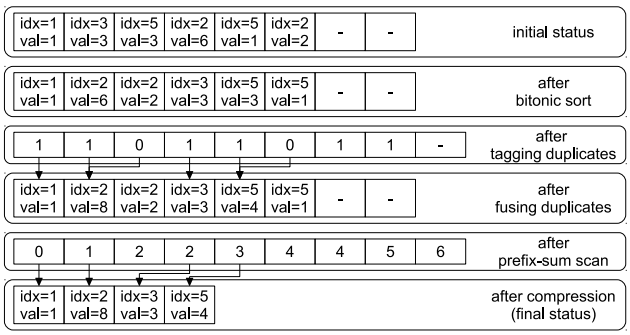


Figure 2. An example of the bitonic ESC method.

the entries in the result sequence are duplicate-free while the initial heap includes duplicate entries. In each delete-max step in our variant heapsort, the root node and the first entry of the result sequence are fused if they share the same index; otherwise the root node is inserted to the head part of the sequence. Our method is also distinguished from a heap-based sparse accumulator given by Gilbert et al. [31] by the mechanism of eliminating duplicate entries. Figure 1 gives two steps of an example of our heap method. Finally, the sorted sequence without duplicate indices is generated in the scratchpad memory and saved to the matrix $\widetilde{C}$ in the global memory. And the numbers of the nonzero entries in the rows of the result matrix C are updated to the sizes of the corresponding result sequences.

For the rows in each bin of the bin group 4, a typical ESC algorithm is used. The method first collects all candidate nonzero entries to an array in the scratchpad memory, then sorts the array by using basic bitonic sort and compresses duplicate indices in the sequence by using prefix-sum scan. Figure 2 shows an example of this procedure. Finally, a sorted sequence without duplicate indices is generated in the scratchpad memory and saved to the matrix $\widetilde{C}$, and the numbers of the nonzero entries in the rows are updated.

For the rows in the bin group 5, our method inserts each input nonzero entry to the corresponding row of the result

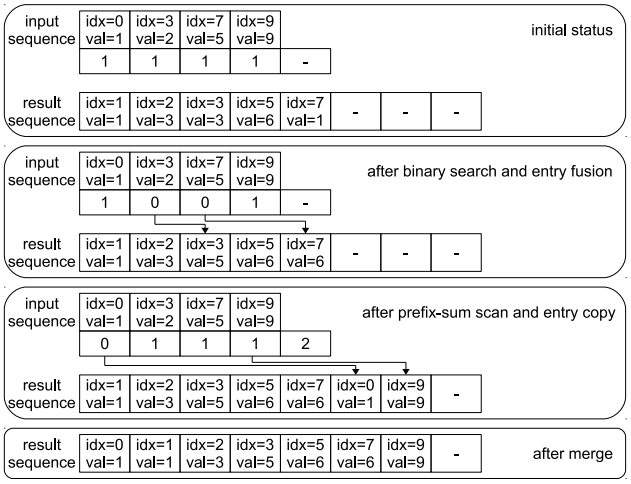


Figure 3. An example of the merge method. The input sequence is in the register file. Its mask sequence and the result sequence are in the scratchpad memory.

matrix C (lines 7–11 in the Algorithm 1) in parallel. We can see that the input sequence (the candidate nonzero entries) and the result sequence (the selected nonzero entries in the current row of C) should always be kept ordered and duplicate-free because of the CSR format. Therefore, we can convert the parallel insert operations to parallel merge operations that merge ordered sequences and the final result sequence is ordered and duplicate-free.

Each parallel merge operation can be split into multiple sub-steps: (1) a binary search operation on the result sequence for fusing entries with the same indices and tagging them, (2) a prefix-sum scan operation on the input sequence for getting continuous position in the incremental part of the result sequence, (3) copying non-duplicate entries from the input sequence to the result sequence, and (4) merging the two sequences in one continuous memory space. Figure 3 shows an example of this procedure. After all input sequences are merged into one result sequence, it is saved to the matrix $\widetilde{C}$, and the numbers of the nonzero entries in the rows are updated.

As we allocate a limited scratchpad memory space for the result sequence, a potential overflow might happen. In this case, we first compare total size of the two sequences (notice the input sequence is in the thread registers, but not in the scratchpad memory yet) with the allocated size of the result sequence in the scratchpad memory. If a merge operation is not allowed, our method records current computation position as a checkpoint and dumps the result sequence from the scratchpad memory to the global memory. Then the host allocates more global memory (we use $2\times$ each time) and re-launches kernel with a $2\times$ large scratchpad memory setting. The relaunched kernels obtain checkpoint information, and load existing results to the scratchpad memory and continue the computation. The global memory dumping and reloading bring an extra overhead, but actually it does not affect the total execution time too much because of three reasons: (1) the global memory access is almost completely coalesced, (2) the latency could be hidden by subsequent computation, and (3) this overhead is only a small factor of large computation (short rows normally do not face this problem). For very long rows exceed the scratchpad memory capacity, our method still allocates a space in the scratchpad memory as a level-1 merge sequence, executes the same merge operations on it and merges the level-1 sequence in the scratchpad memory and the result sequence in the global memory only once before the kernel is ready to return.

It is worth noting that the parameters of the binning depends on specifications (e.g. thread bunch size and scratchpad memory capacity) of the GPU architectures. In this paper, we use the abovementioned fixed-size parameters for assigning the rows into the bins since the current nVidia GPUs and AMD GPUs have comparable hardware specifications. However, the binning strategy can be easily extended for future GPUs with changed architecture designs.

The fourth stage, arranging data, first sums the numbers of the nonzero entries in all rows of the result matrix C and allocates its final memory space. Then our method copies existing nonzero entries from the temporary matrix $\widetilde{C}$ to the result matrix C. For the rows in the bin group 1, copy operation is not required. For the rows in the bin group 2, we use one thread for each row. For the rest of the rows in the bin groups 3–5, we use one thread group for each row. After all copy operations, the SpGEMM computation is done.

B. Evaluating GPU Merge algorithms

Because both the binary search and the prefix-sum scan take fast logarithmic time for each entry in the input sequence, these operations have relatively good efficiency and performance stability on modern GPUs. Therefore, a fast merge algorithm is very crucial for the performance of the merge method in the SpGEMM.

Recently some new merge algorithms [32], [33], [34], [35], [36], [37], [38] have been proposed for the GPUs. But which one is the fastest in practice is still an open question. Because the main objective of the research [36], [37], [38] is efficiently merging large data in the global memory, they still use basic methods, such as bitonic sort and ranking-based merge, as building blocks for small data in the scratchpad memory. Peters et al. [35] proposed a locality-oriented advanced bitonic sort method that can reduce synchronization overhead by merging data in fast private memory instead of relatively slow shared memory. Therefore we evaluate 5 GPU merge algorithms: (1) ranking merge [32], (2) merge path [33], (3) basic oddeven merge [34], (4) basic bitonic merge [34], and (5) advanced bitonic merge [35].

The implementation of the algorithm (2) is extracted from the Modern GPU library [39]. The implementations of the algorithm (3) and (4) are extracted from the nVidia CUDA SDK. We implement the algorithm (1) and (5). Additionally, another reason that we conduct the evaluation is that none of the above literature presented performance of merging short sequences of size less than 2^{12}, which is the most important length (consider the $nnzr(C)$ in the Table 1) for our SpGEMM.

Our evaluation results of merging 32-bit keys, 32-bit key-32-bit value pairs and 32-bit key-64-bit value pairs are shown in Figure 4. The experimental platforms are described in the section VII.A. Each of the five algorithms merges two short ordered sequences of size l into one ordered output sequence of size $2l$. Thus the sorting network methods in our evaluation only execute the last stage. To saturate throughput of the GPUs, the whole problem size is set to size 2^{25}. For example, 2^{14} thread groups are launched while each of them merges two sub-sequences of size $l = 2^{10}$. We execute each problem set through multiple thread groups of different sizes and record the best performance for the evaluation.

We can see that the GPU merge path algorithm almost always outperforms other methods while sub-sequence size is no less than 2^8. Since our merge method starts from size 256, the merge path method is chosen for our SpGEMM implementation. The main advantages of the merge path method are that it can evenly assign work load to threads and can easily deal with the input sequences of arbitrary sizes. Detailed description and complexity analysis of the GPU merge path algorithm can be found in [33].

Other algorithms are not chosen because of various reasons. We can see that the ranking merge is slightly faster than the merge path method in the Figure 4(f). But it is not chosen in our implementation, since the this algorithm is an out-of-place method that requires more scratchpad memory and thus cannot scale to longer sequences. Because the basic bitonic merge and the basic oddeven merge do not show better performance and cannot simply deal with data of arbitrary sizes, none of them is chosen. The advanced bitonic sort method is always the slowest because it loads data from the scratchpad memory to the thread registers for data locality. However, the latency gap between the scratchpad memory and the register file is normally very small and the load operations actually reduce the overall performance. Thus this method should only be used for migrating global memory access to scratchpad memory access.

We can also see that the AMD Radeon HD 7970 GPU is almost always much faster than the nVidia GeForce GTX Titan GPU in all tests. The reason is that the capacity of the scratchpad memory (2048 KB, 64 KB/core $\times$ 32 cores, in the AMD GPU and 672 KB, 48 KB/core $\times$ 14 cores, in the nVidia GPU) heavily influence the performance of merging small sequences. On the other hand, even though the AMD GPU has 64 KB scratchpad memory per core, each instance of the kernel program can only use up to 32 KB. Thus the AMD GPU cannot scale to longer sub-sequences (e.g. 2^{12} with 32-bit key-32-bit value pairs) that can be executed by using the nVidia GPU.

VII. Experimental Results

A. Testbeds

We use two machines shown in Table II for evaluating the SpGEMM algorithms through the benchmark suite.

B. Memory Pre-allocation Comparison

Figure 5 shows the comparison of the three memory pre-allocation methods. We can see that, for small matrices (e.g. "2cubes_sphere"), our hybrid method shows exactly the same space requirements as the upper bound method does. However, for large matrices (e.g. "pwtk"), allocated memory sizes through our hybrid method are much closer to the memory sizes allocated by the precise method. One exception is the matrix "webbase-1M", our hybrid method actually allocates more memory space than the upper bound method. The reasons are that the reduced rate of the intermediate matrix $\widehat{C}$ to the result matrix C is very low (see the Table I) and our $2\times$ progression mechanism just allocates memory across the upper bound size. But overall, our hybrid method saves space allocation of the upper bound method and execution time of the precise method without introducing any significant extra space requirements.

C. SpGEMM Performance Comparison

The absolute and relative performance of the SpGEMM algorithms that compute $C = A^2$ are shown in Figure 6 and 7, respectively. Three GPU methods in the nVidia CUSPARSE v2, CUSP v0.4.0 and BHSPARSE (we call our algorithm set BHSPARSE since this work is under the Project Bohrium [40]) are evaluated on two GPUs: nVidia GeForce GTX Titan and AMD Radeon HD 7970. One CPU method in the Intel MKL v11.0 is evaluated on Intel Xeon E5-2630 CPU. The performance of another recent ESC-based GPU SpGEMM work [12] is not included in the comparison because its source code is not available yet. The Intel MKL SpGEMM program is multithreaded and utilizes all six cores in the Intel Xeon CPU. For the GPU algorithms, data transfer time between the host and the device is not included in our evaluation since the SpGEMM is normally one of the building blocks for more complex problem completely running on the GPUs.

We first compare the performance of the three different GPU SpGEMM algorithms on the nVidia GPU in the machine 1. We can see that BHSPARSE outperforms the CUSPARSE and the CUSP on the most of the sparse matrices. From the perspective of the absolute performance, our method obtains better SpSGEMM and SpDGEMM performance on 17 and 19 matrices out of the whole 23 matrices over the CUSPARSE, and on 21 and 18 matrices over the

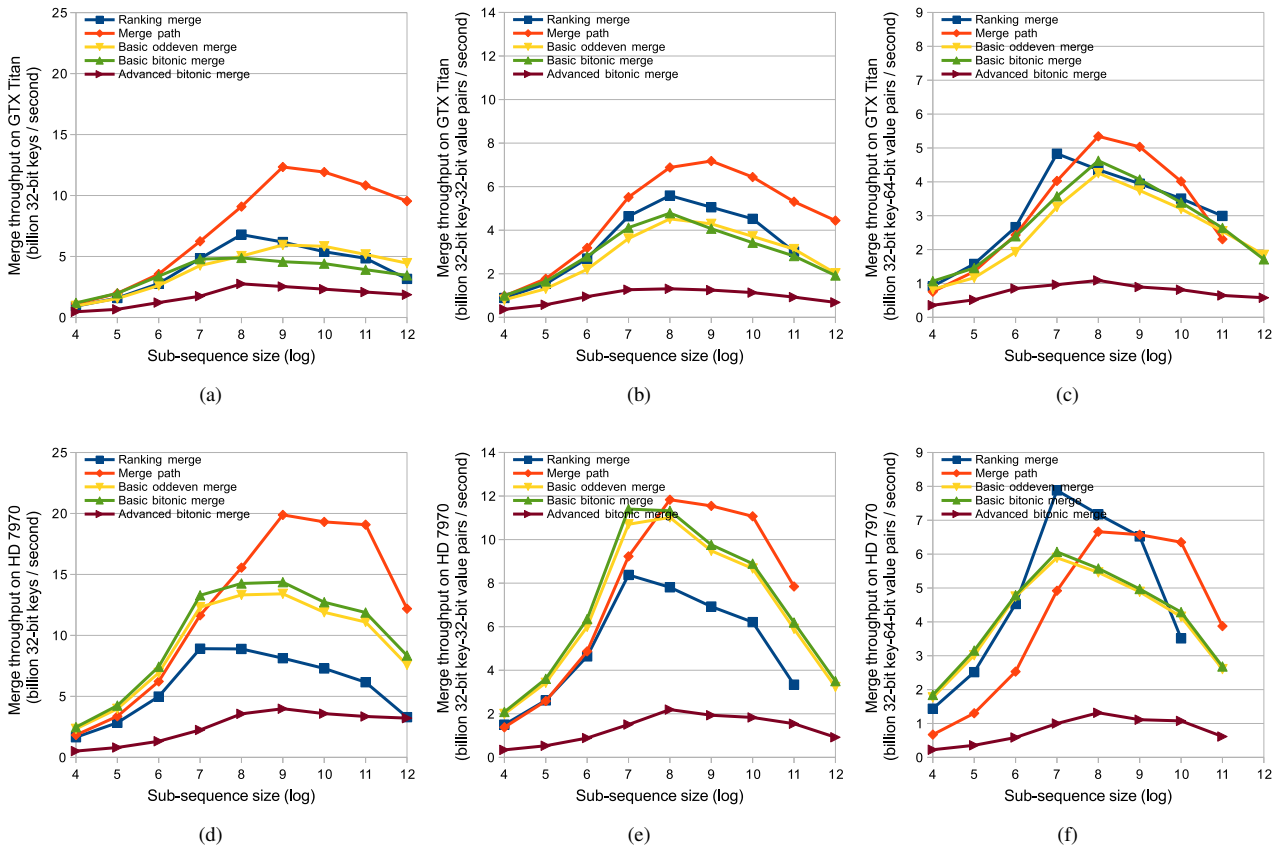


Figure 4. Performance comparison of merging 32-bit keys, 32-bit key-32-bit value pairs and 32-bit key-64-bit value pairs through 5 GPU merge algorithms: ranking merge, merge path, basic oddeven merge, basic bitonic merge and advanced bitonic merge on two different GPUs: nVidia GeForce GTX Titan and AMD Radeon HD 7970.

Table II
TWO MACHINES USED FOR BENCHMARKING

Configuration	Machine 1	Machine 2
CPU	One Intel Xeon E5-2630 (six Sandy Bridge cores, 2.3 GHz, boost up to 2.8 GHz, Hyper-Threading on, 15 MB L3 cache)	One Intel Core i7-3770 (four Ivy Bridge cores, 3.4 GHz, boost up to 3.9 GHz, Hyper-Threading off, 8 MB L3 cache)
System memory	64 GB DDR3-1333 (4 channels, 42.6 GB/s bandwidth)	32 GB DDR3-1600 (2 channels, 25.6 GB/s bandwidth)
GPU	One nVidia GeForce GTX Titan GPU (14 Kepler cores, 2688 CUDA cores, 876 MHz, 4.7 Tflops in single precision, 1.6 Tflops in double precision, 672 KB scratchpad memory)	One AMD Radeon HD 7970 GPU (32 GCN cores, 2048 Radeon cores, 1 GHz, 4 Tflops in single precision, 1 Tflops in double precision, 2048 KB scratchpad memory)
GPU memory	6 GB GDDR5 (288 GB/s bandwidth)	3 GB GDDR5 (288 GB/s bandwidth)
System software and library	Ubuntu Linux 12.04, Intel C++ Compiler 14.0, Intel MKL 11.0, nVidia CUDA SDK 5.5, CUSPARSE v2, CUSP 0.4.0 and GPU driver version 319.32	Ubuntu Linux 12.04, g++ complier 4.6.3, AMD APP SDK 2.8 and GPU driver version 13.4

CUSP, respectively. From the perspective of the relative performance, our method delivers average 2.6x (up to 7.7x) and 4x (up to 8.9x) speedup on SpSGEMM performance over the CUSPARSE and the CUSP, and average 2.7x (up to 7.9x) and 2.4x (up to 5.4x) speedup on SpDGEMM performance over them, respectively.

We can see that the CUSPARSE method outperforms our approach when and only when the input matrices are fairly regular (belong to the first 9 matrices in the Table I). For all irregular matrices and some regular ones, the BHSPARSE is always more efficient. On the other hand, the absolute performance of the CUSP method is very stable since its execution time almost only depends on the number of the necessary arithmetic operations. Therefore this approach is insensitive to sparsity structures. Actually the feature can bring better performance to matrices with

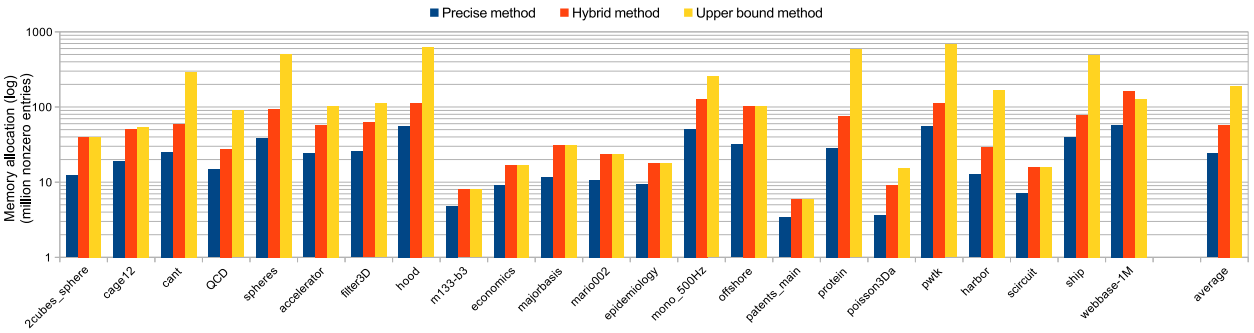
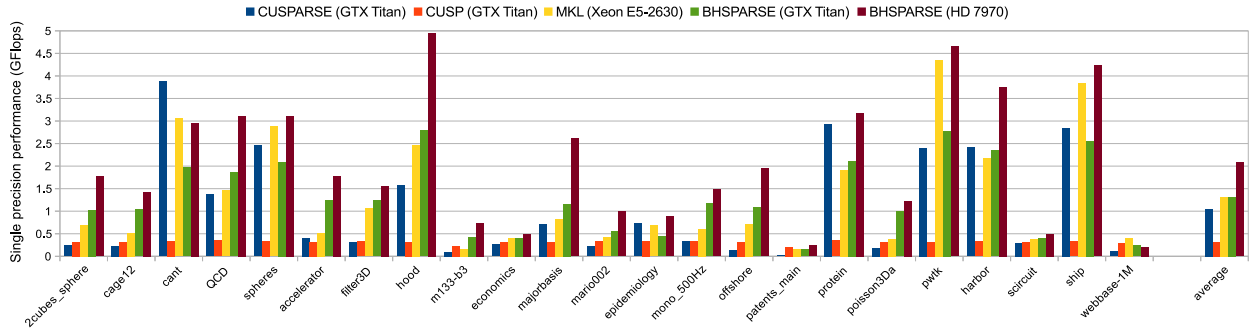
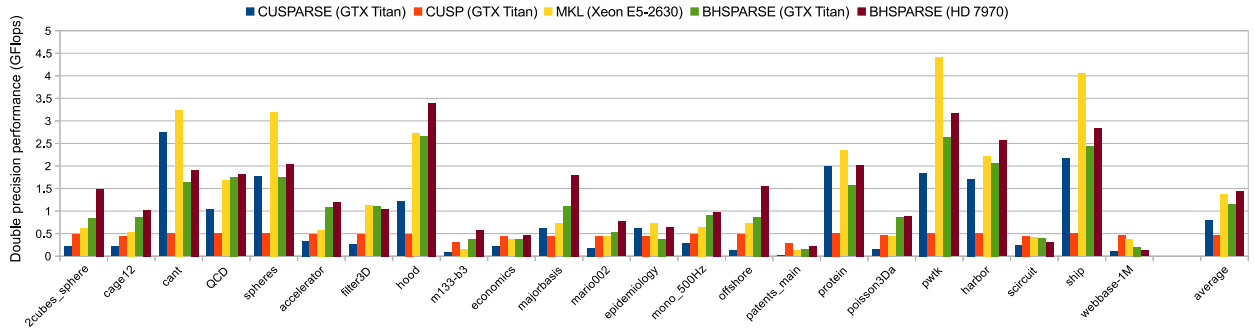


Figure 5. Global memory requirement comparison of the precise method, our hybrid method and the upper bound method on the benchmark suite. The memory requirement of the precise method includes the two input matrices and the result matrix. The memory requirements of the other two methods also contain additional intermediate matrices.



(a)



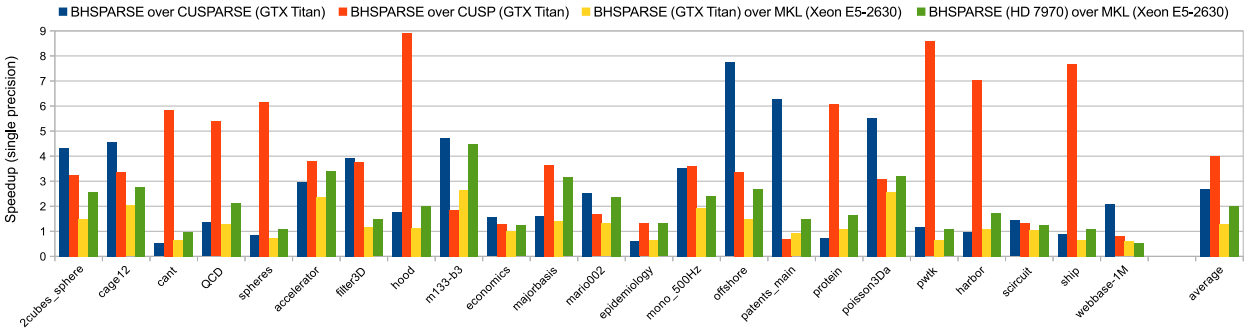
(b)

Figure 6. Absolute performance comparison.

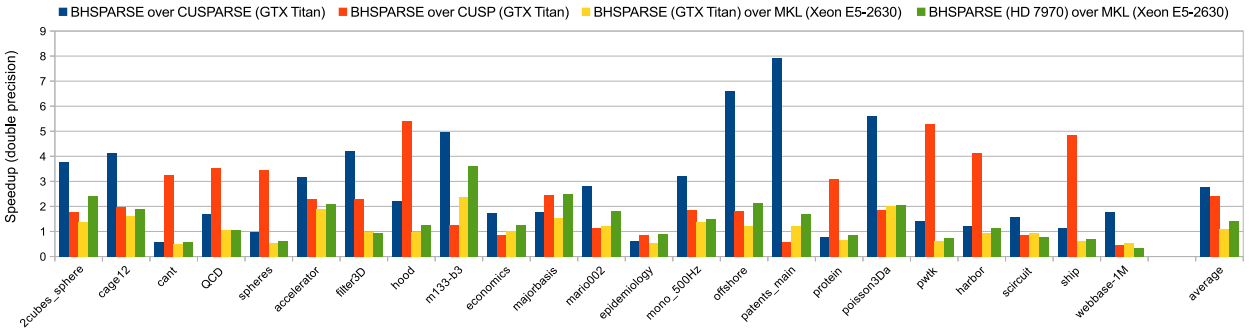
some specific sparsity structures. However in most cases, the CUSP method suffers with higher global memory pressure.

Compared to the Intel MKL on the Intel CPU in the machine 1, our CUDA-based implementation on the nVidia GPU obtains better SpSGEMM and SpDGEMM performance on 16 and 12 matrices, and delivers average 1.3x (up to 2.6x) and 1.1x (up to 1.9x) SpSGEMM and SpDGEMM speedup, respectively. Our OpenCL-based implementation on the AMD GPU in the machine 2 obtains better SpSGEMM and SpDGEMM performance on 21 and 14 matrices, and delivers average 2x (up to 4.5x) and 1.4x (up to 2.4x) SpSGEMM and SpDGEMM speedup, respectively. If we set the Intel MKL SpGEMM performance on this machine as a baseline, our approach is the first GPU SpGEMM that outperforms well optimized CPU method.

Even though the nVidia GPU and the AMD GPU have similar global memory bandwidth and the former one even offers higher peak computational power, the AMD GPU outperforms the nVidia GPU on most matrices. According to the performance behaviors of the merge algorithms in the Figure 4, we can see that the scratchpad memory capacity is very important for the GPU SpGEMM performance.



(a)



(b)

Figure 7. Relative performance comparison.

VIII. Conclusion

In this paper we demonstrated an efficient SpGEMM algorithm on the GPUs for solving the three challenging problems in the SpGEMM. In the experiments on a benchmark suite composed of 23 matrices with diverse sparsity structures, our SpGEMM algorithm delivered excellent absolute and relative performance as well as space efficiency over the previous GPU SpGEMM methods. Moreover, on average, our approach obtained up to twice the performance of the start-of-the-art CPU SpGEMM method.

Acknowledgment

The authors would like to thank Jianbin Fang at the Delft University of Technology for supplying access to the machine with the nVidia GeForce GTX Titan GPU. The authors also thank the anonymous reviewers for their insightful comments on this paper.

References

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

[2] J. Gilbert, S. Reinhardt, and V. Shah, "High-performance graph algorithms from parallel sparse matrices," in *Applied Parallel Computing. State of the Art in Scientific Computing*, ser. Lecture Notes in Computer Science, B. Kågström, E. Elmroth, J. Dongarra, and J. Waśniewski, Eds. Springer Berlin Heidelberg, 2007, vol. 4699, pp. 260–269.

[3] T. M. Chan, "More algorithms for all-pairs shortest paths in weighted graphs," in *Proceedings of the Thirty-ninth Annual ACM Symposium on Theory of Computing*, ser. STOC '07. New York, NY, USA: ACM, 2007, pp. 590–598.

[4] H. Kaplan, M. Sharir, and E. Verbin, "Colored intersection searching via sparse rectangular matrix multiplication," in *Proceedings of the Twenty-second Annual Symposium on Computational Geometry*, ser. SCG '06. New York, NY, USA: ACM, 2006, pp. 52–60.

[5] V. Vassilevska, R. Williams, and R. Yuster, "Finding heaviest h-subgraphs in real weighted graphs, with applications," *ACM Trans. Algorithms*, vol. 6, no. 3, pp. 44:1–44:23, jul 2010.

[6] 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*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 18:1–18:11.

[7] G. Ortega, F. Vázquez, I. García, and E. M. Garzón, "Fast-spmm: An efficient library for sparse matrix matrix product on gpus," *The Computer Journal*, 2013.

[8] F. Vazquez, G. Ortega, J. Fernandez, I. Garcia, and E. Garzon, "Fast sparse matrix matrix product based on ellr-t and gpu computing," in *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, 2012, pp. 669–674.

[9] J. Demouth, "Sparse matrix-matrix multiplication on the gpu," NVIDIA, Tech. Rep., 2012.

[10] NVIDIA. Nvidia cusparse library. [Online]. Available: https://developer.nvidia.com/cuSPARSE

[11] S. Dalton and N. Bell. Cusp : A c++ templated sparse matrix library. [Online]. Available: http://cusplibrary.github.com

[12] S. Dalton, N. Bell, and L. Olson, "Optimizing sparse matrix-matrix multiplication for the gpu," University of Illinois, Tech. Rep., 2013.

[13] Intel. Intel math kernel library. [Online]. Available: http://software.intel.com/en-us/intel-mkl

[14] J. Gilbert, C. Moler, and R. Schreiber, "Sparse matrices in matlab: Design and implementation," *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 1, pp. 333–356, 1992.

[15] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Trans. Math. Softw.*, vol. 4, no. 3, pp. 250–269, sep 1978.

[16] K. Matam, S. Indarapu, and K. Kothapalli, "Sparse matrix-matrix multiplication on modern architectures," in *High Performance Computing (HiPC), 2012 19th International Conference on*, 2012, pp. 1–10.

[17] P. Sulatycke and K. Ghose, "Caching-efficient multithreaded fast multiplication of sparse matrices," in *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing 1998*, 1998, pp. 117–123.

[18] R. Yuster and U. Zwick, "Fast sparse matrix multiplication," *ACM Trans. Algorithms*, vol. 1, no. 1, pp. 2–13, jul 2005.

[19] A. Buluç and J. Gilbert, "On the representation and multiplication of hypersparse matrices," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, 2008, pp. 1–11.

[20] J. Fang, A. Varbanescu, and H. Sips, "A comprehensive performance comparison of cuda and opencl," in *Parallel Processing (ICPP), 2011 International Conference on*, 2011, pp. 216–225.

[21] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Supercomputing, 2007. SC '07. Proceedings of the 2007 ACM/IEEE Conference on*, 2007, pp. 1–12.

[22] A. Buluç and J. R. Gilbert, "Challenges and advances in parallel sparse matrix-matrix multiplication," in *Proceedings of the 2008 37th International Conference on Parallel Processing*, ser. ICPP '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 503–510.

[23] E. Saule, K. Kaya, and Ü. V. Çatalyürek, "Performance evaluation of sparse matrix multiplication kernels on intel xeon phi," in *Proc of the 10th Int'l Conf. on Parallel Processing and Applied Mathematics (PPAM)*, sep 2013.

[24] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, dec 2011.

[25] A. Buluç, S. Williams, L. Oliker, and J. Demmel, "Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, 2011, pp. 721–733.

[26] R. R. Amossen, A. Campagna, and R. Pagh, "Better size estimation for sparse matrix products," in *Proceedings of the 13th International Conference on Approximation, and 14 the International Conference on Randomization, and Combinatorial Optimization: Algorithms and Techniques*, ser. APPROX/RANDOM'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 406–419.

[27] E. Cohen, "On optimizing multiplications of sparse matrices," in *Integer Programming and Combinatorial Optimization*, ser. Lecture Notes in Computer Science, W. Cunningham, S. McCormick, and M. Queyranne, Eds. Springer Berlin Heidelberg, 1996, vol. 1084, pp. 219–233.

[28] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey, "Sort vs. hash revisited: Fast join implementation on modern multi-core cpus," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1378–1389, aug 2009.

[29] G. Ballard, A. Buluç, J. Demmel, L. Grigori, B. Lipshitz, O. Schwartz, and S. Toledo, "Communication optimal parallel multiplication of sparse random matrices," in *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '13. New York, NY, USA: ACM, 2013, pp. 222–231.

[30] A. Buluç and J. Gilbert, "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C170–C191, 2012.

[31] J. R. Gilbert, W. W. Pugh, and T. Shpeisman, "Ordered sparse accumulator and its use in efficient sparse matrix computation," United States Patent US 5983230 A, nov 1999.

[32] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore gpus," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, 2009, pp. 1–10.

[33] O. Green, R. McColl, and D. A. Bader, "Gpu merge path: A gpu merging algorithm," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: ACM, 2012, pp. 331–340.

[34] P. Kipfer and R. Westermann, "Improved gpu sorting," in *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, M. Pharr, Ed. Addison-Wesley, mar 2005, ch. 46, pp. 733–746.

[35] H. Peters and O. Schulz-Hildebrandt, "Comparison-based in-place sorting with cuda," in *GPU Computing Gems Jade Edition*, W.-M. Hwu, Ed. Morgan Kaufmann, Oct 2011, ch. 8, pp. 89–96.

[36] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger, "A novel sorting algorithm for many-core architectures based on adaptive bitonic sort," in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, 2012, pp. 227–237.

[37] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani, "Aa-sort: A new parallel sorting algorithm for multi-core simd processors," in *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, 2007, pp. 189–198.

[38] A. Davidson, D. Tarjan, M. Garland, and J. Owens, "Efficient parallel merge sort for fixed and variable length keys," in *Innovative Parallel Computing (InPar), 2012*, 2012, pp. 1–9.

[39] S. Baxter. Modern gpu library. [Online]. Available: http://www.moderngpu.com/

[40] M. R. B. Kristensen, S. A. F. Lund, T. Blum, K. Skovhede, and B. Vinter, "Bohrium: Unmodified numpy code on cpu, gpu, and cluster," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2013 SC Companion:*, 2013.