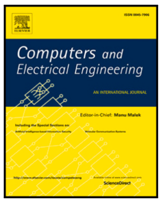
Computers and Electrical Engineering 88 (2020) 106848

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng



Accelerating sparse matrix–matrix multiplication with GPU Tensor Cores[☆]

Orestis Zachariadis [a,*], Nitin Satpute [a], Juan Gómez-Luna [b], Joaquín Olivares [a]

[a] *Department of Electronic and Computer Engineering, Universidad de Cordoba, Córdoba, Spain*
[b] *Department of Computer Science, ETH Zurich, Zurich, Switzerland*

ARTICLE INFO

Keywords:
Sparse matrix multiplication
GPU
Tensor Cores
Parallel computing
SpGEMM

ABSTRACT

Sparse general matrix–matrix multiplication (spGEMM) is an essential component in many scientific and data analytics applications. However, the sparsity pattern of the input matrices and the interaction of their patterns make spGEMM challenging. Modern GPUs include Tensor Core Units (TCUs), which specialize in dense matrix multiplication. Our aim is to re-purpose TCUs for sparse matrices. The key idea of our spGEMM algorithm, tSparse, is to multiply sparse rectangular blocks using the mixed precision mode of TCUs. tSparse partitions the input matrices into tiles and operates only on tiles which contain one or more elements. It creates a task list of the tiles, and performs matrix multiplication of these tiles using TCUs. To the best of our knowledge, this is the first time that TCUs are used in the context of spGEMM. We show that spGEMM, with our tiling approach, benefits from TCUs. Our approach significantly improves the performance of spGEMM in comparison to cuSPARSE, CUSP, RMerge2, Nsparse, AC-SpGEMM and spECK.

1. Introduction

Sparse general matrix–matrix multiplication (spGEMM), similar to its dense counterpart, performs the Matrix Multiplication (MM) of two sparse matrices. The main difference between sparse and dense MM is that we have to account for sparse matrices, which contain mostly zero elements. spGEMM is an important component in scientific and data analytics applications such as Graph analytics [1], Bread th-First-Search (BFS) [2], Algebraic Multigrid (AMG) [3], Schur complement [4], etc. Sparse matrices, which often contain millions of elements, require MM methods that do not waste computing resources on elements that are zero. The diverse structure and density of sparse matrices pose difficulties in regards to memory management and load balancing in parallel systems [5–9].

The re-emergence of deep learning motivated the creation of application specific integrated circuits (ASIC) that specialize in MM. MM is a core component of convolution [10] and these ASICs accelerate the calculation of the convolutional layers significantly in comparison to the normal, general purpose, processing cores. Such ASICs are Tensor Core Units (TCUs) from NVIDIA [11] and TPUs from Google [12]. We utilize the TCUs from NVIDIA to accelerate spGEMM for two reasons. First, accessibility. They are widely available as they are included in the new generation of GPUs from NVIDIA. Second, mixed precision. Mixed precision allows TCUs to mix 16-bit inputs and 32-bit multiplication and accumulation. Typically, in regards to deep learning 16-bits of precision (or less) is sufficient for training purposes and therefore tensor unit manufacturers opt only for 16-bit or lower precisions. Therefore, mixed precision of TCUs widens the application field to scientific problems which are more demanding w.r.t. precision.

[☆] This paper was recommended for publication by associate editor Dr. Raymond Choo.
[*] Corresponding author.
E-mail address: orestis.zachariadis@uco.es (O. Zachariadis).

https://doi.org/10.1016/j.compeleceng.2020.106848
Received 23 December 2019; Received in revised form 8 September 2020; Accepted 10 September 2020
Available online 29 September 2020

Our work is motivated by three key observations. First, blocking sparse matrix storage formats [13], which group the elements of the matrix into rectangular *tiles*, are a good fit for TCUs which expect rectangular matrices as input. Second, TCUs are very efficient even when they are not fully occupied [14]. Third, even though TCUs support only low precision inputs, they can operate in mixed precision mode to perform operations that require higher precision, e.g., GEMM [15,16]. The key idea of our approach is to partition the input in tiles and multiply the tiles with TCUs. Tiles are sparse, but TCUs perform MM efficiently even when not fully occupied. Mixed precision mode is necessary in order to keep sufficient accuracy when multiplying large matrices.

Based on these observations, we propose a new GPU-based framework for spGEMM computation. Our novel methodology groups elements into tiles and uses the fast MM of TCUs to multiply the tiles. To the best of our knowledge, this is the first proposal of using TCUs in the context of spGEMM. Our TCU-based spGEMM methodology, which we name *tSparse*, has two advantages. First, it takes advantage of fast MM of TCUs. Second, by utilizing TCUs to process the MM, we leave the normal processing cores free for non-canonical workloads.

tSparse modifies Expand-Sort-Compress (ESC) methodology [3] of CUSP [17] to *Sort-Expand and Compress* (SEaC), i.e., tSparse brings both multiplication and accumulation after the sorting step. The benefits of this change are twofold. First, tSparse does not have to maintain in memory a large matrix for the intermediate products. Second, tSparse takes full advantage of Multiplication–Accumulation (MAC) operation of TCUs. To that end, we form a task list instead of calculating the intermediate products immediately. Our GPU kernels consume tiles from the task list and perform MM of the tiles using TCUs.

We measure the performance of tSparse in matrix squaring ($A * A$) on matrices from SuiteSparse (formerly known as University of Florida Sparse Matrix Collection) [18]. We compare the performance of our approach to four state-of-the-art libraries: cuSPARSE [19], CUSP [17], RMerge2 [9], Nsparse [20], AC-SpGEMM [7] and spECK [21].

The rest of the paper is organized as follows. Section 2 gives a background on spGEMM. Section 3 presents a high-level overview of tSparse, whereas Section 4 describes tSparse in-depth. Section 5 introduces our test configuration, which we use in Section 6 to evaluate the performance of tSparse. Section 7 presents related work. Finally, Section 8 concludes the paper.

2. Background

In this section we describe: (1) the sparse matrix storage format we use (Section 2.1), (2) the sparse matrix–matrix multiplication (Section 2.2), (3) the challenges in spGEMM, (4) the ESC methodology of CUSP (Section 2.4), (5) precision of real numbers (Section 2.5), and (6) the tensor cores of Nvidia (Section 2.6).

2.1. Storage format

In sparse matrices, typically, the number of non-zero (NNZ) elements is much smaller than the number of zero elements. In order to save memory without degrading the performance of MM, we need an efficient way to store only non-zero (NZ) elements.

2.1.1. COO format

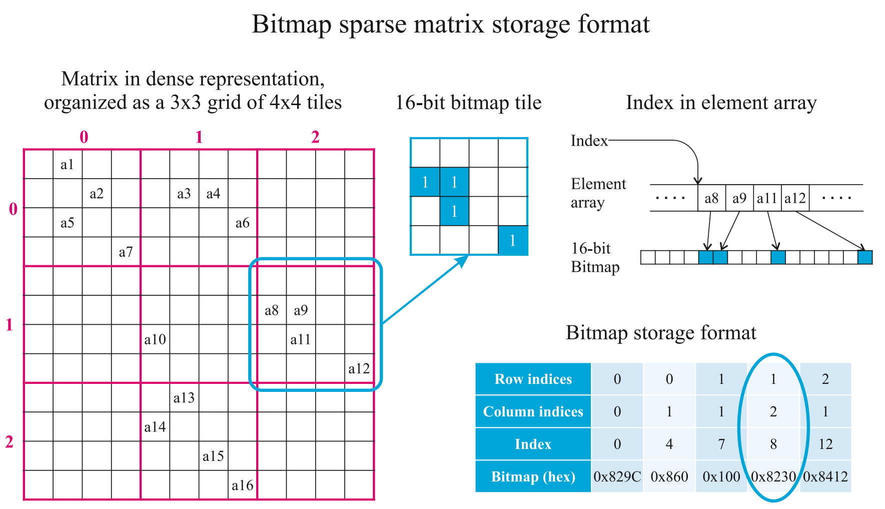
COO format stores each NZ element along with the coordinates the element would have in the dense representation of the matrix [22]. The COO format uses three arrays: for elements, for row indices and for column indices.

2.1.2. Bitmap format

TCUs simultaneously process multiple elements in rectangular structures. However, COO stores only single elements and has no concept of rectangular structures, therefore it is not sufficient by itself as a storage format for tSparse. For this reason, we use a bitmap-based block shaped storage format to store sparse matrices [13].

In our work, we use a bitmap format similar to [13] for three reasons: (1) it is simple and straightforward, (2) square tiles of fixed size fit well to TCUs, and (3) the performance of the format has been evaluated in [13]. The basic idea is to partition the matrices in a grid of 8×8 square blocks and work only on non-empty blocks. We refer to these blocks as *tiles*. The elements inside each tile have the same placement as they have in the dense representation of the matrix. Each element in the tile can be either zero or NZ. To keep track which elements are NZ we use a bitmap, a binary number of which each digit corresponds to one slot of the tile. If a slot contains a NZ we set the respective bit of the bitmap to "1", otherwise to "0". Then, tiles are stored in COO format. The difference with the standard COO format is that, instead of using single elements as values, now we use a tuple pair of two values: (1) an *index* to the *element array*. The element array holds the elements of the tile (elements of the same tile are in consecutive positions of the array), and (2) the *bitmap* of the tile.

Fig. 1 shows how we convert a dense matrix to bitmap format. For simplicity we use 4×4 tiles. We represent the positions of NZ elements as "1"s in the bitmap. We store four values for each tile that has one or more NZ elements: (1) row index like in COO format, (2) column index like in COO format, 3) index into the element array, and (4) bitmap of the location of NZ elements in the tile.


Bitmap sparse matrix storage format
Matrix in dense representation, organized as a 3x3 grid of 4x4 tiles
16-bit bitmap tile
Index in element array
Index
Element array
16-bit Bitmap
a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13 a14 a15 a16
Bitmap storage format

Row indices	0	0	1	1	2
Column indices	0	1	1	2	1
Index	0	4	7	8	12
Bitmap (hex)	0x829C	0x860	0x100	0x8230	0x8412

Fig. 1. A 12×12 matrix in dense (left) and bitmap formats (bottom right). Tiles of 4×4 partition the 12×12 matrix in a 3×3 grid of tiles. Non-zero elements $a8, a9, a11, a12$ of the circled tile are represented as "1" in the *bitmap*. We store the NZ elements of the tile in consecutive locations in the element array. *Index* points to the first element of the tile. On the bottom right of the figure, we circle the representation of the selected tile in bitmap format.

2.2. Sparse matrix–matrix multiplication

The general matrix multiplication (GEMM) has the form:

$$D = A \times B + C \tag{1}$$

where A, B, C are the input matrices and D is the output. In spGEMM, similarly to dense matrices, to get one element of the output, we need to multiply the NZ elements of one row of A with the corresponding NZ elements of one column of B and then accumulate the *intermediate products* (i.e., calculate the inner product). The difference in spGEMM is that we multiply the corresponding elements only if the elements in the corresponding positions of the row of A and the column of B are both NZ and we accumulate only NZ products. The various ways to access the elements of A and B and perform MM are listed in [23]. The same multiplication method applies even if instead of elements we use tiles. In this case, the product of two corresponding tiles is their outer product (or equally MM). We make two important observations. First, (1) takes the form

$$C = A \times B + C \tag{2}$$

when accumulating the tiles, where C is both output and accumulator. We use only the form of (2) for the rest of this work. Second, the matrix Multiplication–Accumulation of small tiles is exactly what the TCUs were designed to do. Fig. 2 shows an example of the $A * B$ sparse MM.

2.3. Challenges of spGEMM

Unlike other sparse matrix operations, in spGEMM both input and output are sparse. Therefore, it is very difficult to utilize the knowledge we infer from the sparsity structure of the input matrices to make arithmetic and memory optimizations. The reasons that make spGEMM more challenging than spMV (sparse matrix–vector multiplication) are three [5–9]:

First, data access is highly irregular because it depends on the sparsity structure of both matrices and the interaction of both structures. MM is not trivial for two reasons: (1) it requires access of possibly distant memory locations to load the inputs, and (2) it requires inserting the result to the output with possibly irregular access patterns.

Second, it is computationally expensive to know the size of the output before the actual MM. According to Liu et al. [6], there are four methods to estimate how much memory we need to allocate for the output. First, the precise method, which makes an estimate that is very close to the actual size of the output, typically by doing a partial execution of the MM algorithm. Second, the upper bound method, which typically uses as upper bound the amount of intermediate products. Third, estimation using probability theory. Fourth, progressive memory increase, which allocates more memory if the previously allocated memory overflows. In all cases, at the end of spGEMM, we remove empty or unused entries from the allocated memory as necessary.

Third, load balancing. The sparsity structure of both matrices and the interaction of the structures determine the distribution of workload. NNZ elements in each row of the input or output may vary significantly, which makes it difficult to partition the workload among threads.

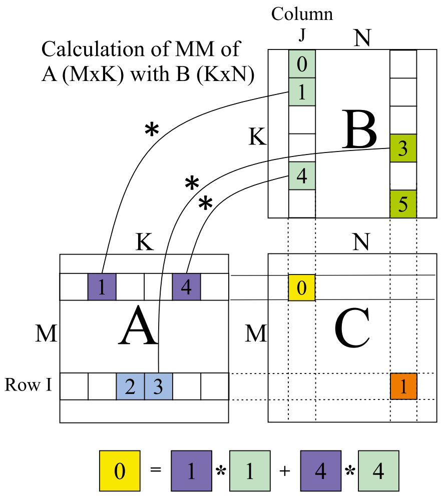


Fig. 2. Example of the sparse matrix–matrix multiplication of two matrices A and B with dimensions $M\times K$ and $K\times N$ respectively. To find the output tile with coordinates $[I,\ J]$, we multiply (MM) the corresponding NZ tiles of row I of A (dark blue) and column J of B (dark red).

2.4. Expand-sort-compress and CUSP

CUSP [17] is a library that specializes in sparse matrix operations. It is open-source and easily accessible on GitHub. It is written in Thrust which makes it easy to read and port to other platforms. Therefore, it provides a good "boilerplate" to test our approach.

CUSP uses Expand-Sort-Compress ESC methodology. ESC performs the spGEMM in three steps [3,17]. First, Expand. ESC multiplies each NZ element $a_{i,j}$ of A with all NZ elements of row $B(j,:)$ of B to get the intermediate products (no accumulation in this step) [23]. Second, Sort. ESC sorts the intermediate products of the previous step so that products that correspond to the same element of C are in consecutive positions. Third, Compress. ESC calculates each element of C by accumulating all respective products, which are in consecutive positions, thanks to the sorting step.

2.5. Real number representation in digital computer systems

Computer systems have to store real numbers in bit representation. Floating point numbers are a common representation. The location of the decimal point and the number of bits determines the precision and range of the represented numbers. We denote the 32-bit representation as *fp32*, whereas the 16-bit as *fp16*. In contemporary systems, typically, we use floating point numbers as defined in IEEE 754 technical standard [24]. Usually, the fewer the number of bits, the faster the processing of the numbers is.

2.6. Tensor core units

NVIDIA, with the latest generations of Graphical Processing Units (GPUs) [25], brought Tensor Cores to the mainstream market. Nvidia TCUs are ASICs that have the purpose of accelerating MM. Therefore, our work on spGEMM has significant benefits by properly adapting spGEMM to TCUs.

TCUs mainly target deep learning, which is not very demanding precision-wise. Therefore, to accelerate MM, TCUs usually work with lower precision number representation (16-bit or less). However, fp16 or lower precision is detrimental to the output because precision and range of fp16 numbers can be insufficient when dealing with physical problems. To rectify this problem NVIDIA provides mixed precision functionality. Mixed precision allows to mix numbers of different precision. The two defining characteristics of the mixed precision implementation of NVIDIA are as follows. First, although inputs A and B are in fp16 precision, their multiplication happens in full precision. Second, the product is stored as fp32 to accumulators C and D [25]. Fig. 3 shows the two characteristics. Markidis et al. and Haidar et al. [15,16] evaluate the performance and precision of GEMM and linear equation solving using the mixed precision mode of TCUs. They show that TCUs can be used in other physical problems, outside deep learning. We use mixed precision functionality to extend the applicability of our work to non deep learning workloads.

3. Overview of our technique

Our TCU-based sparse GEMM technique alters ESC methodology in order to efficiently work with tiles. To that end, tSparse creates a task list which delivers tiles to TCUs to accelerate the MM of the tiles. At this point it is important to make a distinction between the terms *elements* and *tiles*. *Elements* represent a single real or integer number, whereas *tiles* represent a group of elements (Section 2.1.2). This section gives a high level overview of the components of tSparse and how they relate to each other.

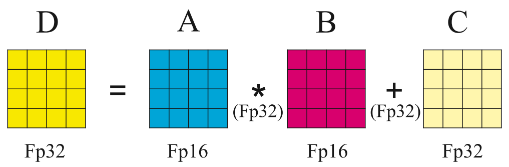


Fig. 3. Mixed precision with CUDA TCUs. Inputs are stored in fp16, whereas the output and addend are stored in fp32. The multiplication and addition are performed in full precision.

3.1. Creating the task list and allocating memory for tiles

tSparse creates the task list in two steps. First, tSparse determines which tiles of A will be multiplied with which tiles of B in order to get each tile of C. Second, unlike the expansion phase of ESC, tSparse does not immediately multiply the corresponding tiles of A and B. Instead, tSparse creates a task list that holds only the locations of the corresponding tuples of A and B. The entries of the task list are sorted so that pairs that correspond to the same tile of C are in consecutive positions. With the help of the task list, tSparse also estimates the number of C tiles in the output and allocates memory accordingly.

3.2. Counting kernel

Dense matrices contain as many elements as the product of their dimensions. However, in sparse matrix–matrix multiplication, the NNZ of the output depend on the structure of the two input matrices and the interaction of these structures throughout the MM. Therefore, we know the exact size of the output only *after* the matrices are multiplied. However, in order to allocate memory for storing the result of the MM we first need to know the exact size to allocate because GPU kernels cannot easily reallocate memory during their execution. The purpose of the counting kernel is to make an estimation of the size of memory we need to allocate in order to store the element array of C, *before* calling the multiplication kernel.

Non-blocking spGEMM approaches allocate memory only for elements. tSparse allocates memory for elements, in addition to memory for tiles. However, allocating memory for elements is not as simple as calling a few parallel primitives because tile multiplication is a sparse MM. The sparsity structure of both input tiles and the interaction of the structures define the memory allocation requirements.

In our implementation, we use the precise method (Section 2.3) to get an estimate of the *count* of elements the output has , i.e., the counting kernel is a partial implementation of the multiplication kernel. The counting kernel requires shorter execution time than the multiplication kernel because it neither loads nor stores any elements. Instead, this kernel uses the bitmap to put zeros and ones in the place of the elements and “simulates” the MM. We allocate memory equal to the NNZ of the output of the simulated MM.

3.3. Multiplication kernel

Once we know how much memory to allocate for tiles and elements we use the multiplication kernel to multiply and accumulate all pairs of tiles of A and B that correspond to each C tile.

3.4. Putting everything together

In order to perform the matrix multiplication of A with B, we (1) determine which products need to be accumulated for each tile of C, and (2) allocate memory for the tiles of C and the element array. Using the task list and the counting kernel we determine the memory allocation size for the tiles of C and the element array, respectively. Subsequently, the multiplication kernel has everything it needs to multiply A with B.

We expect three benefits. First, by placing both the multiplication and accumulation steps of MM in the multiplication kernel we can use TCUs for MM. By moving MM to TCUs, the computational heavy MM is no longer a bottleneck of the spGEMM algorithm. Second, the use of bitmap format reduces memory consumption because one row index and one column index represents up to 64 elements [13]. Third, by grouping elements to tiles, we reduce the amount of values we have to manipulate and therefore there are additional performance benefits (e.g., fewer values to sort during sorting phase of ESC).

4. Implementation details

This section describes the three main parts of our approach, tSparse, in detail: (1) the creation of the task list, (2) the estimation of how much memory to allocate for the elements of the output, and (3) the MM of the tiles. It also describes some smaller components of our implementation.

4.1. Creating the task list and allocating memory for tiles

The four main parts of our algorithm are: (1) finding corresponding tiles of A and B, (2) filtering of zero products, (3) creating a task list with entries that point to the tuples of A and B, and (4) estimating how much memory to allocate for the tuples of the output. In detail:

First, similarly to CUSP, tSparse uses parallel primitives from Thrust library to find the correspondence among tiles of A and tiles of B. The main difference with CUSP is that instead of using single elements we use tiles. This is the part of our methodology that is the same between CUSP and tSparse. We do not change it because it is efficient and accounts for less than 15% of the total execution time of our spGEMM.

Second, the algorithm that finds the correspondence among tiles of A and B considers the whole tile as a single value, i.e., the result is the same regardless if the $8{\times}8$ tile contains one or 64 elements. Therefore, the correspondence algorithm finds a correspondence between tiles of A and B even if the elements inside the two tiles are not corresponding. The resulting tile of the MM of such tiles is a tile with only zero elements. tSparse has a routine that removes this type of corresponding tiles by applying boolean arithmetic on bitmaps. This routine is fast as it does not compute MMs. The culling significantly lightens the workload of the rest of our methodology (fewer tiles to sort, multiply etc.).

Third, tSparse creates the task list. An important consideration is that in ESC methodology the MM and accumulation are in different steps, i.e., MM is in the Expand step and accumulation in the Compress step. However, to fully take advantage of the combined MAC operation of TCUs we need to perform both MM and accumulation of tiles in the same step. In order to put MM and accumulation in the same step we sort the *locations* of the multiplicands, instead of sorting the intermediate products of the MM. By sorting the locations, we defer the MM step until after the Sorting step. Effectively, the locations of the multiplicands form a task list, of which each entry points to one tile of A and one tile of B. Merging the expansion and compression steps and moving them after the sorting step significantly reduces both memory allocations and costly data movement to/from global memory. The reason is that tSparse does not store the intermediate products. Therefore, we save memory by not allocating memory for the 64-bit bitmaps, an undefined number of up to 64 elements and the corresponding indices in the element array. We reduce data movements to/from global memory by not storing the intermediate products before the sorting step and loading them for a second time after the sorting step.

Fourth, our algorithm counts how many of the intermediate products correspond to the same tile of C using a segmented reduce parallel primitive on the sorted indices. We create an offset array from the prefix sum of the counted intermediate products. Our GPU kernels use the offset array for indexing purposes. The prefix sum also gives the total count of tuples in the output. We use this total count to allocate memory.

Sorting. Sorting a long task list is computationally demanding, therefore we need advanced optimization strategies. tSparse works with sorted matrices. We use this knowledge to sort the workload of each row of A separately. For this task, we use the segmented sort by Kaixi et al. [26], which employs a hybrid sorting scheme based on row length.

4.2. Counting kernel

The counting kernel works in four steps. First, it reads the bitmaps of A and B. Second, it creates tiles, wherein each element is set to "1" or "0" based on the corresponding position in the bitmap. Third, it multiplies and accumulates the tiles of A and B that correspond to each individual tile of C using TCUs. Fourth, we count how many elements of the resulting tile are NZ with the `ballot` instruction. We repeat for all tiles of C and accumulate the counts. The counting kernel returns an array of which each value holds an estimation of how many elements each tile of C has.

Each tile contains only "1"s or "0"s, therefore: (1) the estimation of memory requirements can be more than what is actually required because the counting kernel cannot account for possible numerical cancellation (after the actual multiplication there is a compaction stage that removes empty entries from the allocated *element array*), and (2) half precision is enough for executing the MM of the counting kernel because we work only with zero and NZ elements.

4.3. Multiplication kernel

The multiplication kernel performs the actual multiplication and constructs the COO matrix of the output, i.e., it sets the row and column indices, the index and bitmap tuple and the elements of the element array. The multiplication kernel loads the actual elements from A and B and stores the result in the memory allocated by the counting kernel. We emphasize that we use only TCUs for the MM and not TCUs in addition to the standard CUDA cores.

There are two important considerations when multiplying the elements, which are real numbers. First, fp16 arithmetic has a very limited representation range of numbers , which we can easily exceed with multiplication. Therefore, we prefer the mixed precision functionality of TCUs. Second, unlike the counting kernel where we have only positive numbers, when accumulating real numbers, elements get canceled as a result of numerical cancellation. Many tiles may end up empty, something that the counting kernel, which acts on boolean elements, does not predict. For this reason, our multiplication kernel has the additional task of marking for removal *tiles* that are empty.

Fig. 4 compares element loading between the counting and multiplication kernels. The counting kernel does not need to load any actual element. It just creates "1"s based on the bitmap. The multiplication kernel, on the other hand, loads the elements and it places them according to the bitmap.

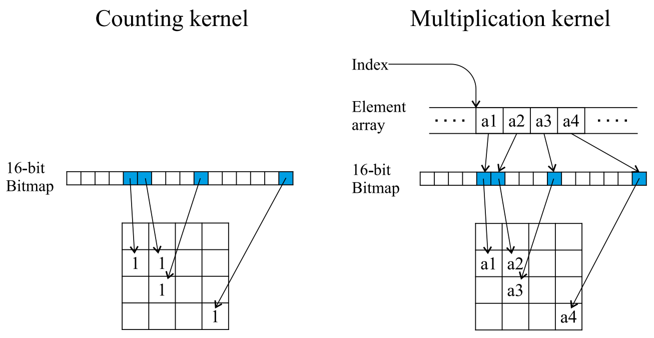


Fig. 4. Comparison of counting and multiplication kernels. The counting kernel (left) places "1"s at the locations indicated by the bitmap. The multiplication kernel (right) loads the actual elements from memory and places them at the locations indicated by the bitmap.

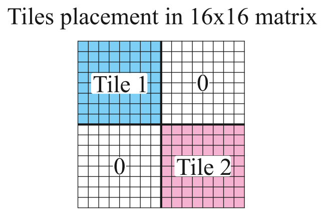


Fig. 5. Placement of two 8×8 tiles in the 16×16 matrix configuration of TCUs.

4.4. Other components

Arrangement of tiles in the TCUs. NVIDIA does not provide an Application Programming Interface (API) for using TCUs with small 8×8 tiles [11]. TCUs execute MM on 256 elements at a time. However, our tiles have a size of 8×8, which means that a large part of the TCU remains unused. Although a TCU does not have to be fully loaded in order to get performance benefits, we can fit two tiles in a single TCU. 16×16 is the only supported matrix configuration that can fit two 8×8 tiles [11]. To put two tiles in the same TCU two steps are necessary. First, we initialize the 16×16 matrix to zero. Second, the tiles must be placed in a diagonal of the 16×16 matrix (Fig. 5). If the tiles were not in the diagonal, but instead side-by-side (top-bottom), the same row (column) of the 16×16 matrix would have elements of two unrelated tiles, which would mix the inner products (of rows of A with columns of B) of the first tile with the inner products of the second tile. The API forces loading to/from TCUs through shared memory. We find the internal layout of the registers of the TCUs (fragments) and we access them directly instead. Loading through registers, which are faster than shared memory, we minimize data movement and increase the performance of MM.

Load balancing. Both counting and multiplication kernels calculate an inner product of tiles (Section 2.2). The number of intermediate products required for each tile of C is different, because the number depends on the sparsity structure of A and B. Therefore the workload for the computation of each tile of C is different, leading to thread blocks with different amounts of work. Thread blocks with different execution times create imbalance among the SMs of the GPU. To tackle this issue, we assign each pair of tiles of C to a different thread block. When a thread block finishes and releases the resources, the scheduler of the SM schedules another thread block to take its place. Therefore, SMs take blocks according to their needs and stay fully occupied until completion of the MM.

Compaction of zeros. MM creates zero elements because of numerical cancellation. To store the output in a strictly sparse format we need to remove all zeros. Therefore, we need a way to detect zeros and remove them or, in other words, *compact* the arrays that hold the elements and the tuples. For the element array, which is just an array, we use a compaction parallel primitive from Thrust. For the array that holds the tuples, we first mark empty tiles in the multiplication kernel (see Section 4.3).

4.5. Putting everything together

Algorithm 1 summarizes tSparse. First, tSparse creates a task list (lines 1–8). Second, it estimates how much memory to allocate for tiles (lines 9–15). Third, it estimates how much memory to allocate for elements (lines 16–17). Fourth, it multiplies the matrices (line 18). Finally, it compacts zero elements (line 19) and empty tiles (line 20).

Fig. 6 illustrates details of the multiplication kernel (Algorithm 1, line 18). In order to get a tile C of the output, we have to accumulate a varied number of products (MMs of tiles of A and B), e.g., $C0 = A1\times B1 + A4\times B4$ and $C1 = A3\times B3$ for the example of Fig. 2 (note the color code of tiles). Each TCU processes two tiles, e.g., TCU0 calculates *C0* and *C1*. However, *C1* has fewer addends. Therefore, when we load the second addend of *C0*, we use *0*s in place of the second addend of *C1*. From the final result, we extract the bitmaps (using `ballot`) and elements of the output.

Algorithm 1 Pseudocode for tSparse

```
for all NNZ tiles $A[i,j]$ in $A[:,:]$ do
    for all NNZ tiles $B[j,k]$ in $B[j,:]$ do
        $task\_list \leftarrow \{row\_ptr(A[i,j]), col\_ptr(B[j,k])\}$
    end for
end for
FilterTilesWithZeroProduct($task\_list$)
SortByKey($B_{cols}[task\_list], task\_list$)
SortByKey($A_{rows}[task\_list], task\_list$)
$tile\_count \leftarrow 0$
for all $c$ in $task\_list$ do
    if $C[A_{rows}[c], B_{cols}[c]]$ is unique then
        $tile\_count \leftarrow tile\_count + 1$
    end if
end for
AllocateMemGPU($tile\_count$)
$element\_count \leftarrow$ CountingKernel($task\_list$)
AllocateMemGPU($element\_count$)
$C_{tiles}, C_{elements} \leftarrow$ MultiplicationKernel($task\_list$)
CompactZeroElements($C_{elements}$)
CompactEmptyTiles($C_{tiles}$)
```

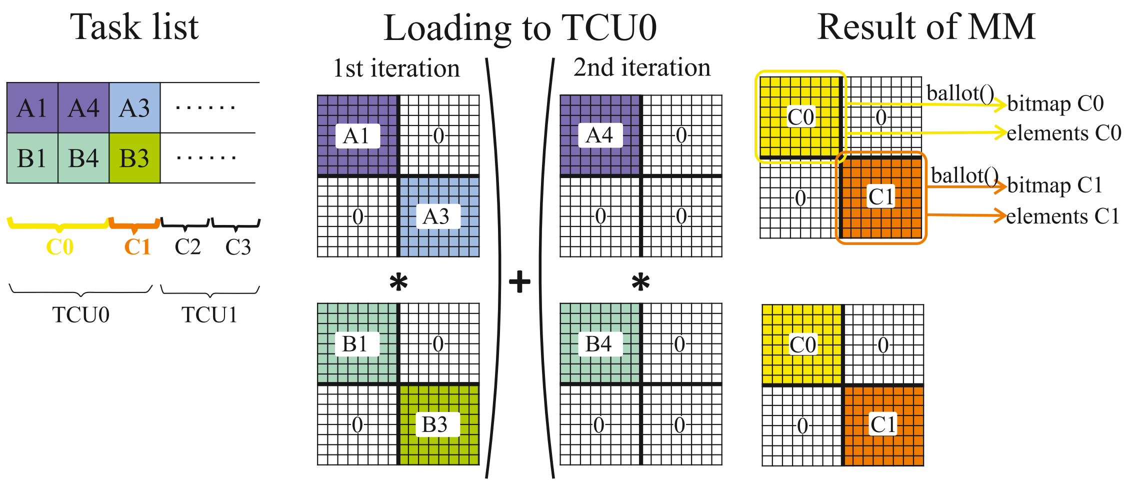


Fig. 6. Detail of the multiplication kernel. Each TCU calculates two tiles of the output C. If the two output tiles require a different number of addends, we use "0"s to make them match. Finally, the output of the TCU gives us the respective output elements of the two tiles and their bitmaps (using ballot operations).

5. Evaluation methodology

We test our approach, tSparse, on two systems: (1) Intel i9-9900@3.6 GHz CPU and NVIDIA Titan RTX GPU, and (2) Intel i7-8700@3.2 GHz CPU and NVIDIA RTX 2070 GPU. Both GPUs are of the Turing architecture [25]. We use CUDA SDK v10. 2 and the accompanying parallel primitives library, Thrust [11], for our GPU code.

We compare tSparse with cuSPARSE from CUDA Toolkit [11], CUSP [17], RMerge2 [9], Nsparse [20], AC-SpGEMM [7], spECK [21]. In addition, to confirm the benefit of using TCUs, we create one implementation of tSparse without TCUs and we name it *nonTCU*. In nonTCU, we use the same method as [13] to multiply the tiles (Algorithm 2). We note that we include the time of memory allocations in the execution time of AC-SpGEMM.

To evaluate the performance of tSparse, we perform the $A * A$ MM, which has the benefit that both matrices have the same sparsity structure. The same structure makes it easier to make observations. In order to facilitate the presentation of the performance of tSparse, we select a subset of 16 matrices from SuiteSparse Matrix Collection [18] for our dataset. All selected matrices are square, as our $A * A$ problem dictates. We select matrices which have elements in the fp16 range.

Table 1 shows the characteristics of our dataset. We denote a matrix stored in bitmap storage format as C_{tiles}. For non-tiling (i.e., single element) approaches, we denote as $\overline{C}$ the total amount of intermediate products. Similarly, for our tiling approach, we denote as $\overline{C}_{tiles}$ the total amount of intermediate products, where each product is a MM between tiles. $NNZ(\cdot)$ of a matrix denotes

Algorithm 2 Matrix multiplication of two 8×8 tiles without TCUs

```
1: $tid$ {The id of a thread}
2: $i \leftarrow 0$
3: while $i < 8$ do
4:     $C\_tile[tid] = C\_tile[tid] + A\_tile[(tid/8) * dim + i] * B\_tile[i * 8 + mod(tid, 8)]$
5: end while
```

Table 1
Matrix characteristics. We list the size of the matrix (number of rows/columns), the number of non-zeros of: the input (NNZ(A)), the output (NNZ(C)), the intermediate matrix (NNZ($\overline{C}$), the number of tiles (NNZ(C_{tiles})), and the number of tiles of the intermediate matrix (NNZ($\overline{C}_{tiles}$)), and the density of the tiles of the input matrix. The upper part corresponds to matrices that are commonly used in the literature, the bottom part to matrices we selected based on our criteria.

Matrix name	Dimensions (square)	NNZ(A)	NNZ(C)	NNZ($\overline{C}$)	NNZ(C_{tiles})	NNZ($\overline{C}_{tiles}$)	Bitmap density (median, mean, std)
mc2depi	525825	2100225	5245952	8391680	718228	1364149	7, 6.4, 4.9
webbase-1M	1000005	3105536	51111996	69524195	2546355	4327469	6, 5.6, 5.3
cage12	130228	2032536	15231874	34610826	2945653	9295217	3, 4.5, 4.1
dawson5	51537	1010777	3616737	21284355	219077	1515543	6, 9.5, 9.1
lock1074	1074	51588	134676	2752056	3050	19520	32, 31.4, 17.3
patents_main	240547	560943	2281308	2604790	2089143	2511808	1, 1.0, 0.2
struct3	53570	1173694	3400384	26704476	146007	543125	21, 18.8, 9.8
wiki-Vote	8297	103689	1831112	4542805	526421	3058660	1, 1.4, 1.0
bcsstk30	28924	2043492	8946070	173481412	252076	1627326	23, 25.9, 17.9
nemeth21	9506	1173746	2578720	146859992	47341	523549	59, 47.0, 21.1
pcrystk03	24696	1751178	7240266	129128312	212471	1514965	15, 23.4, 18.8
pct20stif	52329	2698463	10016951	154237335	323396	1770466	17, 23.4, 18.1
pkustk06	43164	2571768	10596384	179924544	451380	2336176	16, 19.6, 12.9
pli	22695	1350309	8548665	99698581	292851	2262407	14, 15.9, 11.5
net50	16320	945200	40622452	79727280	1037234	8150260	8, 8.6, 11.1
web-NotreDame	325729	1497134	16801350	64593748	693759	1589613	3 , 6.5, 10.7

the NNZ values of the matrix. Finally, from the second and third columns of Table 1 derives the average row size of the input, which is defined as $NNZ(A)/Dimensions$, and we denote as $\widehat{RowA}$.

We divide our dataset into two parts. The first part (upper half of Table 1) consists of matrices that other works use [5–9,13,19,20,27,28]. The second part (lower half of Table 1) consists of matrices that we select after taking into consideration the two criteria that derive from the analysis in Section 6.1: 1) $NNZ(A) > 300000$, and (2) $\widehat{RowA} > 42$.

6. Results and analysis

In this section, we find the criteria that define when tSparse is the recommended spGEMM approach using all qualified matrices from SuiteSparse. Then, we collect four types of measurements using the matrices in Table 1: (1) the speedup of tSparse, (2) the execution time breakdown, (3) the numerical precision, and (4) the memory consumption.

6.1. Finding the selection criteria

We measure the execution time of cuSPARSE, CUSP, RMerge2, Nsparse, AC-SpGEMM, spECK, nonTCU and tSparse on a collection of matrices which we create from SuiteSparse. For this collection, we keep matrices that meet the following four conditions: (1) the input matrix has more than 10000 NNZ elements, (2) the input is square with real numbers in the fp16 range, (3) the bitmap density of the input is more than one, and (4) matrices for which all approaches that participate in the comparison return correct results. We then define criteria based on the characteristics of the matrices. These criteria facilitate the selection of the most appropriate spGEMM approach among tSparse and the other approaches.

We find the criteria in two steps. First, we analyze how each approach works in order to identify which matrix characteristics affect the performance of said approach. Second, we adjust the criteria to our matrix collection manually. Alternatively, we could use a machine-learning method to find the criteria [29], which we leave for future work. We make six major observations.

First, tSparse generally outperforms cuSPARSE in the larger matrices of our collection ($NNZ(A) > 200000$). This happens probably because cuSPARSE does not have sufficient shared memory to store the hash tables and therefore global memory traffic increases. Nevertheless, the size of the matrix is not the only thing that decides the performance of cuSPARSE. To the best of our knowledge, the performance of cuSPARSE also depends on the sparsity structure of the matrix because the structure affects the number of hash conflicts.

Second, tSparse performs better than CUSP when the number of intermediate products of tiles is smaller than the number of intermediate products of elements of CUSP, i.e., $NNZ(\overline{C}_{\text{tiles}}) < NNZ(\overline{C})$. Otherwise, CUSP, which handles only single elements, is faster than tSparse which has an additional overhead for handling tiles.

Table 2
The criteria that define when tSparse is faster in comparison to other approaches. This table evaluates the performance of each condition using the precision and recall metrics from the classification theory. We note that the collection size varies to account for the incorrect results of each approach.

Approach	Condition	Predictions/Collection size	Precision	Recall
cuSPARSE	NNZ(A) >200000	122/260	0.75	0.84
CUSP	NNZ($\overline{C}$)/NNZ($\overline{C}_{tiles}$) ≥ 1	260/260	0.98	1
RMerge2	$\widehat{RowA} > 42$ AND NNZ(A) > 100000	53/233	0.92	0.33
Nsparse	$\widehat{RowA} > 42$ AND NNZ(A) > 100000	53/233	0.75	0.74
AC-SpGEMM	NNZ($\overline{C}$)/NNZ($\overline{C}_{tiles}$) > 9	120/233	0.89	0.86
spECK	$\widehat{RowA} > 42$ AND NNZ(A) > 300000	45/234	0.78	0.8
All	$\widehat{RowA} > 42$ AND NNZ(A) > 300000	45/233	0.76	0.85

Table 3
Speedup of tSparse over the various approaches for matrices selected by our criteria.

	Speedup with tSparse					
	Titan RTX ($\widehat{RowA} < 42$)			RTX 2070 ($\widehat{RowA} < 21$)		
Approach	Gmean	Minimum	Maximum	Gmean	Minimum	Maximum
cuSPARSE	2.8	0.65	11.58	2.98	0.78	9.37
CUSP	36.93	4.17	132.74	26.25	3.55	113.5
RMerge2	11.16	0.89	124.09	11.1	1.17	149.18
Nsparse	1.49	0.65	6.64	1.77	0.84	7.8
AC-SpGEMM	3.72	0.79	19.21	3.4	0.71	12.95
spECK	1.46	0.47	3.17	1.88	0.7	4.67
nonTCU	1.48	1.1	1.97	1.56	1.19	2.14

Third, tSparse is faster than Nsparse when the average row size of A ($\widehat{RowA}$) is greater than 42 and $NNZ(A) > 100000$. Nsparse follows a hash table approach (Section 7.2). Therefore, larger rows possibly create more hash conflicts and hash tables are difficult to keep in shared memory (instead of the slower global memory) with large rows. We also note that web graph matrices (webbase-1M and web-NotreDame) have high irregularity, i.e., although the average row size is small (3.1 and 4.6 respectively), there are rows with thousands of elements (maximum 4700 and 3445 respectively). Therefore they are also taxing for Nsparse.

Fourth, RMerge2 shows behavior similar to Nsparse. tSparse has better performance than RMerge2 when $\widehat{RowA} > 42$. To the best of our knowledge, RMerge2, which follows a hybrid approach (Section 7.3), uses faster kernels when the row size is smaller than the size of a CUDA warp [9]. We note that the low recall score is owed to an instability of RMerge2 (possibly caused by a driver issue) that, in a seemingly random manner, decreases the performance.

Fifth, AC-SpGEMM is an ESC-based approach, like CUSP. Therefore, a similar criterion applies to AC-SpGEMM. The only difference is that, as AC-SpGEMM has much better performance than CUSP, the criterion is much stricter, i.e., $9 * NNZ(\overline{C}_{\text{tiles}}) < NNZ(\overline{C})$.

Sixth, spECK is a hash table approach and has similar behavior to Nsparse. spECK generally performs better than Nsparse, therefore we select a criterion similar to Nsparse's but stricter, i.e., $\widehat{RowA} > 42$ and $NNZ(A) > 300000$.

Global criterion. In summary, a range of matrix characteristics define the performance of each spGEMM approach. To help with selecting tSparse over other approaches, we define two criteria that generally work well to show when tSparse is the most appropriate approach: (1) $NNZ(A) > 300000$, and (2) $\widehat{RowA} > 42$. Table 2 summarizes the various criteria. We note that tSparse performs better on RTX 2070 than on Titan RTX, for the reasons we describe in Section 6.2. Therefore, we can relax the criterion for non-high-end GPUs as follows: $\widehat{RowA} > 21$. Finally, we note that the first criterion may limit the suitability of our approach for applications that reduce the size of input matrices, e.g., AMG.

6.1.1. Evaluating the performance of the criteria

We evaluate the speedup of tSparse over the other spGEMM approaches after applying our global criterion to the matrix collection. Table 3 shows the speedup of tSparse over the other spGEMM approaches when applying the global criterion.

We make three major observations. First, tSparse is faster than all other approaches by $1.46\times - 36.93\times$, which confirms that our global criterion is reliable. Second, the speedup of all approaches follows the same trends for both GPUs. The RTX 2070 GPU performs better w.r.t. execution time, with more relaxed criterion, as the CPU does not bottleneck the execution time as much as on Titan RTX (Sections Section 6.3, 6.2). Third, the *Minimum* columns of Table 3 show speedups < 1. This happens because of the wrong predictions of our proposed criterion (false positives).

6.2. Speedup

To show the benefits of our approach that uses TCUs for MM, we find the speedup over cuSPARSE, CUSP, RMerge2, Nsparse, AC-SpGEMM, spECK and nonTCU for the 16 matrices of our dataset. Figs. 7 and 8 show the speedup of tSparse over the seven approaches when calculating $A * A$ on the Titan RTX GPU. Fig. 7 corresponds to the first part of our dataset (randomly selected

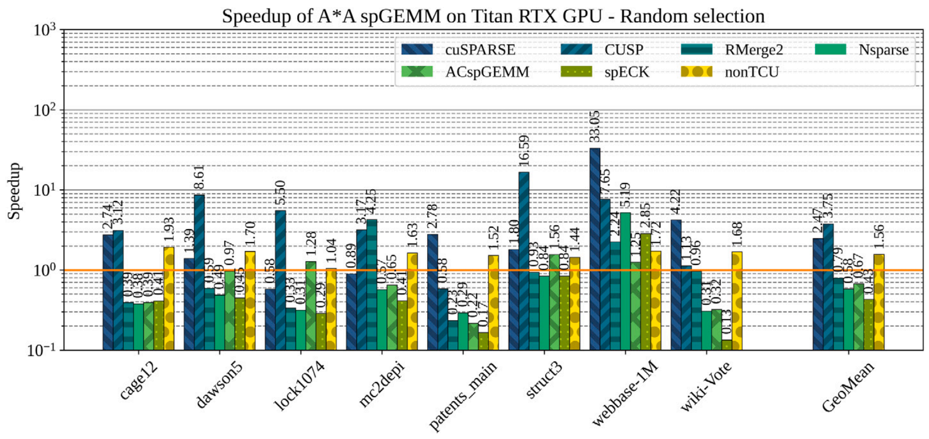


Fig. 7. Speedup of tSparse on $A * A$ spGEMM using randomly selected matrices on Titan RTX.

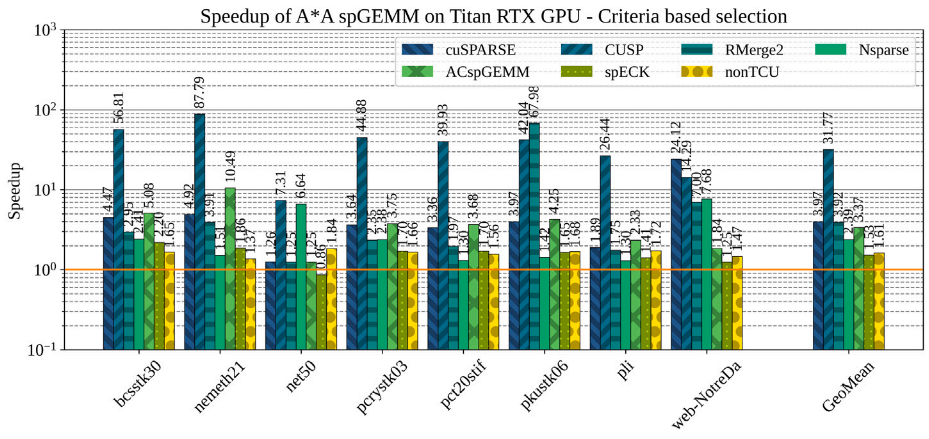


Fig. 8. Speedup of tSparse on $A * A$ spGEMM using matrices selected based on criteria on Titan RTX.

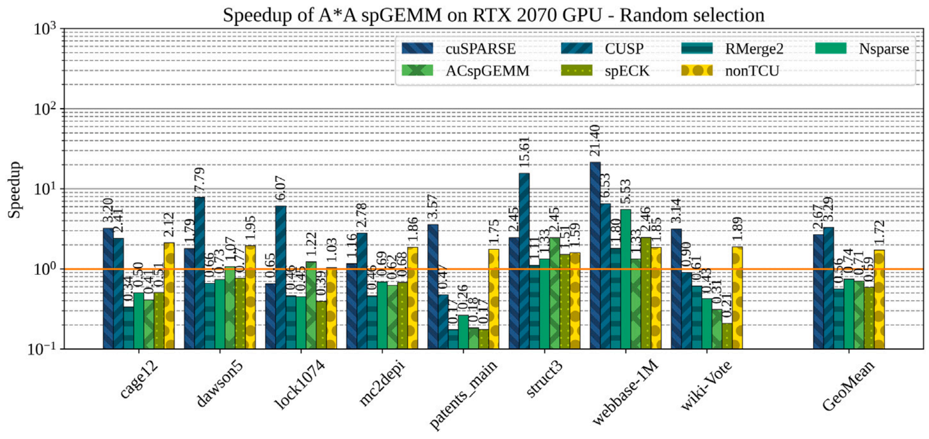


Fig. 9. Speedup of tSparse on $A * A$ spGEMM using randomly selected matrices on RTX 2070.

matrices), whereas Fig. 8 corresponds to the second part of our dataset (matrices selected based on criteria). Figs. 9 and 10 present the speedup on RTX 2070. GeoMean indicates the geometric average of the speedup for the matrices of the respective figure.

We make five major observations.

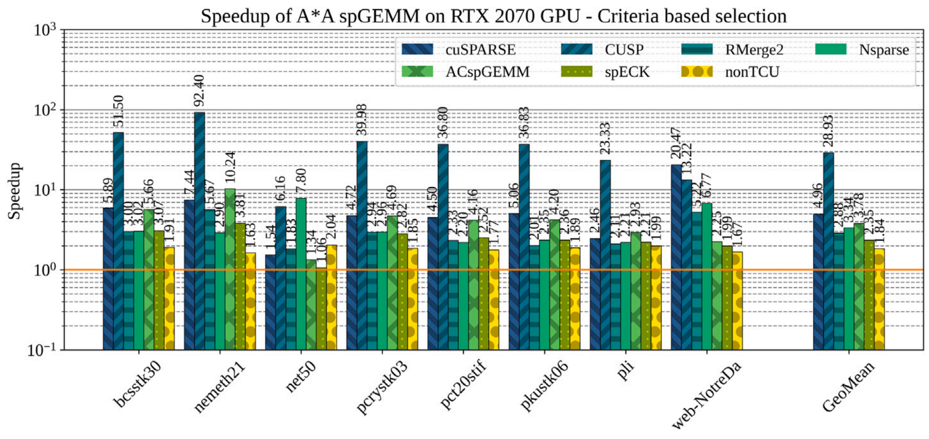


Fig. 10. Speedup of tSparse on $A * A$ spGEMM using matrices selected based on criteria on RTX 2070.

First, tSparse that uses TCUs performs faster than our nonTCU implementation, by an average of 1.68×. This speedup may seem low considering that TCUs in mixed precision promise 4× more flops than normal fp32 operations [25]. There are two reasons that keep it low: (1) our counting and multiplication kernels, without TCUs, occupy about 50% of the total execution time of spGEMM, so according to Amdahl's law we do not expect more than 2× speedup of the total execution time, and (2) tSparse is memory bound, rather than arithmetic bound, because the sorting step results in a task list with entries that point to non-continuous memory locations.

Second, tSparse outperforms cuSPARSE, CUSP, RMerge2, Nsparse, AC-SpGEMM and spECK in 14 (15), 15 (14), 10 (10), 9 (10), 11 (12) and 8 (10) out of the 16 matrices, respectively (the performance of 2070 in parentheses). The speedup of tSparse for the randomly selected matrices is: cuSPARSE 2.47× (2.67×), CUSP 3.78× (3.29×), RMerge2 0.79× (0.56×), Nsparse 0.58× (0.74×), AC-SpGEMM 0.58× (0.71×), spECK 0.58× (0.59×), nonTCU 1.46× (1.72×). With the matrices selected based on our criteria, the speedup of tSparse is: cuSPARSE 3.97× (4.96×), CUSP 31.77× (28.93×), RMerge2 3.92× (2.88×), Nsparse 2.39× (3.34×), AC-SpGEMM 3.37× (3.78×), spECK 1.53× (2.35×), nonTCU 1.61× (1.84×). The total speedup for the sixteen matrices of our dataset on average (geometric mean) is: cuSPARSE 3.12× (3.64×), CUSP 10.91× (9.76×), RMerge2 1.76× (1.27×), Nsparse 1.18× (1.58×), AC-SpGEMM 1.51× (1.63×), spECK 0.81× (1.17×), nonTCU 1.59× (1.78×).

Third, tSparse owes its performance gain to both tiling and TCUs. nonTCU indicates the performance we gain by using tiling only. Across the matrices of our dataset, the speedup of nonTCU, on Titan RTX (RTX 2070), on average (geometric mean) is: cuSPARSE 1.97× (2.04×), CUSP 6.88× (5.49×), RMerge2 1.11× (0.72×), Nsparse 0.74× (0.89×), AC-SpGEMM 0.95× (0.92×), spECK 0.51× (0.66×).

Fourth, although the spGEMM approaches generally maintain their relative ranking, the speedup of tSparse over the other approaches is greater on the RTX 2070 system. The reason is that, although the GPU parts become significantly faster on Titan RTX, the CPU tasks of tSparse (e.g., memory allocation/deallocation) take the same time on both systems. tSparse scales slightly worse than the other approaches on the higher-end GPU due to the CPU bound tasks.

Fifth, tSparse shows better performance with denser tiles. Grouping NZ elements in tiles reduces the number of values to work on. According to [5], the main cost of ESC is sorting. Consequently, fewer values equals to less time sorting. Generally, tSparse has good performance for *bitmap density* greater than five.

6.3. Execution time analysis

In this section, we present the relative execution time of the main parts of tSparse, i.e., Task list creation, Sorting of the task list, Counting elements of the output and the MM itself (Fig. 11). We make three observations. First, creating the task list takes considerable time. The main reason is that during the task list phase we do most of the necessary memory allocations. Second, Sorting takes about 19%, Counting about 11%, MM about 16% and Compaction about 4% of the execution time. Third, the runtime of Counting is not much shorter than MM's. The reason is that Counting includes the execution time for both the counting kernel and the memory allocations that the MM kernel needs.

We note that in the execution time we include all allocations, including the allocation of the output. We also include the time for conversion from fp32 to fp16 (if the input is stored in fp32). However, we do not include the time for conversion to/from bitmap format for fair comparison to other approaches.

6.4. Numerical precision

TCUs accept 16-bit numbers as input, whereas TCUs perform MM and accumulation in 32-bit precision (mixed precision - Section 2.6). In this section, we show how tSparse fares w.r.t. precision in comparison to a full 32-bit approach (we use CUSP

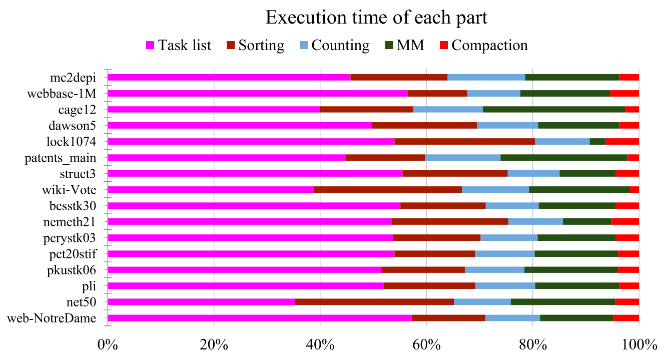


Fig. 11. Relative runtime of the main parts of tSparse on Titan RTX.

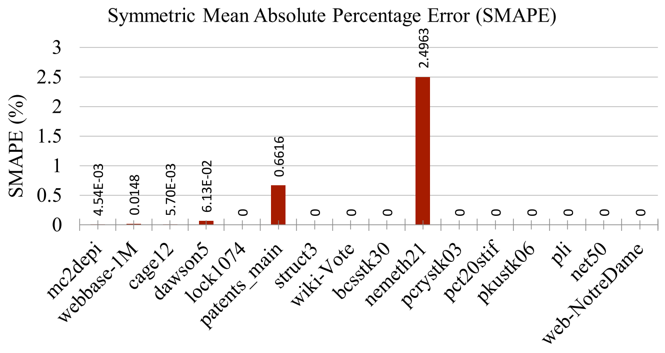


Fig. 12. Numerical precision of spGEMM in Symmetric Mean Absolute Percentage Error (SMAPE).

as base). Fig. 12 illustrates the precision of spGEMM as Symmetric Mean Absolute Percentage Error (SMAPE - Eq. (3)).

$$\frac{100\%}{n}\sum_{i=1}^{n}\frac{|x_i-\hat{x}_i|}{|x_i|+|\hat{x}_i|} \tag{3}$$

We make two observations. First, when the input matrices contain patterns of “1” and “0” the SMAPE is 0%. Second, when the inputs are real numbers the SMAPE is on average 0.02%. Exception are the cases with inputs close to the limits of the fp16 range, where the round-off error is bigger (nemeth21 - 2.49%, patents_main - 0.66%).

6.5. Memory requirements

In this section we present the peak memory consumption of all approaches. Fig. 13 presents the memory consumption for the 16 matrices of our dataset.

We make four major observations. First, hash table approaches require the smallest memory area as they do not have to store huge intermediate matrices. Second, ESC approaches require large amounts of memory. Third, tSparse, which “packs” elements into tiles, typically requires more memory than hash approaches and less than ESC. The amount of memory is proportional to the bitmap density. When the average bitmap density is very low, like in patents_main and wiki-vote, tSparse requires a lot of memory due to the additional overhead for storing tiles. Fourth, RMerge2 performs well as it usually requires only 5 bytes for each row of the left-hand side.

6.6. Summary

We examine the performance of our approach in $A * A$ and compare to cuSPARSE, CUSP, RMerge2, Nsparse, AC-SpGEMM and spECK over the 16 sparse matrices of our dataset. We draw two important conclusions. First, tSparse generally outperforms the other approaches for the larger matrices of our dataset when we have dense tiles and enough work per matrix row. Second, TCUs play an important role in the performance of our approach, speeding up nonTCU 1.68×.

tSparse outperforms the other state-of-the-art approaches 1.53× to 31.77× when our selected criteria are met, without significant loss in accuracy. Therefore tSparse is a suitable alternative for spGEMM.

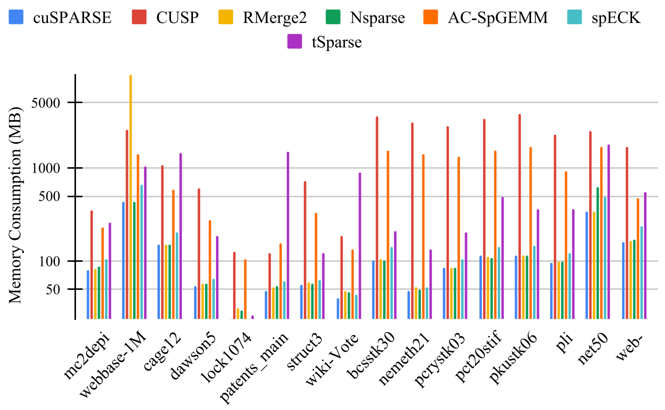


Fig. 13. Memory Consumption in MB.

7. Related work

To our knowledge, this is the first paper to use TCUs in the context of spGEMM. With this approach, we group NZ into tiles. Unlike previous methods, which use normal fp32 cores, we use tensor cores to multiply the tiles. We show that, with this approach, we can increase the performance of spGEMM. In this section we summarize other spGEMM implementations.

7.1. Expansion sorting compression

There exists a substantial body of work on improving ESC methodology that we describe in Section 2.4. Dalton et al. [5] improve on ESC by optimizing sorting, the most time consuming step, and by localizing processing to shared memory. Kunchum et al. [28], in HybridSparse, implement variants of ESC. Winter et al. [7], in AC-SpGEMM, perform ESC locally in shared memory. They use dynamic scheduling of iterations of ESC to keep data longer in shared memory. Thus, they reduce global memory traffic and the cost of sorting a huge intermediate matrix in global memory.

tSparse moves expansion and compression steps after sorting. This way we reduce memory allocation and movements and utilize MAC of TCUs.

7.2. Hash tables

Hash tables can mitigate the cost of sorting and storing huge intermediate matrices. Demouth et al. [19] present one of the first implementations of spGEMM with hash tables. cuSPARSE [11] is based on the work of Demouth et al. Their approach has two drawbacks. First, there is imbalance between threads because different threads of the warp might have to insert a different number of values in the hash table. Second, shared memory space is limited, which results in frequent data movement to global memory. Anh et al. [27], in BalancedHash, and Nagasaka et al. [20], in Nsparse, reduce the consumption of shared memory by partitioning the rows of the input or output, respectively. Nsparse improves on BalancedHash in two ways: (1) by using hash tables of variable size in shared memory, less shared memory is required and more thread blocks can run, and (2) by using fewer auxiliary matrices, it keeps memory traffic low and reduces memory storage requirements. Deveci et al. [30] use two-level hash tables. They adapt the hash tables to the number of threads in order to create a method that is portable to many platforms. Parger et al. [21], in spECK find a trade-off between analysis cost for load balancing and expected gain.

In tSparse, (1) we have less values to sort because elements are grouped into tiles, and (2) we avoid storing the intermediate products by using the task list to directly accumulate them.

7.3. Hybrid

Hybrid methods select among multiple methods/kernels during spGEMM depending on the workload. Dalton et al. [5] improve on ESC methodology by changing the thread granularity of the sorting method based on the size of rows of C. Liu et al. [6], in bhSparse, change the sorting-merging method depending on the size of rows of C. Kunchum et al. [28], in HybridSparse, choose the spGEMM method based on the workload of each row of A. They select among variants of ESC and their own method (a GPU implementation of scatter vectors). Gremse et al. [8,9], in RMerge create different kernels for different row sizes of A. Hybrid methods achieve good load balance thanks to their adaptability to the workload.

We can divide tSparse in two parts. The part that performs ESC on tiles to form the task list and the part that uses our kernels to perform MAC operations. For the first part, we use Thrust which achieves good load balance. For the second part, we let the GPU hardware scheduler manage the workload. This is possible because we employ many thread blocks, which the GPU schedules to SMs based on the available resources of each SM.

8. Conclusion

In this work, we utilize TCUs to increase the performance of spGEMM. To that end, we modify the ESC method and we create a task list of MMs of tiles. The key advantages of our approach, tSparse, are two. First, tiles reduce the number of values that the computationally demanding parts of ESC have to act on. Second, the task list sends the tiles to TCUs, which not only perform MM faster than normal computation cores, but also leave the normal cores free for different workloads.

The results confirm that TCUs increase the performance of MM and the combination of our tiling approach with TCUs provides significant benefits to spGEMM. TCUs increase the performance of tSparse by 68% in comparison to our nonTCU implementation. Our approach is, on average, $1.53\times$ to $31.77\times$ faster than cuSPARSE, CUSP, RMerge2, Nsparse, AC-SpGEMM and spECK when $NNZ(A) > 300000$ and $\widehat{RowA} > 42$. We conclude that our methodology improves the performance of spGEMM by making efficient use of tiles and TCUs. The source code of our approach is available at https://github.com/oresths/tSparse.

CRediT authorship contribution statement

Orestis Zachariadis: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Writing - original draft, Writing - review & editing, Visualization. **Nitin Satpute:** Conceptualization, Writing - review & editing, Formal analysis, Visualization. **Juan Gómez-Luna:** Conceptualization, Writing - review & editing, Supervision, Project administration, Funding acquisition. **Joaquín Olivares:** Conceptualization, Resources, Writing - review & editing, Supervision, Project administration, Funding acquisition.

Acknowledgments

This work is supported by High Performance Soft-tissue Navigation (HIPERNAV - H2020-MSCA-ITN-2016). HIPERNAV has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 722068.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.compeleceng.2020.106848.

References

[1] Davis T. Algorithm 9xx: SuiteSparse: Graphblas: graph algorithms in the language of sparse linear algebra. 2018.
[2] Gilbert JR, Reinhardt S, Shah VB. High-performance graph algorithms from parallel sparse matrices. In: International workshop on applied parallel computing. Springer; 2006, p. 260–9.
[3] Bell N, Dalton S, Olson L. Exposing fine-grained parallelism in algebraic multigrid methods. SIAM J Sci Comput 2012;34(4):C123–52. http://dx.doi.org/10.1137/110838844.
[4] Yamazaki I, Li XS. On techniques to improve robustness and scalability of a parallel hybrid linear solver. In: International conference on high performance computing for computational science. Springer; 2010, p. 421–34.
[5] Dalton S, Olson L, Bell N. Optimizing sparse matrix—Matrix multiplication for the GPU. ACM Trans Math Software 2015;41(4):1–20. http://dx.doi.org/10.1145/2699470.
[6] Liu W, Vinter B. A framework for general sparse matrix-matrix multiplication on GPUs and heterogeneous processors. J Parallel Distrib Comput 2015;85:47–61. http://dx.doi.org/10.1016/j.jpdc.2015.06.010, arXiv:1504.05022.
[7] Winter M, Mlakar D, Zayer R, Seidel H-P, Steinberger M. Adaptive sparse matrix-matrix multiplication on the GPU. In: Proceedings of the 24th symposium on principles and practice of parallel programming. New York, NY, USA: ACM; 2019, p. 68–81. http://dx.doi.org/10.1145/3293883.3295701.
[8] Gremse F, Höfter A, Schwen LO, Kiessling F, Naumann U. GPU-accelerated sparse matrix-matrix multiplication by iterative row merging. SIAM J Sci Comput 2015;37(1):C54–71. http://dx.doi.org/10.1137/130948811.
[9] Gremse F, Küpper K, Naumann U. Memory-efficient sparse matrix-matrix multiplication by row merging on many-core architectures. SIAM J Sci Comput 2018;40(4):C429–49. http://dx.doi.org/10.1137/17M1121378.
[10] Vasudevan A, Anderson A, Gregg D. Parallel multi channel convolution using general matrix multiplication. In: 2017 IEEE 28th international conference on application-specific systems, architectures and processors (ASAP). IEEE; 2017, p. 19–24.
[11] NVIDIA. CUDA C programming guide 10.2, November. 2019, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
[12] Google. Google cloud TPU. Google Cloud 2019. https://cloud.google.com/tpu/.
[13] Zhang J, Gruenwald L. Regularizing irregularity : Bitmap-based and portable sparse matrix multiplication for graph data on GPUs. GRADES-NDA 2018. http://dx.doi.org/10.1145/3210259.3210263, ISBN: 9781450356954.
[14] Dakkak A, Li C, Xiong J, Gelado I, Hwu W-m. Accelerating reduction and scan using tensor core units. In: Proceedings of the ACM international conference on supercomputing. ACM; 2019, p. 46–57.
[15] Haidar A, Tomov S, Dongarra J, Higham NJ. Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In: Proceedings of the international conference for high performance computing, networking, storage, and analysis; 2018. ISBN: 9781538683842.
[16] Markidis S, Chien SWD, Laure E, Peng IB, Vetter JS. NVIDIA tensor core programmability, performance & precision. In: Proceedings - 2018 IEEE 32nd international parallel and distributed processing symposium workshops. 2018, p. 522–31. http://dx.doi.org/10.1109/IPDPSW.2018.00091.
[17] Dalton S, Bell N, Olson L, Garland M. Cusp: Generic parallel algorithms for sparse matrix and graph computations. Version 0.5.1; 2015, http://cusplibrary.github.io/.
[18] Davis TA, Hu Y. The university of florida sparse matrix collection. ACM Trans Math Software 2011;38(1):1.
[19] Demouth J. Sparse matrix-matrix multiplication on the GPU. Nvidia; 2012, p. 21.
[20] Nagasaka Y, Nukada A, Matsuoka S. High-performance and memory-saving sparse general matrix-matrix multiplication for NVIDIA pascal GPU. In: 2017 46th international conference on parallel processing. Bristol, United Kingdom: IEEE; 2017, p. 101–10. http://dx.doi.org/10.1109/ICPP.2017.19.

[21] Parger M, Winter M, Mlakar D, Steinberger M. spECK: accelerating GPU sparse matrix-matrix multiplication through lightweight analysis. In: Proceedings of the 25th ACM sigplan symposium on principles and practice of parallel programming. San Diego California: ACM; 2020, p. 362–75. http://dx.doi.org/10.1145/3332466.3374521.
[22] Barrett R, Berry MW, Chan TF, Demmel J, Donato J, Dongarra J, et al. Templates for the solution of linear systems: building blocks for iterative methods, vol. 43. Siam; 1994.
[23] Matam K, Indarapu SRKB, Kothapalli K. Sparse matrix-matrix multiplication on modern architectures. In: 2012 19th international conference on high performance computing. 2012, p. 1–10. http://dx.doi.org/10.1109/HiPC.2012.6507483.
[24] Committee IS, et al. 754-2008 IEEE standard for floating-point arithmetic. IEEE Comput Soc Std 2008;2008:517.
[25] NVIDIA. Nvidia turing Gpu architecture whitepaper. 2018.
[26] Hou K, Liu W, Wang H, Feng W. Fast segmented sort on GPUs. In: Proceedings of the international conference on supercomputing; 2017. p. 1–0.
[27] Anh PNQ, Fan R, Wen Y. Balanced hashing and efficient GPU sparse general matrix-matrix multiplication. In: Proceedings of the 2016 international conference on supercomputing. New York, NY, USA: ACM; 2016, p. 36:1–36:12, http://dx.doi.org/10.1145/2925426.2926273.
[28] Kunchum R, Chaudhry A, Sukumaran-Rajam A, Niu Q, Nisa I, Sadayappan P. On improving performance of sparse matrix-matrix multiplication on GPUs. In: Proceedings of the international conference on supercomputing. New York, NY, USA: ACM; 2017, p. 14:1–14:11, http://dx.doi.org/10.1145/3079079.3079106.
[29] Xie Z, Tan G, Liu W, Sun N. IA-SpGEMM: An input-aware auto-tuning framework for parallel sparse matrix-matrix multiplication. In: Proceedings of the ACM international conference on supercomputing; 2019. p. 94–105.
[30] Deveci M, Trott C, Rajamanickam S. Multithreaded sparse matrix-matrix multiplication for many-core and GPU architectures. Parallel Comput 2018;78:33–46.

Orestis Zachariadis is a researcher and Ph.D. candidate at University of Cordoba. He works on parallel computing for medical image registration algorithms as part of HiPerNav EU project. In 2018, he worked on medical technology at SINTEF. In 2016, he joined MultiDrone EU project as a research assistant at University of Bristol, where he worked on object and human detection and following in videos shot from multiple drones. He holds a B.Sc. and an M.Sc. in Electrical and Computer Engineering from Aristotle University of Thessaloniki. His research interests include parallel computing, computer vision and robotics.

Nitin Satpute is a researcher and Ph.D. candidate at University of Cordoba. He works on GPU acceleration for medical image enhancement and segmentation algorithms as a part of HiPerNav EU project. In 2018, he worked on liver segmentation at OUH, Oslo, Norway. In 2019, he worked on liver enhancement at NTNU Gjovik Norway, where he developed fast parallel cross modality approach for liver contrast enhancement. He holds Master of Engineering in Embedded Systems from BITS Pilani, India. His research interests include GPU computing, Medical Imaging, Re-configurable Computing and Video Processing.

Juan Gómez Luna is a postdoctoral researcher at ETH Zürich. He received the BS and MS degrees in Telecommunication Engineering from the University of Sevilla, Spain, in 2001, and the Ph.D. degree in Computer Science from the University of Córdoba, Spain, in 2012. Between 2005 and 2017, he was a lecturer at the University of Córdoba. His research interests focus on Processing-in-Memory, GPUs and heterogeneous systems, medical imaging, and bioinformatics.

Joaquín Olivares was born in Elche, Spain. He received the B.S. and M.S. degree in Computer Sciences in 1997, and 1999, respectively, and the M.S. degree in Electronics Engineering in 2003, all from the Universidad de Granada, Spain. He received the Ph.D. degree in 2008 at the Universidad de Cordoba, Spain. He is Associate Professor with the Electronic and Computer Engineering Department at the Universidad de Cordoba, Spain, since 2001. He is founder and head of the Advanced Informatics Research Group. His research interests are in the field of embedded systems, computer vision, and high performance computing.