Releasing the Potential of Tensor Core for Unstructured SpMM using Tiled-CSR Format

Zeyu Xue, Mei Wen*, Zhaoyun Chen, Yang Shi, Minjin Tang, Jianchao Yang, Zhongdi Luo
Key Laboratory of Advanced Microprocessor Chips and Systems, National University of Defense Technology, Changsha, China
{xuezeyu18, meiwen, chenzhaoyun, shiyang14, tangminjin14, yangjianchao16, luoabner}@nudt.edu.cn
* Corresponding Author

***Abstract*—The GPU has become a popular platform for AI applications, thanks in part to its Tensor Cores that address performance issues. However, the Sparse Matrix Multiplication (SpMM) kernel has remained a bottleneck despite significant advances in computing power. Due to the hardware mechanism of the Tensor Core, its programming granularity does not match SpMM. In this paper, we analyze the reasons why the unstructured SpMM kernel is not suitable for the Tensor Core, and propose the Tiled Compressed Sparse Row (Tiled-CSR) compression format. To address the issue of low non-zero rates in Tiled-CSR format, we exploit the row shuffle algorithm to improve the utilization of Tensor Cores and enhance computing density. We also utilize adaptive memory access modes and 3D-Grid tiling for the SpMM kernel to reduce memory access latency. The experimental results on NVIDIA A100 GPU with matrices in the Deep Learning Matrix Collection (DLMC) demonstrate that the Tiled-CSR format improves the utilization of Tensor Cores under different sparsity, with a maximum of 3.89× at 50% sparsity and a minimum of 1.82× at 90% sparsity compared to the SR-BCRS format. Additionally, our kernel achieves an average speedup of 1.54×(up to 2.12×) over *Magicube*.**

***Index Terms*—unstructured sparse matrix multiplication, compression format, Tensor Core, bandwidth efficiency**

I. Introduction

To mitigate memory and computation overhead issue, the methods of reduced precision and sparsity have been employed [1]. Lower precision uses fewer bits for storage and computation, and the low-precision units can improve the computation throughput [2]. In addition, sparsity is exploited jointly to reduce expensive overhead of storage and computation quantity [3]. As one of the basic workload, the performance of Sparse Matrix Multiplication (SpMM) can be improved by exploring the sparsity.

To address the issue of low performance of SpMM, many high-performance kernels were designed based on the Compression Sparse Row (CSR) format [4]. SpMM kernels in *cuSPARSE* are targeted at high sparsity (e.g., $\geq 95\%$) [5]. *Sputnik* is a library proposed by Gale et al. that contains fine-grained SpMM kernels based on CSR format, addressing the issue that only at high sparsity can *cuSPARSE* well work [6]. It is state-of-the-art(SOTA) method on the CUDA Core. However, it does not utilize Tensor Cores with high computing power, but instead only deploys the SpMM workload on CUDA Cores. Chen et al. initially introduce *vectorSparse* that implements SpMM on NVIDIA V100 Tensor Cores [7]. It uses a sparse encoding with 1-D block (e.g., 8×1, 4×1, 2×1) to improve the proportion of computing valid data on Tensor Cores. Li et al. propose *Magicube*, a high-performance SpMM library with low-precision (8-bit, 4-bit) integers on Tensor Cores, which can perform mixed calculations with multiple precision [8]. Similar to *vectorSparse*, the data layout *Magicube* targeted at is 1-D blocks, which means it also lacks universality.

vectorSparse utilizes the 1st generation Tensor Core in V100 [9], while *Magicube* utilizes the 3rd generation Tensor Core in A100 [10], solving the problems of structured sparse data whose block vector-length is 8, 4 and 2. Because Tensor Cores have coarse programming granularity, it cannot adapt for fine-grained sparse data. Oriented to structured data, *vectorSparse* and *Magicube* use a specific compression layout to adapt the data to the input format of Tensor Cores, so as to achieve high proportion of computing valid data on Tensor Cores.

Structured data usually appears as continuous non-zero vector or block, while unstructured data is unordered and does not have spatial regularity. However, data is not always structured, and it has become a challenge to adapt for unstructured sparse data while maximizing the computational utilization of Tensor Cores. In unstructured SpMM workload based on CSR format, corresponding rows in the dense matrix need to be loaded according to the column index of the sparse matrix. The column indexes of different rows in the sparse matrix are inconsistent, which causes difficulties to merge multiple rows for calculation.

To address this issue, we introduce a novel sparse matrix compression format, Tiled Compressed Sparse Row (Tiled-CSR), based on the particularity of the Tensor Core programming. With the Tiled-CSR format applied, the effective computation of Tensor Cores is still not enough, due to possible lack of compressible all-zero columns between adjacent rows. We further propose the row shuffle algorithm to improve the computational utilization of Tensor Cores after data compression. We also propose adaptive memory access modes based on the difference of data reuse. In addition, based on the characteristics of GPU programming model, we divide the thread grid into 3-D grid to improve the bandwidth efficiency. Our main contributions are:

- **Tiled-CSR**: We design the *Tiled-CSR* format to adapt for the input format of Tensor Cores.

- **Row Shuffle**: We use the Algorithm 1, named *row shuffle*, to exchange rows, which can combine similar rows together to improves the computational utilization of Tensor Cores.
- **Adaptive Memory Access Modes**: We use *adaptive memory access modes* on the basis of different data-reuse. It can use different memory access instructions to improve the efficiency of bandwidth.
- **3D-Grid Tiling**: We introduce *3D-Grid tiling* to improve the bandwidth efficiency, which can reduce memory access latency and achieve higher performance.

We use matrices in the Deep Learning Matrix Collection (DLMC) [11] to test the performance of our work with different sparsity. The row shuffle algorithm achieves an increase in the proportion of effective computation on Tensor Cores with an average of 2.66$\times$. The lowest utilization improvement is 1.82$\times$ when the sparsity is 90%, and the highest utilization improvement is 3.89$\times$ when the sparsity is 50%. Results on NVIDIA A100 GPU demonstrate that our work reach the performance that speedup 1.54$\times$ over *Magicube*, and even achieve a 2.12$\times$ performance improvement at 98% sparsity.

II. Background and Motivation

A. Compression in Deep Learning

Deep Learning models can be sparse by applying techniques such as weight pruning, which involves removing unnecessary connections and weights from the models, or by directly training the model to have sparsity patterns. Theoretically, the compressed models require less memory storage and computation during inference or training. However, to efficiently utilize the sparsity in models is a challenge, especially on hardware without an unstructured sparsity mechanism. To address this challenge, many compression formats have been proposed, including Compressed Sparse Row format (CSR), Coordinate format (COO), Compressed Sparse Column format (CSC), Linked List format (LIL), Diagonal format (DIA), etc [4]. Among these compression formats, CSR format is the most typical and widely used, and its compression rule is shown in (b) of Fig. 2.

Several previous work has already utilized certain compression formats to accelerate SpMM. Li et al. have proposed the Stride Row-major Block Compressed Row Storage (SR-BCRS) in *Magicube* [8], which is a compression format based on the CSR format and tailored for structured sparse data. They utilize the K of the WMMA-instruction in the Tensor Core as stride to compress the data and deploy the workload on Tensor Cores. For 1-D block data, the smaller the vector-length(V), the lower the effective calculation rate of the Tensor Core, specifically V/M (M means height of the LHS matrix in the WMMA-instruction). This means that if we want to implement unstructured SpMM based on the CSR format on Tensor Cores, we can only utilize 1/M of the computing power of Tensor Cores. To simultaneously improve the computational efficiency of Tensor Cores while executing unstructured SpMM, in Section III-C, we propose a new compression format based on CSR format.

B. Tensor Core of NVIDIA GPU

TABLE I
Shapes of WMMA of fp16 in A100 Tensor Cores.

Precision	M	K	N
	16	16	16
fp16	8	16	32
	32	16	8

Since the Volta architecture, NVIDIA introduced the Tensor Core which provides warp-level APIs where 32 threads of one warp work together to perform specialized matrix load, matrix multiply and accumulate, and matrix store operations [9, 10]. The APIs define the shape of the matrix multiplication C = $A \times B$, where the output matrix C has a shape of $M \times N$, the Left-Hand-Side (LHS) matrix A has a shape of $M \times K$, and the Right-Hand-Side (RHS) matrix B has a shape of $K \times N$, as shown in Table I [12].

The Tensor Core outperforms CUDA Cores in terms of performance [13], as shown in Table II. In computationally intensive workloads such as matrix multiplication, the introduction of Tensor Cores can take full advantage of the memory bandwidth of the GPU. However, the input and output of the Tensor Core are fixed scale matrices, so its programming granularity is relatively coarse. Considering the significant computational redundancy of SpMM, deploying unstructured SpMM on Tensor Cores and effectively utilizing its computing power has become a key challenge.

TABLE II
Comparison of performance between Tensor Cores and CUDA Cores in V100 and A100. And comparison of performance between fp16 and int8 in A100.

Architecture	fp16(Tensor Core)	fp16(CUDA Core)	Speed Up
V100	125 TFLOPS	31.4 TFLOPS	4$\times$
A100	312 TFLOPS	78 TFLOPS	4$\times$
	fp16(Tensor Core)	int8(Tensor Core)	Speed Up
A100	312 TFLOPS	624 TOPS	2$\times$

C. Motivation

According to the characteristics of unstructured sparse matrix and CSR format, it is difficult to calculate multiple rows at the same time, so deploying the unstructured SpMM workload on Tensor Cores with coarse programming granularity is a challenge. In addition, because the computing power of Tensor Cores is relatively high, the bandwidth cannot satisfy its data needs frequently. Memory access optimization techniques and new memory access methods have been developed, so efforts need to be made on new optimization of bandwidth. Therefore, the motivation for our work are exploiting unstructured SpMM kernels on Tensor Cores and improving the bandwidth efficiency for Tensor Cores.

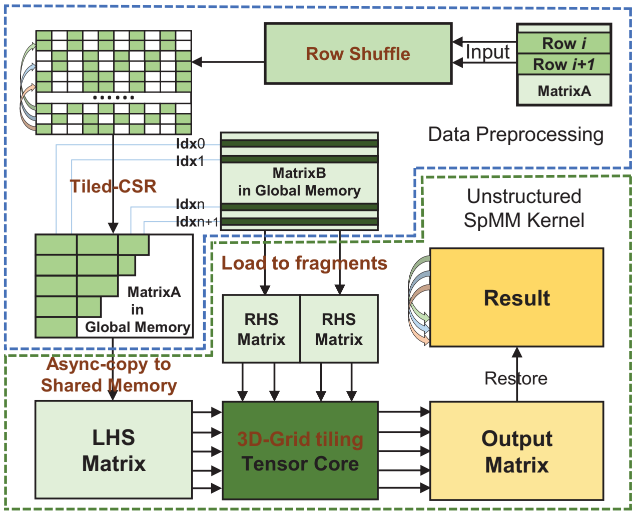
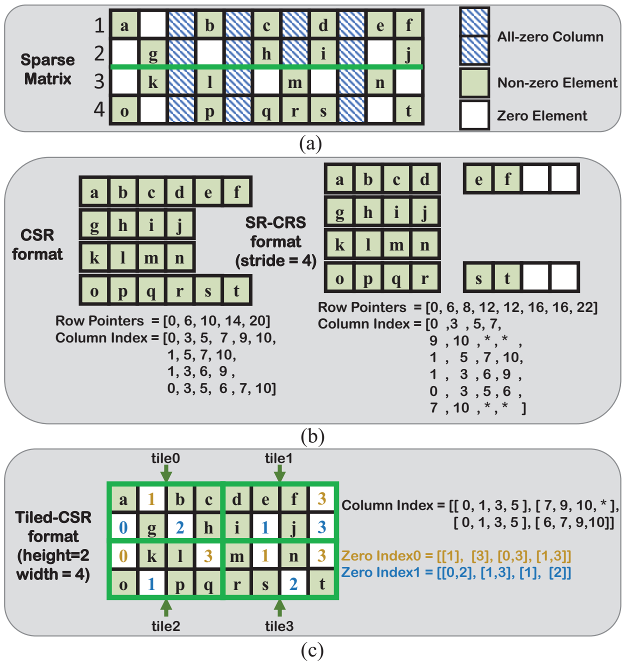


Fig. 1. Overview of our work. Data preprocessing includes Row Shuffle and the Tiled-CSR format compression. Optimization of SpMM kernel includes adaptive memory access modes and 3D-Grid tiling.



Sparse Matrix
All-zero Column
Non-zero Element
Zero Element
(a)
CSR format
SR-CRS format (stride = 4)
Row Pointers = [0, 6, 10, 14, 20]
Column Index = [0, 3, 5, 7, 9, 10, 1, 5, 7, 10, 1, 3, 6, 9, 0, 3, 5, 6, 7, 10]
Row Pointers = [0, 6, 8, 12, 12, 16, 16, 22]
Column Index = [0, 3, 5, 7, 9, 10, *, *, 1, 5, 7, 10, 1, 3, 6, 9, 0, 3, 5, 6, 7, 10, *, *]
(b)
tile0
tile1
tile2
tile3
Tiled-CSR format (height=2 width = 4)
Column Index = [[0, 1, 3, 5], [7, 9, 10, *], [0, 1, 3, 5], [6, 7, 9,10]]
Zero Index0 = [[1], [3], [0,3], [1,3]]
Zero Index1 = [[0,2], [1,3], [1], [2]]
(c)

Fig. 2. CSR format , SR-CRS format in *Magicube* and Tiled-CSR format in our work. (The three formats use the same sparse matrix in (a).)

III. Tiled Compression Sparse Row Format and Row Shuffle Algorithm

Our work is to accelerate unstructured SpMM, and the overview of our work is shown in Fig. 1. Data preprocessing is aimed at efficiently adapting unstructured sparse data for the computing patterns of Tensor Cores. And our work in the optimization of SpMM kernel part is to optimize CUDA code from the perspective of architecture and programming model. Data preprocessing is introduced in Section III, while the optimization of SpMM kernel in Section IV.

To address the issue raised in Section II-A - how to design a compression format for the Tensor Core programming and improve the computational utilization of Tensor Cores. In this section, we propose the Tiled-CSR format based on the Tensor Core input and introduce the row shuffle algorithm to enhance the utilization of Tensor Cores computation.

A. Tiled-CSR Compression Format

Although there are many compression formats, only SR-BCRS format is designed for Tensor Core programming. When using the SR-BCRS format in *Magicube* [8], as shown in (b) of Fig. 2, we can only compress data row by row due to the different column indices of non-zero elements in each row of unstructured sparse matrices. If the LHS matrix is compressed by SR-BCRS format and then entered into Tensor Cores for WMMA operations, the performance of the Tensor Core can only be utilized by 1/M. Because the LHS matrix only has one row of valid data for each computation after compression, while the Tensor Core has M rows.

To increase the proportion of effective Tensor Core computations, we propose the Tiled-CSR format. Unlike other compression formats, the Tiled-CSR format allows the existence of zeros after compression. The main principle of this format is to group M rows together and compress the all-zero columns in these M rows to ensure that each column has at least one non-zero element. The compressed data is then divided into tiles of the same size as the Tensor Core input requires.

As shown in the example of Fig. 2, supposing the input size of the LHS matrix is 2×4, i.e., height=2, width=4. We first delete the all-zero columns in every 2 rows, and then compress and divide the sparse matrix into tiles, each tile consisting of 4 columns.

As described in Section II-B, the WMMA-instruction specifies the shape of the matrix multiplication as M-K-N. To fully utilize the computing power of Tensor Cores, we compress the sparse data using the Tiled-CSR format with a height of M and a width of K. The smaller the value of M, the easier it is to find more all-zero columns, which reduces the size of the sparse matrix and compresses more data. As shown in Table I, there are three shapes for the fp16 WMMA-instruction. Therefore, we choose the instruction WMMA.m8n32k16, with a height of 8 and a width of 16 in the Tiled-CSR format.

Besides, in the Tiled-CSR format, we use Zero Index*i* to mark the position of uncompressed zero elements. Each Zero Index*i* represents the column coordinates of the zero element in the *i*-th row of all tiles. Taking (c) of Fig. 2 as example, height is 2, so there are Zero Index*0* and Zero Index*1*.

B. Row Shuffle Algorithm

Directly compressing sparse matrix with Tiled-CSR format cannot achieve high non-zero rate. Therefore, we propose the algorithm 1, called Row Shuffle, leverages the principles of Algorithm 2 and 3 from Section III-C to traverse the original

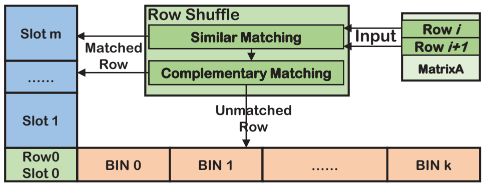


Fig. 3. Workflow of the row shuffle algorithm. The row shuffle algorithm includes similar matching and complementary matching.

Algorithm 1 Row Shuffle

```
Input: Sparse Matrix
Output: new Sparse Matrix
 1: new Sparse Matrix = []
 2: done_ROW = []
 3: for $ROW_i$ in Sparse Matrix do
 4:   Slot = []
 5:   Bin = []
 6:   if $ROW_i$ not in done_ROW then
 7:     add $ROW_i$ to Slot
 8:     add $ROW_i$ to done_ROW
 9:     for $ROW_j$ in Sparse Matrix do
10:       if $ROW_j$ not in done_ROW then
11:         if Similar Matching($ROW_j$, $ROW_i$, threshold)
            then
12:           add $ROW_j$ to Slot
13:           add $ROW_j$ to done_ROW
14:         else if Complementary Matching($ROW_j$,
            Slot, k) then
15:           add $ROW_j$ to Slot
16:           add $ROW_j$ to done_ROW
17:         else
18:           add $ROW_j$ to Bin
19:         end if
20:         if len(Slot) == 8 or len(Bin) == K then
21:           add Slot to new Sparse Matrix
22:           Break
23:         end if
24:       end if
25:     end for
26:   end if
27: end for
```

sparse matrix, identify similar rows, and merge them for compressing using the Tiled-CSR format.

As illustrated in Fig. 3, the main principle of the row shuffle algorithm is to pop slots when either slots or bins are full. When slots are filled, they are compressed as a group. If bins are filled, it means that within the length of the bins, enough similar rows could not be found to fill slots. To maximize the computational efficiency of Tensor Cores, the length of the bins can be set to be the same as the height of the sparse matrix, ensuring that the row shuffle algorithm can traverse each row and not miss any similar rows. However, if the length of the bins is set to the height of the sparse matrix, the time complexity of the row shuffle algorithm would be O(M^2).

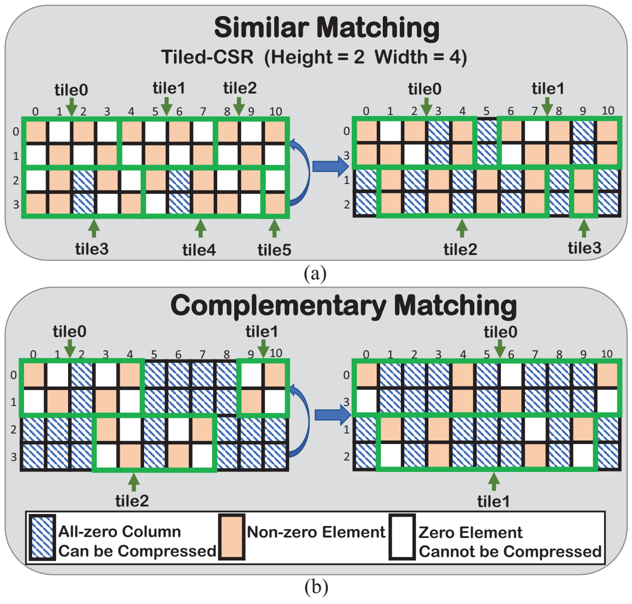

Similar Matching
Tiled-CSR (Height = 2 Width = 4)
tile0
tile1
tile2
tile3
tile4
tile5
Complementary Matching
All-zero Column Can be Compressed
Non-zero Element
Zero Element Cannot be Compressed
(a)
(b)

Fig. 4. Two match conditions for the row shuffle algorithm. Similar matching arranges similar rows together for tile compression. Complementary matching allows dissimilar rows to form tiles as the input of Tensor Cores.

To address the issue of high time complexity, we presort the matrix to cluster potentially similar rows together. Specifically, we presort the rows in the matrix by $\sum col^2$. And after presorting, reducing the length of bins to a constant K would not affect the result of the row shuffle algorithm, which makes the time complexity of the row shuffle algorithm down to O(*KM*). Moreover, the time complexity of sort algorithm (e.g., MergeSort) is O(*MlogM*) [14], and the time complexity of the row shuffle algorithm ,O(*KM*), is lower than sort algorithm. On account of this, we reduce the time complexity of the row shuffle algorithm from O(M^2) to O(*MlogM*).

C. *Match Requirements For the Row Shuffle Algorithm*

Algorithm 2: First, V_1 and V_2 are merged to form V_3. If the length of V_3 is smaller than the threshold times the length of V_1 or V_2, the match is successful, otherwise not. The purpose of this algorithm is to match rows with more overlap, resulting in more all-zero columns in the Tiled-CSR format and reducing zeros that cannot be compressed.

Algorithm 3: Algorithm 3 is suitable for situations where some rows may not meet the matching criteria of Algorithm 2, but their length is small enough such that several rows can be combined without exceeding the tile size. This allows for a single execution of the Tensor Core WMMA-instruction to obtain the result. If these rows are calculated separately, each row contains very few valid elements, and running them separately would waste computing power. Although the

Algorithm 2 Similar Matching

```
Input: $V_1$, $V_2$, Threshold
$Len_1$ = len($V_1$)
$Len_2$ = len($V_2$)
$V_3$ = $V_1 \cup V_2$
$Len_3$ = len($V_3$)
if $(Len_3 < (1 + threshold) \times Len_1) or (Len_3 < (1 + threshold) \times Len_2)$ then
    return $true$
else
    return $false$
end if
```

Algorithm 3 Complementary Matching

```
Input: $V_1$, Slots, k
$V_2$ = $V_1 \cup$ Slots[0] $\cup$ Slots[1] $\cup$ Slots[2] $\cup$ ...
$Len_2$ = len($V_2$)
if $(Len_2 < k)$ then
    return $true$
else
    return $false$
end if
```

effective utilization after combination is not high, it is still several times higher than before.

To further understand Algorithm 2 and 3, take Fig. 4 as an example. In (a) of Fig. 4, there are 22 non-zero elements. Before exchanging rows, only 2 columns of zeros can be fully compressed, requiring six tiles after compression, with an effective computation ratio of 22÷(6×2×4) = 0.4583. After that, there are 9 all-zero columns, requiring only four tiles after compression, with an effective computation ratio of 22÷(4×2×4) = 0.6875. This represents a 22.92% improvement in the computational efficiency of Tensor Cores.In (b) of Fig. 4, the results of similar matching between these four rows are not good. If the compression is performed without exchanging the rows, it would require three tiles to complete the calculation. However, after exchanging ROW_0 and ROW_3, there are only 4 non-zero columns, and the same is true for combining ROW_1 and ROW_2, which means that they only need 2 tiles totally to complete the calculation.

IV. SPMM KERNEL DESIGN

We use kernel code generator to generate CUDA kernels, as shown in Fig. 5. Adaptive Memory Access Modes replace memory access instructions in the kernel. And 3D-Grid Tiling adjusts kernel parameters based on input scale to optimize bandwidth utilization.

```
template <int BlockM, int BlockN, dim3 blockDim>
__global__ void SpMMkernel(int M, int N,
    SparseMatrix A, Matrix B, Matrix C){
int n_blockX = N / BlockN;
int n_blockY = M / BlockM;
int aspect = BlockN / BlockM;
int Grid_n_blockX = sqrt(num_SM / aspect);
int GirdX = n_blockX / aspect;
int GirdY = n_blockY;
int GirdZ = aspect;
dim3 gridDim(GirdX, GirdY, GirdZ);
SpMM<<<$gridDim, blockDim$>>>(M,N,A,B,C);}
```

Fig. 5. Pseudocode for SpMM Kernel

A. Adaptive Memory Access Modes

The A100 SM also supports asynchronous data memory access through its LD/ST modules, which can directly move data from DRAM/L2 cache to Shared Memory. The Tensor core instructions have also been updated to read 2 values per instruction, reducing on-chip storage's bandwidth requirements and L1 cache capacity demands. However, the PTX Inline Assembly is required to access non-contiguous data addresses in the Fragments.

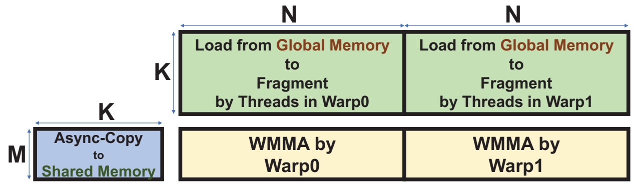

N
N
K
Load from Global Memory to Fragment by Threads in Warp0
Load from Global Memory to Fragment by Threads in Warp1
K
M
Async-Copy to Shared Memory
WMMA by Warp0
WMMA by Warp1

Fig. 6. Adaptive memory access modes. LHS matrix is directly copied to Shared Memory due to its re-usability. RHS matrix is directly copied to Fragments by different warps.

Due to the high reuse of the LHS matrix, it is loaded into Shared Memory using asynchronous copy APIs and waits to be called by each warp in the thread block. On the other hand, like Fig. 6, the RHS matrix has low reuse, and its non-contiguous data is loaded directly into the Fragments through PTX Inline Assembly.

B. 3D-Grid Tiling

The A100 has a total of 108 SMs, and each SM can only accommodate a certain number of thread blocks. Here we assume that one SM can only accommodate one thread block. When the SpMM scale is very large, the number of thread blocks will exceed 108, and they will be scheduled in a loop according to the order of gridDim.z$->$gridDim.y$->$gridDim.x. In general, when designing the programming model, the matrix is considered as a two-dimensional data format, that is, gridDim.z $\equiv$ 0. Thus, the first 108 Blocks are called in the order shown in the left figure of Fig. 7. Although the LHS matrix has good locality, the access locality of the RHS matrix is extremely poor. If the matrix is now split into three dimensions, i.e., expanding

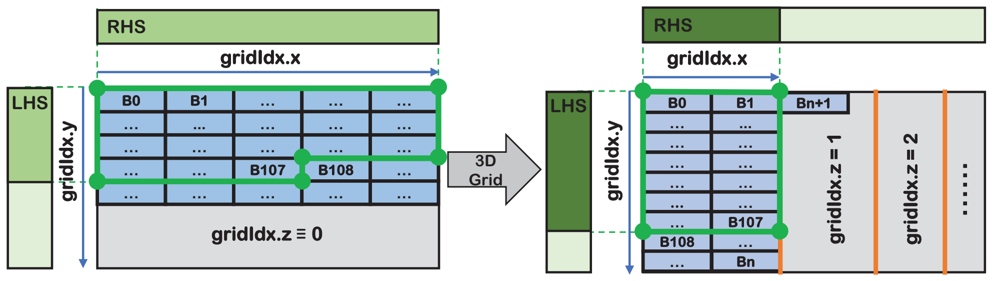


Fig. 7. 2D-Grid tiling vs 3D-Grid tiling. Before 3D-Grid tiling, cache hit ratio of RHS matrix is low. After 3D-Grid tiling, cache hit ratio of LHS and RHS get balanced.

gridDim.z, the scheduling will be as shown in the right figure of Fig. 7. This way, the locality of both the LHS matrix and the RHS matrix is balanced. We call this method 3D-Grid tiling.

Although 3D-Grid tiling cannot change the bandwidth, it increases the L2 cache hit ratio, improves the bandwidth efficiency of L2 Cache, reduces the bandwidth efficiency of DRAM, and thus reduces access latency. In addition, 3D-Grid tiling is only for large-scale matrix multiplication. If there are only a few thread blocks, the performance impact of using this method for scheduling the thread blocks is not significant.

V. Experimental Results

We conduct extensive experiments on the SpMM design. This section shows the advantages of algorithm and performance.

A. Experiment Setup

Hardware Environments: In this section, we elaborate the experimental evaluation of the SpMM kernels on parallel processing architecture: NVIDIA A100 Tensor Core GPU (108 Ampere SMs, 40GB Global Memory with bandwidth of 1555GB/s, 40MB L2 Cache, 192KB configurable L1/Shared Memory unified cache per each SM).

NVCC version is 11.2, and the GPU driver version is 525.85.12. In time statistics, we don't consider the data transfer time from CPU to GPU. We calculate the throughput by this formula: 2 × A_nnz × N / latency.

Matrices Benchmark: We select 2716 sparse matrices in the DLMC [11], whose matrices have different sparsity from the Transformer Model.

Baseline: For the combination of Tiled-CSR format and row shuffle algorithm, we compare with SR-BCRS and unprocessed matrices. For SpMM design, we compare our codes with Sputnik [6], Magicube [8] and cuSPARSE [5], and set the Magicube(L16-R16) as the baseline. In addition, due to *Magicube* executing structured 1-D block SpMM, we set the vector-length of 1-D block to 1 (i.e., unstructured data) to facilitate the comparison.

B. Benefits of Row Shuffle Algorithm

The purpose of row shuffle algorithm is to improve the non-zero rate in Tiled-CSR format. As described in Section III-A, the size of Tiled-CSR format in our work is 8×16. Since it

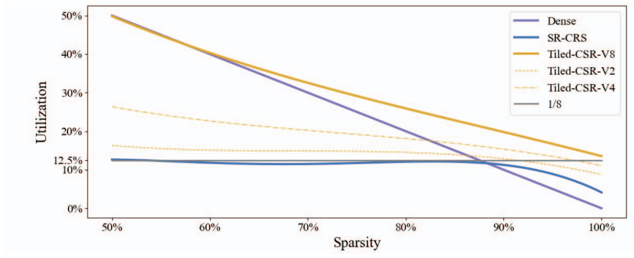


Fig. 8. The proportion of computing valid data on Tensor Cores with different format.

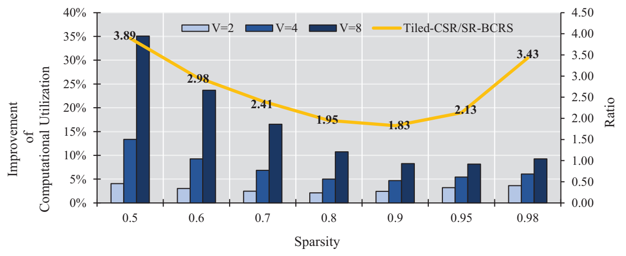


Fig. 9. The improvement of valid computing on Tensor Cores.

is oriented to unstructured sparse data, SR-BCRS format can only utilize one row of computing resources in Tensor Cores for each calculation, so it cannot exceed a utilization rate of 12.5%. If the sparse matrix is unprocessed, the utilization rate is equal to the non-zero rate of the dense matrix. After using the combination of the Tiled-CSR format and the row shuffle algorithm, the computational utilization of Tensor Cores is equal to or slightly higher than that of dense data at low sparsity, while it is much higher than other formats at high sparsity, as shown in Fig. 8. Besides, we set the height in the Tiled-CSR format to 2, 4, and 8 that are consistent with the vector-length in the 1-D block, to test row shuffle algorithm.

As shown in Fig. 9, the improvement in valid computing of the Tiled-CSR format compared to SR-BCRS format on Tensor Cores can be observed. Analysis shows that the highest improvement is at 50% sparsity, which increases the valid computing of Tensor Cores by 35% (equivalent to 3.89 times the original). At 90% sparsity, the improvement is the lowest,

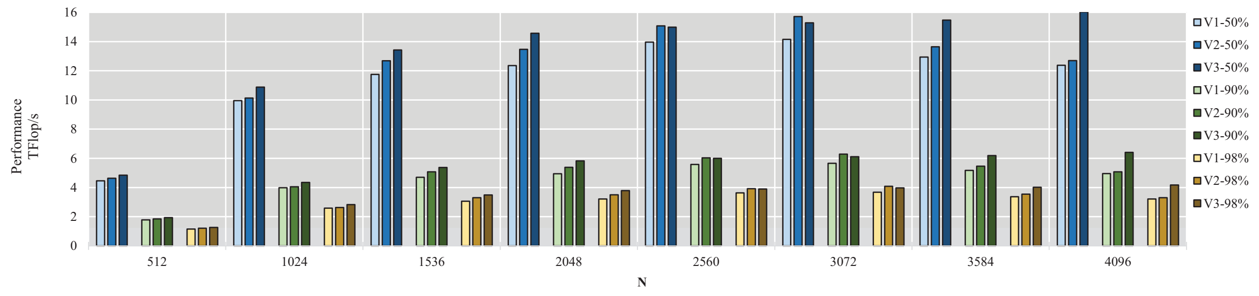


Fig. 10. Ablation Experiment: performance evaluation for two optimization methods of SpMM using matrices with different size(form 512 to 4096) and sparsity(50%, 90%, 98%). V1 is the basic version without neither methods. V2 takes the method of adaptive memory access modes. V3 further takes the method of 3D-Grid tiling.

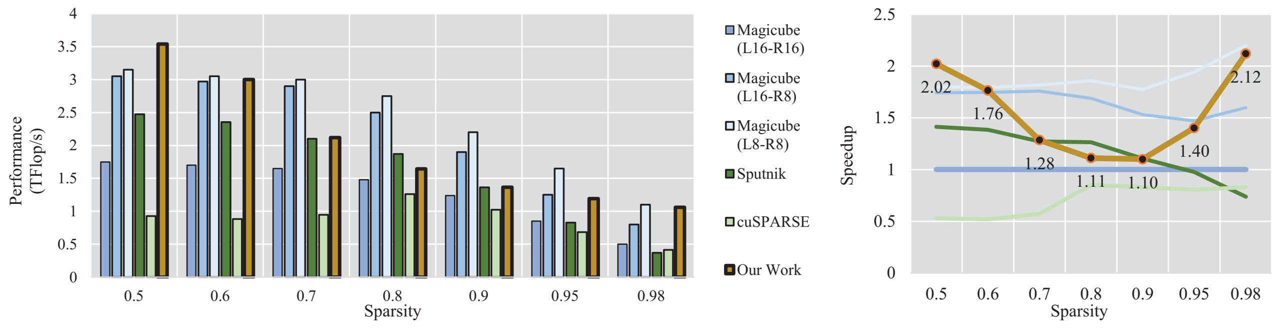


Fig. 11. Left sub-figure is the performance of unstructured SpMM kernels on A100 with different sparsity. Right sub-figure is the speedup of unstructured SpMM kernels over *Magicube*(L16-R16).

only increasing by 8% (equivalent to 1.82 times the original). We analyze this phenomenon from the aspect of Algorithm 2 and Algorithm 3. Although the effect of Algorithm 2 decreases with increasing sparsity, Algorithm 3 can play a role in high sparsity. That is why row shuffle algorithm is superior to other reorder algorithms.

C. Ablation Study

With this ablation study, we can see both the optimization methods discussed in Section IV (including adaptive memory access modes and 3D-Grid tiling) are very effective, as shown in in Fig. 10. We evaluate the optimization brought as the N of SpMM varies from 512 to 4096. The results indicate that optimization methods can achieve a performance improvement of 1.3×.

Adaptive memory access modes are suitable for any situation, as they dynamically use memory access instructions based on data reuse, resulting in improved performance of V2. However, 3D-Grid tiling can only function when it is relatively large in scale. We can observe that as the matrix size increases (e.g.,>3000), the performance of V1 and V2 begins to decrease, as the Cache miss begins to increase. The performance of V3 has remained stable increasing and not been affected by changes in size of matrices.

D. Comparison With Other Work

Next, we compare the performance of our work with other libraries. The left bar chart in Fig. 11 shows the performance of SpMM. Because the target precision of our work is fp16, while *Magicube* is oriented to integer SpMM. Fortunately, the *Magicube* uses an emulation for mixed precision to simulate the results of int16 (i.e., L16-R16). Its throughput is equal to fp16 operations on Tensor Cores. This experiment mainly compares our work with *Magicube* (L16-R16), *Sputnik*, and *cuSPARSE* (fp16). In addition, results of L16-R8 and L8-R8 in *Magicube* are also presented in the chart.

Magicube and *Sputnik* are the SOTA methods on the Tensor Core and the CUDA Core, respectively. And our work always outperforms *Magicube*(L16-R16), even surpassing the performance of lower precision kernels(e.g., *Magicube*(L16-R8), *Magicube*(L8-R8)) at some sparsity (e.g., 50%, 60%, 98%). The performance of our SpMM design basically exceeds that of Sputnik, except at 80% sparsity. We analyze the reason for that is row shuffle algorithm is not suitable for sparsity around 80%, as described in Section V-B. Although the computing power of the Tensor Core is much higher than that of the CUDA Core, the programming granularity of the CUDA Core is much finer.

The line chart of Fig. 11 illustrates the speedup over *Magicube*(L16-R16). The performance of our kernel reaches a maximum of 2.12× speedup at 98% sparsity and a minimum of 1.10× speedup at 90% sparsity. Furthermore, our kernel can ultimately achieve an average performance of 1.54× relative to *Magicube* (L16-R16).

VI. Related Work

As shown in Table III, efforts have been made on SpMM to exploit the benefit of parallel architecture like GPUs. NVIDIA has developed *cuSPARSE* and *cuSPARSELt* [5, 15]. *cuSPARSE* can execute unstructured SpMM that targets 95% or higher sparsity on CUDA Cores. *cuSPARSELt* utilizes the hardware mechanism in the 3rd generation Tensor Core, targeting 2:4 sparsity. Huang et al. introduce *Ge-SpMM*, a SpMM Library for GNN [16]. Gale et al. develop *Sputnik*, a high performance sparse library that targets fine-grained sparsity [6]. Besides, *Ge-SpMM* and *Sputnik* are both based on the CSR format and implement SpMM on CUDA Cores.

Chen et al. introduce *vectorSparse*, which implements the SpMM kernel [7]. Different from previous work, *vectorSparse* implements kernels in Tensor Cores instead of CUDA Cores. Similar to *vectorSparse*, Li et al. develop *Magicube* on Tensor Cores [8]. They propose SR-BCRS format based on CSR format, which is more suitable for Tensor Core programming. However, both *Magicube* and *vectorSparse* can only implement SpMM for 1-D blocks and cannot adapt to unstructured sparse data.

TABLE III
Libraries supporting SpMM on GPU. (TC : Tensor Core)

Library	Precision	Granularity	Support TC
cuSPARSE[5]	fp64/fp32/fp16 bf16/int32/int8	unstructured/ block	No
cuSPARSELt[15]	fp16/bf16/int8	2:4 sparsity	Yes
Ge-SpMM[16]	fp32	unstructured	No
Sputnik[6]	fp32/fp16	unstructured	No
vectorSparse[7]	fp16	1-D block	Yes
Magicube[8]	int16/int8/int4	1-D block	Yes
Our Work	fp16	unstructured	Yes

VII. Conclusion

The main contribution of our work is deploying unstructured SpMM workload on Tensor Cores, which can be divided into two parts: data preprocessing and SpMM kernel optimization. The combination of the Tiled-CSR format and the row shuffle algorithm can adapt unstructured data for Tensor Core programming and improve the proportion of computing valid data. Adaptive memory access modes use different memory access instructions to improve the bandwidth efficiency based on the different characteristics of data reuse in the LHS and RHS matrices. And 3D-Grid tiling can improve the L2 Cache hit ratio.

The experiments on NVIDIA A100 GPU conclude that the combination of the Tiled-CSR format and the row shuffle algorithm can achieve on average 2.65 × utilization of Tensor Cores. And our SpMM kernel can achieve on average 1.54×(up to 2.12×)speedup over *Magicube*(L16-R16). Our approach addresses the challenges of achieving high performance in GPU programming and offers valuable insights into the unique features of GPU architectures for high-performance computing applications.

References

[1] Sambhav Jain et al. “Trained Quantization Thresholds for Accurate and Efficient Fixed-Point Inference of Deep Neural Networks”. In: *MLSys*. 2020.

[2] Ganesh Venkatesh, Eriko Nurvitadhi, and Debbie Marr. “Accelerating Deep Convolutional Networks using low-precision and sparsity”. In: *ICASSP*. 2017.

[3] Bing Li et al. “Running sparse and low-precision neural network: When algorithm meets hardware”. In: *ASP-DAC*. 2018.

[4] Zhaojun Bai et al. *Templates for the Solution of Algebraic Eigenvalue Problems*. Ed. by Zhaojun Bai et al. 2000.

[5] NVIDIA. *cuSPARSE*. 2023. URL: https://docs.nvidia.com/cuda/cusparse/index.html.

[6] Trevor Gale et al. “Sparse GPU Kernels for Deep Learning”. In: *SC*. 2020.

[7] Zhaodong Chen et al. “Efficient Tensor Core-Based GPU Kernels for Structured Sparsity under Reduced Precision”. In: *SC*. 2021.

[8] Shigang Li, Kazuki Osawa, and Torsten Hoefler. “Efficient Quantized Sparse Matrix Operations on Tensor Cores”. In: *SC*. 2022.

[9] *NVIDIA TESLA V100 GPU ARCHITECTURE*. https://images.nvidia.cn/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. NVIDIA. 2017.

[10] *NVIDIA A100 Tensor Core GPU Architecture*. https://images.nvidia.cn/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf. NVIDIA. 2020.

[11] Google. *Deep Learning Matrix Collection(DLMC)*. 2020. URL: https://github.com/google-research/google-research/commits/master/sgk.

[12] Md Aamir Raihan, Negar Goli, and Tor M. Aamodt. “Modeling Deep Learning Accelerator Enabled GPUs”. In: *ISPASS*. 2019.

[13] Jack Choquette and Wish Gandhi. “NVIDIA A100 GPU: Performance Innovation for GPU Computing”. In: *HCS*. 2020.

[14] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 2009.

[15] NVIDIA. *Exploiting NVIDIA Ampere Structured Sparsity with cuSPARSELt*. 2020. URL: https://developer.nvidia.com/blog/exploiting-ampere-structured-sparsitywith-cusparselt/.

[16] Guyue Huang et al. “GE-SpMM: General-Purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks”. In: *SC*. 2020.