Adaptive Sparse Matrix-Matrix Multiplication on the GPU

Martin Winter
Graz University of Technology, Austria
martin.winter@icg.tugraz.at

Daniel Mlakar
Graz University of Technology, Austria
daniel.mlakar@icg.tugraz.at

Rhaleb Zayer
Max Planck Institute for Informatics, Germany
rzayer@mpi-inf.mpg.de

Hans-Peter Seidel
Max Planck Institute for Informatics, Germany
hpseidel@mpi-inf.mpg.de

Markus Steinberger
Graz University of Technology, Austria
steinberger@icg.tugraz.at

Abstract

In the ongoing efforts targeting the vectorization of linear algebra primitives, sparse matrix-matrix multiplication (SpGEMM) has received considerably less attention than sparse Matrix-Vector multiplication (SpMV). While both are equally important, this disparity can be attributed mainly to the additional formidable challenges raised by SpGEMM.

In this paper, we present a dynamic approach for addressing SpGEMM on the GPU. Our approach works directly on the standard compressed sparse rows (CSR) data format. In comparison to previous SpGEMM implementations, our approach guarantees a homogeneous, load-balanced access pattern to the first input matrix and improves memory access to the second input matrix. It adaptively re-purposes GPU threads during execution and maximizes the time efficient on-chip scratchpad memory can be used. Adhering to a completely deterministic scheduling pattern guarantees bit-stable results during repetitive execution, a property missing from other approaches. Evaluation on an extensive sparse matrix benchmark suggests our approach being the fastest SpGEMM implementation for highly sparse matrices (80% of the set). When bit-stable results are sought, our approach is the fastest across the entire test set.

CCS Concepts • **Theory of computation** → **Massively parallel algorithms**; • **Computing methodologies** → *Linear algebra algorithms*; • **Software and its engineering** → Scheduling;

PPoPP'19, February 16–20, 2019, Washington, DC, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6225-2/19/02...$15.00
https://doi.org/10.1145/3293883.3295701

Keywords SpGEMM, GPU, Sparse Matrix, Adaptive, ESC, bit-stable

1 Introduction

Generalized sparse matrix-matrix multiplication (SpGEMM) is one of the key kernels in scientific computing and data analytics, *e.g.*, in algebraic multigrid solvers [5], Schur complement methods [25], betweenness centrality [6] and cycle detection [26]. Algorithmically, SpGEMM consists of building the output matrix **C** from the product of two sparse input matrices **A** and **B**, given by

$$\mathbf{C}_{ij} = \sum_k \mathbf{A}_{ik} \cdot \mathbf{B}_{kj}; \qquad (1)$$

where k spans the colliding non-zeros of the i-th row of **A** and j-th column of **B**. In the sequential setting, efficient treatment of SpGEMM dates back to the pioneering work of Gustavson [18]. As the computing landscape shifts towards ubiquitous parallelism, there is a pressing need for equivalently efficient approaches on modern many-core processors.

Among existing many-core architectures, the graphics processing unit (GPU) is of particular interest. It has emerged as a viable and cost-effective co-processor in both single workstations and large supercomputing clusters. As the GPU is primarily designed for massively parallel, uniform execution and memory access, achieving good speedups on unstructured problems such as SpGEMM remains a challenge.

To provide context to the ensuing discussion, we assume matrices are given in the compressed sparse row (CSR) format, which is probably the most common format in use. Entries are sorted according to rows and their values and column ids are explicitly stored. An additional row pointer array indicates the beginning of each row in the sorted arrays.

Challenges The challenges of SpGEMM on the GPU stem from multiple factors. First, the number of entries in each row of **A** and **B** may vary strongly. This disparity complicates load balancing, as threads may easily receive vastly different work loads, which is especially difficult to manage on single instruction, multiple data (SIMD) devices like GPUs.

Second, there is no way to predict the number of intermediate products ($\mathbf{A}_{ik} \cdot \mathbf{B}_{kj}$) without inspecting the matrices. This makes it difficult to perform intermediate computations within efficient on-chip memory, as it may easily overflow.

Third, memory access patterns are paramount on the GPU. Due to the nature of SpGEMM, the sparsity pattern and content of both matrices **A** and **B** determine the memory access pattern throughout computations.

Strategies Without loss of generality, the landscape of GPU SpGEMM is dominated by the following strategies

- ESC: explicitly *expand* all temporary products, *sort* them and *combine* them to yield the final matrix [5, 7–9].
- Hashing: merge temporary products either in scratchpad memory or globally using hashing [3, 22, 23].
- Merging and Hybrid: choose a fitting method for each row [17, 19, 20].

Most of these approaches are designed with only one or two of the challenges discussed earlier in mind. The ESC strategy achieves excellent load balancing at the cost of high intermediate memory. In fact, in its original form all intermediate products go through slow global GPU memory. Similarly, hashing is notoriously slow in global memory. Operating (partially) in scratchpad memory can increase performance of both approaches. Relying on smart global scheduling [7, 22], overflow of scratchpad memory can be avoided. However, this entails a complete matrix inspection (which can consume up to 30% runtime; *cf.* [22] fig. 6). One major drawback of hashing is that the accumulation order depends on the hardware scheduler and thus might yield different floating point errors during each run.

Merge-based approaches and hybrids focus on preprocessing and row-based scheduling. This may lead to a large number of temporary matrices and significant preprocessing effort. Furthermore, memory access patterns may deteriorate by switching strategies. Again, for this kind of scheduling the matrices need to be inspected fully.

Contribution We propose a new GPU SpGEMM algorithm that tackles the challenges outlined above in a comprehensive way and at the same time cuts down on overhead computations throughout all stages. To this end, we make the following contributions:

- an efficient global load balancing scheme based solely on the non-zeros of **A**. It avoids costly inspection of the involved matrices and achieves entirely uniform load balancing in practice.
- a dynamic work distribution which achieves perfect local load balancing within a block of threads throughout multiple iterations of ESC, mixing partial results with new input.
- an efficient local adaption of the ESC algorithm which operates within scratchpad memory and allows combining temporary results to a complete subset of **C**.
- chunk-based storage of partial results of **C**.
- an efficient, adaptive merge algorithm for partial results.
- an adaptive approach to handle long rows of **B** efficiently.

2 Related work

In the sequential treatment of the problem, the output matrix is filled one row at a time by means of a large bookkeeping array aka sparse accumulator (SPA) [10, 15, 18]. As the sparsity pattern of **C** is not known prior to execution, a preliminary pass for memory allocation counts the number of non-zeros of **C** and a secondary pass computes the entries.

Over the years many strategies have been proposed to adapt SpGEMM to modern parallel architectures. They vary mainly in the way the operations in Eq. 1 are partitioned among **A**, **B**, and **C**. The classification by Ballard et al. [4] offers an elegant dimensionality-based interpretation of strategies by mapping operations to the cube $\mathbf{A}\times\mathbf{B}\times\mathbf{C}$. Within this classification, problems are amenable to solving the underlying hypergraph partitioning. However, the more elaborate the partition, the more preprocessing effort is needed.

The multi-threaded approach by Patwary et al. [24] reduces cache misses by blocking accesses to the SPA. It searches for adequate block partitioning of the columns of **B** and fills individual blocks of **C** independently. In the same spirit, Akbudak and Aykanat [1] rely on row-wise partitioning of **A** to exploit locality in accessing the rows of **B**.

The approach by Bell et al. [5], implemented in CUSP [8], breaks the processing into expansion, sorting, and compression (ESC). This translates into generating a list of intermediate products which are sorted by row and column index and falls within the 3D category. The output is generated by contracting entries with the same indices. Optimizations take into account the sparsity patterns of **A** and **B** to improve the row-wise processing of **C** [9]. Another variant performs ESC locally before merging the results globally [7] at the cost of increased load balancing effort. While we also perform ESC locally, the major advantage of our approach is that we use dynamic scheduling to perform multiple ESC iterations before going to global memory, considerably reducing memory bandwidth, global sorting and compaction costs.

Adopting a partial $1D$ row-wise strategy, rows from **B** can directly be merged, even on the GPU [16]. The RMerge approach [17] optimizes this merge strategy by ensuring operations are completed in efficient memory. This constraint is enforced by splitting **B** into multiple matrices with limited row length and iteratively computing the product of these matrices from right to left. In the bhSparse approach [20] rows are grouped by the number of intermediate products and then a merge-based strategy is adaptively selected based on the number of intermediate products. They also compare against a CPU implementation based on Intel MKL and show average speed up of 2.5/2.2 for single/double precision.

In a similar spirit, the approach by Kunchum et al. [19] selects different strategies depending on the row structure. SpGEMM can be also addressed using hash tables. An early approach [12] keeps a primary hash table in scratchpad memory and a secondary in global memory. An implementation of this approach is used within cuSparse [23]. Deveci et al. [13] also uses a dual hash table approach, whereas global hash tables are only used temporarily and are reclaimed. The BalancedHash approach [3] restricts itself to local hash tables and avoids overflows using "better size estimates" [2].

nsparse [22] follows these approaches and addresses the memory problem by grouping rows based on intermediate products and thus can construct hash tables with different sizes. Deveci et al. [14] build on their previous work [13]. In particular, they combine partitioning for hierarchical parallelism with the use of a two-level hash data structure.

One downside of hash-based approaches is their non-deterministic compaction order leading to different floating point errors during each run.

3 Adaptive SpGEMM

Our adaptive chunk-based GPU SpGEMM approach (AC-SpGEMM) focuses on four major goals

1. performing computations in local on-chip memory
2. coherent memory access
3. independence of row lengths
4. ensuring deterministic results

To achieve these goals, AC-SpGEMM follows a four stage approach, as outlined in Figure 2. In the first stage, AC-SpGEMM prepares data for global load balancing. In the second stage, we perform chunk-based ESC, producing deterministically bit-stable results. With the help of our local work distribution, this stage performs multiple iterations of ESC, fully utilizing local memory resources while keeping temporary results in scratchpad memory. Merging of rows shared across chunks happens in the third stage. Finally, in the fourth stage, we allocate the output matrix **C** and fill it with data from the chunks.

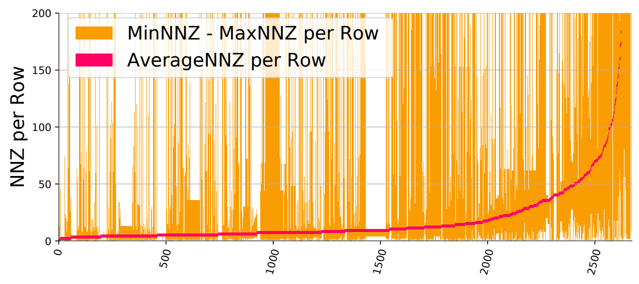


Figure 1. Average non-zeros per row for the SuiteSparse matrix collection, min and max overlayed and clamped.

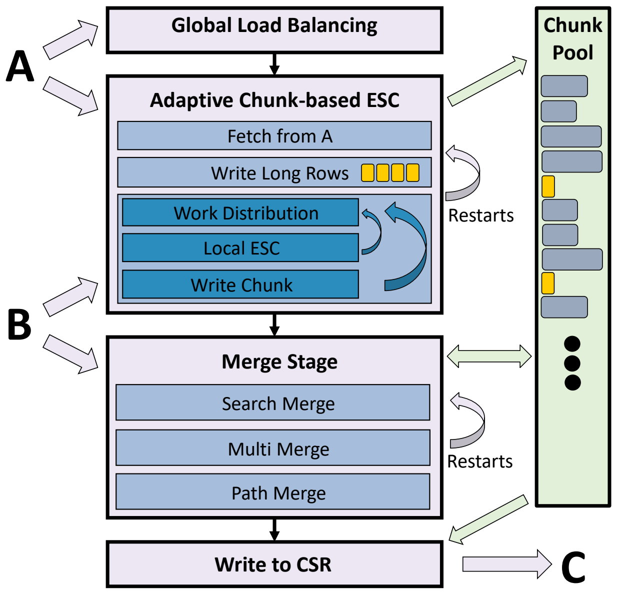


Figure 2. Our approach has four consecutive stages: global load balancing follows a strict non-zero splitting of **A**; our AC-ESC step performs the major part of the SpGEMM, computing entire chunks of **C**; merge combines the rows shared across chunks before the data is copied to **C**.

Before discussing the details of each stage, we motivate some design choices. Analysing the row length of matrices from the SuiteSparse matrix collection [11], it can be observed that the majority of matrices in common problem domains have average row lengths of less than 200 elements, *cf.* Figure 1. Considering register sizes of current GPUs and reasonably small thread block sizes, up to 4000 temporary elements can be held by each block. If there is reasonable overlap between rows, the resulting output can be stored in scratchpad memory for another iteration of ESC. Given 200 entries per row, ideally another 3800 temporary elements can be loaded and compacted. In the best case, these local load and compaction steps continue until yielding completed rows of **C**, without ever going through slow global memory.

3.1 Global Load Balancing

Previous SpGEMM approaches adopt one of two strategies for load balancing. (1) They bin the rows of **A** according to their lengths [20] and choose an appropriate algorithm for each category. This strategy may pull apart sequential rows in **A**, severing memory access patterns to **A** and **C**.

(2) They analyze the number of temporary products that will be produced and distribute them uniformly [3, 7, 22]. This approach needs to analyse the entire temporary data and write identifiers to global memory. The relative cost of load balancing increases with the sparsity of the input

Algorithm 1: Global Load Balancing

```
1 $a \leftarrow row\_ptr[tid]$
2 $b \leftarrow row\_ptr[tid+1]$
3 $blocka \leftarrow \textbf{divup}(a, \textbf{NNZ\_PER\_BLOCK})$
4 $blockb \leftarrow (b-1)/\textbf{NNZ\_PER\_BLOCK}$
5 while $blocka \leq blockb$ do
6     $blockRowStarts[blocka] \leftarrow tid$
7     $blocka \leftarrow blocka + 1$
```

matrices. According to our analysis and the cost break down given by Nagasaka et al. [22], load balancing can consume up to 30% of the overall runtime for very sparse matrices.

To tackle the drawbacks of both strategies, we propose a simple global load balancing scheme and defer the fine grained control to the next stage. Our global load balancing splits the non-zeros of **A** uniformly, assigning the same number to each thread block. In this way, memory requirements from **A** are static. However, the actual workload for each block varies based on the intermediate products generated.

While a static splitting of **A**'s non-zeros does not require processing, the second stage still needs to associate each entry from **A** with a row. To provide this information, global load balancing in parallel checks **A**'s row pointers and writes the required information into an auxiliary array, as outlined in Algorithm 1. The cost of this step is negligible compared to enumerating temporary products.

3.2 Adaptive Chunk-based ESC

Each thread block executing the second stage is assigned an equally sized portion of **A** and performs SpGEMM with **B**. Depending on the sparsity pattern, it can produce any number of output chunks, each representing a partial result of **C**. We propose an adaption of ESC due to its desirable properties to perform SpGEMM. First, after the expansion of the temporary products, every thread performs identical work independent of which row the data comes from. Second, sort can efficiently be implemented within a block, using Radix sort [21]. Third, a stable sort algorithm always yields identical floating point results, free of the problematic scheduling-based effects encountered when using hashing.

The downside of ESC is that sorting intermediate data is more costly than sorting the output data. However, dealing with only a chunk of data at once and keeping all data local, the cost is significantly lower than sorting all temporary elements of the entire product $\mathbf{A} \cdot \mathbf{B}$. While other local ESC approaches have been proposed before [7, 9, 20], they operate on individual rows or a fixed number of temporary products and then always go to global memory. Our approach completely ignores row boundaries and performs multiple iterations of ESC locally, dynamically splitting off completed rows. We only move to global memory after completion or when data cannot be compacted sufficiently.

3.2.1 Fetch A

The first step in AC-ESC fetches non-zeros and column ids of **A** required by the thread block, using a coalesced read pattern. We store this data in scratchpad memory to be available throughout the entire stage. While coalesced loading of **A** is less important for denser matrices where the loads from **B** dominate, it is important for very sparse matrices where loads from **B** are similar in count. In addition, the row ids for all non-zeros in **A** are needed. As **NNZ_PER_BLOCK** is constant, we can reduce the bit length of these row ids by locally remapping them: Using a dictionary we use the index of the first non-zero in that row as local row id.

3.2.2 Local Work Distribution

The most important component in AC-ESC is the local work distribution. As the number of intermediate products processed by a block of threads varies, we have to dynamically decide which elements to load to exactly fill up the available resources. While inspecting **B** for global load balancing is costly, inspecting **B** now comes with little additional cost as it needs to be accessed anyway. Thus, after loading each column index of **A**, we retrieve the number of elements to be fetched from **B** for every entry in **A**.

To determine which elements to load, we propose an expanding work distribution that dynamically supplies local ESC with data. The work distribution offers three methods: `placework`, `size`, `receivework` outlined in Algorithm 2. `placework` receives the number of temporary products that will be generated for each entry in **A**. A prefix sum over that data yields the expansion of temporary products up to a specific entry in **A**. `size` queries the overall sum, *i.e.*, how many elements are still to be processed. `receivework` can be called with a desired number of elements to be drawn from the work distribution (*Consume*). It can deliver multiple elements (N) to each thread. We typically use 8.

To perform the work assignment, we determine in parallel the offset of the first temporary product of each row from **A** (line 17-19). Marking this product (line 20) and performing a max prefix scan over the data (line 24) assigns the corresponding row id (A_{res}) to all *Consume* temporary products. To ensure data is loaded in a coalesced manner, we interleave the temporary products between threads, which is commonly known as moving from a block layout to stripped layout(line 25). Comparing the output id (c) and the cumulative sum up to the first element of the respective row ($WDState[A_{res}[i]]$), the local offset in the row can be computed.

Instead of using the local offset directly, we reverse the order using the offset from the next row $WDState[A_{res}[i]+1]$ and count down (line 29). In this way, if the work distribution splits a row in **B**, we take entries from the end of the row and thus simply act like the row is shorter in the next iteration of ESC. Finally, we reduce all cumulative counts by the number of elements drawn from the work distribution

Algorithm 2: Local Work Distribution

```
1  ScratchPad $WDState$[NNZ_PER_BLOCK]
2  Function placework($elements$[NNZ_PER_THREAD])
3      blockPrefixSumIncl($elements, elements$)
4      for $i \leftarrow 0$ to NNZ_PER_THREAD do
5          $WDState$[tid · THREADS + $i$ + 1] $\leftarrow$ $elements[i]$
6      $WDState[0] \leftarrow 0$
7      syncThreads()
8  Function size()
9      return $WDState$[NNZ_PER_BLOCK]
10 Function receivework($N, Consume$)
11     ScratchPad Offsets[$N$ · THREADS]
12     $A_{res}[N]$
13     $B_{res}[N]$
14     clear($Offsets$)
15     syncThreads()
16     for $i \leftarrow 0$ to NNZ_PER_THREAD do
17         $a \leftarrow WDState$[$i$ · THREADS + tid]
18         $a_n \leftarrow WDState$[$i$ · THREADS + tid + 1]
19         if $a < Consume$ and $a \neq a_n$ then
20             $Offsets[a] \leftarrow i$ · THREADS + tid
21     syncThreads()
22     for $i \leftarrow 0$ to $N$ do
23         $A_{res}[i] \leftarrow Offsets$[$N$ · tid + $i$]
24     blockMaxScanIncl($A_{res}, A_{res}$)
25     blockedToStripped($A_{res}, A_{res}$)
26     for $i \leftarrow 0$ to $N$ do
27         $c$ = tid + $i$ · THREADS
28         if $c < Consume$ then
29             $B_{res}[i] \leftarrow WDState[A_{res}[i] + 1] - c - 1$
30         else
31             $A_{res}[i] \leftarrow \emptyset$
32             $B_{res}[i] \leftarrow \emptyset$
33     syncThreads()
34     for $i \leftarrow 0$ to NNZ_PER_THREAD do
35         $j \leftarrow$ tid + $i$ · THREADS + 1
36         $WDState[j] \leftarrow$ max(0, $WDState[j] - Consume$)
37     syncThreads()
38     return $A_{res}, B_{res}$
```

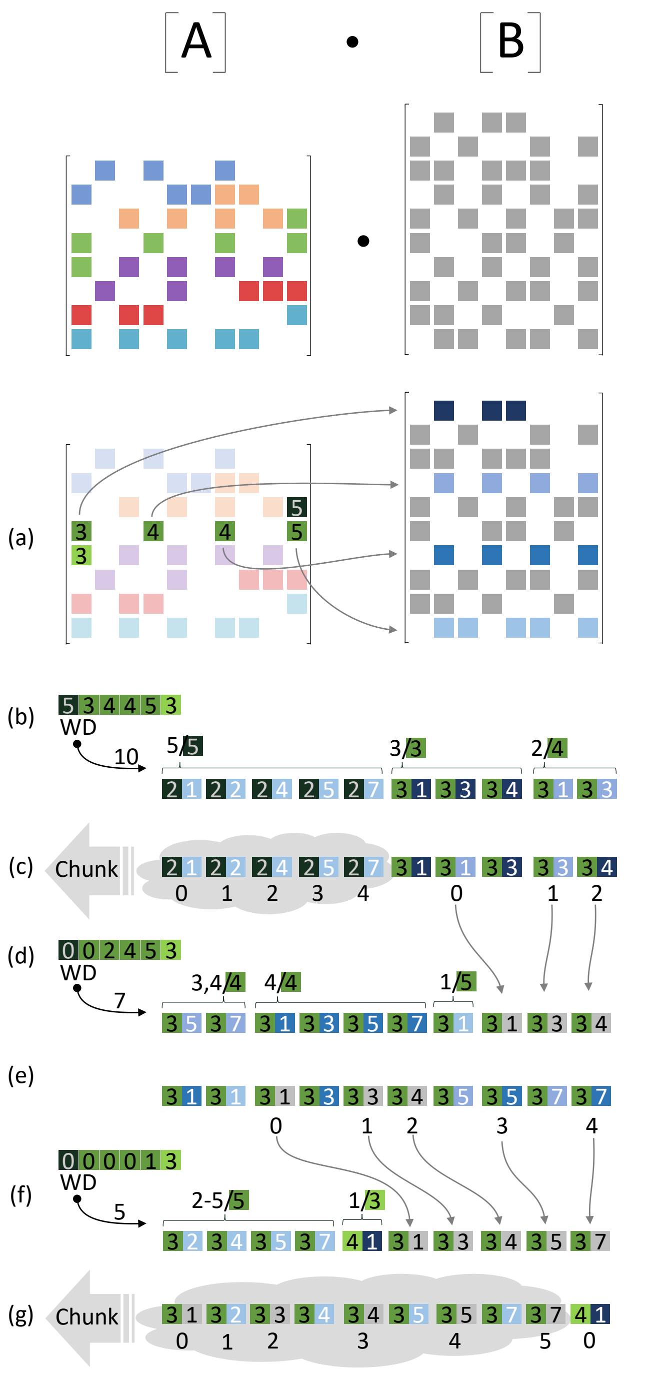


Figure 3. Example of our work distribution-driven local ESC: (a) After global load balancing over **A**'s non-zeros (colors), we fill the work distribution with the row lengths each entry from **A** references in **B**. (b) Taking 10 elements from the work distribution, we expand, sort and compact. (c) A first chunk for row id 2 is split off to global; the remaining 3 elements are kept locally for another iteration. (d) 7 elements are drawn from the work distribution. (e) All elements are kept for another iteration. (f) 5 elements are needed. (g) A complete chunk for row 3 is produced.

(line 36) and return identifiers for all temporary products (A_{res}, B_{res}). The only state the work distribution needs to keep is $WDState$. As AC-ESC might run out of memory during chunk allocation, it needs to be able to continue after an allocation round trip to the host. Thus, the work distribution also needs to support restarts. In case of a restart, we simply store the number of already consumed elements to global memory. When the block continues, we initialize the work distribution as usual, but immediately reduce the workload accordingly, *i.e.*, we execute line 36 with the stored count.

3.2.3 Local ESC

Driven by the work distribution, we perform multiple iterations of local ESC, as indicated in Figure 3. Using the output from the work distribution, every thread loads its assigned element from **B** and multiplies it with the previously loaded value from **A** to complete the expansion. This yields coalesced memory access for elements of the same row in **B**. Of course, as different rows from **B** might be loaded—depending on the column ids of **A**—the exact memory access pattern still depends on the input data.

After the expansion, the intermediate products are moved into radix sort, using their row and column id for sorting. The runtime of radix sort is proportional to the bit length being sorted, and thus reducing sort bit length is important for ESC [9]. While previous work followed a static approach to bit reduction, our approach is more aggressive and completely dynamic: As mentioned earlier, we bound the maximum range of row ids using a dictionary. Unfortunately, the same approach is not applicable to column ids as it would require knowledge of all unique column ids. However, we can bound the range by tracking the minimum and maximum id for all entries we fetch from **B** and thus reduce the number of bits. We do the same for the row ids on top of the dictionary, further reducing their bit range. Thus, the sorting effort adapts to the input data present.

The reduction of the number of sorted bits not only reduces the sorting effort, but also the register requirements. Keeping in mind that the row ids in the worst case require $\log_2$(**NNZ_PER_BLOCK**) bits and the column id of **B** is limited by **B**'s dimension, we choose a 32 bit or 64 bit integer. For example, for a block size of 256 threads and 2 **NNZ_PER_THREAD**, we need 9 bits; thus 32 bit integers are sufficient for matrices up to 8.4 million columns (23 bits).

In the compaction step, we are not only interested in combining the data, but also in the number of entries in every row and how to write the chunk to memory. We perform all three tasks within a single prefix scan using special operators and state. At first, we determine whether neighbouring elements have the same sorting key, *i.e.*, should be combined, and whether their row id bits match, *i.e.* they are from the same row. Every thread can trivially perform this operation for all elements it holds in registers.

For cross-thread communication, we use scratchpad memory. We then encode both facts as individual bits in a 32 bit integer (row ends as the 17th bit (orange) in Algorithm 3; the end of a combine sequence using the first bit (purple)). We split the remaining 30 bits in half to count the elements in each row (green) and overall compacted elements (blue). For each element that ends a combine sequence, we initialize each of the 15 bit counters to 1.

The scan uses the state information to decide whether to reset the accumulation and/or counting bits as outlined in Algorithm 3.

Algorithm 3: Compaction Scan Operator

```
            // initial state for the scan operation
      // end row 0b0000 0000 0000 0011 0000 0000 0000 0011
     // end comp 0b0000 0000 0000 0010 0000 0000 0000 0011
         // none 0b0000 0000 0000 0000 0000 0000 0000 0000
1  Function CombineScanOperator($(a, b)$)
2      if equalRow $(a_{key}, b_{key})$ then
3          $state \leftarrow a_{state}$ & 0xFFFE
4      else
5          $state \leftarrow a_{state}$ & 0xFFFEFFFE
6      if $a_{key} = b_{key}$ then
7          $n_{value} \leftarrow a_{value} + b_{value}$
8      else
9          $n_{value} \leftarrow b_{value}$
10     $n_{key} \leftarrow b_{key}$
11     $n_{state} \leftarrow state + b_{state}$
12     return $n$
```

After completion of the scan, the state bits can still be queried to identify the compacted elements as well as row ends. Additionally, the other bits hold information about each element's position in the chunk as well as the local offset in the row. Using the row counts, we update the global counter for each row, which will serve later on for computing the row pointer array and memory requirements of **C**.

Previous approaches to local ESC assign the exact number of temporary elements that can be handled to a block [7] and go to global memory after ESC. The resulting data however may still need to be combined with temporary results from multiple other blocks. If we directly wrote our results to chunks in global memory, we would face the same issue. Thanks to the flexibility of our work distribution, we keep the results around for another iteration of ESC, combining the temporary results with new data from **B**.

By keeping the temporary results for the next iteration of ESC and reducing the number of elements drawn from the work distribution accordingly, we reduce the global memory traffic significantly. While avoiding additional chunks is reasonable, keeping elements from multiple output rows for the next iteration is not, as all but the last row are already completed. Conversely, it does not make sense to have a few elements of a new row at the end of a chunk, as these will require merging in a later stage. Thus, we only keep the last row for the next round and write other rows to a global chunk.

3.2.4 Chunk Management

When we decide to write a chunk of **C** to global memory, a thread of the block uses the compaction result to compute the chunk size and allocates a chunk from the pool. To this end, it increments an atomic counter by the chunk size. To write both the column ids and values to the chunk, we take a

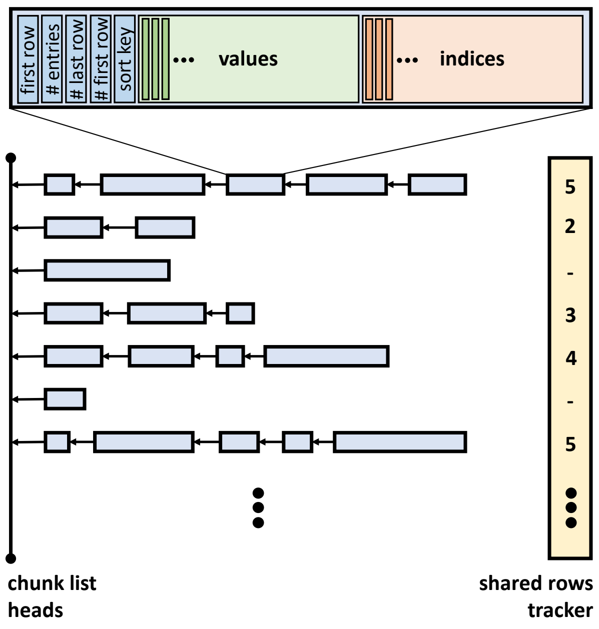


Figure 4. The chunk lists are constructed as per row linked lists while the shared rows tracker holds per row chunk counts if merging is required for a certain row.

compacting round trip through scratchpad memory to ensure coalesced writes to global memory.

When a chunk is generated, we update the restart information for this thread block. If a complete chunk is written, information about how many elements have been drained from the work distribution is sufficient. If the last row is kept for the next iteration, we write the row id as restart information. The next time the block is launched it can then completely ignore these rows when loading data from **A**.

In addition to the data needed for **C**, each chunk holds its starting row, element count and number of elements in the first and last row. To perform chunk copy in parallel in the final stage, we keep an array of pointers to all chunks. Using a full pointer allows chunks to be placed in memory arbitrarily. Thus, expanding the chunk pool is as easy as adding another memory region to be used as pool.

Finally, we require information about which chunks have data for the same row to start merging. To identify all chunks that contribute to one output row, we construct a linked list of chunk pointers for every row, with a list head being available for every row, as shown in Figure 4. The list insertion uses an atomic exchange operation on the list heads, making the list order dependent on the hardware scheduler. To ensure bit-stable results, we further store a chunk identifier from the block id and a per-block running chunk number, which yields a global ordering of chunks. To increase the efficiency of chunk merging, we use one bit of the list heads to indicate whether there is only a single chunk in the list. Upon adding a second chunk to the list, we insert the row id into an array holding all rows with two or more chunks. This array serves as the basis for identifying how many merge blocks need to be started in the next stage.

3.3 Chunk Merging

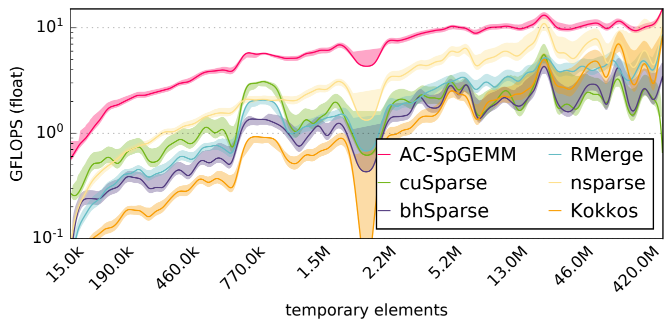
After AC-ESC, the merge stage combines rows shared between chunks to generate the final result. To guarantee a deterministic merge order, we perform an initial sort of the chunks based on their global chunk order. This sort is negligible compared to the merge itself. As any number of elements may need to be merged, one could launch a single block for each shared row. However, typically, a shared row is covered by two chunks, *i.e.*, because global load balancing splits the entries of one row across two blocks. In this case, the number of entries in both chunks may be low, potentially wasting resources when using a complete block.

Similarly to AC-ESC, we want to work on multiple rows to fully use the available resources. To this end, we run a prefix scan over all shared rows, using the row count that we summed up atomically for all rows during AC-ESC. For shared rows, this count represents the number of remaining intermediate products.

Throughout the scan we combine row range identifiers, if the sum of their respective elements does not overflow the number of elements we can handle in one block. This approach may not fill up blocks completely, but yields significantly better resource usage than a one-block-per-row strategy. When launching these *Multi Merge* blocks, we once again build on our work distribution and execute the remaining steps of our AC-ESC, creating new chunks for all merged rows. At the same time, we set the row count for each shared row to the correct value after merging.

The scope of *Multi Merge* is limited to rows that were split over two chunks. We therefore propose two additional algorithms to deal with rows yielding more than two chunks: *Path Merge* and *Search Merge*. The former is applicable up to a predefined number of chunks, while the latter can handle an arbitrary number. We decide which algorithm to use based on the length of each row's chunk list.

Search Merge uses binary search sampling in all chunk column ids to find overlapping ranges that can be handled at once. At first, we compute the minimum and maximum column id over all involved chunks. Then, we uniformly sample this range, according to ((max − min)/**THREADS**), assigning one thread to each sample. Using binary search, every thread finds the next higher column id in all chunks and computes the sum over all elements that are below across all chunks. The thread with the largest sum that still fits into the available resources, delivers the data to be merged. Using AC-ESC on that data yields the first part of the merged row. Reducing the count of all samples by the number of consumed elements yields the next cut and so on. In case the sampling is too coarse we sub-sample the range.



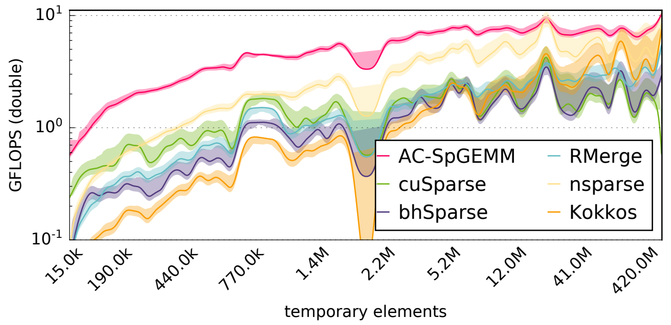


Figure 5. Trend line of the SpGEMM performance for all tested methods over highly sparse matrices (average row length ≤ 42) from the SuiteSparse collection. Line thickness indicates variance.

Path Merge avoids global memory binary search by placing samples uniformly over the entries of every chunk. For each sample we fetch the column id and sort them across the entire block, while carrying the sample number along with the sort. Next, we perform a custom scan over the sorted data to find the correspondences between samples from different chunks, *i.e.*, identify possible paths through all chunks.

As every sample originally only holds the sample number from its own chunk, we set the other counts to zero, using only as many bits as necessary to represent each sample number. The max scan over these individual bit ranges delivers the combined merge path, *i.e.*, the matching cut through each chunk. For each path, we compute the number of temporary elements from the combined sample locations and chunk sizes. Choose the one that fits into memory, we run AC-ESC. The stored paths are again used for the next iteration.

All three approaches produce new chunks, and thus must support restarts. In *Multi Merge*, a restart simply starts from scratch, as it has only one iteration. On the other hand, both *Search Merge* and *Path Merge* store the last used sample and therefore a restart therein equals sampling a reduced range.

3.4 Long Rows

Processing long rows that exceed the available local resources during AC-ESC would load and sort them without any change and write them to the chunk pool. To avoid these unnecessary computations, we identify long rows during *Fetch A* and immediately create a chunk that only points to the data in **B** and attach the factor from **A**.

By removing the row from the work distribution, we avoid processing it during AC-ESC, but have to make slight modifications to this stage. If the long row has to be merged with values otherwise placed in the middle of a chunk, we break up chunks at the position of the long row to allow for merging afterwards in the merge stage.

3.5 Output Matrix and Chunk Copy

Once all chunks have been finalized, generating the final result is straightforward: A device-wide prefix sum over the row counts yields the row pointer array and **C**'s memory requirement for allocation of the values and column id arrays. Then, in parallel, we iterate over all chunks and copy their data to the newly allocated **C**. Each chunk uses a complete block of threads to copy data in a coalesced fashion.

4 Evaluation

To provide a realistic assessment of the performance of our approach, we benchmarked the entire SuiteSparse matrix collection [11], which contains more than 1800 unique matrices of non-trivial size ($\geq 10^4$ NNZ) from various application domains with different matrix characteristics. Very small matrices ($\leq 10^4$ NNZ) are excluded as they do not provide sufficient parallelism for execution on the GPU and thus CPU implementations are typically faster. From about 10^4 NNZ upwards, our approach outperforms state-of-the-art CPU implementations [14] on a consumer grade CPU of similar cost (Intel Xeon E5-2630 16 GB of memory). We compare our approach to cuSparse [23], bhSparse [20], RMerge [17], nsparse [22], and Kokkos [14]. All approaches work directly with CSR and were compiled with CUDA Toolkit 10.0. We compute $\mathbf{A} \cdot \mathbf{A}$ for square matrices and $\mathbf{A} \cdot \mathbf{A}^T$ for non-square, where we precompute $\mathbf{A}^T$. As test platform we use an Intel i7 7700 CPU at 3.60GHz and an NVIDIA Titan Xp (compute capability 6.1).

Our algorithm is written in CUDA and uses a block size of 256/512 non-zeros for global load balancing, sorts 8 elements per thread and keeps up to 4 elements per thread from one iteration to the next. We use conservative memory estimates for all helper data structures. For the initial chunk pool, we rely on a simplistic memory estimate S of **C**, using the average row length as a measure of row overlaps, *i.e.*, pretend matrices have the same number of uniformly distributed elements in each row. More precisely, for **A** of size $n_A \times m_A$, the average row length is given by $\bar{a} = |A|/n_A$, where $|A|$ indicates the number of non-zeros, and the estimated probability for a collision is $p_a = \bar{a}/m_A$. For the product **AB**, the memory estimate is given as $S \approx n_A \cdot \bar{b} \cdot (1-(1-p_b)^{\bar{a}})/p_b$. We multiply this factor by 1.2 to account for the chunk meta

		≈1500 highly sparse matrices ($\bar{a} \leq 42$)					≈300 denser matrices ($42 < \bar{a}$)				
		speed up of AC-SpGEMM			better than	best	speed up of AC-SpGEMM			better than	best
		min	max	h. mean	AC-SpGEMM		min	max	h. mean	AC-SpGEMM	
float	cuSparse[†]	0.78	614.17	4.01	0%	0%	0.55	674.51	1.66	6%	2%
	bhSparse	1.00	96.60	5.56	0%	0%	0.71	429.04	1.77	9%	0%
	RMerge	0.60	23.50	3.73	0%	0%	0.34	10.21	2.00	3%	0%
	nsparse[†]	0.26	76.80	2.29	3%	3%	0.19	33.02	0.76	66%	60%
	Kokkos[†]	0.48	767.71	6.27	1%	1%	0.34	148.27	1.17	45%	7%
	AC-SpGEMM	-	-	-	-	96%	-	-	-	-	31%
double	cuSparse[†]	0.51	470.44	3.78	0%	0%	0.62	608.74	1.50	10%	3%
	bhSparse	1.00	88.60	5.12	0%	0%	0.71	387.04	1.56	17%	0%
	RMerge	0.45	24.90	3.70	0%	0%	0.40	9.50	1.81	5%	1%
	nsparse[†]	0.31	43.06	2.01	6%	5%	0.18	28.83	0.67	70%	61%
	Kokkos[†]	0.37	582.87	5.09	2%	1%	0.30	120.58	0.92	53%	10%
	AC-SpGEMM	-	-	-	-	94%	-	-	-	-	26%

Table 1. Relative speedup of AC-SpGEMM over competing approaches and percentage where the approaches achieved better performance than AC-SpGEMM / achieved the best performance. AC-SpGEMM dominates the performance for highly sparse matrices and still achieves the best performance for about 1/3 of denser matrices. [†] does not produce bit-stable results.

data and divergences from the average row length and apply a lower bound of 100MB. In case the estimate is not sufficient, the restart functionality will increase the chunk pool.

4.1 Runtime Overview

To better analyze the runtime performance, we split the evaluation into highly sparse and denser matrices, with a split factor of 42 non-zeroes per row, classifying 80% of the matrices in SuiteSparse as highly sparse (see also Figure 1). Summary plots for SpGEMM on highly sparse matrices are shown in Figure 5, and relative speedups against the evaluated methods for all matrices in Table 1. For highly sparse matrices, our approach clearly dominates performance, achieving average speedups of at least 2× over other approaches. For denser matrices, nsparse achieves the best performance, with an average speedup of 1.32/1.49 over AC-SpGEMM, with AC-SpGEMM being on par with Kokkos.

Over the entire data set, our approach achieves an average speedup of 3.27/3.05, 4.17/3.80, 3.30/3.21, 1.74/1.53, and 3.76/3.02, over cuSparse, bhSparse, RMerge, nsparse, and Kokkos, respectively. The trend plots and table already indicate the strengths and weaknesses of our approach. While ESC leads to deterministic, bit-stable results, the search and merge overhead increases as the number of non-zeros per row increase. Thus, even though our approach performs multiple iterations of ESC in efficient on-chip memory, our approach loses ground to hash-based approaches, like nsparse, for denser matrices. However, for very sparse matrices the overhead of ESC is not severe and the advantages of local scheduling and single-run chunk generation shine. Compared to other bit-stable approaches, AC-SpGEMM overall clearly achieves the best performance.

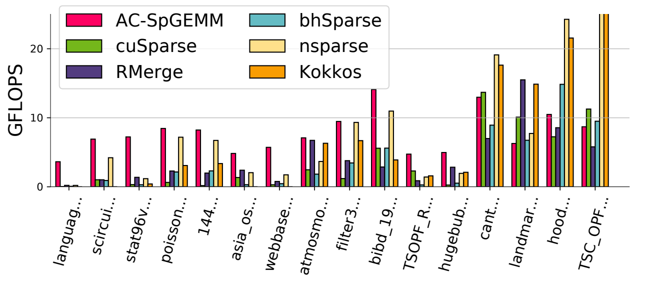


Figure 6. Double precision performance for commonly benchmarked matrices and additional cases for which our approach is bested by others. Note that nsparse achieves 45 GFLOPS for the last matrix.

4.2 Runtime Details

A performance overview for commonly benchmarked matrices from various fields and selected matrices that are difficult for our approach are provided in Figure 6; matrix statistics are given in Table 2. The most difficult scenarios for our approach include `TSC_OPF_1047`, `cant`, and `hood`, which, in spite of having strongly different structure, all feature a large average row length, produce a high number of intermediate products and compact them significantly (up to a compaction factor of 150). The cost of ESC is simply too high in comparison to hashing, even while keeping hundreds of iterations in local memory. For `TSC_OPF_1047`, nsparse achieves the highest speedup over AC-SpGEMM: 5×. Another matrix with many temporary products but a lower compaction factor is `landmark`. Interestingly, due to its very special structure it is one of the few cases where RMerge takes the lead.

	A					C			
	rows	cols	nnz	len	max	nnz	len	max	temp
language	0.40	0.40	1.22	3.0	11.5k	4.61	11.6	32.0k	5.5
scircuit	0.17	0.17	0.96	5.6	353	5.22	30.5	1.9k	8.7
stat96v2	0.03	0.96	2.85	98.1	3.2k	0.35	12.1	1.6k	8.7
poiss…	0.01	0.01	0.35	26.1	110	2.96	218.8	584	11.8
144	0.14	0.14	2.15	14.9	26	10.42	72.0	116	33.0
asia_osm	11.95	11.95	25.42	2.1	9	42.75	3.6	24	56.9
webb…	1.00	1.00	3.11	3.1	4.7k	51.11	51.1	12.4k	69.5
atmos…	1.49	1.49	10.32	6.9	7	36.49	24.5	25	71.6
filter3D	0.11	0.11	2.71	25.4	112	20.16	189.4	550	86.0
bibd_19_9	0.01	0.09	3.3	19.4k	19.4	0.03	171.0	171	119.7
TSOPF…	0.04	0.04	16.17	424.2	983	74.32	1.9k	3.3k	128.0
hugebu…	21.20	21.20	63.58	3.0	3	132.69	6.3	7	190.7
cant	0.06	0.06	4.01	64.2	78	17.44	279.3	375	269.5
landmark	0.07	0.00	1.15	16.0	16	101.82	1.4k	1.6k	549.2
hood	0.22	0.22	10.77	48.8	77	34.24	155.3	231	562.0
TSC_O…	0.01	0.01	2.02	247.8	1.5k	8.83	1.1k	3.5k	1352.4

Table 2. Matrix overview: values in millions except for row statistics (average length and maximum row length).

	helper	chunk	used	%	u/o	R	mpL
language	15.05	100.00	58.23	50.88%	1.10	0	98.73%
scircuit	13.09	100.00	63.15	55.18%	1.06	0	97.68%
stat96v2	29.91	122.11	6.68	5.47%	1.65	0	97.98%
poisson3Da	6.06	129.96	42.61	32.78%	1.26	0	91.02%
144	27.04	460.75	129.89	28.19%	1.09	0	98.38%
asia_osm	416.96	1091.05	497.10	45.56%	1.02	0	99.77%
webbase-1M	100.03	710.80	605.79	85.23%	1.04	3	98.05%
atmosmodl	130.69	1033.36	435.36	42.13%	1.04	0	99.70%
filter3D	38.06	983.23	271.19	27.58%	1.18	0	98.78%
bibd_19_9	1.06	100.00	19.20	16.78%	57.38	0	99.21%
TSOPF_…	605.60	3044.69	1593.56	52.34%	1.87	0	99.23%
hugebub…	928.54	1991.35	1545.96	77.63%	1.02	0	99.45%
cant	66.76	3072.00	276.51	9.00%	1.39	0	99.48%
landmark	228.97	6144.00	3333.51	54.26%	2.86	1	97.29%
hood	162.85	3072.00	508.58	16.56%	1.30	0	99.73%
TSC_O…	37.99	3072.00	310.41	10.10%	3.07	0	99.03%

Table 3. Overall memory consumption in MB for helper data structures, chunk pool, actual chunk pool used, and used chunk memory relative to the output matrix (u/o); as well as number of restarts (R) and lowest multiprocessor load (mpL).

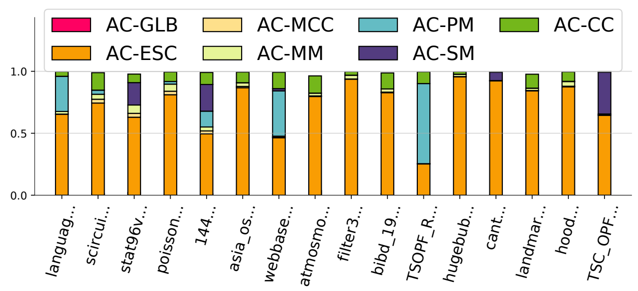


Figure 7. Relative runtime of the different steps of our approach: global load balancing (GLB), AC-ESC, Merge Assignment (MCC), Multi Merge (MM), Path Merge (PM), Search Merge (SM), and Chunk Copy (CC).

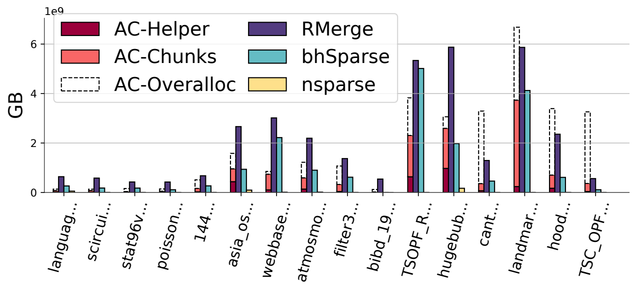


Figure 8. Memory consumptions. Due to our simplistic memory estimate, the effectively used memory is very small compared to the allocation.

As the compaction factor gets lower, our approach is competitive, leading the performance for many commonly tested matrices, in various domains, such as fluid dynamics (`poisson3Da`), linear programming (`stat96v2`), or graphs (`webbase-1M`). This performance edge is independent of the size of the matrix (compare `hugebubbles-00020` and `144`). Also, local dense areas (`TSOPF_RS_b2383`), specific structures (`hugebubbles-00020`), individual long rows (`webbase-1M`), non-square matrices (`stat96v2`), and very long rows (`bibd_19_9`) are handled efficiently by our approach. The best speedup is achieved by our approach if the matrix is highly sparse, but has few long rows (`language`), where our efficient ESC and the special treatment of long rows go hand in hand, outperforming the best other approach by 20×.

The timing breakdown of our approach in Figure 7 shows that we operate under ideal conditions for many matrices, spending most time in AC-ESC within on-chip memory.

For matrices with long rows (`TSOPF_RS_b2383`), a considerable amount of time is spent on *Merge*. Although *Multi Merge* handles more shared rows than *Search* and *Path Merge* in all cases, its time is negligible, as every block handles many rows. Assigning the merge cases and copying the final matrix make up between 1-20% of the runtime; global load balancing is negligible, underlining the efficiency of our approach.

At the same time, multi processor load is virtually perfect in all cases (*cf.* last column of Table 3), indicating that pairing our global and local load balancing with the GPU hardware scheduler achieves ideal workload distributions.

4.3 Memory Consumption

Our memory consumption is detailed in Table 3 and compared to others in Figure 8. nsparse requires hardly any additional memory.

We allocate similar memory as RMerge and bhSparse, of which we typically only use a fraction (% in Table 3). The fact that the used chunk memory is only slightly higher than the memory required for **C** (u/o) in Table 3, shows that local iterations of ESC essentially produce completed chunks of the output matrix (note that our lower bound of 100MB leads to the high value for `bibd_19_9`). This highlights the advantages of our local work distribution. Our chunk pool estimate is conservative in most cases and only few matrices require restarts, *cf.* Table 3.

Although we require three restarts for `webbase-1M`, we achieve a 4.4× speedup over the best other approach, indicating that restart is efficient. To evaluate the cost of restarts, we reduced the chunk pool size for `webbase-1M`, where we measured a runtime of 22.0, 23.6, 24.5, 26.6, 30.8, and 39.7, and 48.6*ms* for 0, 3, 5, 10, 21, 42, and 63 restarts, respectively. Even with 63 restarts we still beat *nsparse* by a factor of 2×. Additionally, a less conservative chunk estimate could significantly reduce memory with little impact on performance due to restarts.

4.4 Summary

Overall, it can be noted that AC-SpGEMM is the fastest approach when bit-stable results are required for virtually all tested matrices (RMerge is better in 1% of cases). AC-SpGEMM is the fastest approach for very sparse matrices. For matrices with many temporary products and large compaction factors, ESC strategies tend to fall behind hash-based approaches like nsparse as the per-product cost is simply too high. Nevertheless, across the entire test suit, AC-SpGEMM takes the performance lead in 83% of all cases.

5 Conclusion

On massively parallel processors such as GPUs, any performance gains require bringing fine grained parallelism forward. AC-SpGEMM achieves this goal by a comprehensive take on the problem. Our main contribution is a fully adaptive local work distribution, allowing for multiple iterations of local ESC avoiding costly global memory round trips. Paired with a novel, adaptive chunk management approach, special case handling of long rows, and a series of optimizations, AC-SpGEMM forms a highly efficient complete SpGEMM solution, which achieves bit-stable results. Experimental results on 2000 matrices across various fields reveal dominating performance for highly sparse matrices. Only matrices with very large numbers of temporary products can be handled more efficiently using the most recent hash-based alternative—which is not bit-stable.

An obvious improvement for our approach is reducing the overallocation of chunk memory. Furthermore, extending the adaptive behaviour of our chunk-based approach to choose between alternative approaches (ESC, hashing, merging) depending on the load currently seen by the work distribution may lead to a further improvement of performance in those scenarios where other strategies shine.

Our approach is open source and can be downloaded from ACM DL.

Acknowledgment

This research was supported by the German Research Foundation (DFG) grant STE 2565/1-1, and the Austrian Science Fund (FWF) grant I 3007. The GPU for this research was donated by NVIDIA Corporation.

References

[1] Kadir Akbudak and Cevdet Aykanat. 2017. Exploiting Locality in Sparse Matrix-Matrix Multiplication on Many-Core Architectures. *IEEE Transactions on Parallel and Distributed Systems* 28, 8 (Aug 2017), 2258–2271. https://doi.org/10.1109/TPDS.2017.2656893

[2] Rasmus R. Amossen, Andrea Campagna, and Rasmus Pagh. 2010. Better Size Estimation for Sparse Matrix Products. In *Proceedings of the 13th International Conference on Approximation, and 14 the International Conference on Randomization, and Combinatorial Optimization: Algorithms and Techniques (APPROX/RANDOM'10)*. Springer-Verlag, Barcelona, Spain, 406–419.

[3] Pham N. Q. Anh, Rui Fan, and Yonggang Wen. 2016. Balanced Hashing and Efficient GPU Sparse General Matrix-Matrix Multiplication. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*. ACM, New York, NY, USA, Article 36, 12 pages. https://doi.org/10.1145/2925426.2926273

[4] Grey Ballard, Alex Druinsky, Nicholas Knight, and Oded Schwartz. 2016. Hypergraph Partitioning for Sparse Matrix-Matrix Multiplication. *ACM Trans. Parallel Comput.* 3, 3, Article 18 (Dec. 2016), 34 pages. https://doi.org/10.1145/3015144

[5] Nathan Bell, Steven Dalton, and Luke N. Olson. 2012. Exposing Fine-Grained Parallelism in Algebraic Multigrid Methods. *SIAM Journal on Scientific Computing* 34, 4 (2012), C123–C152. https://doi.org/10.1137/110838844

[6] Aydin Buluc and John R. Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, Miami, FL, USA, 1–11. https://doi.org/10.1109/IPDPS.2008.4536313

[7] Steven Dalton, Sean Baxter, Duane Merrill, Luke Olson, and Michael Garland. 2015. Optimizing Sparse Matrix Operations on GPUs Using Merge Path. In *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, Hyderabad, India, 407–416.

[8] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. 2014. Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. http://cusplibrary.github.io/ Version 0.5.0.

[9] Steven Dalton, Luke Olson, and Nathan Bell. 2015. Optimizing Sparse Matrix-Matrix Multiplication for the GPU. *ACM Trans. Math. Softw.* 41, 4, Article 25 (Oct. 2015), 20 pages. https://doi.org/10.1145/2699470

[10] Timothy A. Davis. 2017. SuiteSparse: A Suite of Sparse matrix packages. http://www.cise.ufl.edu/ davis/.

[11] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1 (Dec. 2011), 1:1–1:25.

[12] Julien Demouth. 2012. *Sparse matrix-matrix multiplication on the GPU.* Technical Report. NVIDIA.

[13] Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam. 2017. Performance-portable sparse matrix-matrix multiplication for many-core architectures. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, Lake Buena Vista, FL, USA, 693–702. https://doi.org/10.1109/IPDPSW.2017.8

[14] Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam. 2018. Multithreaded sparse matrix-matrix multiplication for many-core and GPU architectures. *Parallel Comput.* 78 (oct 2018), 33–46. https://doi.org/10.1016/j.parco.2018.06.009

[15] John R. Gilbert, Cleve Moler, and Robert Schreiber. 1992. Sparse Matrices in MATLAB: Design and Implementation. *SIAM J. Matrix Anal. Appl.* 13, 1 (1992), 333–356. https://doi.org/10.1137/0613024

[16] Oded Green, Robert McColl, and David A. Bader. 2012. GPU Merge Path: A GPU Merging Algorithm. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS '12)*. ACM, New York, NY, USA, 331–340.

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

[18] Fred G. Gustavson. 1978. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.* 4, 3 (Sept. 1978), 250–269. https://doi.org/10.1145/355791.355796

[19] Rakshith Kunchum, Ankur Chaudhry, Aravind Sukumaran-Rajam, Qingpeng Niu, Israt Nisa, and P. Sadayappan. 2017. On Improving Performance of Sparse Matrix-matrix Multiplication on GPUs. In *Proceedings of the International Conference on Supercomputing (ICS '17)*. ACM, New York, NY, USA, Article 14, 11 pages. https://doi.org/10.1145/3079079.3079106

[20] Weifeng Liu and Brian Vinter. 2015. A Framework for General Sparse Matrix-matrix Multiplication on GPUs and Heterogeneous Processors. *J. Parallel Distrib. Comput.* 85, C (Nov. 2015), 47–61. https://doi.org/10.1016/j.jpdc.2015.06.010

[21] Duane Merrill. 2015. CUB: CUDA Unbound, a library of warp-wide, block-wide, and device-wide GPU parallel primitives.

[22] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. 2017. High-Performance and Memory-Saving Sparse General Matrix-Matrix Multiplication for NVIDIA Pascal GPU. In *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, Bristol, UK, 101–110. https://doi.org/10.1109/ICPP.2017.19

[23] NVIDIA. 2019. *The API reference guide for cuSPARSE, the CUDA sparse matrix library.* (v9.1 ed.). NVIDIA.

[24] Md. Mostofa A. Patwary, Nadathur R. Satish, Narayanan Sundaram, Jongsoo Park, Michael J. Anderson, Satya G. Vadlamudi, Dipankar Das, Sergey G. Pudov, Vadim O. Pirogov, and Pradeep Dubey. 2015. *Parallel Efficient Sparse Matrix-Matrix Multiplication on Multicore Platforms.* Springer International Publishing, Cham, 48–57.

[25] Ichitaro Yamazaki and Xiaoye S. Li. 2011. On Techniques to Improve Robustness and Scalability of a Parallel Hybrid Linear Solver. In *Proceedings of the 9th International Conference on High Performance Computing for Computational Science (VECPAR'10)*. Springer-Verlag, Berlin, Heidelberg, 421–434. http://dl.acm.org/citation.cfm?id=1964238.1964281

[26] Raphael Yuster and Uri Zwick. 2004. Detecting Short Directed Cycles Using Rectangular Matrix Multiplication and Dynamic Programming. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '04)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 254–260. http://dl.acm.org/citation.cfm?id=982792.982828

A Artifact Description Appendix: Adaptive Sparse Matrix-Matrix Multiplication on the GPU

A.1 Abstract

The following appendix provides the necessary information to acquire the framework and rerun the experiments used to evaluate the framework.

A.2 Description

A.2.1 Check-list (artifact meta information)

- **Data set:** Matrix data set found on SuiteSparse Matrix Collection (Formerly the University of Florida Sparse Matrix Collection)
- **Hardware:** Recent GPU hardware from NVIDIA, tested on NVIDIA GTX 1080TI, NVIDIA GTX TITAN X Pascal and NVIDIA GTX TITAN Xp
- **Output:** The output is provided in the output stream as well as in a *.csv* file
- **Experiment workflow:** Run *runall.bat/.sh* file to gather results for all matrices in a folder or run individual matrices through the framework
- **Experiment customization:** The number of iterations used for the timing measurements can be altered. A CPU implementation can be enabled as well to confirm the results of the framework output
- **Publicly available?:** ACM DL

A.2.2 How software can be obtained (if available)

The framework is downloadable from ACM DL.

A.2.3 Hardware dependencies

Recent GPU hardware from NVIDIA, tested on NVIDIA GTX 1080Ti, NVIDIA GTX TITAN X Pascal and NVIDIA GTX TITAN Xp. Remaining system specifications should have a comparatively small impact on performance, performance was tested on an Intel Core i7-7700 paired with 32 GB RAM on Ubuntu 16.04 LTS.

A.2.4 Software dependencies

- CMake 3.2 or higher
- CUDA 9.1 / 9.2 / 10.0
- C++14 compliant compiler, tested on:
  - MSVC (Visual Studio 2017)
  - GCC 6 / GCC 7
- CUB v1.8.0

A.2.5 Datasets

The framework itself can parse *Matrix Market Format (.mtx)* files (Matrix data set found on SuiteSparse Matrix Collection (Formerly the University of Florida Sparse Matrix Collection)) and upon first parsing a matrix in this format also performs a conversion into a binary format that will be used for consecutive runs, which greatly reduces loading times. These are stored with the *.hicoo* extension.

A.3 Installation

On Linux simply run the provided *setup.sh* script which will clone CUB, setup and build the project. On Windows, download and extract CUB into the folder *include/external*, then create a build folder in the top directory and use CMake to setup the project to build.

A.4 Experiment workflow

The framework can be operated in one of two modes:

- Single Matrix
  - In this setup the framework can be run for a single matrix and optionally confirm the resulting output matrix by comparing it to a host-based solution
- Complete testrun
  - A runall.bat/.sh file is provided that consecutively calls the framework with all matrices provided in a folder, this was used to run the large testcases

The output is provided in the output stream as well as in a separate *.csv* file which includes all important matrix stats (rows, columns, nnz, average nnz per row, etc.) as well as timing measurements. This script is setup such that each test run is done as a separate process, such that failed launches do not impede launches after that. The output of the script are timing measurements and when enabling *Debug* within the framework (the template instantiation must have this enabled) also detailed measurements as well as memory measurements.

For all matrices in the testset the framework computes $C = A \cdot A$ (or $C = A \cdot A^T$ if the input matrix is not square) and measures the time required for the multiplication procedure, only the input matrix A is provided directly on the device to the procedure and the result matrix C is also returned as a device matrix. Conversion operators are provided to transfer matrices between host and device as well as convert the *COO* format to *CSR* if required.

A.5 Evaluation and expected result

To replicate the results gathered in this paper it suffices to run the runall.bat/.sh file on a folder containing matrices found in the SuiteSparse Matrix Collection (Formerly the University of Florida Sparse Matrix Collection) dataset. The next page holds two plots detailing the exact performance measurements for the complete test set compiled using the hardware described above.

A.6 Experiment customization

The number of iterations used for the timing measurements can be altered. A CPU implementation can be enabled as well to confirm the results of the framework output. Debug can be enabled to get more detailed timing/memory results.

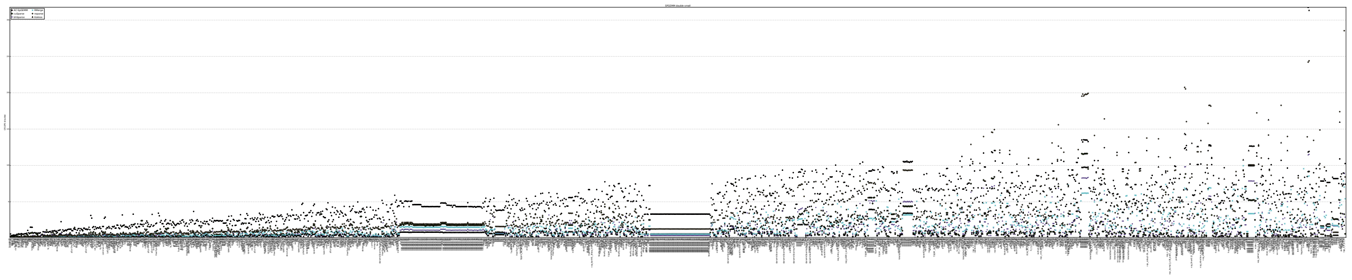


Figure 9. Marker plot for the complete test set using double precision for small matrices with average row length $\bar{a} < 42$

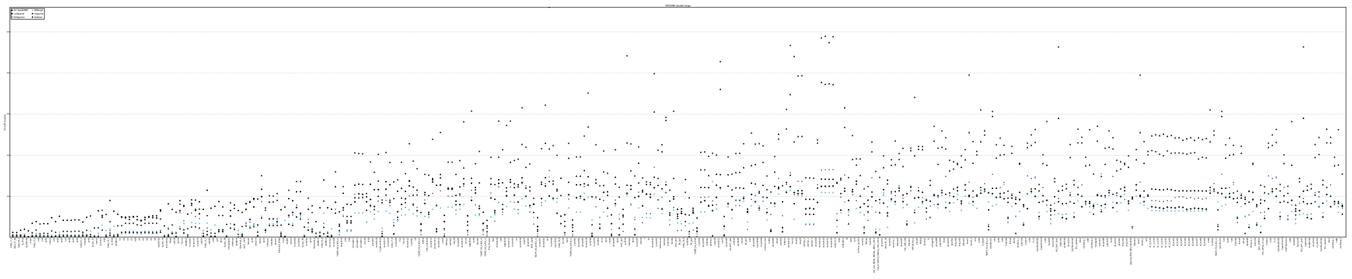


Figure 10. Marker plot for the complete test set using double precision for large matrices with average row length $\bar{a} \geq 42$

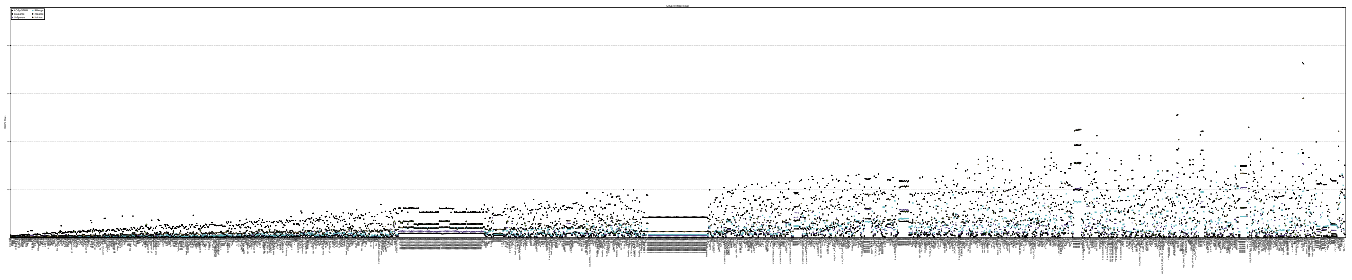


Figure 11. Marker plot for the complete test set using single precision for small matrices with average row length $\bar{a} < 42$

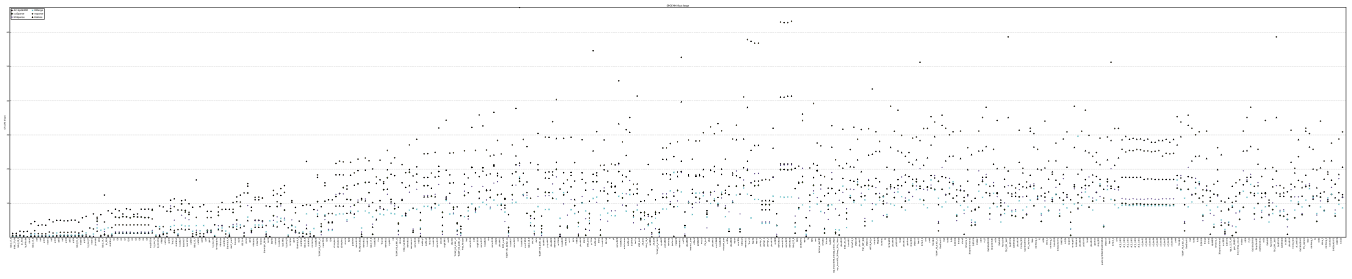


Figure 12. Marker plot for the complete test set using single precision for large matrices with average row length $\bar{a} \geq 42$