Regularizing Irregularity: Bitmap-based and Portable Sparse Matrix Multiplication for Graph Data on GPUs

Jianting Zhang
Dept. of Computer Science
City College of New York
New York City, NY, 10031
jzhang@cs.ccny.cuny.edu

Le Gruenwald
School of Computer Science
University of Oklahoma
Norman, OK 73071
ggruenwald@ou.edu

ABSTRACT

Graphs can be naturally represented as sparse matrices. The relationship between graph algorithms and linear algebra algorithms is well understood and many graph problems can be abstracted as Sparse General Matrix-Matrix Multiplication (SpGEMM) operations. While quite some matrix storage formats, including bitmap-based ones, have been proposed for sparse matrices, they are mostly evaluated on the simpler Sparse Matrix-Vector Multiplication (SpMV) problems. In this study, we have developed data parallel algorithms to pair up bitmap-indexed sparse matrix blocks for SpGEMM using data parallel primitives for portability. Experiments on the WebBase-1M dataset with more than one million rows and columns and three million non-zero values have shown that our technique on squaring the large-scale sparse matrix using a 2013 GTX Titan GPU can complete in about 300 milliseconds. The runtime is 2.4X faster than CUSP and 3.5X faster than bhSPARSE, the two leading open source SpGEMM packages. Furthermore, our bitmap-indexed sparse matrix blocks can be efficiently converted to regular small dense matrices and subsequently utilize new hardware accelerations, such as tensor cores inside Nvidia Volta GPUs and Google Tensor Processing Units (TPUs), for more efficient implementations.

Categories and Subject Descriptors

Keywords

bitmap-indexing, Sparse Matrix Multiplication, graph operations, data parallel design, GPU

1. INTRODUCTION

Graph data are becoming increasingly important in real world applications, ranging from social networks to physics-based numeric simulations. Graphs can be naturally represented as sparse matrices and the relationship between graph algorithms and linear algebra algorithms is well understood [1] [2] [3]. Graph libraries targeting at new hardware (GPU in particular), such as nvGraph [4] from Nvidia and BelRed [5] from AMD, provide APIs for sparse matrix operations. There are increasing interests in utilizing sparse matrix operations for graph applications [6] [7] [8] [9].

Dozens of sparse matrix storage formats have been proposed for sparse matrices [10]. Previous studies have shown that storage formats have a significant impact on the performance of sparse matrix operations, including Sparse Matrix-Vector Multiplication (SpMV) and Sparse General Matrix-Matrix Multiplication (SpGEMM) on both multi-core CPUs and GPUs. While those storage formats are applicable to both SpMV and SpGEMM in principle, they are mostly tested for SpMV and very few have been tested for SpGEMM, especially on new hardware. To our knowledge, commercial and open SpGEMM libraries mostly support traditional storage formats, such as Coordinate list (COO) and Compressed Sparse Row (CSR). An interesting question to ask is: are there alternative sparse matrix formats that can be used to efficiently support SpGEMM for graph data?

.

Bitmap indexing, as a classic data structure, has been used extensively for various applications on different hardware architectures. Bitmap-based sparse matrix storage format has been proposed for SpMV on CPUs [11]. Different from CSR that stores row numbers and COO that stores both row numbers and column numbers explicitly, each bit in a bitmap is used to indicate the existence or absence of a non-zero value in the corresponding location, which can potentially save memory footprint significantly. Bitmap has also been used as an auxiliary data structure in some sophisticated spare matrix formats, such as CSR5 [12] and LSRB-CSR [13]. However, they have been only tested for SpMV on CPUs [11] and GPUs [12] [13] and their performance for SpGEMM is largely unknown.

Compared with operations on dense matrices, including Matrix-Vector Multiplication (MV) and Matrix-Matrix Multiplication (MM), their sparse counterparts (SpMV and SpGEMM, respectively) have much more irregularities on data accesses due to sparsity and hence are more technically challenging to achieve high efficiency. Compared with SpMV where the sparsity is on the matrix side whereas the vector side is still dense, SpGEMM involves two input matrices and needs to handle sparsity on both sides, in addition to the obvious fact that a large two-dimensional matrix is more complex than a vector (or multiple vectors). From a database perspective, SpGEMM can be considered as a special join, namely Dot-Product Join [14], where each pair may involve O(n) data elements and O(n) operations which are much more complex than relational joins, where only O(1) elements and O(1) operations are required for a pair.

The hardware landscape has been under rapid changes in the past few years, including multi-core CPUs, many-core GPUs and Intel Many Integrated Core accelerators (MICs) [15]. While the achievable floating point computing power has been increasing significantly for dense matrix operations (MV and MM) due to their inherent parallelisms that match with Single Instruction Multiple Data (SIMD) computing power well, it is much harder to utilize SIMD for sparse matrix operations (SpMV and SpGEMM) due to irregular data accesses. With the ever-increasing computing power demands from the deep learning community, more specialized hardware for matrix multiplications, such as ASIC-based Tensor Processing Units (TPU), start to become available inside Nvidia Volta-based GPUs [16] and Google Clouds [17]. However, these hardware accelerators, while extremely fast and powerful, are designed to support small blocked dense matrix only (e.g., 16*16), which renders most of the current sparse matrix storage formats, including blocked/segmented ones (on a single row or multiple rows) inappropriate for such hardware accelerators.

In this study, we aim at examining the suitability of bitmap indexing for SpGEMM in the context of graph data applications on GPUs and present our bmSPARSE technique as both a sparse matrix format and a SpGEMM algorithm. Different from previous studies that focus on either supporting SpMV only or exploiting sophisticated optimizations to push the performance limit of SpGEMM on new hardware, we aim at re-targeting the classic bitmap-indexing technique in relational data management for spare matrix/graph data and develop data parallel algorithms for SpGEMM to support graph applications. Our technique decomposes a large-scale sparse matrix into regularly shaped 2D blocks (8*8 sub-matrices in particular). Non-empty blocks are then indexed by their block identifiers (ids) for binary-search based paring in the task list generation stage in SpGEMM. Within a non-empty block, the bitmap of non-zero values is represented as a *uint64_t* value. By utilizing population counting intrinsic functions that are available on modern hardware (such as *__popc* and *__popcll* on Nvidia GPUs and *__builtin_popcount* and*__builtin_popcountll* on modern CPUs supporting GCC), it is efficient to compute the number of non-zero elements in a block and compute the positions of non-zero elements within and across blocks for easy data accesses. The tradeoff between real-time computation and storage overheads (explicitly storing row/column information) is well justified on modern hardware. Blocks in our technique play a bridging role in regularizing irregularity in SpGEMM: (1) The distributions of blocks are considered sparse and they are used as the basic units for indexing and pairing for task list generation to determine the structures of output matrices; (2) Non-zero values within a block are indexed with bitmaps which can be used to transform the non-zero values between sparse format and dense format in real time in both directions efficiently on fast shared memory or caches to compute the output matrices in the bitmap format.

We demonstrate that our bmSPARSE technique is effective not only in reducing the memory footprint of large-scale sparse matrices but also in supporting high-performance SpGEMM which has numerous graph applications. Applying to the public WebBase-1M dataset [18] with more than one million rows and columns (graph nodes) and three million non-zero values (graph edges), squaring the matrix takes approximately 300 milliseconds on a 2013 Nvidia GTX GPU, which is 2.4X times faster than CUSP [19] and 3.5X faster than bhSPARSE [20], the two leading open source SpGEMM packages, on the same GPU device. Perhaps more importantly, our work has shown that, bitmap, as both a sparse matrix storage format and an indexing technique, and together with data parallel designs, can effectively regularize the significant irregular data accesses on massively data parallel hardware with much less technical complexity. The resulting semi-regular data access and blocked sub-matrix computation patterns is also friendly to future hardware accelerations such as TPUs, and thus our technique is likely to benefit from such hardware accelerations.

The rest of the paper is arranged as follows. Section 2 introduces the background and discusses the related work. Section 3 presents our bitmap-based storage format and the bmSPARSE technique for SpGEMM. Section 4 is the experiments using the WebBase dataset. Finally, Section 5 is the conclusion and future work.

2. BACKGROUND AND RELATED WORK

Graph and Sparse Matrix are strongly related, not only very often they share the same storage format (e.g., COO and CSR) [4] [5], but also many graph algorithms can be expressed as linear algebra operations [1] [2] [3] which could be easier to understand at a high level, especially for those having a numeric computing background. Practically, it has been shown that, Breadth-First-Search (BFS) based on SpMV has achieved better performance on Intel Xeon CPUs and Intel Xeon Phi MICs than native implementations [3]. Park et al. implemented the popular *PageRank* algorithm using SpMV formulation on both Hadoop and Spark platforms and achieved better performance and/or scalability when compared with native leading graph libraries, including PEGASUS, GraphLab, GraphX and Giraph [21]. In a sense, research on sparse matrix operations can be an interesting bridge between the discrete and the numeric computing communities and synergize advantages of both sides for large-scale problems (e.g. GraphBLAS initiative [22] and ExaGraph project [23]). The bridging role is becoming increasingly important given that deep learning algorithms and systems are now ubiquitous and sparsity in deep learning algorithms are being actively explored and exploited for faster and more efficient training and inferencing [24].

To address sparsity in matrices, dozens of sparse matrix formats have been developed and we refer to [10] for systematic studies of design considerations and metrics on performance. While memory footprint alone is an important factor for storage format, performance measurements with respect to runtimes on various sparse matrix operations are also important in many applications. Unfortunately, most of existing performance studies on different storage formats are based on SpMV alone (see [25] for a review), which leaves their performance on SpGEMM largely unclear. Due to the complexity of SpGEMM, existing SpGEMM implementations are mostly based on popular storage formats (CSR in particular) and few efforts have been given to alternative storage formats for SpGEMM. Nevertheless, several storage formats originally developed for SpMV share certain similarities with our bmSPARSE technique and we will provide a brief review next before discussing the state-of-the-art works on SpGEMM. We note that, those techniques originally developed for SpMV could be extended to support SpGEMM and some of them can be incorporated into our bmSPARSE technique for further performance improvements. We leave the integration and comparison for our future work.

Mapped Blocked Row (MBR) [11] uses bitmaps to index non-zero values in blocks whose sizes can vary. In addition to *b_bmp* array for bitmaps blocks, it also has *row_start* and *col_idx* arrays to keep row and column information for blocks. The format is very similar to our bmSPARSE technique, except that we use row and column numbers directly at block level and combine them into a single long integer variable (e.g. *uint64_t*) as block identifier. We do not use position information (*row_start*) as they can be efficiently computed from the number of *1s* in bitmaps on the fly. We also need block identifies to match blocks in the two input sparse matrices in SpGEMM, which is not required in SpMV.

Derived from CSR, in Compressed Multi-Row Storage (CMRS) [26], rows are divided into strips. A *StripPtr* array is used to point to the beginning positions of non-zero values in strips that have multiple rows. The *RowInStrip* array records the row numbers within strips, which can be calculated as the remainders of row numbers divided by the numbers of rows in the strips. This format, compared with the classic CSR format that has only one level of granularity on rows, has two level of granularities on rows instead. CMRS is somewhat similar to delta encoding and can potentially has a lower memory footprint than CSR, in addition to be more GPU friendly in exploring fine-granular parallelisms.

The Local Segmented Reduction based CSR (LSRB-CSR) [13], in a way similar to its predecessor CSR5 [12], partitions non-zero values into blocks with each block having approximately the same number of non-zero values. Both CSR5 and LSRB-CSR use a bitmap to indicate whether a matrix element is the first element in a row, which is different from the bitmap in [11] and our technique. Compared with CSR5, LSRB-CSR is simpler, more memory efficient and performs better for SpMV according to [13]. LSRB-CSR is similar to our technique in the sense that certain position information is computed from the bitmap structure (which also contributes to its memory efficiency). However, LSRB-CSR divides row-major ordered none-zero values sequentially and the resulting blocks are essentially irregularly shaped chunks of rows. It can be seen that, CMRS, CSR5 and LSRB are designed to adopt a 1D (along rows) blocking strategy and to exploit fine-grained parallelisms on GPUs. While the strategy may be suitable for SpMV, it is unclear how to adapt them for SpGEMM, where 2D blocking (along both rows and columns) seems to be more suitable.

Among the SpMV techniques that adopt 2D blocking (but do not necessarily involve bitmap), Blocked CSR (BCSR) [27] might be the earliest proposal that divides a large sparse matrix into 2D blocks and then applies CSR for each block. The work has motived several efforts to develop efficient SpMV techniques on both CPUs and GPUs. The Cache-Oblivious Extension Quadtree (COEQT) technique proposed in [28] divides a sparse matrix into four quadrants recursively until the blocks corresponding to leaf nodes can fit into cache, which can be considered as a special 2D-blocking strategy. While the idea is interesting in the sense that derived quadrants are equivalent to blocks and bitmaps can be used to index non-zero values inside blocks, it seems that the technique primarily adopts COO format and the only difference is re-ordering non-zero values according a DFS traversal order of the resulting quadtree.

The Blocked Row-Column (BRC) [29] format partitions both rows and columns where the block size is parameterized by a predefined number of non-zero values in a block, instead of a sub-matrix space size. It can be seen that equal partition based on sub-matrix space size (e.g., BCSR and COEQT) may result in variable numbers of non-zero values among blocks while equal partition-based on non-zero values (e.g., BRC) may result in irregularly shaped blocks. To the best of our knowledge, there have been no efforts to adapt those 2D blocking formats to SpGEMM. Our bmSPARSE technique chooses equal partition based on sub-matrix space size as it is easier to pair equal and regularly shaped blocks in SpGEMM.

Most of the recent works on SpGEMM adopt CSR or COO formats and many of them are targeting at GPUs. The Expansion, Sorting, and Contraction (ESC) algorithm [30] generates T(i,j,k) pairs by matching column j in A_{ij} with row j in B_{jk} (based on $C_{i,k}$=sum(A_{ij}*B_{jk})) and then performs reduction using (i, k) as key to aggregate the intermediate values in T(i,j,k) to C_{ik}. While operations required by ESC has high parallelisms on GPUs, its performance is hurt by generating large volumes of intermediate data and several optimization efforts have been developed as documented in the CUSP open source library [19].

The bhSPARSE framework [20] adopts a four-stage strategy. The first stage estimates the number of non-zero elements in each row of the output matrix and the second stage puts the rows into different bins for load balancing purpose. Subsequently, the third stage computes the resulting matrix, generates non-zero values and outputs them to a temporal matrix. Finally, the last stage compacts the temporal matrix and generates the final output matrix. Similar to CUSP, the source code of bhSPARSE has been released [31]. Independent studies [32] [33] have confirmed that bhSPARSE is generally more efficient than CUSP, which can be partially contributed to the binning strategy and better load balancing.

The BalancedHash technique [32] also estimates non-zero values in the output matrix first, but it partitions non-zero values (instead of rows as in bhSPARSE) for load balancing. With additional optimizations including efficient hashing on GPU shared memory to aggregate intermediate results and atomic operations to fully utilize more recent GPU hardware support, it is reported that BalancedHash can achieve impressive speedups over bhSPARSE. The HybridSparse technique [33] combines the classic ESC technique [30] with a GPU-based Scatter-Vector approach that was originally developed for CPUs [34] and has demonstrated higher performance than both bhSPARSE and BalancedHash. Unfortunately, the source code of BalancedHash and HybridSparse is not available and direct comparisons are not possible.

Extensive optimizations to maximize the utilizations of GPU hardware are applied in CUSP, bhSPARSE, BalancedHash and HybridSparse. While the performance is significantly boosted, sophisticated algorithmic designs make their codebases difficult to understand, maintain and improve. An interesting work in [35] focuses on both performance and portability in SpGEMM as the work targets at heterogeneous platforms, including multicore CPUs, GPUs, Intel MICs and clusters, where the Kokkos framework [36] is used for portability. While currently we target at GPUs only, we would like to keep portability in mind from the very beginning. Our idea is to utilize parallel primitives [37] [38] as much as possible and only resort to custom GPU kernels when necessary. We argue that the behavior of parallel primitives, such as sort, scan, reduction, expand and compact, are well understood and well supported on multiple platforms with high portability. As detailed in Section 3, our technique utilizes the Thrust library [39] (which comes with CUDA SDK) to generate task lists and to compact intermediate outputs, which are portable among multiple hardware platforms. The kernels to estimate the output sizes (i.e., memory counting) and to perform the actual sub-matrix computation are very simple. In addition to using OpenCL for portability (similar to bhSPARSE), the implementations can be easily ported to other platforms using their native programming languages for higher efficiency. While portability and efficiency may conflict with each other, in Section 4, we show that our technique is able to outperform both CUSP and bhSPARSE significantly for the public WebBase dataset [18], which might indicate that our proposed technique is effective for graph data.

There are several aspects of sparse matrix operations that are relevant to our work. For efficient implementations of SpGEMM on GPUs, similar to the long-lasting debate between hashing and merging for relational joins, there are also works that adopt merging strategies on GPUs [40] [41]. Architecture-specific optimization techniques for SpGEMM (e.g., [42]), while may not necessarily be portable, can be used as a performance benchmark. Techniques developed for optimizing SpGEMM on multi-core CPUs [43] and traditional clusters [44] are valuable for developing portable SpGEMM techniques. The idea of designing a unified sparse matrix format [45] or automatically choosing the best sparse matrix format [46], and dynamic allocation and task scheduling for SpMV [47] [48] [49] seems to be applicable to SpGEMM as well, which is left for our future work.

3. THE PROPOSED BMSPARSE TECHNIQUE

For notation convenience, we term our technique as ***bmSPARSE*** where *bm* stands for bitmap. We first present our bitmap-based sparse matrix storage format and then we will introduce our SpGEMM algorithm based on the format, which includes three stages: the symbolic computing stage to generate a task list to associate the blocks of the two input matrices and the output matrix, the block multiplication stage to multiply input blocks and assemble the resulting output blocks, and the compaction stage to remove zero values in the resulting output matrix. The block multiplication stage follows a two-phase procedure with the first phase to determine the upper bounds of non-zero values in the output blocks for efficient memory allocation purpose and the second phase to actually write out block multiplication results.

3.1 bmSPARSE format for Sparse Matrix

As shown in Fig. 1, bmSPARSE partitions a sparse matrix into 8*8 blocks and each block is represented as a (*key*, *bmp*) pair. *Key* can be either a 32-bit integer for small (# of row/col less than 65536) and 64-bit integer for large sparse matrices. We use 64-bit key for generalizability, although there are possibilities for compression and memory saving. For *bmp*, naturally an 8*8 bitmap requires a 64-bit integer. Given that a (row, column) pair is represented as a 64-bit integer, it can be seen that, as long as there are more than one non-zero element in a block, the bitmap representation is memory-efficient and can be up to 32 times more efficient than using the classic COO format. This is because when a block is fully dense, COO requires 64*2 32-bit integers while bmSPARSE requires only two 64-bit integers. The non-zero values in a block are then re-ordered based on their row and column numbers within the block to make the one-to-one mapping between a 1-bit and a non-zero element.

Clearly, the number of non-zero elements in a block can be efficiently computed using hardware intrinsic functions on both CPUs and GPUs, which typically takes only one or a small number of machine cycles and is much more efficient than na ïve counting using a loop. An exclusive scan (prefix-sum) [37] can be performed on the derived numbers of non-zero elements in all blocks to compute the starting positions of non-zero elements in the blocks in parallel for both computation and data accesses. Compared with the CSR and COO formats that use two arrays for row and column numbers for non-zero elements, our technique uses only a single array for blocks and is much more memory efficient. While a more comprehensive analysis comparing with CSR and COO is left for future work, as discussed above, the memory efficiency can be significantly higher than COO. We will empirically compare memory footprints of COO, CSR and our bmSPARSE for the WebBase dataset in Section 4.

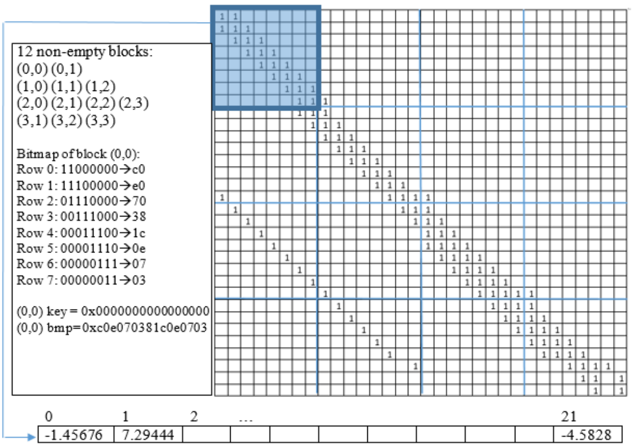

Fig. 1 Illustration of bmSPARSE Format

3.2 Generating Task List in SpGEMM

The purpose of generating the task list is to pair up blocks in the two input matrices, i.e., A and B, respectively, to determine the corresponding blocks in the output matrix, i.e., C, according to the general rule in SpGEMM, i.e., C_{ik}=sum(A_{ij}*B_{jk}). The stage (Fig. 2) is essential in regularizing irregularity in SpGEMM as a block in C can involve variable pairs of blocks in A and B, which reflects the irregular non-zero element distributions in the two input matrices. Once the task list in the form of vector of (*i*,*j*,*k*) triplets is generated, multiplying and adding 8*8 blocks according to the list is highly parallelizable and the irregular distributions of non-zero elements in the blocks are bounded.

Assuming the two input matrices, A and B, are given as a list of (*key*, *bmp*) pairs of blocks, the stage to generate the task list involves *key* only, which is a concatenation of (row, column) numbers. We use A' and B' to denote boolean matrices indicating the non-empty blocks of A and B, respectively. The first step in the task list generation stage is to count the number of non-empty blocks in each row of B', which can be easily implemented as a *reduce_by_key* parallel primitive by using the row numbers of B' as key. The second step is to match each column of the non-empty blocks in A' with the rows in B'. Clearly, an A' column may be matched with multiple B' rows whose numbers can be retrieved from the resulting vector in step 1 by using a *Gather* parallel primitive in Thrust, as shown in the middle-top part of Fig. 2.

The third step in the stage is to expand A' blocks, identified as (*i*, *j*) pairs, by the numbers of matched pairs to generate (*i*, *j*, *idx*) triples where *idx* belongs to 0..n_j-1 and n_j is the number of columns in the j^th row of B'. Note that *idx* is NOT column number of non-empty blocks in B' and thus k in (*i*, *j*, *k*) triple needs to be computed from *idx*. By using an auxiliary array keeping the starting positions of non-empty blocks in the rows of B' (vector *pos*), which can be computed in parallel using an exclusive scan parallel primitive as shown in the middle of Fig. 2 (labelled as "#3 Exclusive Scan"), the columns of B' blocks matching with A' blocks, i.e., *k*, can be retrieved by accessing the *key* vector of B. In our implementation, the (*i*, *j*, *k*) triple vector is computed by taking the matrix A *key* vector and the *idx* vector after expansion (the procedure labelled as "#3 Expand" in the middle of Fig. 2, see the next paragraph for details) as the inputs and apply a user-defined functor in a *transform* parallel primitive. Given a (*Akey*, *Bidx*) pair, *i* and *j* are the first and last 32 bits of *Akey* and *k* can be calculated as the last 32 bits of *wjk*, which is calculated as *B.key*[*pos[j]+idx*].

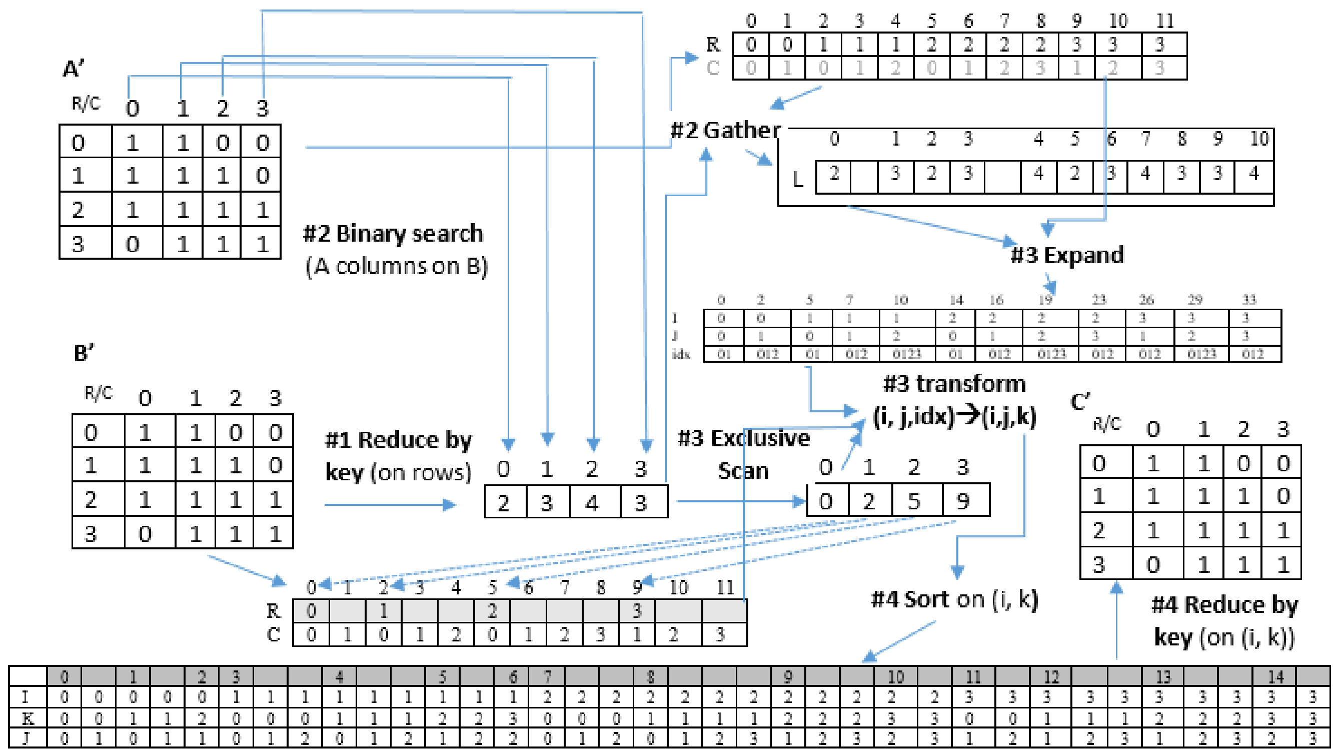
A'
B'
C'
#1 Reduce by key (on rows)
#2 Binary search (A columns on B)
#2 Gather
#3 Exclusive Scan
#3 Expand
#3 transform (i, j,idx)→(i,j,k)
#4 Sort on (i, k)
#4 Reduce by key (on (i, k))

Fig. 2 Task List Generation Using Data Parallel Primitives

While the behavior of the *expand* parallel primitive, i.e., expanding a vector (*E*) by making the numbers of copies of its elements specified in the second vector (*N*), is well-known [37], it is not directly supported in Thrust. Fortunately, it can be implemented by chaining four parallel primitives in Thrust (not shown in Fig. 2): *exclusive_scan* on *N* to compute the positions of the first copies of the elements to be expanded, *gather* to put *E* elements at these positions as the first copies and *maximum-inclusive scan* to create the desired number of copies of E elements, and finally *exclusive_scan_by_key* to generate the *idx* vector by using *E* elements as the keys on a sequence of 1s.

The fourth step in the task list generation stage is to generate keys for output blocks, i.e., C_{ik}. After the vector of (*i*, *j*, *k*) triples are generated, it is sorted based on (*i*, *k*) so that input block pairs, i.e., (*i*, *j*) and (*j*, *k*), that contribute to the same output block (*i*, *k*) are neighboring to each other and can be processed by a same thread block in the next stage (block multiplication, Section 3.3). By performing a *reduce_by_key* parallel primitive on (*i*, *k*), the number of non-empty blocks of the output matrix is computed, the key vector of output matrix *C* (by concatenating *i* and *k*) can be derived and the approximate layout of *C* is thus determined. To facilitate accessing blocks in *A* and *B*, this stage also searches block keys of (*i*, *j*) and (*j*, *k*) in the block key vectors of *A* and *B*, respectively, so that the bitmaps and the starting positions of non-zero values can be directly accessed using the positions.

3.3 Block Multiplication

In the block multiplication stage, a GPU thread block is assigned to compute a block of *C* based on $C_{i,k}$=sum(A_{ij}*B_{jk}). As a C_{ik} block may need to accumulate multiple intermediate blocks of A_{ij}*B_{jk}, a loop is applied. For an 8*8 block, 64 threads are assigned to a thread block and each thread is responsible for computing an element in the output C_{ik} block, i.e., thread *idx* is responsible for block element (*idx*/8, *idx*%8) in block C_{ik}. Although the floating point computation is simple, the more complex work for each thread is to retrieve non-zero elements from A_{ij} and B_{jk} blocks based on their bitmaps, and to generate the bitmap and its corresponding non-zero values for block C_{ik} in output matrix *C*.

As illustrated in Fig. 1, an 8*8 bitmap is represented as a 64-bit integer and its 1-bits are used to index non-zero values. To efficiently retrieve non-zero values from A_{ij} and B_{jk} for subsequent block multiplication, the positions of the non-zero values, i.e., non-zero value array offsets, need to be computed in parallel by all threads. While each thread can retrieve its bit from a bitmap and then perform a thread block level scan (prefix-sum), we found that it is more efficient to use population count intrinsic function that are available on modern GPUs. Since we use 64-bit integer for bitmap, *__popcll* is suitable. The offset can be simply computed as *L-__popcll*(*w*>>(*threadIdx.x*)) in CUDA where *w* is the bitmap of a block; and *L=__popcll(w)*, which can be set as a shared variable and computed by a single thread. Assuming that the non-zero values of A_{ij} and B_{jk} blocks are loaded into the 8*8 double array d_{ij} and d_{jk}, respectively, the output can be computed in parallel as following, where *t* is a looping variable 0..7:

out[*threadIdx.x*]+=*dij*[(*threadIdx.x*/8)*8+*t*]*djk[*t**8+(*threadIdx.x*%8)]

The remaining task is to compact the 8*8 *out* array and generate the bitmap and non-zero values for the output block. Here we use another intrinsic function *__ballot* to ask all threads in a thread warp to test whether *out*[*threadIdx.x*] is zero. Since a thread warp has 32 threads, the bitmap values can be set up by combining the voting results using *__ballot* in the two warps in a thread block (64 threads in total). To write non-zero values in the *out* array in accordance with the resulting bitmap, the offsets of the non-zero values are determined in the same way as computing the offsets for retrieving elements of the input blocks.

In our design, the d_{ij}, d_{jk} and *out* arrays have a size of 8*8 and are shared among the 64 threads in a thread block. As such, each thread block only needs 3*64*sizeof(double) bytes shared memory. Adding several other scalar shared variables at block level, the total required shared memory is less than two kilobytes. As such, different from many other GPU kernels for linear algebra applications, shared memory is not a limiting factor for performance in our design. However, using only 64 threads per block seems to make occupancy relatively low (50%). This is because occupancy is now limited by the maximum number of thread block per multi-processor on CUDA-enabled GPUs (16 for GTX Titan supporting Compute Capacity 3.5). Despite using shared memory to lower the overall data access overhead has been actively exploited, the kernel requires quite significant data accesses to GPU global memory which have long delays. As such, it is desirable to improve occupancy by using a larger number of threads per block so that more warps can be scheduled to amortize memory I/O overhead to achieve higher throughput. One idea is to use a larger thread block to process multiple C_{ik} blocks in parallel and we leave this for future work.

Another important issue to discuss is how to allocate memory for non-zero values in output matrix *C*. One solution is to utilize the fact that a block can have at most 64 non-zero values and pre-allocate a value array with a size of 64**num_block* and then compact unused memory space in the last stage (to be discussed next, Section 4.4). One of the disadvantages of the solution is large-memory footprint when the output matrix is sparse and the number of non-zero values is far less than 64. Another solution is to follow a classic two-phase approach, i.e., execute part of the kernel in the first phase to count the number of non-zero values and then perform a prefix sum to calculate the starting positions of non-zero values of the blocks before actual computation is done in the second phase. The second solution essentially trades off space with time by spending more time on computation in order to save unnecessary memory allocation.

The tradeoff can be well justified in many cases, including when output matrices are sparse. We have implemented both solutions and decided to adopt the two-phase approach. We note that the first solution does work for the WebBase dataset [18] on the GTX Titan GPU with 6GB memory and actually runs faster than the second solution as measured by end-to-end runtime. However, for the first solution, in addition to large memory footprint, we found that compacting a large array with large portions of unwanted data items (Section 3.4) is also expensive, which largely offsets the advantage of not requiring the memory counting stage for the first solution. To enable our implementation to handle larger graph datasets without running into the out-of-memory problem, we believe the second solution is more desirable. That being said, as reported in Section 4, the first phase takes about 1/3 of the runtime of block multiplication stage and is only slightly less expensive than the runtime for the task generation phase. As such, memory counting overhead cannot be neglected. We are working on optimizing the performance of the memory counting kernel to further improve the overall performance.

3.4 Compaction

Compaction is needed to remove zero values in the output after block multiplication. When each block is allocated 64 doubles to hold the maximum possible number of non-zero values (solution 1 above), there is a significant portion of zero values that need to be compacted. Even for the second phase solution 2, since the counting stage does not involve actual floating point computation on blocks, it is possible that multiplying and adding non-zero values result in zero values, which is not uncommon in matrix multiplication and they need to be compacted. Compaction is actually highly parallelizable and the *copy_if* parallel primitive in Thrust is designed for the right purpose. Compaction works better when there are few elements that need to be compacted as the cost for data movement is small in this case. As reported in Section 4, by adopting solution 2 in the block multiplication stage, the compaction stage incurs the least overhead during the end-to-end process.

4 EXPERIMENTS AND RESULTS

4.1 Data and Setup

We use the WebBase-1M dataset (denoted as WebBase) from University of Florida Sparse Matrix Collection [18] for experiments. The dataset has 1,000,005 rows (*n_row*) and 1,000,005 columns (*n_col*) and 3,105,536 non-zero values (*nnz*). It was first used by Samuel Williams et al [50] as a dataset representing web connectivity matrix for experiments on SpMV and later was used by several studies for SpGEMM, including CUSP, bhSPARSE, BalancedHash and HybridSparse on GPUs. As the source code of BalancedHash and HybridSparse is not available, we will compare our technique with CUSP and bhSPARSE only. In previous studies, bhSPARSE performs generally better than CUSP on many datasets, but performs worse than CUSP on the WebBase dataset. This makes it interesting to examine the suitability of alternative storage formats and techniques for SpGEMM on the web graph dataset.

Our experiments are performed on a 2013 Nvidia GTX Titan (Kepler) GPU with 2,688 CUDA cores and 6 GB GDDR5 memory. CUDA 8.0 and the Thrust library that comes with the SDK are used for GPU implementation. All programs are compiled with –O3 optimization flag. We have also implemented a CPU version using C++ Standard Template Library (STL) for verification and comparison purposes. The machine is a dual 8-core Ubuntu box with Intel E5-2650 processors running at 2.6 GHZ and 16 GB DDR3 memory but only a single core is used in the experiments for the CPU implementation.

4.2 Results

We next report the memory footprint and runtime of the proposed bmSPARSE technique, in comparison with CUSP and bhSPARSE. We assume the row numbers and column numbers are stored as 32-bit integers (4 bytes) and the non-zero values are stored as doubles (8 bytes). While CUSP supports multiple formats, COO, which is the original format of the WebBase dataset, is used for CUSP in the experiments. The memory footprint of COO can be calculated as (2*sizeof(int32)+sizeof(double))**nnz,* again *nnz* is the number of non-zero values of the input. Given that *nnz*=3,105,536 for the WebBase dataset, the memory footprint of CUSP is thus ~47.39 MB. The benchmark code of bhSPARSE uses the CSR format and the memory footprint can be computed as *n_row**sizeof(int32)+*nnz**sizoef(int32)+*nnz**sizeof(double) and thus the memory footprint is ~39.35MB. Since both the *key* and *bmp* of our bmSPARSE technique are represented as int64 datatype (8 bytes), the memory footprint can be calculated as *n_block**sizeof(int64)+*n_block**sizeof(int64)+*nnz**siozeof(doub le). As *n_block* is 550761for the dataset, the memory footprint is thus ~32.10 MB. Clearly, bmSparse has the smallest memory footprint among the three. When non-zero values are excluded and only indexing overhead is considered, as shown in Table 1, the overhead for bmSPARSE is only about 1/2 of CSR and 1/3 of COO.

Table 1 Overall Result Comparisons

	Memory (MB)		Runtime (ms)	Runtime-Speedup
	All	Index		
CUSP (COO)	47.39	23.69	760.495	2.4X
bhSPARSE (CSR)	39.35	15.66	1104.74	3.5X
bmSPARSE	32.10	8.40	306.302	1X

To better understand the performance of bmSPARSE on the WebBase dataset, Table 2 lists the runtimes of different modules in bmSPARSE. Transferring the input matrix in the COO format from CPU to GPU takes about 15ms and the conversion from COO to bitmap format takes about 78ms. While significant, they do not dominant the overall runtime. Note that these two parts are not included in the total runtime when comparing with CUSP and bhSPARSE, for consistency and fairness reasons.

Table 2 Runtime Breakdown

CPU to GPU Data Transfer Time (ms)	14.963
COO2BMP Time (ms)	78.108
1:Task Generation Time (ms)	67.517
2:Memory Counting Kernel Time (ms)	64.514
3:Block Multiplication Kernel Time (ms)	172.126
4: Compaction Time (ms)	12.982
Total GPU runtime (1+2+3+4) (ms)	317.139
Total CPU time (single core) (ms)	8290.960

Among the four runtimes that are included in the total runtime for comparison, it is clear that block multiplication kernel takes more than half of the total runtime time and is the bottleneck of the overall process. As such, our bmSPARSE technique is likely to benefit from future hardware (e.g. TPUs) acceleration.

The memory counting kernel is about 1/3 of the block multiplication kernel time (2+3). A closer look reveals that the memory counting kernel likely suffers from load unbalancing among threads. As our implementation assigns a thread to count the required memory for nonzero-values in a C_{ik} output block, a thread is required looking into a variable number of (A_{ij},B_{jk}) pairs and the number may vary significantly among threads in a warp and hence hardware could be underutilization. A possible improvement would be to sort the blocks of the output matrix based on the numbers of (A_{ij},B_{jk}) pairs that they associate for load balancing, which is left for our future work.

From Table 2 it can be seen that the GPU implementation is about 26.1X time faster than the serial CPU implementation, despite that the CPU implementation largely follows the data parallel design for GPUs. It seems that the container classes in C++ STL, including vector and map, while convenient to use, are less performant. We are in the process of porting kernels to Intel Thread Building Block (TBB) [37] based implementations on multi-core CPUs so that they can be integrated with existing parallel primitive based implementations that can be re-targeted for multi-core CPUs. The performance comparisons on multi-core CPUs are also left for future work.

While the performance is encouraging on the WebBase dataset that we currently target at, preliminary experiments on additional sparse matrix datasets from the University of Florida Sparse Matrix Collection have shown that our bmSPARSE technique does not always perform better than CUSP or bHSPARSE. This may indicate that our technique, as a very different alternative technique to existing ones, may be suitable to one or more specific categories of sparse matrices. The performance on the WebBase dataset suggests that graph data could be of these categories although further investigations are needed.

5. CONCLUSION AND FUTURE WORK

In this study, we have introduced our preliminary design and implementation of a bitmap-based SpGEMM technique namely bmSPARSE. Experiments on the WebBase-1M dataset with more than one million rows and columns and three million non-zero values have shown that SpGEMM on squaring the large-scale sparse using a 2013 GTX Titan GPU can complete in about 300 milliseconds. The runtime is 2.4X faster than CUSP and 3.5X faster than bhSPARSE, the two leading open source SpGEMM packages. The memory footprint is considerably lower than the popular COO and CSR formats as well.

For future work, in addition to the optimization plans discussed inline, we would like to perform more experiments on additional graph datasets and compare the performance of different components in both our bmSPARSE technique and others, including CUSP and bhSPARSE, to better understand the strengths and weaknesses of the techniques. We hope to identify deeper and more meaningful patterns in SpGEMM for different types of sparse matrices to intelligently choose different techniques for different types of datasets to improve the overall performance. Our decisions to embrace data parallel designs and use parallel primitives are largely driven by portability among existing and new hardware with the understanding of efficiency-portability tradeoffs. In addition to looking into the opportunity for TPU hardware accelerations, we also plan to port existing design and implementation to multi-core CPUs with Vector Processing Units (VPUs) to explore SIMD computing power on CPUs for graph operations.

ACKNOWLEDGEMENT

This work is supported through NSF Grants IIS-1302423 and IIS-1302439 and PSC-CUNY 60777-00 48.

REFERENCES

1. J. Kepner and J. Gilbert, Graph Algorithms in the Language of Linear Algebra, SIAM, 2011, 348 pages.
2. S. Che, B. M. Beckmann and S. K. Reinhardt, "Programming GPGPU Graph Applications with Linear Algebra Building Blocks," International Journal of Parallel Programming, vol. 45, no. 3, pp. 657-679, 2017.
3. M. Besta, F. Marending, E. Solomonik and T. Hoefler, "SlimSell: A Vectorizable Graph Representation for Breadth-First Search," in Proc. IEEE IPDPS'17, 2017.
4. Nvidia, "NVGRAPH Library User's Guide," Online, 2017.
5. S. Che, B. M. Beckmann and S. K. Reinhardt, "BelRed: Constructing GPGPU graph applications with software building blocks," in Proc. IEEE HPEC'14, 2014.
6. A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarath and P. Sadayappan, "Fast Sparse Matrix-Vector Multiplication on GPUs for Graph Applications," in Proc. SC'14, 2014.
7. X. Yang, S. Parthasarathy and P. Sadayappan, "Fast Sparse Matrix-vector Multiplication on GPUs: Implications for Graph Mining," Proc. VLDB Endow., vol. 4, no. 4, pp. 231--242, 2011.

8. R. Zayer, M. Steinberger and H.-P. Seidel, "A GPU-Adapted Structure for Unstructured Grids," Computer Graphics Forum, vol. 36, no. 2, pp. 1467-8659, 2017.
9. Y.-Y. Jo, S.-W. Kim and D.-H. Bae, "Efficient Sparse Matrix Multiplication on GPU for Large Social Network Analysis," in Proc. ACM CIKM'15, 2015.
10. D. Langr and P. Tvrd ík, "Evaluation Criteria for Sparse Matrix Storage Formats," IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 2, pp. 428-440, 2016.
11. R. Kannan, "Efficient sparse matrix multiple-vector multiplication using a bitmapped format," in Proc. IEEE HiPC, 2013.
12. W. Liu and B. Vinter, "CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication," in Proc. SC'15, 2015.
13. L. Liu, M. Liu, C. Wang and J. Wang, "LSRB-CSR: A Low Overhead Storage Format for SpMV on the GPU Systems," in Proc. IEEE ICPADS'15, 2015.
14. B. Qin and F. Rusu, "Dot-Product Join: An Array-Relation Join Operator for Big Model Analytics," arXiv:1602.08845, 2017
15. J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach (5th Ed.), Morgan Kaufmann, 2011.
16. Nvidia, "Inside Volta: The World's Most Advanced Data Center GPU," 2017.
17. N. P. Jouppi, C. Young, N. Patil and D. Patterson, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in Proc. ISCA'17, 2017.
18. University of Florida, "Sparse Matrix Collection," [Online]. https://www.cise.ufl.edu/research/sparse/matrices/Williams/webbase-1M.html.
19. S. Dalton, N. Bell, L. Olson and M. Garland. [Online]. Available: http://cusplibrary.github.io/.
20. W. Liu and B. Vinter, "A Framework for General Sparse Matrix-matrix Multiplication on GPUs and Heterogeneous Processors," J. Parallel Distrib. Comput., pp. 47--61, 2015.
21. B. Park, H.-M. Park, M. Yoon and U. Kang, "PMV: Pre-partitioned Generalized Matrix-Vector Multiplication," arXiv:1709.09099v1, 2017.
22. GraphBLAS. [Online]. Available: http://graphblas.org/.
23. ExaGraph. [Online]. Available: http://www.pnnl.gov/science/highlights/highlight.asp?id=4558.
24. H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang and W. J. Dally, "Exploring the Granularity of Sparsity in Convolutional Neural Networks," in IEEE CVPRW'17, 2017.
25. S. Filippone, V. Cardellini, D. Barbieri and A. Fanfarillo, "Sparse Matrix-Vector Multiplication on GPGPUs," ACM Trans. Math. Softw., vol. 43, no. 4, pp. 30:1--30:49, 2017.
26. Z. Koza, M. Matyka, S. Szkoda and Ł. Mirosław, "Compressed Multirow Storage Format for Sparse Matrices on Graphics Processing Units," SIAM J. Sci. Comput., vol. 36, no. 2, pp. 219-239.
27. E.-J. Im, K. Yelick and R. Vuduc, "Sparsity: Optimization Framework for Sparse Matrix Kernels," Int. J. High Perform. Comput. Appl., vol. 18, no. 1, pp. 135--158, 2014.
28. J. Zhang, J. Wan, F. Li, J. Mao, L. Zhuang, J. Yuan, E. Liu and Z. Yu, "Efficient sparse matrix–vector multiplication using cache oblivious extension quadtree storage format," Future Generation Computer Systems, vol. 54, pp. 490-500, 206.
29. A. Ashari, N. Sedaghati, J. Eisenlohr and P. Sadayappan, "An Efficient Two-dimensional Blocking Strategy for Sparse Matrix-vector Multiplication on GPUs," in Proc. ACM ICS'14, 2014.
30. S. Dalton, L. Olson and N. Bell, "Optimizing Sparse Matrix-Matrix Multiplication for the GPU," ACM Trans. Math. Softw., vol. 41, no. 4, pp. 25:1--25:20, 2015.
31. W. Liu and B. Vinte, "bhSPARSE," [Online]. Available: https://github.com/bhSPARSE/bhSPARSE.
32. P. N. Q. Anh, R. Fan and Y. Wen, "Balanced Hashing and Efficient GPU Sparse General Matrix-Matrix Multiplication," in Proc. ACM ICS'16, 2016.
33. R. Kunchum, A. Chaudhry, A. Sukumaran-Rajam, Q. Niu, I. Nisa and P. Sadayappan, "On Improving Performance of Sparse Matrix-matrix Multiplication on GPUs," in Proc. ICS'17, 2017.
34. F. G. Gustavson, "Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition," ACM Trans. Math. Softw., vol. 4, pp. 250-269, 1978.
35. M. Deveci, C. Trott and S. Rajamanickam, "Performance-portable sparse matrix-matrix multiplication for many-core architectures," in 2017 Proc. IEEE IPDPSW, 2017.
36. H. Edwards, C. R.Trott and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," Journal of Parallel and Distributed Computing, vol. 74, no. 12, pp. 3202-3216, 2014.
37. M. McCool, A. Robison and J. Reinders, Structured Parallel Programming: Patterns for Efficient Computation, Morgan Kaufmann, 2012.
38. D. B. Kirk and W.-m. W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, 2nd ed., Morgan Kaufmann, 2012.
39. Nvidia, "Thrust parallel algorithms library," [Online]. Available: https://thrust.github.io/.
40. S. Dalton, S. Baxter, D. Merrill, L. Olson and M. Garland, "Optimizing Sparse Matrix Operations on GPUs Using Merge Path," in Proc. IEEE IPDPS'15, 2015.
41. F. Gremse, A. Höfter, L. O. Schwen, F. Kiessling and U. Naumann, "GPU-Accelerated Sparse Matrix-Matrix Multiplication by Iterative Row Merging," SIAM Journal on Scientific Computing, vol. 37, no. 1, pp. 54-71, 2015.
42. Y. Nagasaka, A. Nukada and S. Matsuoka, "High-Performance and Memory-Saving Sparse General Matrix-Matrix Multiplication for NVIDIA Pascal GPU," in Proc. ICPP'17, 2017.
43. M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Park, M. J. Anderson, S. G. Vadlamudi, D. Das, S. G. P. O. Pirogov and P. Dubey, "Parallel Efficient Sparse Matrix-Matrix Multiplication on Multicore Platforms," in Proc. SC'15, 2015.
44. A. Buluç and J. R. Gilbert, "Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments," SIAM J. Sci. Comput., vol. 34, no. 4, pp. 170-191, 2012.
45. M. Kreutzer, G. Hager, G. Wellein, H. Fehske and A. R. Bishop, "A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units," SIAM J. Sci. Comput., vol. 36, no. 5, pp. 401-423, 2014.
46. N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy and P. Sadayappan, "Automatic Selection of Sparse Matrix Representation on GPUs," in Proc. ICS'15, 2015.
47. A. Derler, R. Zayer, H.-P. Seidel and M. Steinberger, "Dynamic Scheduling for Efficient Hierarchical Sparse Matrix Operations on the GPU," in Proc. ACM ICS'17, 2017.
48. J. King, T. Gilray, R. M. Kirby and M. Might, "Dynamic sparse-matrix allocation on GPUs," in Proc. ACM ICS'16, 2016.
49. M. Steinberger, R. Zayer and H.-P. Seidel, "Globally Homogeneous, Locally Adaptive Sparse Matrix-vector Multiplication on the GPU," in Proc. ACM ICS '17, 2017.
50. S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick and J. Demmel, "Optimization of Sparse Matrix-vector Multiplication on Emerging Multicore Platforms," in Proc. SC'07, 2007