spECK: Accelerating GPU Sparse Matrix-Matrix Multiplication through Lightweight Analysis

Mathias Parger
Graz University of Technology
Austria
mathias.parger@icg.tugraz.at

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

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

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

Abstract

Sparse general matrix-matrix multiplication on GPUs is challenging due to the varying sparsity patterns of sparse matrices. Existing solutions achieve good performance for certain types of matrices, but fail to accelerate all kinds of matrices in the same manner. Our approach combines multiple strategies with dynamic parameter selection to dynamically choose and tune the best fitting algorithm for each row of the matrix. This choice is supported by a lightweight, multi-level matrix analysis, which carefully balances analysis cost and expected performance gains. Our evaluation on thousands of matrices with various characteristics shows that we outperform all currently available solutions in 79% over all matrices with >15k products and that we achieve the second best performance in 15%. For these matrices, our solution is on average 83% faster than the second best approach and up to 25× faster than other state-of-the-art GPU implementations. Using our approach, applications can expect great performance independent of the matrices they work on.

• **Theory of computation** → **Massively parallel algorithms**; • **Computing methodologies** → *Linear algebra algorithms.*

SpGEMM, Sparse Matrix, GPU, Analysis

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PPoPP '20, February 22–26, 2020, San Diego, CA, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6818-6/20/02...$15.00
https://doi.org/10.1145/3332466.3374521

1 Introduction

Sparse general matrix-matrix multiplication (SpGEMM) is ubiquitous in many applications like the algebraic multigrid method [2], graph processing [12] and mesh operations [20]. While dense matrices, with identical numbers of products per output element and predetermined memory access patterns, can be accelerated well with single instruction, multiple threads (SIMT) processors like the graphics processing unit (GPU), SpGEMM poses a significant challenge on that kind of hardware. Due to locally varying non-zero (NZ) patterns and unpredictable numbers of intermediate and output elements, SpGEMM leads to unbalanced workloads and uncoalesced memory accesses, resulting in low performance on GPUs. Since GPUs offer high theoretical peak performance and are gaining importance in supercomputing, optimizing GPU SpGEMM is of particular interest.

SpGEMM computes the sum of products for each element of the matrix C given two sparse input matrices A and B:

$$\mathbf{C}_{ij} = \sum_{k} \mathbf{A}_{ik} \cdot \mathbf{B}_{kj}, \tag{1}$$

where i and j are the row and column indices of the NZ elements of $\mathbf{A}$ and $\mathbf{B}$, and k is the set of colliding indices.

In the following, we assume that the matrices are stored in the compressed sparse rows (CSR) format, which is the most commonly used format. CSR stores the NZ elements sorted row-major and column-minor. Every entry consists of its value and its column index. Additionally, a sorted array of row offsets indicates the beginning of each row.

Challenges There are multiple challenges in computing SpGEMM on SIMT hardware. First, the number of non-zeros (NNZ) per row may vary from row to row in both input matrices as well as the output matrix. Thus, achieving a uniform work distribution and a good memory access pattern is non-trivial. Second, determining the exact size of C is similarly complex as the SpGEMM itself. On this account, memory requirement estimation and workload distribution is either done heuristically or involves extensive matrix analysis and counting of temporary elements, *i.e.*, the intermediate products before accumulation. Third, the distribution of elements inside one row of the output matrix may vary between a single element, small dense regions, or widely spread elements throughout the row. Thus, choosing a well-suited accumulator of temporary elements is difficult, especially when targeting the use of fast on-chip scratchpad memory.

While existing GPU solutions work well for the kind of matrices they were optimized for, they fail to adapt to matrices with different characteristics. As a consequence, the performance difference of the best performing algorithms is often orders of magnitudes apart when targeting matrices with different sparsity patterns. It is difficult to determine the best-suited approach by only inspecting the input matrices upfront. Other factors, such as temporary product counts, number of products per resulting element and variability between rows may completely alter the efficiency of an approach. Attempting to analyze these factors easily becomes more costly than performing the complete SpGEMM. Furthermore, these characteristics may vary strongly across the output matrix and thus might require different approaches for parts of the output. However, switching the algorithm locally may worsen memory access patterns and caching, reducing performance.

Contribution Considering the aforementioned challenges, SpGEMM needs to balance the cost of analysis and the gains by adaption of the SpGEMM algorithm. In this paper, we investigate this trade-off and propose *spECK* (*SpGEMM achieving Efficient Computation for all Kinds of matrices*), which achieves high performance independent of the input matrix characteristics. We make the following contributions:

- We propose a lightweight, multi-level matrix analysis, which balances the expected gains and costs of the analysis based on information gathered on-the-fly.
- We describe an adaptable, hash-based accumulator that adjusts to different row characteristics and optimizes memory access patterns and thread utilization.
- We show that switching between our hashing, dense accumulation and direct referencing allows to locally adapt SpGEMM to a variety of matrix characteristics.
- We use the full *SuiteSparse Matrix Collection* [6] to design *spECK*'s multi-level analysis and algorithm selection, which ensures universal applicability.

Comparing *spECK* to six state-of-the-art GPU SpGEMM implementations, *spECK* achieves the best performance for 79% and the second best for 15% out of 2263 matrices from the *SuiteSparse Collection*. As *spECK*'s performance is highly consistent throughout the entire data set, we achieve an average speedup of 83% over the second best approach.

2 Related Work

Researchers have proposed many SpGEMM approaches for GPUs over the past years. These algorithms can be divided in four categories: *Expand, Sort and Compress* (ESC), *Hashing*, *Merging* and *Dense accumulation*. An overview of their properties is given in Table 1.

ESC was introduced in *CUSP* [3] and has been adapted to work locally in *bhSPARSE* [14], by Dalton et al. [4, 5] and *AC-SpGEMM* [19]. ESC stores all intermediate products in temporary memory (*expand*), *sorts* them by (row and) column index and finally accumulates the values with colliding indices (*compress*). During sort and compress, it works on all intermediate products, independent of their origin and thus achieves good load balancing and memory access patterns automatically. ESC is fast for matrices with small numbers of products per output element (low compaction), but it usually requires large amounts of temporary memory. Furthermore, when used for large rows with high compaction, sorting the intermediate products becomes increasingly expensive compared to only sorting the resulting elements.

Hash-based solutions [1, 7] use hashmaps as accumulators. *nsparse* [16] and *cuSPARSE* [17] are well known hash-based examples. Hashmaps come with the cost of atomic operations during accumulation. However, the efficiency of atomic operations in scratchpad memory makes hashing viable on the GPU [16]. Hashing approaches typically rely on one or two analysis steps: (a) estimate temporary memory requirements and (b) a symbolic SpGEMM pass to determine the output matrix size. Both introduce an overhead. Furthermore, global load balancing is often used to assign rows of A to bins according to their temporary memory requirements. Typically, a fixed number of threads is assigned to each row, which reduces memory access- and local load balancing efficiency compared to ESC. If scratchpad memory is insufficient, performance may drop significantly, as hashmaps must be stored in slower global memory [7]. Thus, hashing performs well for medium sized rows which fit into scratchpad memory.

Merging uses sorted lists for intermediate results and combines them using a merge-sort-like algorithm [10, 11]. *RMerge* [10] decomposes the input matrices into sub-matrices that can be merged with an efficient algorithm. *bhSPARSE* [14] dynamically chooses between different merging solutions (and optionally uses ESC). Merge-based solutions share characteristics with ESC, such as high performance for matrices with low compaction and high memory usage for large matrices with high compaction. As merging typically uses equally-sized temporary arrays, they suffer from bad utilization for matrices strongly varying in density or show high preprocessing costs to transform the data into usable portions.

Dense accumulators [15, 18] store and accumulate intermediate values in dense arrays. In contrast to hashing solutions, they directly use the column indices to access the

Table 1. Comparison between state-of-the-art GPU SpGEMM algorithms and their used accumulation type.

	CUSP [3]	*nsparse* [16]	*RMerge* [10]	*AC-SpGEMM* [19]	*bhSPARSE* [14]	*spECK*
Accumulation Type	ESC	Hashing	Merging	ESC	Hybrid	Hybrid
Analysis Costs	none	med	high	low	med	adapt
Memory Requirements	high	low	high	high	high	low
Memory Access	good	rand	good	good	rand	adapt
Load Balancing	good	binning	fixed	adapt	binning	adapt
Work load	high	low	med	med	med	low
Best Performance	-	med to denser	very thin	very thin to med	-	all

array, thereby avoiding hash calculation and collision handling at the cost of lower memory utilization. Unsurprisingly, dense accumulators can achieve high performance for matrices with relatively dense output rows, while high memory demands limit the performance for sparse results.

spECK uses an adaptive version of hashing and dense accumulation, which we configure based on a lightweight analysis. Our hashing in combination with local load balancing is faster than previous hashing approaches and most often outperform merging and ESC in their respective domains. Additionally, *spECK* shows the lowest temporary memory requirements, enabling the multiplication of larger matrices.

3 Background and Motivation

Unlike previous SpGEMM solutions, which work well only on certain types of matrices, *spECK* aims for high performance on all kinds of matrices. As a basis for our discussion, we use the full *SuiteSparse Collection* with over 2800 matrices.

The customary way to perform SpGEMM with CSR inputs is to compute one row of **C** by accumulating all rows of **B** referenced within one row of **A**. A natural way for parallelization is over the rows of **A**, since the output for each row can be calculated independently using one of the before mentioned accumulation approaches. Depending on the rows being computed, different approaches work well. Thus, multiple solutions select alternative accumulators depending on the matrix characteristics: *bhSPARSE* [14] chooses an accumulator based on the number of intermediate products. Likewise, Kunchum et al. [13] bin the rows using intermediate product counts and sub-bin based on the NNZ per row. *nsparse* [16] uses a special row accumulator for very low NNZ. *AC-SpGEMM* [19] handles long rows differently.

With *spECK* too, we target different accumulators. Depending on the number of temporary products and density for each row, we plan to switch between three accumulation approaches: For sparse output rows with widely spread NZs, hashmaps are a good choice due to their low memory requirements. If the space between first and last NZ in the output rows is densely populated, dense arrays can avoid the overhead of hashing and sorting. And if the current block is only accessing a single row in **B**, we can avoid accumulation all together and directly reference the row in **B**. However, to achieve high performance SpGEMM, switching between these accumulators and configuring them must not incur complex analysis. Especially, if the involved matrices are uniform, *i.e.*, the row lengths only vary slightly, the overhead can reduce performance below that of simpler approaches that fit the matrix type.

3.1 Global Load Balancing

Good load balancing is essential for achieving high performance on the GPU. Unequal load and thus unused resources may significantly slow down execution. Load balancing typically consists of two steps: *global load balancing*, *i.e.*, splitting the work into blocks and *local load balancing*, *i.e.* distributing the work of a block to all threads of the block. Commonly, global load balancing is combined with binning over the rows of **A** for switching between different accumulators [14, 16]. The rows are often assigned to bins depending on the number of temporary products. Thus, by assigning each bin to a different block size, rows with few products use fewer threads than large rows; balancing the average number of products per thread.

For hashing, using the hash map size for different bins may be suboptimal. As the output rows become shorter, large hash maps become increasingly detrimental to performance, as scratchpad memory is wasted and extracting data from nearly empty hash maps still requires to inspect all (empty) entries. Thus, instead of binning over the temporary products, we use binning to fully utilize the available scratchpad memory, which is the most performance critical factor according to our experiments. Similar to *nsparse* [16], we can bin each row of **A** depending on the amount of memory needed for accumulation. In this way, we can optimize the utilization of scratchpad memory and ensure that all temporary elements can be stored locally (as long as the required memory does not exceed hardware limits).

We design a load balancer that finds fitting hash map sizes after binning and sets up blocks of threads with matching thread sizes. Ideally, both the symbolic and the numeric SpGEMM step complete in scratchpad memory. However, scratchpad memory sizes and numbers of threads per block

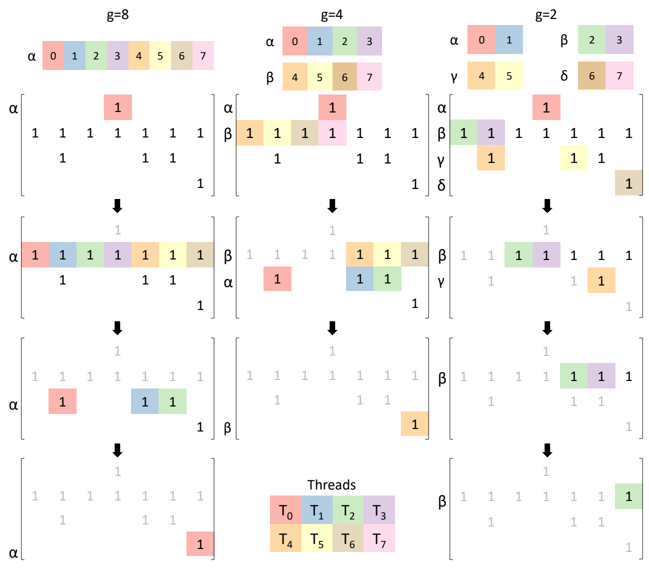


Figure 1. Local load balancing using different group sizes g per row of input matrix **B**. Given a hypothetical eight threads per block, the three columns show the access patterns for different numbers of threads per row. $g = 8$ finishes the multiplication in 4 iterations with coalesced memory access. $g = 4$ requires 3 iterations, $g = 2$ requires 4 with a worse memory access pattern. ($g = 1$ would take 7).

cannot be scaled down below certain hardware limits when trying to achieve full hardware utilization. Single rows are often too short to provide enough parallelization even for the smallest configuration. To tackle this issue, we combine multiple short rows into a single block to maximize the scratchpad memory usage and increase the thread utilization.

Still, binning comes at a cost. Many matrices have a nearly uniform number of temporary elements per row, in which case all rows fall into the same bin. In these cases, binning (and the involved deep analysis), creates unnecessary overhead. Note that in nearly 40% of the matrices in the *SuiteSparse Collection*, the highest number of temporary elements is less than twice as large as the average. With *spECK*, we aim to detect those cases where the row with the highest memory demands does not exceed the average demands greatly and avoid binning. Similarly, for small matrices the expected performance gained by binning is small compared to the overhead and we also want to avoid binning.

3.2 Local Load Balancing

The assignment of threads to entries in **A** and **B** has a big effect on performance. Ideally, memory access to the rows of **B** should be coalesced. However, as **B**'s rows may vary in size, it is difficult to assign a fitting number of threads to each row. Previous work [10, 14, 16] usually uses a constant number of threads. Assigning many threads will yield perfect memory access, however, for small rows many threads are idle. Contrary, using few threads will assign work to all threads, but memory access patterns worsen. An ideal local load balancer would assign exactly as many threads to each row in **B** as there are entries [19]. However, the costs of such dynamic load balancing are high. In contrast, static local load balancing omits further analysis, but often leads to low utilization of the threads, see Figure 1. For example, *nsparse* assigns a fixed number of 32 threads to each row of **B**, independent of the block size and the NNZ of the row. Over 50% of the matrices in *SuiteSparse Collection* have less than eight NZ per row on average, in which case only a fourth of the available threads are utilized while the remaining threads are idle. Idling also occurs, if few rows are assigned to a block: For example, if a block only accesses eight rows of **B** and the block size is 1024 threads, 768 threads ($1024 - 32 \cdot 8$) do not contribute any work.

With *spECK* we aim to improve the thread utilization compared to static local load balancers, while requiring less expensive analysis than *AC-SpGEMM*. By choosing a suitable number of threads per row of **B** for each block, we can find a balance between the two extremes. Since every block accesses different sets of rows in **B** and a single SpGEMM may use millions of blocks for the SpGEMM kernels, the local load balancer still must be lightweight to minimize overhead. Our experiments show that it is not necessary to take the length of every row of **B** into account to get well-suited load balancing. Rather, the number of rows and the average and maximum row length in **B** are sufficient to choose suitable values of g. In this way, we can reduce the overall number of iterations for accumulation and ensure good memory access patterns while avoiding complex, time consuming analysis.

3.3 Conditional Lightweight Analysis

Most of our optimizations rely on matrix analysis. While extracting detailed information about the NZ distribution can help selecting the best suited method and load balancing, the cost of the required analysis could easily exceed the performance gains compared to using the same method for every matrix and row. For example, *nsparse* [16] on average requires about 30% of the execution time to determine intermediate memory requirements and count the number of output elements. In extreme cases, up to 60% may be spent on these steps. If an approach naturally fits the input matrices, analysis would significantly reduce performance. While avoiding the analysis may be tempting, tackling SpGEMM with no knowledge about the expected workload or output sizes involves heavy memory overheads and complicates load balancing. For example, *AC-SpGEMM* [19] may overallocate temporary memory by a factor of 10×. *spECK* aims to minimize the memory requirements, allowing for larger matrices to be multiplied.

Our goal to achieve high performance for all matrices requires information about the matrices involved in the multiplication while ensuring analysis overheads stay low. Thus,

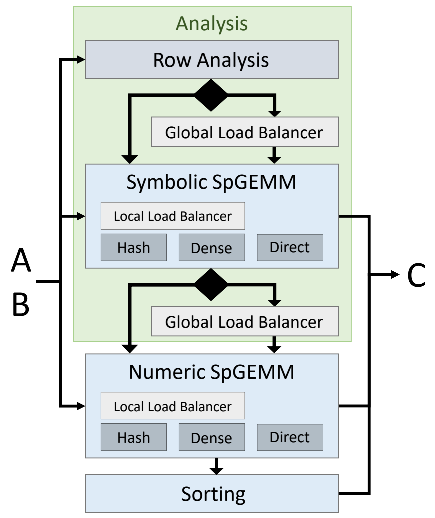


Figure 2. Procedure used in *spECK*. Load balancer and SpGEMM select the best suited methods depending on the previously obtained analysis.

we propose a lightweight, first analysis to gather general information about the involved matrices and decide whether a deeper analysis and load balancing is advisable. While other approaches also run a first analysis [14, 16], they fall short of our approach: Previous work only considers the number of temporary elements generated per row, which can be determined by summing over the row lengths referenced in **B**. While this yields an upper bound for the temporary memory, it does not provide sufficient data to make an informed decision between different algorithms or local load balancing. Analyzing each referenced row in **B** in detail, *i.e.*, looking at all column indices, would give better insights, but incurs high additional overhead. We found that, for algorithm selection and adaption, general statistics among the referenced rows, like average and maximum row length as well as maximum and minimum column index of **B** (and thus **C**), are sufficient. Using this $O(NNZ_A)$ analysis, we guide global and local load balancing strategies, decide and configure the accumulation strategy and decide whether to perform additional analysis and load balancing before numeric SpGEMM.

4 *spECK*

spECK consists of six major stages, as illustrated in Figure 2: First, a lightweight row analysis gathers information about the involved matrix rows. Depending on its outcome, load balancing is performed. Then, a symbolic SpGEMM step determines the memory requirements for the output matrix and gathers additional information about the execution. The symbolic pass chooses between our adaptable hashing, dense accumulation and direct referencing and performs local load balancing. After a second, optional global load balancing step, numeric SpGEMM is performed. Finally, the output data is sorted in case the accumulation did not already provide sorted NZs and **C** is generated.

4.1 Row Analysis

spECK parallelizes SpGEMM over the rows of **A**, which can be processed independently. To this end, we start by analyzing the rows of **A** alongside the referenced rows of **B**. A thorough analysis of each NZ in **B** would induce a large overhead comparable to the actual execution of the SpGEMM itself. Thus, we extract as few features as possible while still being able to select a well suited accumulator and parameters for each matrix and row. For each row of **A**, *spECK* extracts the following information: a) the total number of products, b) the number of products in the longest referenced row of **B** and c) the minimum and maximum column index of all referenced rows of **B**. Furthermore, we extract the maximum number of products over all rows of **A**.

Algorithm 1: Row Analysis

```
1  $prod \leftarrow 0, prod_{max} \leftarrow 0$
2  **forall** $row_{\mathbf{A}}$ *of* **A do**
3      $prod_r \leftarrow 0, prod_{r,max} \leftarrow 0$
4      **forall** $col_{\mathbf{A}}$ *of* $row_{\mathbf{A}}$ **do**
5          $id_{\mathbf{B},0} \leftarrow off_{\mathbf{B}}[col_{\mathbf{A}}], id_{\mathbf{B},n} \leftarrow off_{\mathbf{B}}[col_{\mathbf{A}}+1]$
6          $col_{min} \leftarrow cols_{\mathbf{B}}\left[id_{\mathbf{B},0}\right]$
7          $col_{max} \leftarrow cols_{\mathbf{B}}\left[id_{\mathbf{B},n}-1\right]$
8          $prod_r \leftarrow prod_r + id_{\mathbf{B},n} - id_{\mathbf{B},0}$
9          $prod_{r,max} \leftarrow max(prod_{r,max}, id_{\mathbf{B},n} - id_{\mathbf{B},0})$
10     $prod \leftarrow prod + prod_r$
11     $prod_{max} \leftarrow max(prod_{max}, prod_r)$
```

Note that this analysis is still $O(NNZ_A)$ and we parallelize it over the NZ of **A**, see Algorithm 1.

It provides all information needed to decide if the global load balancer should be used in the symbolic step, and if so, which bin each row falls into, and if not, which bin is used instead, as well as the number of threads the local load balancer assigns per row of **B**. Together with the NNZ per row of **C**, determined in the symbolic SpGEMM stage, this analysis also provides the information required to decide if the global load balancer should be used for the numeric step and which bins are required, and if a block is dense enough to benefit from dense accumulation. The details about the usage are described below.

4.2 Global Load Balancing

The goal of the global load balancer is to find fitting hash map sizes to maximize the scratchpad memory usage and to set up blocks of threads with matching thread sizes.

Configuration *spECK* uses five kernel configurations. The first and largest uses the maximum available scratchpad memory (48 KB on Titan V) and maximum kernel size (1024 threads) which guarantees full hardware utilization. Each successive kernel configuration uses half the amount of scratchpad memory and half the number of threads, ensuring that kernel launches fully use the available resources. Note that the Titan V allows to use double the shared memory size (96 KB) with the same maximum of 1024 threads, essentially halving the number of concurrently active blocks (halving the occupancy). We additionally use this configuration to enable a largest possible hash map size in efficient scratchpad memory, resulting in six kernels in total. As the load for small output rows reaches below efficient kernel sizes, we merge multiple small blocks to be processed within a single block to avoid underutilization of the hash maps

To choose between those kernel configurations for the symbolic step, the load balancer can use the information from the row analysis. With the goal of ensuring that processing can be completed in scratchpad memory, we use the number of products as a conservative estimate for the kernel configuration, *i.e.*, the case where no products are combined and no compaction occurs. According to our experiments, this is a highly conservative estimate: Over the *SuiteSparse Collection*, the average compaction factor is seven, whereas there is a correlation with the matrix size. For example, for matrices with fewer than 10 million products, the average compaction factor is only two. While we could use these factors to speculate about an ideal hash map size, using the conservative estimate guarantees that the accumulation can complete in scratchpad memory. Furthermore, it is important not to fill the hashmap to keep hash collisions low and performance high, which we achieve automatically with the conservative size estimate. For the numeric step, the load balancer does not have to rely on estimates. As the symbolic step computes the row sizes of **C**, the load balancer can use the exact output size to choose the appropriate kernel size. To ensure hashmaps are not getting too full, we choose scratchpad memory such that the final hashmap occupancy does not exceed 66%.

Binning To perform load balancing, *i.e.*, the assignment of rows to kernels, we use binning. Previous approaches pull apart neighboring rows by using atomic operations to insert single rows to bins at a time [14, 16]. We found that this can significantly hurt performance, which we attribute to the fact that matrices often show internal structures, *e.g.*, diagonal-like patterns or local clustering (see Figure 8). To mitigate this issue, we perform binning locally in each block first and append them globally in a single transaction. For local binning we use an efficient prefix sum to ensure the ordering is kept according to the row order in the CSR format. We use parallel prefix scans for each bin that can potentially be non-zero, *i.e.* we skip the bins for the largest configurations if they are not required for the matrix. By using the maximum block size of 1024 threads, we ensure high consistency and improve the cache hit rate for rows with overlapping NZ column indices. At the same time, we avoid costly global sorting within all bins.

To tackle the issue of having rows too short to fully utilize the smallest kernel size, our load balancer performs a block merging procedure for the smallest bin. This merging is related to the NP-complete bin packing problem, *i.e.*, finding the ideal subset of rows to fully utilize the available scratchpad memory. Again, considering the order of input rows, we only consider merging neighboring elements. Still, computing the ideal solution is—to the best of our knowledge—infeasible in parallel. Greathouse and Daga [9] faced a similar problem in their sparse matrix vector multiplication. They opted for serial CPU preprocessing, which limits their approach to work efficiently only on architectures where both CPU and GPU have efficient access to the matrix. With the goal of running our load balancer on the GPU for unknown inputs, we present a parallel approach for bin merging.

Algorithm 2: Block Merge

```
1 for $i \leftarrow 0$ to 5 do
2     $k \leftarrow 0, step \leftarrow 2^i$
3     while $k \leq n$ do
4         if $b_k + b_{k+step} < mem_{min}$ then
5             $b_{k+step} \leftarrow b_k + b_{k+step}$
6         $k \leftarrow k + 2 \times step$
```

Algorithm 2 and Figure 3 illustrate the merging procedure where the while loop in line 3 is run in parallel on all threads of a block. Our approach merges two neighboring blocks as long as their combined temporary memory requirements do not exceed the available memory limit. We do 6 iterations of the merging procedure, since our hashing accumulator can handle up to 32 rows per block. This parallelization is similar to running a prefix sum. In the worst case, we achieve a solution which is within 50% of the maximum: two neighbors can always be merged if their sum is less than 100%. Hence, if they cannot be merged, their average utilization must be higher than 50%.

No load balancing While aforementioned considerations lead to good performance for the load balancer, allocating memory for the bins and performing load balancing still yields additional overhead. Thus, we execute the load balancer only if we expect a performance improvement higher than the computational cost of the load balancer. For example, small matrices or matrices with uniform distribution of NNZ per row are processed using a fixed number of rows per block with a single kernel size that has enough memory to store all entries of the longest row. However, if there is a large variety in row length, using equal kernel sizes may lead to high underutilization of scratchpad memory. In this case, using the load balancer will result in a significant speed-up.

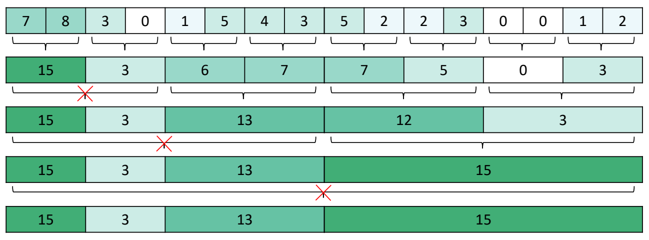


Figure 3. Parallel reduction example for combining neighboring blocks until they fully utilize available memory. Given a maximum of 16 elements per block, neighboring blocks with same row counts are combined as long as their sum of elements is below 16. Our load balancer reduces the number of blocks from 16 to 4. However, the optimal solution would combine the blocks of size 3 and 13 and further reduce the number of blocks to 3.

To decide whether to run the load balancer, we rely on the information from row analysis and the symbolic SpGEMM pass. We provide a detailed analysis of the best performing choices alongside the other influenced stages in Section 5.

4.3 SpGEMM

We use the same approach for computing SpGEMM in a symbolic pass to count the elements of **C** during analysis and in the numeric pass for generating the final result.

In both cases, we decide per block which method to use for expected best performance: direct referencing, hashing or dense accumulation. Furthermore, both use a local load balancer to decide which thread should work on which entry in **A** and referenced entries in **B**.

Local load balancing The task of the local load balancer is to assign threads to rows of **B** such that the workload is balanced and the number of iterations is minimal. At the same time, the number of threads per row should be high enough to achieve coalesced memory access. *spECK*'s local load balancer again tries to balance the cost and performance gains. In the beginning of the SpGEMM step, we divide all T threads of a block into k groups of size $g = T/k$. We then assign those groups successively to the NZ of **A** and thus the referenced rows in **B**, see Figure 1. In this way, we are more dynamic than with a fixed g, but incur less cost than completely dynamic load balancing.

Obviously, good performance depends on k and thus g. To this end, we again use the information available from the row analysis. We use the average row length over all rows that are referenced in the block as a starting point for g. While this seems a natural fit, individual long rows may require many iterations if only few threads are assigned to such a row ($iter_{max} = \frac{elements_{max}}{g}$). To counteract this issue, we consider the following heuristic: If every group has to work through many rows, the many iterations required for one row may balance out overall. Thus, we compare $iter_{max}$ to the number of rows every group will process $n_{rows} = NNZ_A/k$. If $iter_{max} > 2n_{rows}$, we increase g, *i.e.*, we assume that being one iteration off on average is acceptable. The new group size is $g_{new} = g\frac{iter_{max}}{2n_{rows}}$. Similarly, if $n_{rows} > 2iter_{max}$, we reduce g so that more rows of **B** can be processed simultaneously, using $g_{new} = g\frac{iter_{max}}{n_{rows}}$. The idea behind this adjustment is that we bring the maximum number of iterations closer to the number of rows being processed, while prioritizing a low n_{rows} over a low $iter_{max}$. Note that changing g changes both $iter_{max}$ and n_{rows}.

Finally, if k is larger than NNZ_A, *i.e.*, if there are more groups than rows of **B** to work on, we reduce k accordingly, to ensure each thread is assigned to at least one NZ of **A**.

Since we use powers of two as potential thread block sizes, we also round g to the closest power of two, ensuring that all groups are of the same size and all threads are used.

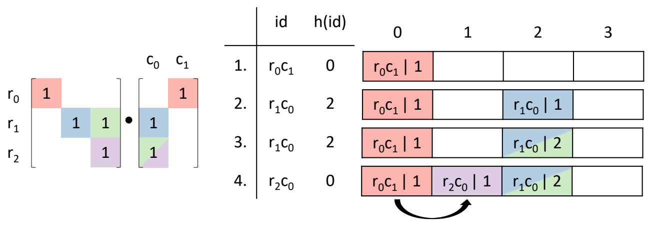
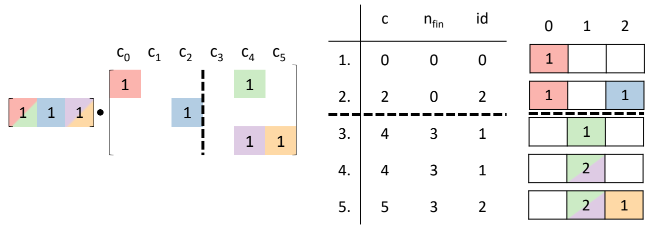

c_0 c_1
r_0 r_1 r_2
1 1 1 1 1 1
id h(id)
0 1 2 3
1. r_0c_1 0 r_0c_1 | 1
2. r_1c_0 2 r_0c_1 | 1 r_1c_0 | 1
3. r_1c_0 2 r_0c_1 | 1 r_1c_0 | 2
4. r_2c_0 0 r_0c_1 | 1 r_2c_0 | 1 r_1c_0 | 2

Figure 4. Working principle of hash-based SpGEMM, where all NZ are stored in one hashmap of size 4. **C** has 3 rows, 2 columns and 3 NZ. Hashed row and column ids ($h(id)$) index into the map. As the hashes of r_0c_1 and r_2c_0 collide, r_2c_0 is inserted in the next available slot.

Sparse Rows of C Blocks with sparse result rows are handled using a hashmap with linear probing [8] in scratchpad memory (Figure 4). Hashmaps allow for fast indexing as long as the map is sparsely filled and for keeping the memory requirements low compared to other methods. When hashmaps are filling up, the number of collisions rises and linear probing becomes increasingly expensive. As mentioned before, global load balancing considers this fact. During the symbolic step, the average map utilization is about 15%, due to the average compression rate of seven. For the numeric step—for which we have exact information—we use a maximum utilization of 66%.

Our hash function multiplies the element index with a prime number and uses the modulo operation to create an array index inside the range of the hash map. The column index of **B** is used as element index when only a single row is processed by a block. For blocks that work on multiple rows, we use a compound integer consisting of 5 bits for local row index and 27 bits for the column index. This way, we reduce the size of the indices to 32 bit, which introduces a limitation to a maximum number of columns of 2^{27}. For larger matrices, we switch to 64 bit integers.



	c	n_{fin}	id
1.	0	0	0
2.	2	0	2
3.	4	3	1
4.	4	3	1
5.	5	3	2

Figure 5. Dense accumulation in an array of size 3, for **C** with 1 row and 6 columns. During the first iteration c_0 to c_2 are accumulated and then stored in global memory. For the second iteration, the array is reset, the start offset advanced and the remaining 3 columns are accumulated.

In the symbolic step, the hashmap is used to count the number of output elements, thus simply storing the index is sufficient. In the numeric step, the values are accumulated, requiring an additional array. Using 32 bit per index and 64 bit (double) per value, the symbolic step can store three times as many elements as the numeric step.

While we choose the kernel sizes such that the hashmaps stay below a 66% fill rate, this is not possible for the largest kernel size, where arbitrarily sized rows may be accumulated. If the local hashmap runs out of space, we move all locally accumulated elements to a global hash map. To this end, we pre-allocate just as many global hash maps as we may need in parallel, *i.e.*, the minimum between: the maximum number of blocks that can run concurrently on the given GPU and the number of blocks requiring global hash maps. If a block needs a global hash map, it allocates from this pool and returns it after completion. We turn to the global hash map if we detect that not all threads may be able to insert a new element into the local map. We then move all entries in parallel to the global map and reset the local map. This way, we always hash locally first before moving large numbers of element together into the global map.

Dense Rows of C For large and dense rows, hashing becomes inefficient. Thus, we switch to a special dense accumulator which stores elements directly in a linear array. As we do not need to store column indices to identify entries, the dense accumulator can store a larger number of elements. Furthermore, it does not require sorting or collision handling, as all entries are stored in order.

If the range from minimum to maximum column index in the resulting row does not fit in scratchpad memory, the dense accumulator needs multiple iterations on different column ranges, successively progressing through the output row, as shown in Figure 5. To efficiently advance through the rows in **B**, we store the positions of the last element that could be processed in the current iteration for each row.

In the symbolic step, we use atomic operations to set bit masks to capture NZ elements, which enables us to store more than 500 000 entries in scratchpad memory compared to roughly 24 000 when using hashmaps.

In the numeric step, the size advantage is not that significant. However, reducing the number of collisions, avoiding global hash maps and avoiding sorting can significantly improve performance. After one iteration of dense accumulation, we use a prefix-sum to compact the output data and write the partial results to **C**.

Single entry rows of A We implement a third, efficient SpGEMM method for rows with only a single NZ in **A**. In this case, we can count the resulting NZ in the symbolic step using only the row offset pointers of **B** without investigating the specific elements. Numeric SpGEMM can directly write the products to the resulting array in the order of appearance in **B** without the need for temporary storage, exploiting the characteristics of sorted indices in the CSR format. While the number of rows with a single NZ element is small overall, 1112 of the 2672 tested matrices contain at least one such row, indicating that this is common in sparse matrices.

Symbolic SpGEMM The symbolic SpGEMM step counts the exact number of resulting elements without calculating their values. This information is used to allocate the output matrix and to compute the row offsets for the CSR format using an exclusive prefix-sum. Furthermore, as the exact memory requirements are known, the load can be balanced more precisely for the numeric step. In the symbolic step, we use dense accumulation only for rows which are more than twice as large as the largest kernel size can store. Due to the average compaction factor of about seven for large matrices, rows with twice as many products than available scratchpad memory can usually still be stored in the hashmap and thus global hash maps are most often avoided. However, hashmaps in global memory must still be allocated as fallback if the compaction for those rows is unexpectedly low.

Numeric SpGEMM The numeric SpGEMM kernel calculates and accumulates intermediate products and stores the resulting values and indices in row-major order. Sorting of the hash results is expensive and often requires temporary memory for efficient computation. Thus, we use different sorting implementations for different row lengths and avoid sorting for medium sized rows by using the dense accumulator if their density is above 18%, *i.e.*, if the number of iterations required by the dense accumulator is less or equal three. The smallest three block sizes sort the resulting elements in scratchpad memory before writing the results to the output array by counting the number of elements in the hashmap with smaller indices. For larger kernels, this approach becomes too expensive due to $O(n^2)$ runtime. The resource requirements for local radix sort would reduce overall occupancy. Thus, these kernels compact the data and store it unsorted in global memory. An additional sorting pass using radix sort arranges them in the right order. The largest kernel size would require a different, slower sorting algorithm with additional memory allocations and potentially the use of slow global hash maps. On this account, we always use dense accumulation for long rows, i.e. rows which require the largest kernel size.

5 Parameter Selection

Choosing the best suited method and parameters has a strong influence on the performance of the multiplication. The different numbers of threads per block, the influence of the sorting implementations and various matrix characteristics make the choice of good metrics and thresholds a difficult task. Many parameters are derived by the hardware specifications, like the optimal number of threads per block to achieve maximum occupancy. Other parameters, like the number of threads per row of **B**, are calculated by the local load balancer by following the goal of minimizing the iterations for accumulation. Some decisions can be made by benchmarking different methods and selecting the fastest. This way, we found that hashing, together with scratchpad memory sorting, works best for small rows and dense accumulation should be used when possible to avoid global memory sorting for medium sized rows. The decision, whether the cost of global load balancing is higher or lower than the gains, is more complex. Using a single threshold, *e.g.*, a minimum number of products required for global load balancing, does not suffice to achieve the best performance for all matrices. On the other hand, using additional analysis for this decision could have a higher cost than always using the load balancer. On this account, we decided to use only information which can be acquired at low cost during row analysis and symbolic SpGEMM, and rely on auto-tuning to achieve a high accuracy with only a few parameters.

Global Load Balancing To estimate the cost and gains for global load balancing, we use the ratio between the maximum and average required scratchpad memory (m_{max}/m_{avg}) as a measure for the variance among the computations and thus the gains the load balancer can achieve. If this ratio is above a threshold and the number of rows ($rows_{\mathrm{C}}$) is sufficiently high to benefit from binning, we use the load balancer. Since we use different sorting for the three smallest kernel sizes in the numeric step, which leads to strongly different performance measures, we use two sets of thresholds: one set of ratio and minimum number of rows in case the longest row requires one of the three largest kernels and one set independent of the used kernels. For the symbolic step, we again use two sets of parameters. However, we use one set of parameters for only the two largest kernel sizes and one for all kernels.

Auto-Tuning We use line-search to optimize these thresholds and to obtain the best overall performance. We first benchmark all matrices with the four combinations of global load balancing (none, symbolic only, numeric only, both) and use the slowdown of the currently selected approach compared to the best possible as a loss. Thus, we tune the parameters not to select the best suited approach for as many matrices as possible, but to minimize the average slowdown compared to the best approach. This approach fits the design of *spECK*, since we aim to achieve good performance for all matrices.

We use an inverse 3-fold cross validation to evaluate our parameters. Using only a third of the matrices for tuning and two thirds for evaluation, the average slowdowns compared to the best performing decision is 1.9% and 2.1%. The auto-tuned values for the three training sets converge to similar values, less than 10% apart. Since all test sets achieve similar performance, we average the parameters over the three training sets for the final parameters used in *spECK*. This way, we reduce the slowdown to 1.7%, and we select the fastest of the four combinations for 85% of the matrices. The final parameters are listed in Table 2.

	m_{max}/m_{avg}	$rows_{\mathrm{C}}$	m_{max}/m_{avg} *	r_{C} *
Symbolic	39.2	28000	6.0	5431
Numeric	10.5	23006	1.3	1238

Table 2. Auto-tuned thresholds used to decide if the global load balancer should be used for symbolic and numeric SpGEMM. Columns marked with * are the parameters used for the largest kernel sizes (three out of six kernels in symbolic and two out of six kernels in numeric SpGEMM).

6 Evaluation

We use the *SuiteSparse Matrix Collection*, consisting of over 2800 matrices, for evaluation of *spECK*. Square matrices are multiplied using $C = AA$, rectangular matrices are multiplied using $C = AA^T$, where A^T is precomputed. For FLOPS measurements, we count each temporary product as two operations, *i.e.*, multiply and add. In addition, we exclude all matrices that cannot be multiplied by at least one GPU method because of memory limitations. The remaining data set consists of 2672 matrices. The test system uses an NVIDIA TITAN V with 12 GB of memory, CUDA 10.1 and an Intel i7 7700 CPU with 16GB RAM on Ubuntu 18.04.

We compare *spECK* to six state-of-the-art solutions: *cuSPARSE* [17] (cu), *AC-SpGEMM* [19] (AC), *nsparse* [16] (n), *RMerge* [10] (r), *bhSPARSE* [14] (bh), and *KokkosKernels* [7] (kk). Additionally, we use the CPU-based *Intel MKL* to compare the advantages and disadvantages of GPU- and CPU-based solutions. In all benchmarks, we measure the execution time of the approaches using double precision. The memory allocation of the output matrix is not measured since every implementation has to allocate the same amount of memory. All other memory allocations during the multiplication are included in the execution times except for *AC-SpGEMM*.

	cu	AC	n	r	bh	ours	kk	mkl
#best	7	335	86	10	0	1875	3	356
#best*	6	304	86	10	0	1777	3	77
#inv.	0	55	49	56	75	0	815	29
t_{avg}†	43.6	5.46	12.6	10.7	40.8	2.75	-	44.3
$^{m}/_{m_b}$†	1.01	5.57	1.87	2.66	4.36	1.00	-	-
$^{m}/_{m_b}$ † *	1.01	5.33	1.87	2.66	4.36	1.00	-	-
$^{t}/_{t_b}$	12.3	2.29	4.13	5.03	13.1	1.48	46.5	9.16
$^{t}/_{t_b}$ *	13.6	1.98	3.26	4.83	12.9	1.08	27.3	10.6
#5×	1.2k	180	387	973	2.0k	71	2.4k	1.6k
#5× *	1.1k	87	204	836	1.8k	3	2.0k	1.6k

Table 3. Number of matrices with best performance and invalid computations, average computation time (in ms) over all finished executions t_{avg}, peak memory usage relative to *spECK* $^{m}/_{m_b}$, average relative computation time compared to the fastest $^{t}/_{t_b}$ and number of matrices where the execution time is more than 5× above the best. Rows marked with * are evaluated on multiplications with more than 15k products. Rows marked with † are evaluated on matrices that could be completed by all GPU approaches except *KokkosKernels*.

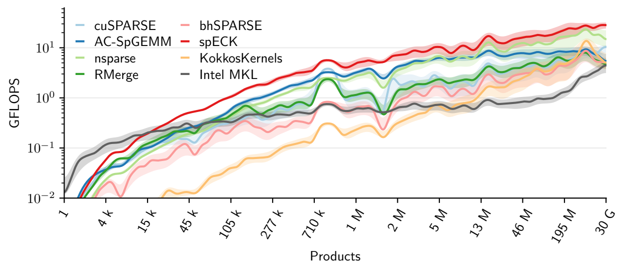


Figure 6. Smoothed GFLOPS achieved over all matrices ordered by number of products. Line thickness indicates the deviation. Benchmarks, which a method failed to compute, are replaced by the slowest valid timing for the matrix.

AC-SpGEMM allocates very large chunks at the beginning and usually has a high over-allocation—they leave exact memory estimates to future work. We decided to exclude their initial allocation time, as it takes longer than the actual multiplication for the greater part of the test set. There is one more important difference between the tested methods: *KokkosKernels* returns unsorted columns and thus violates the CSR specification. This way, *KokkosKernels* omits one of the most expensive steps in SpGEMM for large matrices, which can take up to 40% of the total computation time.

6.1 Overall performance

Table 3 shows the overall performance statistics with trend plots in Figure 6. *spECK* achieves the best performance for 1875, *i.e.*, 70.2%, of all matrices and the second best in 15.3%. 335 and 86 matrices are computed fastest with *AC-SpGEMM* and *nsparse*. *RMerge*, *cuSPARSE*, *KokkosKernels* and *bhSPARSE* achieve the best performance in 10, 7, 3 and 0, respectively.

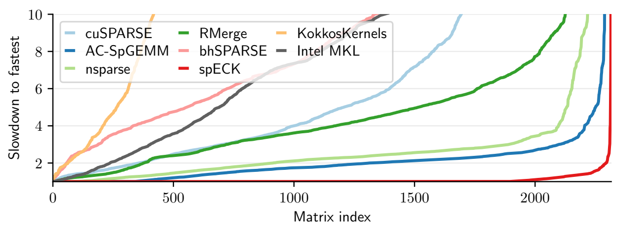


Figure 7. Slowdown compared to fastest method per matrix over all matrices with > 15k products. A ratio of 1 means the method is the fastest for this matrix.

Intel MKL is fastest for 356 matrices, most of them being small matrices for which the overhead of using a GPU is too large and the parallelization is low. Considering only matrix multiplications with more than 15k products, *Intel MKL* achieves the best performance for 77 out of 2263 matrices, while GPU approaches win the remaining 96.6%.

The trend plot clearly shows that our approach achieves the best GPU performance, independent of the input size. Furthermore, it also shows that 15k products form the boundary where GPU-based methods outperform *Intel MKL* on a standard CPU. Thus, we use 15k products as a minimum for the further evaluation and statistics, focusing on the different GPU approaches. Looking at the average relative computation time compared to the individual best approach, it becomes clear that *spECK* achieves exceptional performance throughout the entire test set, with only 108% of the minimum computation time among all methods. *AC-SpGEMM*, *nsparse* and *RMerge* require 198%, 326% and 483% with a large gap to *bhSPARSE*, *cuSPARSE* and *KokkosKernels* with 1290%, 1360% and 2730%. Even for those matrices, where our solution is not the best, we require only 137%.

Figure 7 shows the slowdown of each method compared to the respective fastest method for each matrix. *spECK* shows the best performance overall and is always close to the best performing method. *nsparse* and *AC-SpGEMM* achieve similar results, but *nsparse* exhibits slowdowns of more than 5× for a larger number of matrices. The share of matrices which are computed five times slower compared to the fastest method is 0.1%, 3.8%, 9.0%, 36.9%, 50.1%, 77.6% and 89.3% for *spECK*, *AC-SpGEMM*, *nsparse*, *RMerge*, *cuSPARSE*, *bhSPARSE* and *KokkosKernels*, respectively.

cuSPARSE and *spECK* are the only approaches which are able to complete the multiplication of all tested matrices. In the following comparisons, we evaluate only on the 2092 matrices that could be completed by all methods and have more than 15k products. We exclude *KokkosKernels*, as it fails to compute 815 matrices and would significantly reduce the remaining number of matrices.

The average computation time of *spECK* over these matrices is 2.75 ms. The second fastest method, *AC-SpGEMM*, already takes twice the time for computation with 5.46 ms

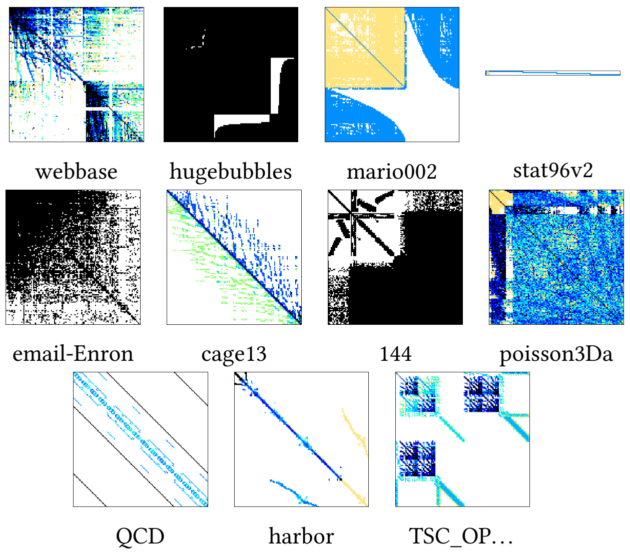


Figure 8. Non-zero patterns of common matrices used in evaluation.

Matrix	Rows	Cols A	NNZ A	Prod.	NNZ C
webbase	1000	1000	3.1	69.5	51.1
hugebu…	21.2k	21.2k	63.6	190.7	132.7
mario0…	389.9	389.9	2.1	12.8	6.4
stat96…	29.1	957.4	2.9	8.7	0.4
email-…	36.7	36.7	0.4	51.5	30.5
cage13	445.3	445.3	7.5	137.3	60.6
144	144.6	144.6	2.1	33.0	10.4
poisso…	13.5	13.5	0.4	11.8	3.0
QCD	3.1	3.1	0.1	4.7	0.6
harbor	46.8	46.8	2.4	156.5	7.9
TSC…	8.1	8.1	2.0	1352.4	8.8

Table 4. Statistics about often-used matrices for in-depth comparisons. Rows and columns are given in thousands, NNZ and products are given in millions.

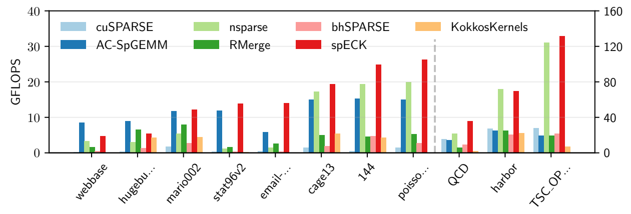


Figure 9. GFLOPS achieved in commonly used matrices.

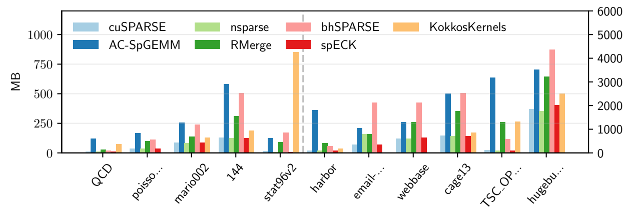


Figure 10. Peak memory consumption in MB during computation of commonly used matrices.

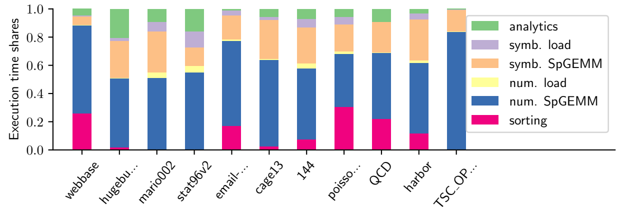


Figure 11. Share of duration for all stages of *spECK* in commonly used matrices. *symb. load* and *num. load* are the load balancers for symbolic and numeric SpGEMM.

excluding their initial chunk allocation. *RMerge* and *nsparse* achieve a comparable average performance with 10.69 ms and 12.55 ms. *bhSPARSE* and *cuSPARSE* require more than ten fold the time of *spECK* to complete the computation with 40.8 ms and 43.6 ms.

Looking at the average peak memory usage, hash-based methods clearly outperform ESC and merging implementations. We include all allocations done during SpGEMM, including the allocation of the matrix **C**. *spECK* has the lowest average peak memory usage per matrix. *cuSPARSE* achieves nearly the same memory usage, followed by *nsparse* with 87% higher average memory usage respectively. We attribute our lower memory requirements to a better analysis of the requirements for global hashing and only allocating memory for the global load balancer if we use binning.

6.2 Common Matrices

We provide detailed analysis on 11 matrices, as shown in Table 4 and Figure 8 & 9. These matrices are used in evaluation of many prior SpGEMM implementations and likely represent the subset of matrices these methods are optimized for. While *AC-SpGEMM*, *nsparse* and *RMerge* together are able to achieve a higher performance than *spECK* in 3 of 11 cases, *spECK* is always only slightly behind. In contrast, *AC-SpGEMM*, *nsparse* and *RMerge* often fall back significantly, achieving only fractions of the highest performance. The memory consumption for the common matrices again clearly shows the difference between hashing and other methods, *cf.* Figure 10.

It can be observed that *nsparse* and *spECK* achieve comparable speed in many of these matrices, but significantly differ for *QCD*, *hugebubbles*, *stat96v2* and *email-Enron*. These are examples where their default setup, *i.e.*, always using hashing and having no local load balancing hurts their performance. For example, *stat96v2* has medium to long rows in *A*, but very short rows in *B*. While our solution chooses appropriate thread counts to access *B*, *nsparse* always uses 32 threads, leading to a utilization of only 9%.

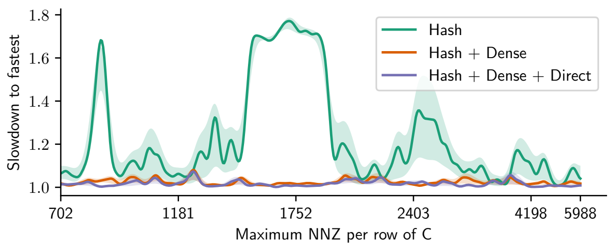


Figure 12. Comparison between using hash only and adding dense accumulation and direct referencing. The results are ordered by the length of their longest rows, clamped to a minimum of 702 (the smallest kernel size with dense accumulation).

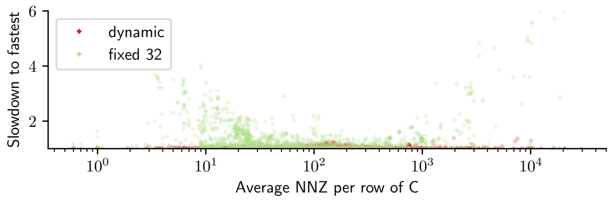


Figure 13. Execution time comparison between dynamic local load balancing and a fixed uniform number of 32 (as used by *nsparse*).

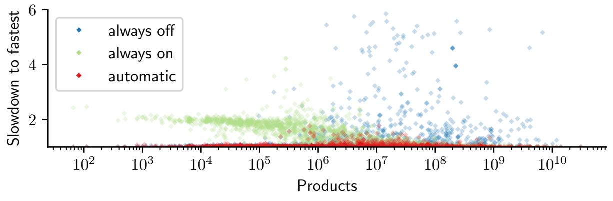


Figure 14. Slowdown compared to fastest approach using permanently enabled/disabled global load balancer and our automatic on/off decision.

6.3 Additional Evaluations

The cost of each stage in *spECK* is shown in Figure 11. The majority of the computation time is spent in the actual numeric SpGEMM kernel which calculates the resulting matrix values. The row analysis step is very cheap and introduces only a small overhead of less than 10% in most cases. Running the load balancer for both SpGEMM stages is roughly as expensive as the row analysis step on average. Even though *spECK* only sorts a fraction of the rows which are computed with hashmaps, sorting of the results can take up to 40% of the computation time.

spECK has many similarities with *nsparse*, but provides a set of improvements for different aspects in SpGEMM computation. For example, we use dense accumulation for medium to large rows to avoid expensive sorting, which can improve the performance more than 60% compared to using only hashmaps (Figure 12). For rows exceeding the size of the largest scratchpad memory hashmap, the performance increases up to 40×, *e.g.*, for the matrix *208bit*, as slow global memory is avoided. Another improvement is the local load balancer with dynamic selection of g, the number of threads per row of B. The dynamic selection of g has a significant impact on the performance, accelerating the computation up to 8× (Figure 13). Outside of the sweet spot around 300 NZ per row of C, where $g = 32$ achieves a good occupancy, the performance decreases strongly for larger and smaller rows. Using our dynamic selection of g, the average number of iterations compared to the best value of g is only 1.02. Contrary to *nsparse*, our load balancer is only used for matrices where we expect a performance improvement. Thus, we reduce the overhead strongly and achieve twice the performance for small matrices (Figure 14). Again, the average slowdown using our decision if load balancing should be used, compared to always selecting the best performing decision, is below 2%.

7 Conclusion

Computing SpGEMM on GPUs is difficult due to varying sparsity patterns and the different sizes of the matrices. We propose a solution that addresses these challenges by combining multiple approaches and enhancing them with a multi-level analysis as well as global and local load balancing. Considering the cost and gains of the analysis and activating load balancing, *spECK* achieves high performance for all kinds of matrices. We achieve the best performance for 79% out of 2263 matrices from the *SuiteSparse Matrix Collection* and we always achieve performance close to the best implementation for the remaining 21%. Furthermore, our approach has the lowest temporary memory requirements. A limitation of *spECK* is the necessity to keep both input matrices and the output matrix in memory during the computation. Even with one of the lowest memory demands, the limited amount of memory available on GPUs limits the size of matrices which can be multiplied, forcing the use of slow CPU-SpGEMM for large matrices instead. We plan to solve that in future work with partial multiplications of large matrices on single GPUs and shared matrix storage in multi-GPU setups.

Acknowledgments

This research was supported by the German Research Foundation (DFG) grant STE 2565/1-1 and the Austrian Science-Fund (FWF) grant I 3007.

References

[1] Pham Nguyen Quang Anh, Rui Fan, and Yonggang Wen. 2016. Balanced Hashing and Efficient GPU Sparse General Matrix-Matrix Multiplication. (2016), 1–12. https://doi.org/10.1145/2925426.2926273

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

[3] Nathan Bell and Michael Garland. 2012. Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. http://cusp-library.googlecode.com

[4] 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, 407–416. https://doi.org/10.1109/IPDPS.2015.98

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

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

[7] M. Deveci, C. Trott, and S. Rajamanickam. 2017. Performance-portable sparse matrix-matrix multiplication for many-core architectures. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 693–702. https://doi.org/10.1109/IPDPSW.2017.8

[8] Donald Knuth. 1963. *NOTES ON OPEN ADDRESSING*. Technical Report. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.56.4899{&}rep=rep1{&}type=pdf

[9] Joseph L. Greathouse and Mayank Daga. 2014. Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 769–780. https://doi.org/10.1109/SC.2014.68

[10] Felix Gremse, Andreas Höfter, Lars Ole 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. https://doi.org/10.1137/130948811

[11] Felix Gremse, Kerstin Küpper, and Uwe Naumann. 2018. Memory-Efficient Sparse Matrix-Matrix Multiplication by Row Merging on Many-Core Architectures. *SIAM Journal on Scientific Computing* 40 (01 2018), C429–C449. https://doi.org/10.1137/17M1121378

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

[13] 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. 11 (2017), 1–11. https://doi.org/10.1145/3079079.3079106

[14] Weifeng Liu and Brian Vinter. 2014. An efficient GPU general sparse matrix-matrix multiplication for irregular data. *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS* (2014), 370–381. https://doi.org/10.1109/IPDPS.2014.47

[15] Md Mostofa, Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, Jongsoo Park, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, Sergey G Pudov, Vadim O Pirogov, and Pradeep Dubey. 2015. Parallel Efficient Sparse Matrix-Matrix Multiplication on Multicore Platforms. *High Performance Computing* (2015), 48–57. https://doi.org/10.1007/978-3-319-20119-1

[16] 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, 101–110. https://doi.org/10.1109/ICPP.2017.19

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

[18] Viral Shah and John R. Gilbert. 2010. *Sparse Matrices in Matlab*P: Design and Implementation.* Technical Report. 144–155 pages. https://doi.org/10.1007/978-3-540-30474-6_20

[19] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. 2019. Adaptive sparse matrix-matrix multiplication on the GPU. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming - PPoPP '19*. ACM Press, New York, New York, USA, 68–81. https://doi.org/10.1145/3293883.3295701

[20] Rhaleb Zayer, Markus Steinberger, and Hans-Peter Seidel. 2017. A GPU-Adapted Structure for Unstructured Grids. *Computer Graphics Forum* 36, 2 (2017), 495–507. https://doi.org/10.1111/cgf.13144 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13144

A Artifact description

A.1 Getting Started

1. Install CUDA 10.1 or 10.2 from https://developer.nvidia.com/cuda-downloads
2. Download cub 1.8.0 from https://nvlabs.github.io/cub/ and extract content into include/external
3. Install g++ >7 and gcc on Linux or url Studio with "Desktop development with C++" workload
4. Install CMake 3.15.5 or newer from https://cmake.org/
5. Clone spECK from Github https://github.com/GPUPeople/spECK
6. Set spECK_STATIC_MEM_PER_BLOCK and spECK_DYNAMIC_MEM_PER_BLOCK in include/Multiply.h line 9 & 10 to the values your hardware supports
   - spECK_STATIC_MEM_PER_BLOCK should be 49152 for all recent devices
   - spECK_DYNAMIC_MEM_PER_BLOCK should be 49152 for all devices before Volta and Turing, 65536 for Turing devices and 98304 for Volta devices
   - if you do not know your GPU generation or hardware limits, compile and run spECK and it will throw errors with information about the correct values
7. Build
   - Windows: (use CMake GUI to setup project) or build the project using "cmake -DCUDA_BUILD_CC70=TRUE -S . -B build -A x64" (set CCXX to correct Compute Capability) followed by opening "runSpeck.sln". Select "Release" configuration and build
   - Linux: Set the correct ComputeCapability (Default is CC70) in "linuxsetup.sh" and run "./linuxsetup.sh"
8. Run
   - Windows: ".\build\Release\runspECK.exe <path-to-matrix> config.ini"
   - Linux: "./build/runspECK <path-to-matrix> config.ini"

A.2 Using spECK

spECK is compiled into a library "spECKLib.lib" and an executable "runspECK.exe" (or linux equivalents). The executable comes with an .mtx (Matrix Market File Format) reader which converts the .mtx file into and saves an ".hicsr" binary file for faster runtimes. runspECK takes 2 input parameters:

- path to matrix in .mtx file format (required)
- path to config.ini (optional)

Config.ini contains helpful options for:

- TrackCompleteTimes: enable/disable benchmarking
- TrackIndividualTimes: enable/disable benchmarking of all stages of speck (comes with performance overhead)
- CompareResult: enable/disable result matrix structure comparison with CUSPARSE. Prints an error message if column indices do not match
- IterationsWarmUp/IterationsExecution: set number of warm up and execution iteration for benchmarking. WarmUp is helpful to make sure that the GPU is running at it's highest clock rate
- InputFile: override input matrix - if an input file is defined in the config, this will override the first command line parameter

A.3 Detailed Results

The achieved GFLOPS of all methods over all 2672 matrices can be found in Figure 15. The GLOPS are calcucated using twice the number of products per multiplication (1× multiply, 1× add) divided by the duration of the SpGEMM.

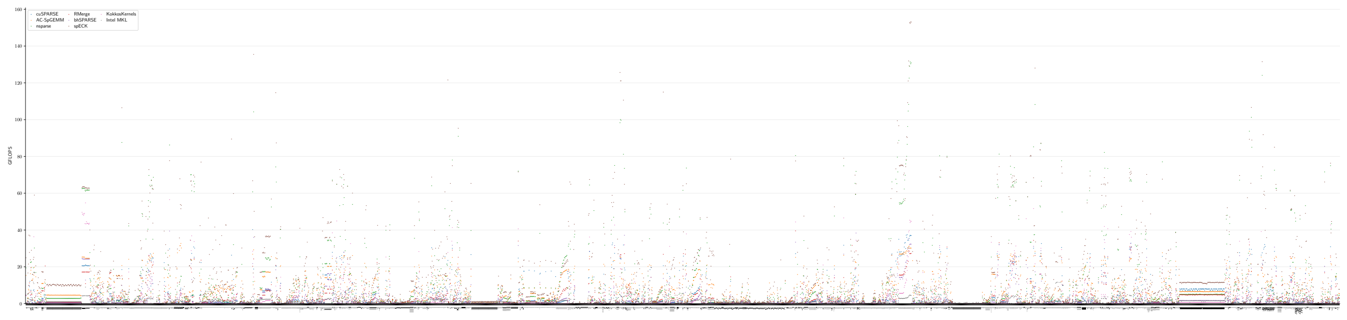


Figure 15. GFLOPS achieved by all methods for each matrix. We use twice the number of products as number of operations (multiply and add) divided by the duration of the computation for all methods.