Fast Sparse Matrix Multiplication on GPU

Lukas Polok
Brno University of
Technology, Faculty of
Information Technology
Bozetechova 2, 612 66 Brno,
Czech Republic
ipolok@fit.vutbr.cz

Viorela Ila
Brno University of
Technology, Faculty of
Information Technology
Bozetechova 2, 612 66 Brno,
Czech Republic
ila@fit.vutbr.cz

Pavel Smrz
Brno University of
Technology, Faculty of
Information Technology
Bozetechova 2, 612 66 Brno,
Czech Republic
smrz@fit.vutbr.cz

ABSTRACT
Sparse matrix multiplication is an important algorithm in a wide variety of problems, including graph algorithms, simulations and linear solving to name a few. Yet, there are but a few works related to acceleration of sparse matrix multiplication on a GPU. We present a fast, novel algorithm for sparse matrix multiplication, outperforming the previous algorithm on GPU up to $3\times$ and CPU up to $30\times$. The principal improvements include more efficient load balancing strategy, and a faster sorting algorithm. The main contribution is design and implementation of efficient sparse matrix multiplication algorithm and extending it to sparse block matrices, which is to our best knowledge the first implementation of this kind.

Author Keywords
parallel sparse matrix multiplication; parallel linear algebra; matrix-matrix multiplication; GPGPU

ACM Classification Keywords
G.1.3 Numerical Linear Algebra: Sparse, structured, and very large systems (direct and iterative methods); G.4 Mathematical Software: Parallel and vector implementations

INTRODUCTION
This paper presents a novel and highly efficient parallel algorithm for sparse matrix multiplication. Sparse matrix-matrix multiplication is an important algorithm, useful in a wide variety of scientific tasks, including among others computational chemistry and physics, graph contraction, breadth-first search from multiple vertices, algebraic multigrid methods, finite element methods or solving (non)linear systems using Schur complement [29].

The sparse matrix algorithms are usually tightly coupled to the sparse matrix storage formats they use. Two of the popular formats are compressed sparse column (CSC) [12] and compressed sparse row (CSR). Those are closely related; matrices stored in one are transposes of the matrices stored in the other. CSC stores matrices as a vector of prefix sums of numbers of nonzero elements in each column and two vectors storing element values and their respective rows. It is common for the elements in each column to be ordered by their row number. The use of the CSC format is assumed in the rest of this paper, unless specified otherwise.

HPC 2015, April 12 - 15, 2015, Alexandria, VA, USA
Copyright ©2015 Society for Modeling & Simulation International (SCS)

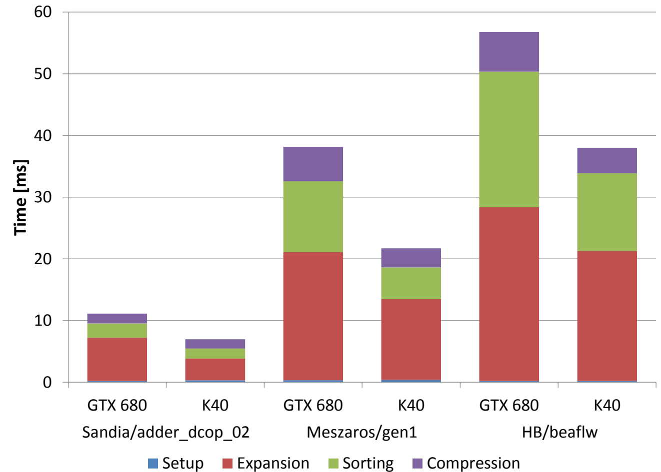

Figure 1. Time of different stages of the proposed algorithm.

Let us recall that in matrix multiplication $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$, each element of the product $\mathbf{C}_{i,j}$ is a sum of products of the corresponding elements in the ith row of $\mathbf{A}$ and the jth column of $\mathbf{B}$. The number of columns of $\mathbf{A}$ must match the number of rows of $\mathbf{B}$. In CSC, it is straightforward to look up elements by column ($O(1)$) but not to look up elements by row ($O(N)$ in the number of nonzero elements), which would be needed to calculate the elements of $\mathbf{C}$ in ordered fashion (*gather*).

The original algorithm for sequential sparse matrix multiplication [16] is implemented e.g. in the popular CSparse package [12] (used by Google's Ceres solver and Street View), and is work-efficient in terms of its complexity being proportional to the number of floating point operations (FLOP). It is worth mentioning that this level of efficiency is only reached for the price of calculating a partially unordered representation of the product, which is still useful in practice, but it is not the canonical form.

The algorithm [16] is efficient by traversing the elements of $\mathbf{B}$ column by column (assuming the CSC storage is used; for CSR all the terms are transposed), where each element $\mathbf{B}_{i,j}$ multiplies all the elements of $\mathbf{A}$ in the ith column (the one corresponding to the *row* of the particular element $\mathbf{B}_{i,j}$). Many of the other sparse matrix multiplication algorithms use this strategy. It produces partially ordered partial products (*scatter*), which need to be summed up. The authors of [16]

came up with an elegant way of quickly merging these partially ordered sequences to form the (unordered) result.

Parallel sparse matrix multiplication algorithms (PSpGEMM in BLAS terminology), however, generally decompose the matrices to band or block submatrices and distribute the computation of the partial products to different processors. Similarly like in the previous case, the results need to be merged to form the final product, using sparse matrix addition in this case. This approach is further referred as a coarse-grain work subdivision, since the submatrices are typically relatively large. Packages [4, 2] use this approach.

RELATED WORK

Unlike dense matrix multiplication [3, 13, 9, 15, 14] which is very well researched and widely understood, sparse matrix multiplication [8, 7, 25] is much more challenging - and even more so in the hardware. Many papers titled "sparse matrix multiplication" actually refer to sparse matrix-dense matrix multiplication [17, 28], which is an extension of sparse matrix-vector multiplication (PSpGEMV or PCSRMM), an equally useful but nonetheless different algorithm.

The work of Buluç [8] discusses the challenges of designing and implementing scalable sparse matrix multiplication in distributed memory systems. Coarse-grain 1D decomposition of the work is considered, and two novel 2D algorithms are presented. The identified challenges are the load imbalance, the amount of work for partial result reduction and the communication overheads.

Matam et. al. [17] explore several variations of the work division in a hybrid CPU + GPU algorithm: row-column, column-row and row-row. Therefore, the technique is based on the coarse grain algorithm. A heuristic is proposed for the fastest row-row case that efficiently balances computational load between the CPU and the GPU. The load balancing is further extended for a special case of banded matrices. The work contains highly efficient implementations of both multiplication of two sparse matrices and a sparse with a dense matrix. The sparse kernel achieves 240.663 MFLOP/s on average on matrices from the SNAP dataset, using quad-core Intel Core i7 920 CPU and Tesla C2050 GPU.

The work of Bell [5] is strongly influential in the context of the later developments of GPU algorithms. It proposes the Expansion Sorting Compression (ESC) algorithm. The expansion stage is based on the scattering of partial products to a matrix stored in the triplet form (also known as the COO format), using the same operation ordering as used in [16]. To convert to CSC, the partial products need to be sorted and the entries contributing to the same element of the product need to be *reduced* (summed up) at sorting and compression stages. The parallel primitives considered here are amenable to fine-grain work distribution. The proposed method is also inspired by the ESC algorithm.

The work of Bell was further refined in [10]. Their implementation is public as [1] and it was used for comparisons with the algorithm proposed in this paper. It focuses more intensively on the GPU platform-specific optimizations, such as avoiding passing data through global memory in favor of local memory and registers, especially in the sorting stage. The CSR storage is used, which is reflected in the three following paragraphs.

A permutation matrix is introduced, which orders the left operand by the work required to process a single row, facilitating load balancing. The product is later reordered, but the cost of doing so is reportedly relatively low.

The memory traffic of the expansion phase is further optimized for more regular coalesced accesses by casting the expansion process as a depth-first search on a layered bipartite graphs of the nonzero elements of both factors.

The sorting phase is also optimized, by realizing that the produced expansion is partially sorted, the expansions of individual rows are contiguous, and only intra-row sorting is required. This is implemented as sorting many rows in the local memory at once. Very long rows which would not fit in the local memory are sorted using a global sort. Further reduction in sorting is achieved by a priori knowledge of the distribution of the bits of the sorted keys, and by copying lower bits of column indices to unused upper bits of row indices, effectively avoiding to have to sort simultaneously or sequentially by two keys.

Well known parallel algorithms, such as parallel sum (*reduction*) and parallel prefix sum (*scan*) [6, 27] or parallel sort [26, 18, 19] are used by the proposed technique. In the remainder of this paper, it is assumed that the reader is familiar with them.

In our previous work, we showed the usefulness of sparse *block* matrices in solving nonlinear least squares problems [23, 22] such as Simultaneous Localization and Mapping (SLAM) in robotics or Bundle Adjustment (BA) and Structure from Motion (SfM) in computer vision, where the block structure naturally occurs. Some instances of Finite Element Method (FEM) problems may also exhibit block structure. We also showed performance gains of performing arithmetic operations exploiting the block structure [21] on CPU. In this paper, we propose an efficient algorithm for parallel sparse matrix multiplication which also extends to sparse block matrices, which is to our best knowledge the first implementation of *blockwise* PSpGEMM in hardware (note that sparse block matrix-vector multiplication was recently implemented on GPU [24]). Our implementation, unlike some others, runs entirely in the GPU, leaving the CPU available for other tasks.

The remainder of the paper is structured as follows. The following section contains the analysis of the algorithm and possible improvements. Section Implementation details the proposed implementation and optimizations used. Section Results shows the performance of the proposed solution through benchmarks and time comparisons with the exiting implementations. Conclusions and future work are given in Section Conclusions and Future Work

ALGORITHM DESIGN

The algorithm introduced in this paper is based on the ESC algorithm [5, 10]. However, the focus is on removing load

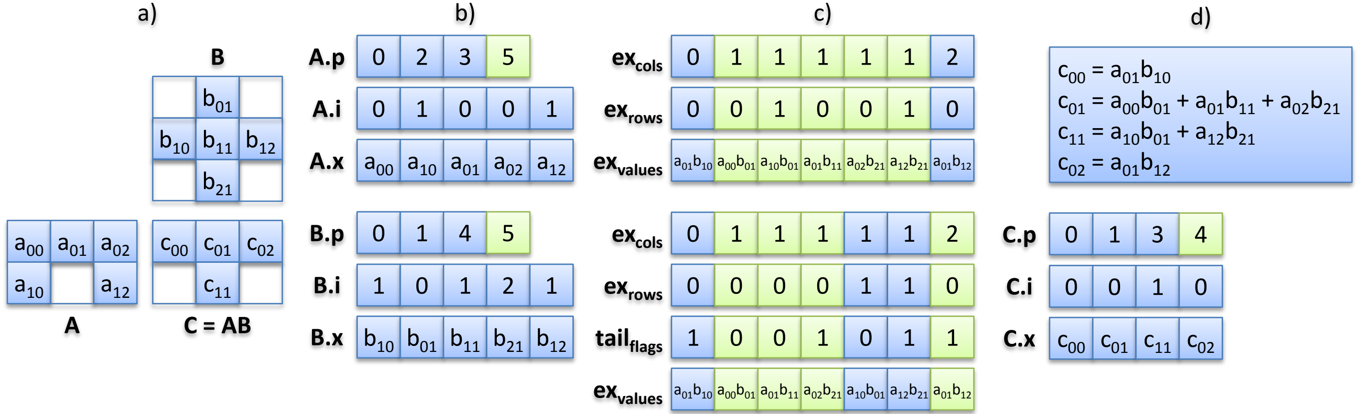

Figure 2. Data at the individual stages of the ESC algorithm[2]: a) the factors and their product, b) CSC representation of the factors, c) top: expansion of the product, segments of product columns indicated by alternating color, bottom: sorted expansion, segments of product elements indicated by alternating color, d) values of the product and its final CSC form.

imbalances and on simplicity, as especially the improved ESC algorithm in [10] handles many special cases, depending on the memory space (local or global) and granularity (thread, warp or thread group) of each particular operation. In contrast, the proposed implementation only requires six custom kernels, some of which are merely a fusion of multiple general purpose operations such as scan, created for performance purposes only.

Expansion Stage

Although the first conceptual stage of the algorithm is expansion, on GPU it is not possible to directly proceed, without first knowing its size, as all the memory needs to be allocated before starting the computation. From [16], it is trivial to derive the exact size of the expansion:

$$expansion(\mathbf{A}, \mathbf{B}) = \sum_{j=1}^{cols(\mathbf{B})} \sum_{k=1}^{nnzc(\mathbf{B},j)} nnzc(\mathbf{A}, row(\mathbf{B}, j, k)), \quad (1)$$

where $cols(\cdot)$ gets the number of columns of a matrix, $nnzc(\cdot, \cdot)$ returns the number of nonzero elements in a specified column of a specified matrix, and $row(\cdot, \cdot, \cdot)$ is the row of the given element in a column of a matrix. Note that all those are $O(1)$ array look-ups if the matrix is stored in CSC format. Also note that the expansion size is closely related to the number of FLOPs required to carry the multiplication out.

The expansion size dictates the memory cost of the ESC algorithm (the proposed variant as well as [5, 10]). Fig. 3 plots a ratio of expansion size to the number of nonzeros in the product. In certain cases $100\times$ more storage than the final product is required (please, refer to Section Results for the description of the dataset). Fortunately, it is possible to transparently subdivide the product by cutting the $\mathbf{B}$ matrix to several column slices, producing one slice of the product at a time.

The choice of granularity of expansion is crucial to load balancing. The proposed algorithm achieves perfect load balancing in the expansion stage by using granularity of individual

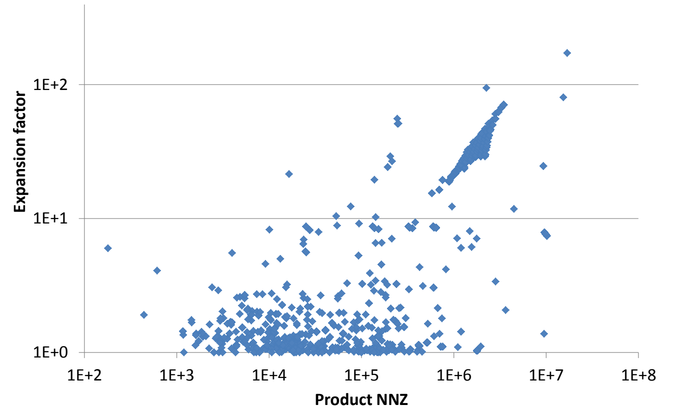

Figure 3. Expansion factor by the number of product NNZ.

scalar products. To do that, it is necessary for each thread to find the elements of $\mathbf{A}$ and $\mathbf{B}$ to process. Here, the *interpolation search* [20] algorithm is employed. It is a special case of binary search where the pivot is chosen based on linear interpolation of the values of the endpoints of the searched interval with the needle as the argument. The average complexity is $O(log\ log\ N)$, worst case being $O(N)$ (for comparison, binary search is $O(log\ N)$ in both cases). Interpolation search is not popular on CPU, as the linear interpolation is too expensive to outperform a regular binary search. However, it is a perfect fit for GPU where linear interpolation nicely hides under memory access latency and allows to find the needle in fewer steps, with much less branching.

The expanded scalar products are essentially the product matrix in the COO format; they can be stored in three vectors of the same length, $\mathbf{ex_{cols}}$ contains columns of the elements, $\mathbf{ex_{rows}}$ contains rows of the elements and $\mathbf{ex_{values}}$ contains values of the elements (see Fig. 2c). Note that the sparse multiplication algorithm generates a partially ordered expansion, where $\mathbf{ex_{cols}}$ is ordered and $\mathbf{ex_{rows}}$ consists of many short ordered runs (given by the rows of elements in the columns of $\mathbf{A}$, which are typically ordered).

[1] An interactive demonstrator is available online at `http://www.fit.vutbr.cz/~ipolok/esc`.

Sorting Stage

The approach in [5] is to use a single global sort. On GPU, the most efficient sort implementations use radix sort [18] with complexity $O(kN)$ where k is proportional to the number of bits of the key. In the case of keys generated by the expansion, the number of bits is given by base 2 logarithm of the number of rows and columns, respectively, and this knowledge can be used to accelerate the sort.

The radix sort may, however, not be the most efficient for a sequence which is already nearly sorted. As the sort starts, the expansion will be first ordered by the least significant bits of the keys, corresponding to the row indices. This will shuffle the column indices which were already ordered at the beginning. The elements are moved by long distances, leading to large amount of potentially uncoalesced global memory traffic. These will be ordered again in the later stages of the sort, by the most significant bits of the keys which correspond to the column indices, leading to long distance movement again.

The radix sort is efficient on GPU if the sorted elements are only reordered by small distances, as such reordering can be performed in the local memory. This is achieved by sorting it in segments corresponding to the individual columns of the product (Fig. 2c top), instead of sorting the whole expansion at once. The individual segments can be sorted in parallel. Now the elements are only reordered by relatively short distances, leading to better write coalescing and leaving ample opportunity to do the sorting in the local memory. However, load balancing issues arise, as the lengths of expansions of the individual columns can vary wildly [10].

In the context of GPU computing, some operations have *segmented* variants, e.g. a *segmented scan*. Its input is a vector of values to calculate the scan of, and a vector of *head flags*, a binary vector with ones at the positions of segment starts. Note that segmented operation is performed on the bulk of data rather than on the individual segments, and thus requires no explicit load balancing. Unfortunately, radix sort is not a good candidate for segmented implementation, as it would lead to both runtime and space tolls: the key bit histograms would need to be evaluated per each segment and the reordering would also need to take place per segment, leading to more load balancing issues. Fortunately, for merge sort, segmented variants exist[0], and the performance toll, compared to the non-segmented variant, is negligible. By using segmented sort, the time of the sorting stage was significantly reduced; one can compare Fig. 1 where sorting takes only 34%, with Fig. 4 in [10] where it is closer to 63% of the total runtime.

Compression Stage

Once the expansion is sorted, the compression is a simple task of calculating sums of elements with the same row and column, which are now in contiguous segments of the expansion (see Fig. 2c bottom). A simple segmented reduction can be used to calculate the sums, while the head flags can be calculated as a difference of row and column numbers between consecutive expansion elements. Note that similarly to expansion stage, the size of the compressed form needs to be calculated first (e.g. as a sum (reduction) of the head flags) so that the memory to store the results can be allocated, unless the size of the product is known beforehand.

When handling matrices with large elements, such as long double or especially block matrix elements (i.e. dense blocks), it is beneficial to reorder the operations slightly: instead of storing the *values* of the partial products in the expansion, store only the *pointers* to the operands and calculate the products themselves during compression. This reduces both the size of the expansion and the memory traffic of sorting it.

[0]One such implementation can be found at **`http://nvlabs.github.io/moderngpu/segsort.html`**.

IMPLEMENTATION

The proposed algorithm was implemented in OpenCL, and is presented in Algorithms 1, 2 and 3. This separation to parts is given by the need to allocate memory, which requires CPU intervention. The algorithm therefore requires GPU - CPU synchronization twice, at the beginning and after the end of Algorithm 2. This may be omitted if the allocation sizes or their upper bounds are known beforehand. Note that all the allocated buffers reside in the GPU memory.

In the algorithms, several conventions are followed. For any matrix $\mathbf{M}$ stored in the CSC format, $\mathbf{M}.\mathbf{p}$ is the prefix sum of nonzeros in each of its columns, $\mathbf{M}.\mathbf{i}$ is the vector of row indices of nonzero elements and $\mathbf{M}.\mathbf{x}$ is the corresponding vector of values of the elements. The parallel GPU *kernel* calls are denoted by **kernel**, and the (one-dimensional) execution domain is specified as $i = 0 \ldots N$, where i is the name of the variable holding the thread id, and N is the required number of threads (thread with id $N-1$ is the last thread).

In the setup stage (Algorithm 1), the $\mathbf{b_{cols}}$ vector is filled with column indices of each corresponding element of $\mathbf{B}$, making $\mathbf{B}$ available in both COO (intermediate) and CSC (input) formats. This allows $O(1)$ lookup of column of any element of $\mathbf{B}$ in the later stages of the algorithm. Additionally, each element of $\mathbf{b_{prods}}$ contains the amount of work required to multiply all the *preceding* elements of $\mathbf{B}$. This will be further used to facilitate load balancing at the expansion stage. The last element contains the total amount of work, which equals

Algorithm 1 Setup stage of `PSpGEMM`.

1: **function** GEMM($\mathbf{A}$, $\mathbf{B}$)
2: $\quad \mathbf{b_{cols}} = \text{ALLOCINT}(\text{NNZ}(\mathbf{B}))$
3: $\quad \mathbf{b_{prods}} = \text{ALLOCINT}(\text{NNZ}(\mathbf{B}) + \mathbf{1})$
4: $\quad$ **kernel** $(i = 0 \ldots \text{NNZ}(\mathbf{B}))$
5: $\quad\quad \mathbf{b_{cols}}[i] = 0$
6: $\quad\quad row = \mathbf{B}.\mathbf{i}[i]$
7: $\quad\quad \mathbf{b_{prods}}[i] = \mathbf{A}.\mathbf{p}[row + 1] - \mathbf{A}.\mathbf{p}[row]$
8: $\quad$ **end kernel** ▷ the last element of $\mathbf{b_{prods}}$ not initalized
9: $\quad$ **kernel** $(i = 0 \ldots \text{COLS}(\mathbf{B}))$
10: $\quad\quad \mathbf{b_{cols}}[\mathbf{B}.\mathbf{p}[i + 1] - 1] = 1$
11: $\quad$ **end kernel**
12: $\quad \mathbf{b_{cols}} = \text{EXCLUSIVESCAN}(\mathbf{b_{cols}})$
13: $\quad \mathbf{b_{prods}} = \text{EXCLUSIVESCAN}(\mathbf{b_{prods}})$
14: $\quad exp_size = \mathbf{b_{prods}}[\text{NNZ}(\mathbf{B})]$ ▷ expansion size

the expansion size. Note that the kernel at line 9 needs to be modified if $\mathbf{B}$ is known to be rank deficient (then the number of succeeding empty columns needs to be added to each 1 in $\mathbf{b_{prods}}$, and care must be taken to not write to index -1). These changes were omitted in sake of space.

The expansion stage (Algorithm 2) begins by allocation of the arrays to hold the expanded values. The expansion is performed by the number of threads necessary to saturate the GPU (denoted $GPU_{hardware\ threads}$), or less if the expansion is smaller than that. Each thread will calculate the same number of scalar products, as discussed in Section Expansion Stage A range of scalar products to carry out ($begin$, $count$) is allocated for each thread, which then looks up $\mathbf{b_{prods}}$ for the element of $\mathbf{B}$ where to start multiplying (line 22). UPPER_BOUND is a standard binary search function: for an ordered vector and a value, it returns the right-most position where this value could be inserted without violating the ordering. The inner loop at line 28 loops over elements of a particular column of the $\mathbf{A}$ matrix, while the outer loop (line 24) takes care of advancing onto the next columns. Note that col_skip is used to start the loop in the middle of a column, should that be required to equally balance the workloads. Also note that if $\mathbf{A}$ is known to be rank deficient, the outer loop may need to advance multiple times, until reaching a non-empty column (such that $elemA < endA$ before entering the inner loop).

Once the expansion is calculated, the $\mathbf{ex_{rows}}$, $\mathbf{ex_{values}}$ pairs can be sorted while using the head flags as segment markers (note that the beginning of the first segment is implied and does not need to be explicitly represented). Finally, once the expansion is sorted, the boundaries of the elements and the columns can be easily spotted, and the number of nonzeros of the final product can be calculated, using the kernel at line 42. The variable $block_size$ refers to the size of the blocks of the EXCLUSIVESCAN kernel, which is selected at runtime to best fit the target GPU.

In the final compression stage (Algorithm 3), the storage for the product is calculated. In the first kernel of this phase, the expansion is scanned for column tails (changes in $\mathbf{ex_{cols}}$, line 55) and element tails (changes also in $\mathbf{ex_{rows}}$, line 56). The scan of the element flags gives the element index in the compressed matrix. Note that the reduction of these flags was already calculated in the previous stage (line 46), which could

Algorithm 2 Expansion and sorting stages.

15: $\mathbf{ex_{cols}} = \textsc{AllocInt}(exp_size)$
16: $\mathbf{ex_{rows}} = \textsc{AllocInt}(exp_size)$
17: $\mathbf{ex_{values}} = \textsc{AllocFloat}(exp_size)$
18: $\mathbf{ex_{hf}} = \textsc{AllocBit}(exp_size)$ ▷ head flags bit array
19: **kernel** $(i = 0 \dots (N = GPU_{hardware\ threads}))$
20: $begin = \lfloor exp_size \cdot i/N \rfloor$
21: $count = \lfloor exp_size \cdot (i+1)/N \rfloor - begin$
22: $elemB = \text{UPPER_BOUND}(\mathbf{b_{prods}}, begin) - 1$
23: $col_skip = begin - \mathbf{b_{prods}}[elemB]$
24: **for** $(prod = 0; prod < count; \text{++}\ elemB)$ **do**
25: $rowB = \mathbf{B}.\mathbf{i}[elemB]$
26: $elemA = col_skip + \mathbf{A}.\mathbf{p}[rowB]$
27: $endA = \mathbf{A}.\mathbf{p}[rowB + 1]$
28: **while** $(elemA < endA$ **and** $p < count)$ **do**
29: $dest = begin + p$
30: $cur_col = \mathbf{ex_{cols}}[dest] = \mathbf{b_{cols}}[elemB]$
31: $\mathbf{ex_{rows}}[dest] = \mathbf{A}.\mathbf{i}[elemA]$
32: $\mathbf{ex_{values}}[dest] = \mathbf{A}.\mathbf{x}[elemA] \cdot \mathbf{B}.\mathbf{x}[elemB]$
33: $\mathbf{ex_{hf}}[dest] = cur_col > \mathbf{b_{cols}}[elemB - 1]$
34: $\text{++}\ elemA, \text{++}\ prod$
35: **end while**
36: $col_skip = 0$ ▷ skip in the first iteration only
37: **end for**
38: **end kernel**
39: $\textsc{SegmentedSort}(\mathbf{ex_{hf}}, \mathbf{ex_{rows}}, \mathbf{ex_{values}})$
40: $tail_blocks = \lceil exp_size/block_size \rceil$
41: $\mathbf{tail_counts} = \textsc{AllocInt}(tail_blocks + 1)$
▷ or reuse $\mathbf{b_{prods}}$ which is not needed below
42: **kernel** $(i = 0 \dots exp_size - 1)$
43: local int **flags**$[block_size]$ ▷ in local memory
44: **flags**$[i] = \mathbf{ex_{cols}}[i] < \mathbf{ex_{cols}}[i+1]$ **or** $\mathbf{ex_{rows}}[i] < \mathbf{ex_{rows}}[i+1]$
45: $g = \lfloor i/block_size \rfloor$ ▷ cooperating thread group
46: $\mathbf{tail_counts}[g] = \text{COOPERATIVE_REDUCE}(\textbf{flags})$
47: **end kernel**
48: $\mathbf{tail_counts} = \textsc{ExclusiveScan}(\mathbf{tail_counts})$
49: $product_NNZ = \mathbf{tail_counts}[tail_blocks] + 1$

Algorithm 3 Compression stage.

50: $\mathbf{C}.\mathbf{p} = \textsc{AllocInt}(\textsc{cols}(\mathbf{B}) + \mathbf{1})$
51: $\mathbf{C}.\mathbf{i} = \textsc{AllocInt}(product_NNZ)$
52: $\mathbf{C}.\mathbf{x} = \textsc{AllocFloat}(product_NNZ)$
53: **kernel** $(i = 0 \dots exp_size - 1)$
54: $g = \lfloor i/block_size \rfloor$ ▷ cooperating thread group
55: $col_tail = \mathbf{ex_{cols}}[i] < \mathbf{ex_{cols}}[i+1]$
56: $elem_tail = \mathbf{ex_{rows}}[i] < \mathbf{ex_{rows}}[i+1]$ **or** col_tail
57: local int **flags**$[block_size]$ ▷ in local memory
58: **flags**$[i] = elem_tail$
59: **flags** $= \text{COOPERATIVE_SCAN}(\textbf{flags})$
60: $compressed_index = \mathbf{tail_counts}[g] + \textbf{flags}[i]$
61: **if** $(elem_tail$ **and** $i < exp_size)$ **then**
62: $\mathbf{C}.\mathbf{i}[compressed_index] = i$ ▷ write indices of
63: **end if** ▷ reduced values of elements in expansion
64: **if** $(col_tail$ **and** $i < exp_size - 1)$ **then**
65: $\mathbf{C}.\mathbf{p}[\mathbf{ex_{cols}}[i] + 1] = compressed_index + 1$
66: **end if** ▷ write positions of beginnings of columns
67: **end kernel**
68: $\mathbf{C}.\mathbf{p}[0] = 0$ ▷ need to write this explicitly
69: $\mathbf{ex_{values}} = \textsc{SegmentedReduction}(\mathbf{C}.\mathbf{i}, \mathbf{ex_{values}})$
70: **kernel** $(i = 0 \dots product_NNZ)$
71: $expansion_index = \mathbf{C}.\mathbf{i}[i]$
72: $\mathbf{C}.\mathbf{i}[i] = \mathbf{ex_{rows}}[expansion_index]$
73: $\mathbf{C}.\mathbf{x}[i] = \mathbf{ex_{values}}[expansion_index]$
74: **end kernel**
75: **return C**
76: **end function**

be promoted to a scan to avoid recalculation, but storing the scans would require $O(\text{expansion size})$ memory and would be disadvantageous from both memory requirements and computational time standpoints.

Once the global index in the compressed matrix is known, indices of the final values of the elements in $\mathbf{ex_{values}}$ can be written ($\mathbf{C.i}$ is used as temporary storage), and $\mathbf{C.p}$ can be filled. Again, if the product is rank deficient, care needs to be taken: $\mathbf{C.p}$ might contain runs of multiple occurrences of the same index (including the zero index at the beginning), corresponding to the runs of empty columns.

Finally, the expansion values are summed up using segmented reduction, with $\mathbf{C.i}$ serving as tail flags, leaving the final values of the elements of the product at the tail positions in $\mathbf{ex_{values}}$ (line 69). The last kernel (line 70) merely copies these values to their compressed destinations in $\mathbf{C.p}$ and rewrites $\mathbf{C.i}$ by the corresponding row indices. Note that this kernel could be fused with the segmented reduction.

RESULTS

In this section, the timing results of sparse matrix multiplication performed using the proposed implementation[1] are compared with a similar state of the art implementation, CUSP 0.3.1 [1]. It was also compared to CSparse 1.2.0 [12], which runs on the CPU[2]. Despite all effort, we were unable to find any existing OpenCL `PSpGEMM` implementations.The evaluation was performed by all-to-all multiplication of sparse matrices from The University of Florida Sparse Matrix Collection [11] and their transposes (for matrices which share a common dimension). This collection was chosen because it contains sparse matrices corresponding to a diverse set of problems, and as such it is suitable for testing of general purpose linear algebra implementations.

All the tests were performed on a computer with NVIDIA GeForce GTX 680 (3 GB RAM) and Tesla K40 (12 GB RAM), a pair of AMD Opteron 2360 SE CPUs running at 2.5 GHz and 16 GB of RAM. In both cases, the program was compiled as x64, and both CUDA and OpenCL used 64-bit pointers. The latest GPU drivers (version 344.48) were used. CUDA implementations were linked against CUDA 6.5 SDK libraries. During the tests, the computer was not running any time-consuming processes in the background. Each test was run at least ten times until cumulative time of at least 5 seconds was reached, and the average time was calculated in order to avoid measurement errors, especially on smaller matrices. Explicit CPU - GPU synchronization was always performed, using `cuCtxSynchronize()` or `clFinish()`, respectively. ECC was disabled on the Tesla GPU.

Our implementation works with the CSC format. The implementations working with CSR format had their matrices converted (transposed) accordingly. Recorded times do not include the conversion or data transfers. The benchmarked version of the proposed algorithm handles all the rank-deficient

[1]The implementation of the proposed algorithm is available, at **https://sourceforge.net/projects/blockmatrix/**.

[2]CSparse is used as an orientative example, more efficient CPU implementations exist.

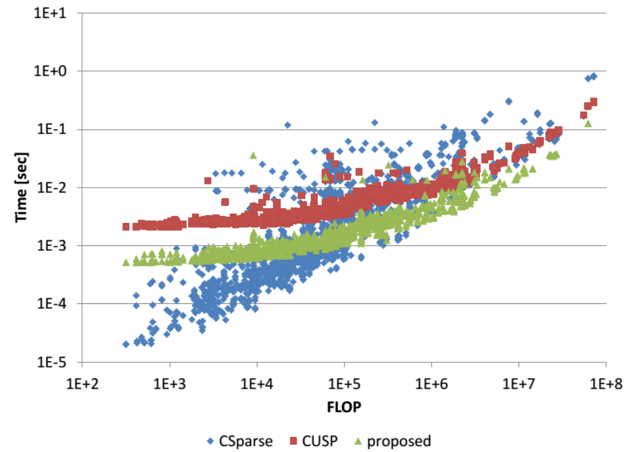

Figure 4. Performance scaling comparison on Tesla K40. Note that both axes are logarithmic.

cases described in the Section Implementation in a fully general way, without requiring prior detection or specialized kernels. The memory for the expansion and the product was allocated as outlined in Section Implementation, without any prior knowledge of the size of either. All the calculations were carried out in double precision.

Timing results for the all-to-all product benchmarks are on Fig. 4. Note that for very small matrices of less than ten thousand FLOP, CSparse is the fastest. For larger matrices, the proposed implementation takes over. Note that time of CSparse scales linearly with the number of FLOPs, as can be expected from a serial implementation of [16]. The times of the parallelized implementations grow slowly before the GPU gets saturated, then also scale approximately linearly. Least squares was employed to estimate the saturated costs to 27.7 ms/MFLOP for CSparse, 4.2 ms/MFLOP for CUSP and finally 3.0 ms/MFLOP for the proposed.

A more conventional comparison is presented in Table 1. This comparison was performed on the SNAP subset of the University of Florida Sparse Matrix Collection. It contains 9 different classes of matrices, a single matrix was chosen from each of them, much like evaluation in [17]. Each of the matrices was multiplied by itself (or in case of rectangular matrices, by its transpose). The proposed solution maintains the best times for most of the matrices, except for $roadNet-CA$, where the number of scalar products per element of the $\mathbf{B}$ matrix is very low, yielding high thread divergence in the proposed implementation. On smaller matrices such as $p2p-Gnutella31$, CUSP does not scale well and is slower despite the divergence. Reducing this divergence is the subject of the future work. Note that on $cit-Patents$, both the proposed and CUSP ran out of memory on GTX 680, and on $as-Skitter$ there was not enough system memory to perform the multiplication even on the CPU. This is not a principal problem of the algorithm, rather it is an implementation issue. One would only need to add an extra parameter of how many columns of the $\mathbf{B}$ matrix should be processed at a time (corresponding to the same number of columns of

Table 1. **Performance comparison on the SNAP subset, the best times are in bold (all times in seconds). The last two columns indicate relative speedup over CSparse and CUSP.**

				GF GTX 680		Tesla K40				
Matrix	nnz/row	FLOP	CSparse	CUSP	ours	CUSP	ours	MFLOP/s	×CSp.	×CUSP
roadNet-CA	2.807	22.138 M	0.774	**0.156**	0.199	0.103	**0.099**	223.662	7.823	1.038
web-Google	5.571	91.665 M	5.312	0.447	**0.433**	0.315	**0.249**	368.356	21.347	1.265
email-Enron	10.020	72.510 M	1.150	0.360	**0.271**	0.247	**0.173**	418.961	6.645	1.429
amazon0312	7.987	42.368 M	1.659	0.209	**0.241**	0.141	**0.123**	344.389	13.488	1.148
ca-CondMat	8.081	5.899 M	0.140	0.035	**0.027**	0.024	**0.015**	394.591	9.347	1.575
p2p-Gnutella31	2.363	539.035 k	0.032	0.015	**0.007**	0.008	**0.003**	190.560	11.304	3.003
wiki-Vote	12.497	7.254 M	0.082	0.036	**0.024**	0.025	**0.015**	482.961	5.482	1.633
cit-Patents	4.376	95.457 M	13.414	*out of RAM*		0.497	**0.446**	214.127	30.089	1.114
as-Skitter	13.081	53.771 G	*out of RAM*[3]							

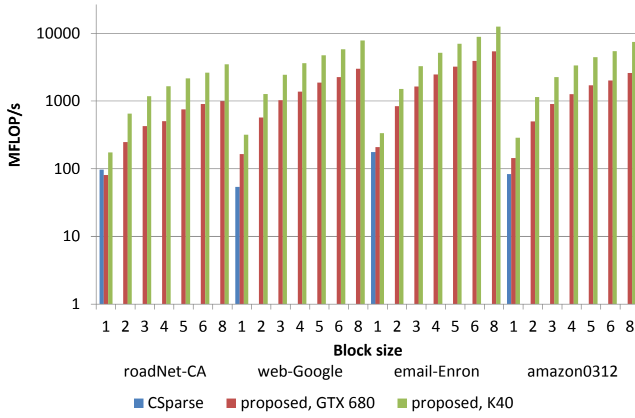

Figure 5. Performance scaling comparison of sparse block matrix multiplication on the first four matrices of SNAP.

the result), and the CPU would schedule the multiplication as several calls of the original algorithm.

For a synthetic benchmark of the sparse *block* matrix multiplication, the matrices from SNAP were used again. Each element was replaced by a dense block, while the block size was varied between the different tests. The results of this benchmark are on Fig. 5. As expected, the proposed implementation exhibits performance increase with increasing block sizes.

CONCLUSIONS AND FUTURE WORK

We presented a novel algorithm for sparse matrix multiplication and demonstrated its extension to sparse block matrices. The algorithm yields on average 329.7 MFLOP/s, outperforms CUSP by a factor of $1.53\times$, and outperforms CSparse running on a single CPU by a factor of $13.19\times$. The sparse block matrix multiplication exhibits further performance scaling with increasing block size, yielding up to 1.26 GFLOPS on Tesla K40 ($email - Enron$, 8×8 blocks). This makes it attractive in problems with block structure such as FEM, SLAM, BA or SfM. To improve the performance even more, multi-GPU or hybrid CPU-GPU extensions could be implemented. The implementation needs to be improved to handle large matrices by splitting the computation to bands, when the expansion does not fit in the GPU memory at once. Currently, only constant block size compressed column format (CBC) is supported. This can be extended to variable block size compressed column (VBC), once addressing possible thread divergence problems. Also, to better integrate with the existing CPU pipelines which use the SSE instruction set, allocation of the product matrix with the proper memory alignment of the blocks needs to be solved.

ACKNOWLEDGEMENTS

The research leading to these results has received funding from the European Union, 7th Framework Programme grants 316564-IMPART and the IT4Innovations Centre of Excellence, grant n. CZ.1.05/1.1.00/02.0070, supported by Operational Programme Research and Development for Innovations funded by Structural Funds of the European Union and the state budget of the Czech Republic.

We gratefully acknowledge the support of NVIDIA Corporation with the donation of the GPUs used for this research.

REFERENCES

1. *CUSP: Generic parallel algorithms for sparse matrix and graph computations*. NVIDIA Corporation, 2009. http://code.google.com/p/cusp-library/.
2. *Intel Math Kernel Library. Reference Manual*. Intel Corporation, Santa Clara, USA, 2009. 630813-054US, `http://software.intel.com/intel-mkl/`.
3. Agarwal, R. C., Balle, S. M., Gustavson, F. G., Joshi, M., and Palkar, P. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development 39*, 5 (1995), 575–582.
4. Balay, S., Gropp, W. D., McInnes, L. C., and Smith, B. F. Efficient management of parallelism in object oriented numerical software libraries. In *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds., Birkhäuser Press (1997), 163–202.
5. Bell, N., Dalton, S., and Olson, L. N. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing 34*, 4 (2012), C123–C152.

[3]Note that with only 16 GB of RAM, this matrix is too large even for CSparse: the product would take 26.4 GB.

6. Blelloch, G. E. *Vector models for data-parallel computing*, vol. 75. MIT press Cambridge, 1990.
7. Bowler, D., Miyazaki, T., and Gillan, M. Parallel sparse matrix multiplication for linear scaling electronic structure calculations. *Computer physics communications 137*, 2 (2001), 255–273.
8. Buluç, A., and Gilbert, J. R. Challenges and advances in parallel sparse matrix-matrix multiplication. In *Parallel Processing, 2008. ICPP'08. 37th International Conference on*, IEEE (2008), 503–510.
9. Choi, J., Walker, D. W., and Dongarra, J. J. Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience 6*, 7 (1994), 543–570.
10. Dalton, S., Bell, N., and Olson, L. Optimizing sparse matrix-matrix multiplication for the gpu. *Technical Report* (2013).
11. Davis, T. The university of florida sparse matrix collection. In *NA digest*, Citeseer (1994).
12. Davis, T. A. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Society for Industrial and Applied Mathematics, 2006.
13. Dekel, E., Nassimi, D., and Sahni, S. Parallel matrix and graph algorithms. *SIAM Journal on computing 10*, 4 (1981), 657–675.
14. Fatahalian, K., Sugerman, J., and Hanrahan, P. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ACM (2004), 133–137.
15. Gunnels, J., Lin, C., Morrow, G., and Van De Geijn, R. A flexible class of parallel matrix multiplication algorithms. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International... and Symposium on Parallel and Distributed Processing 1998*, IEEE (1998), 110–116.
16. Gustavson, F. G. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS) 4*, 3 (1978), 250–269.
17. Matam, K. K., Bharadwaj, S. R. K., and Kothapalli, K. Sparse matrix matrix multiplication on hybrid cpu+ gpu platforms. In *Proc. of 19th Annual International Conference on High Performance Computing (HiPC), Pune, India* (2012).
18. Merrill, D., and Grimshaw, A. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for gpu computing. *Parallel Processing Letters 21*, 02 (2011), 245–272.
19. Merrill, D. G., and Grimshaw, A. S. Revisiting sorting for gpgpu stream architectures. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ACM (2010), 545–546.
20. Perl, Y., Itai, A., and Avni, H. Interpolation search - a log log n search. *Communications of the ACM 21*, 7 (1978), 550–553.
21. Polok, L., Ila, V., and Smrz, P. Cache efficient implementation for block matrix operations. In *Proceedings of the 21st High Performance Computing Symposia*, ACM (2013), 698–706.
22. Polok, L., Ila, V., Solony, M., Smrz, P., and Zemcik, P. Incremental block cholesky factorization for nonlinear least squares in robotics. In *Proceedings of the Robotics: Science and Systems 2013* (2013).
23. Polok, L., Solony, M., Ila, V., Zemcik, P., and Smrz, P. Efficient implementation for block matrix operations for nonlinear least squares problems in robotic applications. In *Proceedings of the IEEE International Conference on Robotics and Automation*, IEEE (2013).
24. Rennich, S. Leveraging matrix block structure in sparse matrix-vector multiplication. In *NVIDIA GPU Technology Conference* (2012).
25. Saravanan, C., Shao, Y., Baer, R., Ross, P. N., and Head-Gordon, M. Sparse matrix multiplications for linear scaling electronic structure calculations in an atom-centered basis set using multiatom blocks. *Journal of computational chemistry 24*, 5 (2003), 618–622.
26. Satish, N., Harris, M., and Garland, M. Designing efficient sorting algorithms for manycore gpus. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, IEEE (2009), 1–10.
27. Sengupta, S., Harris, M., Zhang, Y., and Owens, J. D. Scan primitives for gpu computing. In *Graphics Hardware*, vol. 2007 (2007), 97–106.
28. Shah, M., and Patel, V. An efficient sparse matrix multiplication for skewed matrix on gpu. In *High Performance Computing and Communication & 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, IEEE (2012), 1301–1306.
29. Zhang, F. *The Schur complement and its applications*, vol. 4. Springer, 2005.