Scaling Sparse Matrix Multiplication on CPU-GPU Nodes

Yang Xia
Computer Science and Engineering
The Ohio State University
xia.425@osu.edu

Peng Jiang
Computer Science Department
University of Iowa
peng-jiang@uiowa.edu

Gagan Agrawal
Computer and Cyber Sciences
Augusta University
gagrawal@augusta.edu

Rajiv Ramnath
Computer Science and Engineering
The Ohio State University
ramnath@cse.ohio-state.edu

***Abstract*—Multiplication of two sparse matrices (SpGEMM) is a popular kernel behind many numerical solvers, and also features in implementing many common graph algorithms. Though many recent research efforts have focused on implementing SpGEMM efficiently on a single GPU, none of the existing work has considered the case where the memory requirements exceed the size of GPU memory. Similarly, the use of the aggregate computing power of CPU and GPU has also not been addressed for those large matrices. In this paper, we present a framework for scaling SpGEMM computations for matrices that do not fit into GPU memory. We address how the computation and data can be partitioned across kernel executions on GPUs. An important emphasis in our work is overlapping data movement and computation. We achieve this by addressing many challenges, such as avoiding dynamic memory allocations, and re-scheduling data transfers with the computation of chunks. We extend our framework to make efficient use of both GPU and CPU, by developing an efficient work distribution strategy. Our evaluation on 9 large matrices shows that our out-of-core GPU implementation achieves 1.98-3.03X speedups over a state-of-the-art multi-core CPU implementation, our hybrid implementation further achieves speedups up to 3.74x, and that our design choices are directly contributing towards achieving this performance.**

I. Introduction

Sparse general matrix-matrix multiplication (SpGEMM) is one of the key kernels of preconditioners such as algebraic multigrid method [7] and some graph algorithms [22], [8], [13], [29], [35]. Given two sparse input matrices A and B, SpGEMM computes the sum of products for each element of the sparse matrix C as follows:

$$C_{ij} \;=\; \sum_{k} A_{ik} \times B_{kj}$$

where i and j are the row and column identifiers of the non-zero elements of matrices A and B, and k is the set of colliding indices. Due to the sparsity structure of both input matrices and the output matrix, achieving high performance for SpGEMM is very challenging. As a result, a number of recent efforts have been done to develop GPU implementations for SpGEMM [4], [10], [16], [24], [25].

One important area that has received no attention is creating an *out-of-core* GPU implementation, i.e., existing publications are limited to handling cases where the entire computation can be done with the available memory in (one) GPU. In both scientific computing and graph processing, i.e., the two common types of applications of SpGEMM, sparse matrices can be large. For example, even the most memory-efficient and state-of-the-art SpGEMM implementation [30] is not able to handle a number of matrices from the SuiteSparse Matrix Collection [11] on a GPU because of memory limitations. (SuiteSparse collection has matrices arising from both scientific computations and graph applications.) Typically, matrices capturing graphs can be large – for example, one survey [32] identified that graphs with more than 100 million vertices and more than 1 billion edges (and requiring more than 1 Terabytes for uncompressed storage) are quite common. The nodes in these graphs can represent entities ranging from humans, scientific artifacts, products, or other digital elements, and thus arise from a number of different use cases. Recently, *unified memory* are available and can be used to solve the issue of memory limits. It allows the applications to access the memory on the host side transparently, and load the data to GPU memory when there are page faults. Thus, this feature can significantly ease the programming. However, without the knowledge of the SpGEMM, the loaded memory pages may contain some data which are useless and waste the bandwidth. Besides, there are overheads with page faults. Thus, in this work, we proposed an out-of-core implementation to solve the issue.

Out-of-core GPU implementations have been developed for many applications – MapReduce based graph processing [34], ray casting (visualization) [14], LU solver (numerical computation) [26], deep learning [5], and others. However, SpGEMM out-of-core implementation involves significant challenges not normally associated with other applications. First, when two sparse matrices are multiplied, the output matrix typically has a much larger number of non-zero elements than the two input matrices combined. Take matrix *com-LiveJournal* in Table II as an example. Here, the number of non-zero elements of result matrix $C = A \times A$ is 70x more than the matrix A itself. Thus, storing partial results and transferring them to CPU chunk by chunk can be a significant challenge for SpGEMM – an issue typically not seen with other applications. For example, in a general framework proposed for out-of-core GPU implementations [20], the authors assume that the data not fitting in GPU memory is read-only. However, for SpGEMM, the larger challenge is output data.

This paper addresses a number of challenges associated with creating an out-of-core SpGEMM solver. The first issue is partitioning the problem space – we use a row panel of A and a column panel of B to output the final set of values for a chunk of C. Because transferring these chunks to CPU can be a very dominant cost, overlapping computation and transfer of data between GPU and CPU is crucial for performance. However, with current versions of CUDA, asynchronous execution requires that the code does not use dynamic memory (de)allocation, whereas underlying GPU implementations of SpGEMM use memory in complex ways to make it more efficient. Thus, we avoid dynamic allocations with memory pre-allocations. Besides, there are a number of data transfers in the same direction within the procedure of the computation. However, GPU only supports one data transfer in one direction at one time, which makes the computations wait for data transfers and lose concurrency. We address this issue through careful scheduling of data transfers. Also, through a reordering of the output chunks on the basis of the number of floating-point operations (flops), we make the overhead of data transfers completely hide the overhead of computations.

Further, we extend our framework to utilize both GPU and CPU computation resources to further improve performance. The design choice we made is to assign chunks that involve more flops to the GPU and the others to the CPU. Besides, we determine the amount of workload of GPU/CPU based on a static ratio of flops across all matrices.

We evaluate our work on a set of matrices, which are selected from the SuiteSparse Matrix Collection [11]. Our experimental results demonstrate that our GPU implementation can achieve high performance: compared with a state-of-the-art multi-core CPU implementation, our implementation performs 1.98-3.03X faster. The hybrid implementation further improves performance by 3.74X. We also show that our asynchronous implementation can effectively improve the performance of our out-of-core GPU implementation through overlapping data transfers with the real SpGEMM computations. Finally, we show that our workload assignment for the GPU and the CPU is effective: with reordering, we achieve speedups over the default implementation; with a fixed ratio of flops, we almost always achieve the best results.

II. Background

This section provides background for our work. We first introduce a compressed sparse row (CSR) representation, which is the most commonly used representation for sparse matrices. Then, we briefly review some research works on the GPU execution of SpGEMM computation.

Algorithm 1 Sequential implementation of Gustavson's Row-Row SpGEMM algorithm.

```
1:  ▷ Initialize all elements in matrix C as zeroes
2:  for $A_{i*}$ in matrix $A$ do
3:      for $A_{ik}$ in row $A_{i*}$ do
4:          for $B_{kj}$ in row $B_{k*}$ do
5:              $value = A_{ik} \times B_{kj}$
6:              if $C_{ij} \notin C_{i*}$ then
7:                  $insert(C_{ij}, C_{i*})$
8:                  $C_{ij} \leftarrow value$
9:              else
10:                 $C_{ij} = C_{ij} + value$
11:             end if
12:         end for
13:     end for
14: end for
```

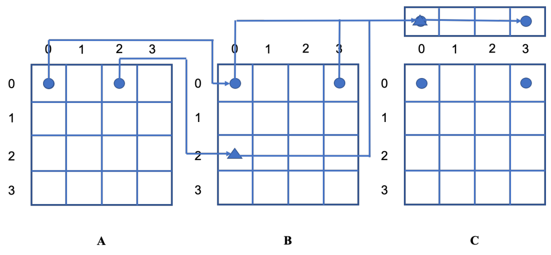
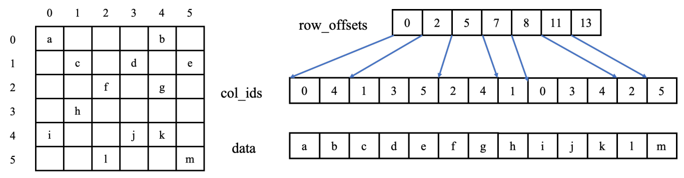


Fig. 2: A simple example of matrix multiplication. The non-zero elements are represented by dots and triangles.



(a) A sparse matrix.

(b) CSR format of the matrix.

Fig. 1: Compressed sparse row representation of a sparse matrix. (Example is taken from [21])

A. Compressed Sparse Row Representation

In this work, we adopt the compressed sparse row (CSR) format, which is widely used for storing and processing sparse matrices, including in recent work on SpGEMM[28], [30]. As shown in Figure 1, three arrays are used: *row_offsets*, *col_ids* , and *data*. The array *col_ids* stores the column indices of the non-zero elements in the sparse matrix row by row. In this work, the column ids are sorted for each row. The corresponding values of these non-zero elements are stored in the array *data*. *row_offset* indicates the beginning index of each row in the matrix in the array *col_ids* and *data*. For example, the value of the third element of the array *row_offsets* is 5, which means that for the third row, the starting column id is *col_ids[5]* and the corresponding value is *data[5]*.

B. Implementations of Sparse General Matrix-Matrix Multiplication (on GPUs)

For a matrix A, let A_{i*}, and A_{*i} denote the ith row and the ith column of matrix A respectively. $I_i(A)$ denotes the indices of the non-zero elements in the ith row of matrix A. One approach for SpGEMM is the *row-column* formulation. Here, to get C_{ij}, we multiply A_{i*} with B_{*j}, i.e., $C_{ij} = A_{i*} \times B_{*j}$. In the *Row-Row Formulation*, to compute C_{i*}, we multiply each non-zero element in A_{i*} with the corresponding row in the matrix B. Then, we add the corresponding scaled B rows to get the C_{i*}. Thus, $C_{i*} = \sum_{j \in I_i(A)} A_{ij} \times B_{j*}$. Note that *column-row*, and *column-column* approaches are also feasible.

Many recent research efforts on optimizing SpGEMM on GPUs [28], [30], [23], [31], [16], [17] generally follow Gustavson's algorithm [19], which is shown as Algorithm 1. Gustavson's algorithm follows the *row-row Formulation*: For each non-zero element of the matrix A, we read the corresponding row of the matrix B using column identifier. This is followed by accumulating non-zero elements in the output matrix C as shown in lines 5 to 9.

Unlike dense matrix multiplication, a key challenge of SpGEMM computations is that the output of the SpGEMM computation is still a sparse matrix, which is also stored in CSR representation. Since each element of the output matrix is computed as a result of operations on a sparse row and a sparse column, it is not trivial to determine which element of the output matrix will be non-zero. Take Figure 2 as an example, the $0th$ and $2nd$ elements of the row 0 of the matrix A are non-zeroes. Thus, we read row 0 and row 2 of the matrix B. As shown in the figure, non-zero elements in these two rows have only two distinct column ids, which are 0 and 3. As a result, the first row of the result matrix contains only two non-zero elements, which are C_{00} and C_{03}. Thus, memory allocation for the output matrix is a hard problem. Note that in Algorithm 1, we assumed that a new value can be inserted in a row of C – however, use of a dynamic structure like a linked list that can support such insertions (and membership lookup) will clearly be very expensive.

Similar to many other implementations, we use a *two-phase* strategy to solve this issue. The first phase is the *symbolic phase*, where they first count the number of non-zero elements of each row in the output matrix. The second phase is called *numeric phase*, which starts with the knowledge of the number of non-zero elements in the output matrix, and thus, space allocation is now feasible. After allocating space, we compute the actual values in the output matrix.

We also note that the algorithm is well-suited for parallel execution, with the main issue being how to combine the intermediate products to non-zero elements of the output matrix. Since the output matrix is also sparse, it is hard to efficiently combine intermediate products. For example, in Figure 2, row 0 and row 2 both have an element whose column id is 0. Thus, we need to combine these two elements to a single value in the results matrix C. Building on prior work in this area [28], [31], we use two accumulation methods: *hashmap* and *dense accumulation*. The hashmap method first allocates memory space based on an upper bound estimation of the size of the hash table. It then inserts values using the column ids of the intermediate results as the key. Then, it sorts the values of each row of the result matrix according to their column ids to obtain the final result matrix compressed with a sparse format [28]. The dense accumulation method [31], [30] stores and accumulates intermediate values in a dense array for each row. In contrast to the hashing solution, this method directly uses the column ids to access the array. As expected, this dense accumulation method can achieve high performance for matrices with relatively dense output rows, while low density leads to low memory utilization of the dense array.

Algorithm 2 Overall procedure of out-of-core iterative SpGEMM computation.

```
1: partition_rows(A, num_row_panels)
2: partition_cols(B, num_col_panels)
3: for $row = 0; row < num\_row\_panels; row++$ do
4:     for $col = 0; col < num\_col\_panels; col++$ do
5:         $C[row][col]$ = SpGEMM($A[row]$, $B[col]$)
6:         Transfer chunk $C[row][col]$ back to host side
7:     end for
8: end for
```

Algorithm 3 Overall procedure of out-of-core iterative SpGEMM computation.

```
1: ▷ $num\_row\_panels$ refers to the number of row panels in A
2: ▷ $num\_col\_panels$ refers to the number of column panels in B
3: partition_rows(A, num_row_panels)
4: partition_cols(B, num_col_panels)
5: for $row = 0; row < num\_row\_panels; row++$ do
6:     for $col = 0; col < num\_col\_panels; col++$ do
7:         ▷ $A[row]$ is a row panel
8:         ▷ $B[col]$ is a column panel
9:         $C[row][col]$ = SpGEMM($A[row]$, $B[col]$)
10:        Transfer chunk $C[row][col]$ back to host side
11:    end for
12: end for
```

III. Design

In this section, we first introduce the overall framework for executing SpGEMM in an out-of-core fashion on a GPU. Then, we describe the implementation method we used for in-core SpGEMM computation for clarity. We further extend our framework to support concurrent executions on both CPU and GPU. Finally, we briefly introduce our partitioning implementation.

A. Overall Framework of Out-of-Core GPU Implementation

Algorithm 3 briefly illustrates the overall procedure of our out-of-core GPU implementation. As shown in line 3-4, we partition matrix A into *row panels* while for matrix B, we partition it into *column panels*, which are simply sets of consecutive rows/columns. Given a row (column) panel identifier i, $A[i]$ ($B[i]$) denotes the row panel of A (column panel of B). Similarly, $C[i][j]$ represents the chunk of the result matrix C that is output by processing $A[i]$ and $B[j]$. For each row panel or column panel, we store data using CSR format on device memory because it is the mostly commonly used data format. Then, we iterate over each row panel and column panel to get a corresponding chunk of matrix C using an in-core GPU implementation. Finally, we transfer the result chunk back to the CPU.

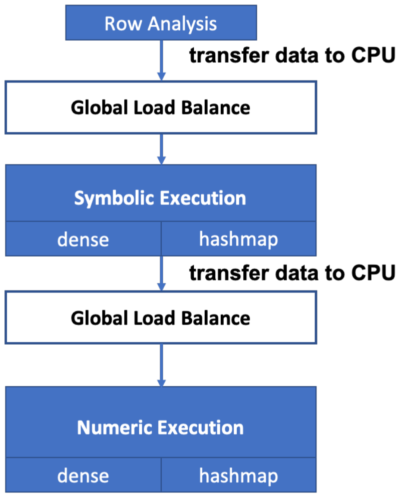


Fig. 3: Overall framework of out-of-core GPU SpGEMM (Blue box denotes GPU side and white box denotes CPU side.

As shown in the algorithm, our framework uses the *Row-Column Formulation*. With our ultimate goal of continuing to scale SpGEMM computations to arbitrarily large matrices, we partition both matrix A and matrix B into smaller chunks (to handle cases where either matrix of them is too large to reside in device memory). Thus, both the *Row-Row Formulation* and the *Column-Column Formulation* are not suitable in our case. For example, in the *Row-Row Formulation*, there are no limits on the number of non-zero elements of a single row in matrix A. As a result, it is not feasible to partition matrix B under the *Row-Row Formulation*. Also, we do not choose *Column-Row Formulation* because it requires accumulation operations for obtaining the output matrix from the partial results. In contrast, in the *Row-Column Formulation*, final values within a chunk of the output matrix C are independent.

B. In-core GPU Implementation

As shown in Algorithm 3, after partitioning matrix A into row panels and matrix B into column panels, a function is invoked to multiply these row and column panels to produce a chunk of output C. Our GPU implementation for this builds on the in-core effort in spECK [30], which in turn specializes the two-phase strategy mentioned in the last section. Considering the challenge of not just memory allocation but also partitioning between processing cores, such GPU implementations consist of three major stages (shown in Figure 3). First, we launch a kernel to do *row analysis* of input matrices, i.e., computing the number of floating-point operations associated with each row. Then, we transfer this collected information from device memory to the host memory. Based on the information captured, we assign rows of the matrix A to different groups to achieve load balance on the host. Second, we launch a kernel for each group and calculate the number of non-zero elements in each row of the result matrix, which is typically called *symbolic execution*. Once this step has been completed, we know the space allocation and the memory location where each row group can start writing its output.

Algorithm 4 Overall procedure of our hybrid implementation.

```
1: ▷ $num\_row\_panels$ refers to the number of row panels in A
2: ▷ $num\_col\_panels$ refers to the number of column panels in B
3: $nun\_chunk = num\_row\_panels \times num\_col\_panels$
4: ▷ Compute flops of each chunk C on GPU
5: $total\_flop = 0$
6: for $row = 0; row < num\_row\_panels; row++$ do
7:     for $col = 0; col < num\_col\_panels; col++$ do
8:         $chunk\_id = row \times num\_col\_panels + col$
9:         $flop =$ GetFlops($A[row]$, $B[col]$)
10:        $flops[chunk\_id] = flop$
11:        $total\_flop = total\_flop + flop$
12:    end for
13: end for
14: Sort the chunks based on the flops with a decreasing order
15: ▷ Get the number of chunks processed on GPU
16: $num\_gpu = 0$
17: $gpu\_flop = 0$
18: for $ch = 0; ch < num\_chunk; ch++$ do
19:     $gpu\_flop = gpu\_flop + flops[ch]$
20:     if $(gpu\_flop/total\_flop) >= Ratio$ then
21:         $num\_gpu = ch + 1$
22:         break
23:     end if
24: end for
25: Parallel GPU thread to process $num\_gpu$ chunks
26: Parallel CPU thread to process the remaining chunks
```

Finally, we re-assign rows of matrix A based on the number of non-zero elements to achieve global load balance again and invoke kernels to do the actual computations, which is referred as *numeric execution* in the figure. As shown in the figure, we use dense accumulation for dense rows and the *hashmap* methods for sparse rows to accumulate results, as it is well established through prior work [30] that they are more efficient for their respective cases.

C. GPU and CPU Hybrid Implementation

Our GPU implementation was extended to support concurrent executions on both CPU and GPU. The motivation of this was that though our GPU implementation outperformed multi-core CPU implementation, CPU performance was also quite competitive (details in Section V). Thus, we seek opportunities to utilize both GPU and CPU resources to perform SpGEMM computations.

We considered two critical performance issues for our hybrid implementation. The first is which computing device should a chunk be assigned to. The second is how to determine the ratio of workload for GPU or CPU. For the first question, we observed that GPU is more suitable to process dense chunks and the throughput of GPU is larger, we reorder chunks of result matrix C based on the number of floating-point operations (flops). Then, we assign chunks with more flops to GPU and the remaining to CPU. In terms of the fraction of the workload assigned to two units, this is done by considering the expected relative speedup of GPU over CPU (denoted by S). Specifically, we use the number of flops as the indicator for the workload. We seek a value g, where first g chunks contain at least $S/(S+1)$ of the number of flops, and assign these g chunks to GPU. We denote $Ratio = S/(S+1)$, and our experimental results show that a fixed value of 65% can achieve good performance for all of our input matrices. However, this ratio is just suitable for our experimental setups, and it might change if we use another GPU or CPU, but we should still be able to use a ratio to assign the workloads.

Algorithm 4 shows the procedure of our hybrid implementation. We keep row panels and columns panels both on device memory and host memory. We first compute the number of flops of each chunk in lines 6-13 and order the chunks based on the number of flops calculated in line 14. Note that the overhead of computing the flops of each chunk is really small compared with SpGEMM computations on GPU [30], [28].Then, we get the number of chunks assigned to GPU based on the $Ratio$ in line 18-23. Finally, we launch two parallel threads: one thread for GPU and one for CPU. Both GPU and CPU threads generate the final matrix chunk by chunk. The GPU implementation of each chunk is described in the above section. On CPUs, a recent high-performance multi-core implementation from Nagasaka *et al.*[27] was invoked for each chunk (more specifically, the hashmap implementation available from them). We also considered MKL library [36] from Intel as another alternative. However, since MKL Library only supports *integer* as the data type for the arrays $row\_offsets$ and col_ids, it can not handle large matrices. Moreover, hashmap implementation from Nagasaka *et al.*[27] also outperformed the MKL Library for small matrices.

D. Partitioning Implementation

One of the requirements for the framework described above is to efficiently partition the matrices A and B to row and column panels, respectively. Note that with the use of the CSR format (which stores each sparse row contiguously), partitioning the matrix A to row panels is straight-forward and it is easy to make it parallel.

However, partitioning of the matrix B is more difficult because we are not able to identify elements of the same column directly using the CSR format. As a result, we use a two-stage partitioning: We first calculate the number of non-zero elements for each column panel, and then we allocate the memory for the column panel. Finally, we fill in the column ids and values of each column panel using the CSR format.

A simplistic implementation would iterate over the column panels to be computed. The range of columns of the original matrix that belongs to a given column panel is computed and denoted by $[start\_col, end\_col]$. For each column panel, we iterate over all the rows in the matrix and identify the non-zero elements in this row that are in the column range we computed. Recall that *row_offset[r]* stores the starting index in the *col_id* and *data* arrays that correspond to the row r. Thus, we iterate over *row_offset[r]* and *row_offset[r+1]*, and determine the elements in the sparse matrix that are within the column range we are looking for.

It is easy to see that this algorithm can be quite inefficient, particularly as the size of rows and the number of column panels increases. We optimize this process by creating an additional data structure, which is *col_offset*. For a given column panel, *col_offset[j]* stores the earliest location in *data* and *col_id* arrays where elements corresponding to the row j and this particular column panel are stored. Now, as each column panel is processed, each row r is traversed starting not from *row_offset[r]* but from *col_offset[i]*. Subsequently, after finding the first element in this row that is not within the range of columns for this column panel, *col_offset[r]* is set to help processing of the next column panel. We also parallelize the partitioning in a prefix sum fashion.

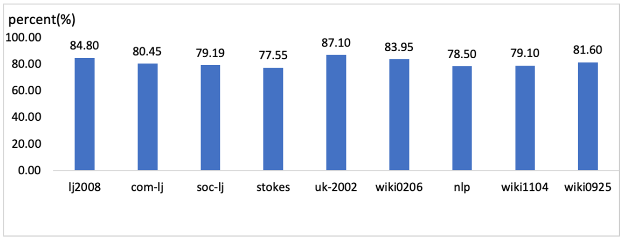


Fig. 4: Percentage of data transfer time over total execution time for synchronous spECK.

IV. Asynchronous Execution

As we had shown in Algorithm 3, the output matrix C is computed and moved from device memory to CPU memory chunk by chunk because the size of the output matrix is typically significantly larger than input matrices and exceeds the limit of device memory. Experimental results show that such data transfers can take a large amount of time. Thus, overlapping data movements with real computations turns out to be important for performance. In turn, designing an implementation that can support such asynchronous transfers impacts the overall design in several ways. This section explains these implementation issues and our solutions.

A. Motivation for Asynchronous Execution

We first demonstrate that the overhead of data transfers is significant. For this purpose, the following experiment was conducted. We modified the implementation of state-of-the-art method spECK [30] to follow the structure shown in Algorithm 3. Data movement was done synchronously – we refer to this version as synchronous, partitioned spECK, or simply synchronous spECK.

Figure 4 shows the percentage of data transfer time over total execution time using synchronous spECK using 9 matrices (More details of matrices and experimental environment are given later in Section V). The percentage varies with the chunk size. Thus, we select the results when synchronous spECK achieves the best performance. As can be seen, the data transfers time are quite high – ranging from 77.55% to 89.65% of the total execution time depending upon the matrix. Thus, these results point to the possibility of up to 22.45% reduction in total execution time if all computation can be overlapped with these transfers.

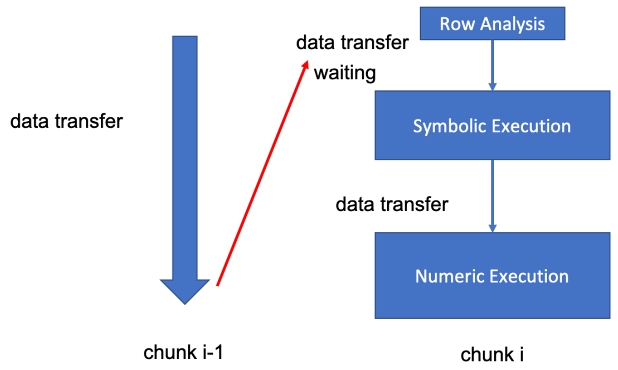


Fig. 5: Illustration of the data transfer issue for asynchronous executions.

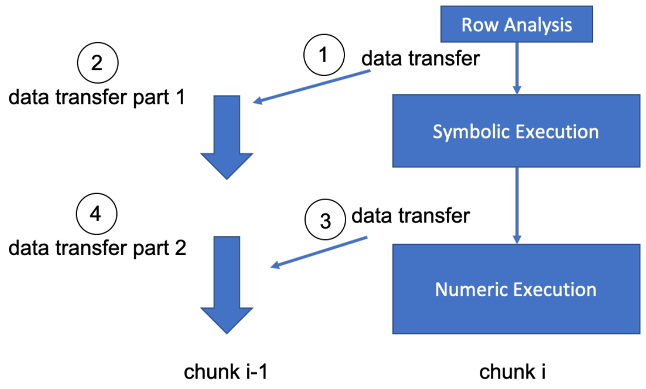


Fig. 6: Illustration of the solution to solve the data transfer issue. The numbers show the order of data transfers.

B. *Implementation Issues of Asynchronous Transfers*

As illustrated above, it is attractive to transfer data from device memory to CPU memory asynchronously to hide the overhead of real SpGEMM computations. A simple implementation is that we create two *streams* and two *buffers* on device. When we finish the SpGEMM computations using the first stream and store the results on the first chunk, we can start transferring the data from the first buffer to CPU pinned memory and start SpGEMM computations of the next chunk using the second stream. We use the streams and buffers alternatively to complete the overall procedure.

However, this simple idea does not work for our SpGEMM computations. First, according to the Programming Guide [18], two commands from different streams can not run concurrently if the host issues any device memory allocation and de-allocations. As a result, we have to avoid dynamic memory allocations during SpGEMM computations to support the asynchronous executions. On the other hand, an efficient SpGEMM implementation typically requires many dynamic memory allocations, to achieve load balance or make efficient use of device memory [30]. For example, as shown in Figure 3, only when we finish the *symbolic execution* and get the number of non-zero elements in the result matrix C, we can allocate device memory to store the array *col_ids* and *data*. This is to avoid allocating too much device memory. Another example is when we finish the row analysis, we need to assign rows to different groups. Thus, we need to allocate device memory to store the group information.

Second, there is only one engine for each direction of data transfer because we used PCI-e, which means that we can only have one data transfer from host to device at one time (and similarly, only one data transfer from device to host). However, an efficient SpGEMM implementation requires multiple data transfers during the computations [30], [28]. For example, in the row analysis stage, we launch a kernel to compute the number of floating-point operations in each row and store this information on device memory. Then, we need to transfer these data from device memory to host memory so that CPU can assign rows to different groups based on the results of row analysis, which is shown in Figure 3. However, this causes a problem to enable asynchronous executions. As illustrated in Figure 5, the SpGEMM computations of chunk i will wait until the data transfer of chunk $i-1$ finished before *symbolic execution*. As a result, we lose the opportunity to have concurrency execution.

Pre-Allocation to Avoid Dynamic Memory Allocation: To solve the first challenge, we allocate device memory before we execute SpGEMM computations as shown in Algorithm 3. An alternative solution we considered is to compute the upper bounds of the sizes of the data structures we used. However, the gap between upper bounds and the actual sizes are really large. As an example, consider estimating the upper bound of the number of non-zero elements in an output chunk. A commonly used approach would be to look at the number of floating-point operations (which are analyzed towards the goal of load balancing). In the worst case, every single multiplication of elements of matrices A and B could lead to a distinct element in C. However, we find that the difference between the estimate and the actual observed value can be large. These differences not only impact the efficiency of execution but also limits the size of chunk. Furthermore, a fixed sized allocation will imply that the chunk size used would be limited by certain chunks that are extremely dense, and thus require large allocation.

Our solution involves doing our own memory management. A large chunk of memory is pre-allocated on device memory and shared by all dynamic data structures. For each data structure, we maintain an offset, which is assigned incrementally as memory requirements are determined and a structure needs memory. This method turned out to be more effective than trying to work with worst-case estimates of memory requirements.

Dividing and Scheduling Data Transfers: Our solution to overcome the second challenge is shown in Figure 6. When we finish SpGEMM computations of the chunk $i-1$, we do not transfer the result chunk back to the CPU side immediately. Instead, we first finish the row analysis stage of the chunk i and transfer the collected data back to the host side. Subsequently, we transfer data of chunk $i-1$, and overlap it with the *symbolic execution* phase of the chunk $i-1$, as shown in the figure. The motivation is that overhead of row analysis is typically very small compared with other phases. Thus, we give up concurrency opportunities during the row analysis stage. However, we can continue to process the chunk i without waiting for the data transfer of the chunk $i-1$ to be finished. As a result, data transfer of chunk $i-1$ can hide the overhead of *symbolic execution* phase.

Further, when we finish the *symbolic execution* phase of chunk i, we get the number of non-zero elements of each row in result matrix C. We also need to transfer this information from the GPU to the CPU. This also conflicts with the data transfer of chunk $i-1$. Thus, we divide the output of the chunk $i-1$ into two portions as shown in Figure 6. The transfer of the first portion is overlapped with the *symbolic execution* phase, and the second portion's transfer is overlapped with the *numeric execution* phase. Since the *numeric execution* typically takes a longer time than the *symbolic execution* phase, in our implementation the first portion contains 33% of the total number of rows. Our experimental results show that the time spent on both *symbolic* and *numeric* execution phases can be hidden by data transfer under this ratio for our selected input matrices.

C. *Scheduling Execution Order of Chunks with GPU*

As illustrated above, the overhead of data transfers typically dominates the overall overhead. Thus, we seek opportunities to schedule the executions of chunks so that the overhead of data transfers can completely hide the overhead of computations. The size of each chunk, and thus the time spent computing it and transferring it, varies very significantly. If we attempt to overlap computations on a large chunk with transfer of a very small chunk, the benefits may be very small. Our solution in this work is to reorder the chunks to make the data transfer overhead in decreasing order. In this way, the overhead of SpGEMM computations of the chunk i tends to be smaller than the overhead of data transfer of the chunk $i-1$, so that the computation overhead can be hidden by the data transfer overhead.

It should be noted that the overhead of data transfer depends on the number of non-zero elements of the chunk. However, calculating the number of non-zero elements itself counts as a large overhead itself. To overcome this problem, we use the number of flops of each chunk instead because the number of non-zero elements and the number of flops typically have a positive correlation.

V. Experimental Results

In this section, we present our experimental results on a set of large matrices to demonstrate the effectiveness of our design for an out-of-core GPU implementation and a CPU-GPU hybrid implementation. We first show the features of the selected input matrices and our experimental environment. This is followed by a comparison of both our GPU implementation and the hybrid implementation against a multi-core CPU implementation. We also focus on evaluating the impact of our design choices, i.e., use of asynchronous transfers, and parameter for distributing work between CPU and GPU.

TABLE I: Nvidia Tesla V100 Specifications.

GPUs	**Tesla V100**
Architecture	Volta
#SM	80
Size of device memory	16GB
FP32 CUDA Cores/GPU	5120
Memory Interface	4096-bit HBM2
Register File Size / SM (KB)	65536
Max Registers / Thread	255
Shared Memory Size / SM (KB)	Configurable up to 96 KB
Max Thread Block Size	1024

TABLE II: Features of input matrices used in our experiments. All numbers except compression ratio are in millions.

matrix	**abbr.**	**n**	**nnz(A)**	**flop**(A^2)	**nnz**(A^2)	**compression ratio**
ljournal-2008	lj2008	5.36	79.02	7828.66	4245.41	1.84
com-LiveJournal	com-lj	4.00	69.36	8580.90	4859.09	1.77
soc-LiveJournal1	soc-lj	4.85	68.99	5915.63	3366.05	1.76
stokes	stokes	11.45	349.32	9424.18	2115.15	4.46
uk-2002	uk-2002	18.52	298.11	29206.61	3194.99	9.14
wikipedia-20070206	wiki0206	3.57	45.03	12796.04	4802.94	2.66
nlpkkt200	nlp	16.24	440.23	24932.82	2425.94	10.28
wikipedia-20061104	wiki1104	3.15	39.38	10728.99	4018.47	2.67
wikipedia-20060925	wiki0925	2.98	37.27	10030.09	3750.38	2.67

A. *Experimental Environment*

We conducted our experiments on an Nvidia Tesla V100. The specifications of this GPU are shown in Table I. The GPU is attached to an Intel(R) Xeon(R) CPU E5-2680 (2013 Ivy Bridge) running at 2.4 GHz. The CPU contains 14 physical cores and provides hyper-threading with 2 threads for each core. Thus, in our experiments, we use 28 threads for multi-core implementation. The size of the host memory is 128GB in our experiments. The host operating system for our experiments is CentOS Linux release 7.4.1708 (Core). Our GPU implementations are based on CUDA 10.1 toolkit and NVCC V10.1.168 is used to compile our programs.

B. *Input Matrices*

From the SuiteSparse Matrix Collection, we select 9 matrices [11] for detailed study and analysis. Most of these matrices (except for *stokes* and *nlpkkt200*) are collected from graphs. Our experiments use *double* as the data type for the array *data*. We selected these matrices because even the state-of-the-art spECK [30], which is shown to be more memory efficient as compared to previous implementations, can not handle any of these matrices because of memory limits on device memory of NVIDIA Tesla V100 GPU. Since these matrices are square, the multiplication operation we conduct is $C = A \times A$, as is the convention in other studies on SpGEMM [28], [30], [31], [37]. Table II shows the main features of these matrices and the matrix (A^2) obtained by their multiplication. Here, n denotes the number of rows (or columns), $nnz(A)$ denotes the number of non-zero elements in the matrix, $flop(A^2)$ denotes the number of floating point operations during the multiplication(a mutiply-add counts as 2 flops) , $nnz(A^2)$ denotes the number of non-zero elements in the result matrix, and *compression ratio* denotes the ratio between the $flop(A^2)$ and $nnz(A^2)$, which is an important performance metric. Also, for clarity of figures in this paper, we show the abbreviation of matrices named in the second column. Note that the storage requirements of the output matrix (A^2) can be as high as 60 GB, which well exceeds the storage of a Volta GPU.

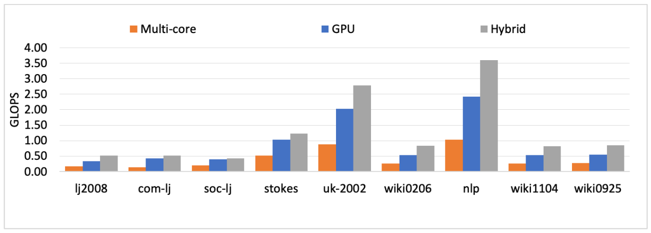


Fig. 7: Comparing GFLOPS of our hybrid implementation with multi-core CPU implementation and out-of-core GPU implementation.

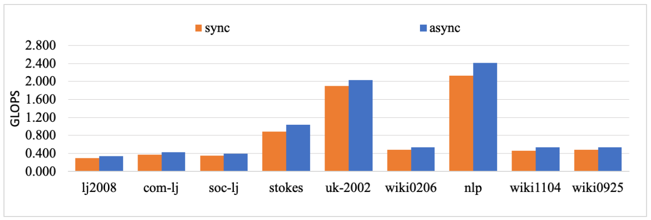


Fig. 8: Comparing our asynchronous GPU implementation with synchronous (spECK) GPU implementation.

C. Comparison with Multi-core CPU Implementation

In this section, we compare both our out-of-core GPU implementation and the hybrid implementation against a multi-core CPU implementation. The multi-core CPU implementation we use is modified from Nagasaka *et al.*[27]. As stated earlier, this implementation can handle large matrices (unlike MKL) while outperforming MKL on small matrices. The execution times measured for GFLOPS calculation include the time for transferring all chunks of the output matrix to the CPU memory. The results are shown in Figure 7.

First, we note that the GFLOPS achieved by our GPU implementation ranges from 0.34 to 2.03. We observed that the performance of SpGEMM computations is closely related with the *compression ratio*. Specifically, the compression ratio of matrices *nlpkkt200*, *uk-2002* and *stokes* are highest (10.28, 9.14, 4.46, respectively), whereas the compression ratios are under 3 for all other matrices. Correspondingly, the GFLOPS achieved by these three matrices are highest (2.42, 2.03, 1.03, respectively). Recall that the major overhead of our out-of-core implementation comes from data transfers, so the performance is positively correlated with compression ratio, which denotes the ratio between the amount of computation (proportional to $flops$) and the data transfer costs (proportional to $nnz(A^2)$). Also, we observed that the data skewness of the martices is related with the compression ratio. Typically, regular matrices such as *nlpkkt200* and *stokes* typically have a higher compression ratio.

Next, in comparing GPU implementation with a CPU baseline, we find that the speedup of our out-of-core GPU implementation over the multi-core implementation are between 1.98 and 3.03, with most values around 2. Note that the ratio remains relatively similar even as the absolute GFLOPS vary – this is because sparse matrices with more (less) regularity achieve better (worse) performance on both GPU and CPU. Finally, when comparing hybrid implementation over GPU implementation, we see speedups between 1.16 and 1.57, with most around 1.5. Note that the speedups of GPU implementation over multi-core implementation is around 2X in most cases. Thus, with GPU and CPU combined, the ideal case is that the total execution time is reduced by 1/3rd, or a 1.5X speedup. Thus, actually achieving this speedup for most cases shows that the workload assignment our hybrid implementation is effective and balanced. Note that the matrix where GPU to CPU ratio is the highest, i.e, 3.03 (*com-LiveJournal*, the additional speedup with the combined version is also the lowest.

D. Effectiveness of Asynchronous Executions

One of the challenges we have addressed in this work is to achieve asynchronous execution on GPU, i.e., overlap transferring of data from GPU to CPU with GPU execution. To demonstrate the benefits of this, we took the spECK GPU implementation [30] and modified it to work on our class of matrices. This was achieved through the same partitioning of the output matrix as in our implementation. Note that this implementation performs dynamic memory allocation, unlike our work.

The results are shown in Figure 8. We can observe that the asynchronous implementation achieves a speedup between 6.8% - 17.7% over the synchronous implementation. The speedup of the asynchronous implementation is mainly limited by the overhead of data transfers. As shown in Figure 4, the percentage of data transfers overhead over total execution time range from 77.5% to 89.6%.

E. Workload Assignment for GPUs

We now focus on our implementation related to work allocation in our hybrid implementation: reordering of chunks (sorting by number of flops required for the chunk) and the ratio of work allocation between GPU and CPU.

Effect of Reordering: We show the importance of reordering the executions of chunks in Figure 9. In the reordering implementation, we always assign chunks with more flops to the GPU and the remaining to the CPU. In the default implementation, we simply assign chunks to the GPU until the number of flops achieves the given ratio of work distribution between the CPU and GPU. This ratio (65% to GPU) and the chunk sizes of each input matrices used of these two implementations stay the same. The experimental results show that with our reordering, it can achieve significant performance improvement over the default implementation. This is because GPU is more suited for execution of chunks with higher density.

In our hybrid implementation with reordering, the ratio of flops assigned to GPUs is a key parameter for the performance In Figure 10, we compare the GFLOPS of two representative matrices under different values of the ratio. As indicated in the figure, the GFLOPS typically increases as we increase the ratio, but then drops. Our experimental results show that a fixed value for the ratio can achieve reasonable efficiency for our matrices.

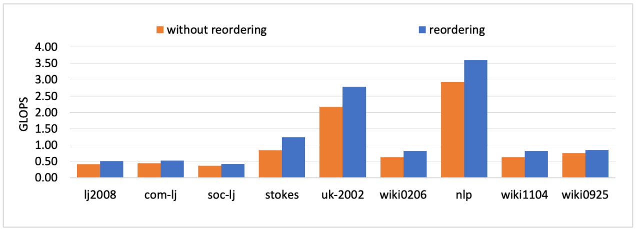


Fig. 9: Hybrid implementation with and without reordering.

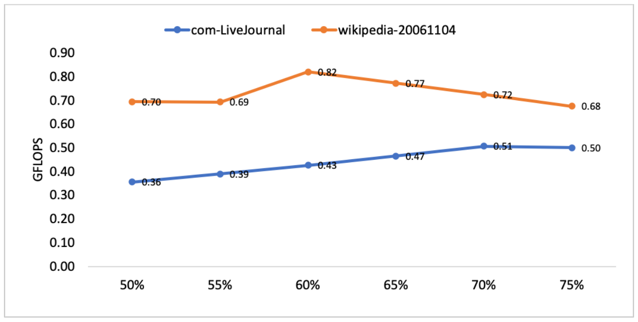


Fig. 10: Comparing performance on two matrices varying GPU/CPU chunk allocation ratio.

We further examine this issue through another experiment, reported in Table III. We compare the chunks assigned to GPUs when the $ratio$ is 65% with the number of chunks at which best performance of hybrid version is achieved (determined through exhaustive search). Note that because we have sorted the chunks by flops and most dense chunks are scheduled on the GPU, the number of chunks allocated to GPU is relatively small. As can be seen in the table, in 7 out of 9 cases, the number of chunks at which best performance is achieved is also the same as what one would get by 65% ratio. Further, for the remaining 2 matrices, the number of chunks we selected is close to the bast value and the performance drops are very small (2.95% and 4.30% respectively). Thus, this simple ratio is able to achieve best performance almost always.

VI. Related Work

In this section, we first discuss other research efforts on GPU acceleration of SpGEMM computations. Then, we introduced techniques proposed for SpGEMM in the context of CPU architecture. Finally, we list work on partitioning of matrices for SpGEMM computation.

A large body of work exists on accelerating the computation of SpGEMM on GPUs. Bell*et al.* [7], [9] proposed the ESC approach, which breaks the computation into Expansion, Sorting, and Compression. It first generates intermediate products (Expand), then it sorts these immediate results by their row and column identifies (Sort). Finally, it combines the values with colliding indices (Compress). Dalton*et al.* [10] improve that performance by storing intermediate products in shared memory. Winter *et al.* [37] further improve on this work by achieving both global load balance and local load balance. Merging is another important accumulation method, which uses sorted intermediate results and merge rows of the matrix directly. This method is like the merge-sort algorithm [15]. RMerge [16] splits the matrix into sub-matrices with limited row length and computes the product of these matrices in an iterative way. The benefit is they can utilize the register resources. Gremse *et al.* [17] further improve the performance through utilizing the shared memory so that they can merge rows of the matrix without splitting the matrix. bhSPARSE [24] dynamically chooses between different merging solutions based on the number of intermediate products. None of these efforts have supported out-of-core implementation for GPUs or considered utilizing combining the aggregate processing power of CPUs or GPUs. Recently, a CPU-GPU joint distributed SpGEMM algorithm called pipelined Sparse SUMMA was proposed [33]. They also implemented a probabilistic memory requirement estimator for efficiency.

In the context of CPUs, Patwary *et al.* [31] suggested that using dense arrays more efficient than using hash tables for multi-core platforms. They also proposed to partition the matrix to column panels so that the dense array can reside in L2 cache and reduce the data movement overhead. As a result, they achieve 3.8X speedups over the MKL library. Nagasaka *et al.* [27] proposed a multi-thread implementation on the Intel KNL processor and compare performance of different implementations on different kinds of matrices. Akbudak *et al.* [2] proposed to apply hypergraph and bipartite graph models to exploit both spatial and temporal locality for row-wise SpGEMM computations. To make efficient use of multi-level memory Deveci *et al.* [12] designed different chunking-based SpGEMM algorithms for Intel KNL. Again, none of this work has considered combining GPU execution with CPU execution.

Matrix partitioning is a significant problem for SpGEMM computation because a reasonable partitioning can not only improve the load balance, but also improve the data locality. There has been a significant amount of research work on hypergraph Partitioning to improve the locality. Akbudak *et al.* [2] applied the hypergraph model to improve the data locality for the Xeon Phi architecture. Akbudak *et al.* [1] presented a hypergraph partitioning model to minimize communication cost, while make sure that workload is balanced. Ballard *et al.* [6] proposed a framework to prove communication lower bounds for both parallel and sequential SpGEMM. They also converted identifying a communication-optimal algorithm for given input matrices to solving the HP problem. Akbudak *et al.* [3] also applied the HP model on the outer-product, inner-product, and row-by-row-product parallel SpGEMM algorithms.

TABLE III: Comparing the number of chunks assigned to GPU - Fixed ratio 65% vs. Best Case

matrix	Best No. of GPU chunks	65% GPU allocation No. of chunks
ljournal-2008	4	4
com-LiveJournal	3	3
soc-LiveJournal1	5	5
stokes	5	5
uk-2002	2	2
wikipedia-20070206	3	2
nlpkkt200	3	2
wikipedia-20061104	5	5
wikipedia-20060925	5	5

VII. Conclusion

This work is motivated by the observation that despite significant interest in SpGEMM implementation on GPUs, the limited memory of GPU hinders execution when the matrices involved are large, as is often the case in practice. To address this gap, we proposed an iterative *out-of-core* GPU implementation. A number of optimizations and design choices were critical to this process. Since data transfer overhead is a dominant factor in performance, we overlapped execution with data transfers. In the process, we had to create an implementation that does not require dynamic memory (de)allocation, and is judicious about data transfers. We extend our iterative implementation to a hybrid implementation, which offloads a fraction of chunks to the CPU to further improve performance. Our evaluation with 9 large matrices shows that we outperform a state-of-the-art CPU implementation, our hybrid implementation further achieves a speedup by 3.74X, and reordering and asynchronous execution turn out to be important for performance.

Acknowledgements. This research was partially supported by NSF awards CCF-1629392; CCF-2007793 and OAC-2034850.

References

[1] Kadir Akbudak and Cevdet Aykanat. Simultaneous input and output matrix partitioning for outer-product–parallel sparse matrix-matrix multiplication. *SIAM Journal on Scientific Computing*, 36(5):C568–C590, 2014.

[2] Kadir Akbudak and Cevdet Aykanat. Exploiting locality in sparse matrix-matrix multiplication on many-core architectures. *IEEE Transactions on Parallel and Distributed Systems*, 28(8):2258–2271, 2017.

[3] Kadir Akbudak, Oguz Selvitopi, and Cevdet Aykanat. Partitioning models for scaling parallel sparse matrix-matrix multiplication. *ACM Transactions on Parallel Computing (TOPC)*, 4(3):1–34, 2018.

[4] Pham Nguyen Quang Anh, Rui Fan, and Yonggang Wen. Balanced hashing and efficient gpu sparse general matrix-matrix multiplication. In *Proceedings of the 2016 International Conference on Supercomputing*, pages 1–12, 2016.

[5] Ammar Ahmad Awan, Khaled Hamidouche, Jahanzeb Maqbool Hashmi, and Dhabaleswar K Panda. S-caffe: Co-designing mpi runtimes and caffe for scalable deep learning on modern gpu clusters. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 193–205, 2017.

[6] Grey Ballard, Alex Druinsky, Nicholas Knight, and Oded Schwartz. Hypergraph partitioning for sparse matrix-matrix multiplication. *ACM Transactions on Parallel Computing (TOPC)*, 3(3):1–34, 2016.

[7] Nathan Bell, Steven Dalton, and Luke N Olson. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing*, 34(4):C123–C152, 2012.

[8] Timothy M Chan. More algorithms for all-pairs shortest paths in weighted graphs. *SIAM Journal on Computing*, 39(5):2075–2089, 2010.

[9] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations. *Version 0.5. 0*, 2014.

[10] Steven Dalton, Luke Olson, and Nathan Bell. Optimizing sparse matrix—matrix multiplication for the gpu. *ACM Transactions on Mathematical Software (TOMS)*, 41(4):1–20, 2015.

[11] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), December 2011.

[12] Mehmet Deveci, Simon D Hammond, Michael M Wolf, and Sivasankaran Rajamanickam. Sparse matrix-matrix multiplication on multilevel memory architectures: Algorithms and experiments. *arXiv preprint arXiv:1804.00695*, 2018.

[13] John R Gilbert, Steve Reinhardt, and Viral B Shah. High-performance graph algorithms from parallel sparse matrices. In *International Workshop on Applied Parallel Computing*, pages 260–269. Springer, 2006.

[14] Enrico Gobbetti, Fabio Marton, and José Antonio Iglesias Guitián. A single-pass gpu ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer*, 24(7-9):797–806, 2008.

[15] Oded Green, Robert McColl, and David A Bader. Gpu merge path: a gpu merging algorithm. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 331–340, 2012.

[16] Felix Gremse, Andreas Hofter, Lars Ole Schwen, Fabian Kiessling, and Uwe Naumann. Gpu-accelerated sparse matrix-matrix multiplication by iterative row merging. *SIAM Journal on Scientific Computing*, 37(1):C54–C71, 2015.

[17] Felix Gremse, Kerstin Kupper, and Uwe Naumann. Memory-efficient sparse matrix-matrix multiplication by row merging on many-core architectures. *SIAM Journal on Scientific Computing*, 40(4):C429–C449, 2018.

[18] Design Guide. Cuda c programming guide. *NVIDIA, July*, 2013.

[19] Fred G Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)*, 4(3):250–269, 1978.

[20] Takahiro Harada. A framework to transform in-core gpu algorithms to out-of-core algorithms. I3D '16, page 179–180, New York, NY, USA, 2016. Association for Computing Machinery.

[21] Peng Jiang, Changwan Hong, and Gagan Agrawal. A novel data transformation and execution strategy for accelerating sparse matrix multiplication on gpus. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 376–388, 2020.

[22] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. Mathematical foundations of the graphblas. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9. IEEE, 2016.

[23] Rakshith Kunchum, Ankur Chaudhry, Aravind Sukumaran-Rajam, Qingpeng Niu, Israt Nisa, and P Sadayappan. On improving performance of sparse matrix-matrix multiplication on gpus. In *Proceedings of the International Conference on Supercomputing*, pages 1–11, 2017.

[24] Weifeng Liu and Brian Vinter. An efficient gpu general sparse matrix-matrix multiplication for irregular data. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 370–381. IEEE, 2014.

[25] Weifeng Liu and Brian Vinter. A framework for general sparse matrix–matrix multiplication on gpus and heterogeneous processors. *Journal of Parallel and Distributed Computing*, 85:47–61, 2015.

[26] Xing Mu, Hou-Xing Zhou, Kang Chen, and Wei Hong. Higher order method of moments with a parallel out-of-core lu solver on gpu/cpu platform. *IEEE Transactions on Antennas and Propagation*, 62(11):5634–5646, 2014.

[27] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. High-performance sparse matrix-matrix products on intel knl and multicore architectures. In *Proceedings of the 47th International Conference on Parallel Processing Companion*, pages 1–10, 2018.

[28] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. High-performance and memory-saving sparse general matrix-matrix multiplication for nvidia pascal gpu. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 101–110. IEEE, 2017.

[29] Qingpeng Niu, Pai-Wei Lai, SM Faisal, Srinivasan Parthasarathy, and P Sadayappan. A fast implementation of mlr-mcl algorithm on multi-core processors. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10. IEEE, 2014.

[30] Mathias Parger, Martin Winter, Daniel Mlakar, and Markus Steinberger. speck: accelerating gpu sparse matrix-matrix multiplication through lightweight analysis. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 362–375, 2020.

[31] Md Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, Jongsoo Park, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, Sergey G Pudov, Vadim O Pirogov, and Pradeep Dubey. Parallel efficient sparse matrix-matrix multiplication on multicore platforms. In *International Conference on High Performance Computing*, pages 48–57. Springer, 2015.

[32] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proceedings of the VLDB Endowment*, 11(4):420–431, 2017.

[33] Oguz Selvitopi, Md Taufique Hussain, Ariful Azad, and Aydın Buluç. Optimizing high performance markov clustering for pre-exascale architectures. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 116–126. IEEE, 2020.

[34] Koichi Shirahata, Hitoshi Sato, and Satoshi Matsuoka. Out-of-core gpu memory management for mapreduce-based large-scale graph processing. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 221–229. IEEE, 2014.

[35] Virginia Vassilevska, Ryan Williams, and Raphael Yuster. Finding heaviest h-subgraphs in real weighted graphs, with applications. *ACM Transactions on Algorithms (TALG)*, 6(3):1–23, 2010.

[36] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi*™, pages 167–188. Springer, 2014.

[37] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. Adaptive sparse matrix-matrix multiplication on the gpu. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 68–81, 2019.