Bitmap-Based Sparse Matrix-Vector Multiplication with Tensor Cores

YuAng Chen
The Chinese University of Hong Kong
yuangchen@cuhk.edu.hk

Jeffery Xu Yu
The Chinese University of Hong Kong
yu@se.cuhk.edu.hk

ABSTRACT

Sparse matrix-vector multiplication (SpMV) plays a crucial role in various scientific and engineering tasks. Thus, extensive research efforts are devoted to enhancing its performance. In this work, we investigate the utilization of the tensor cores — hardware originally designed for dense matrix multiplications — for SpMV. By reverse engineering the architecture of tensor cores, we gain important insights into their internal register layout and develop a technique for direct access to these registers. Building on these findings, we propose *Spaden*, a method for accelerating SpMV using tensor cores. Spaden comprises two main components: (1) a bitmap-based format that achieves compression for the sparse matrix while preserving its rectangular shape, and (2) a pairing kernel, facilitating efficient execution of SpMV on tensor cores by enabling precise register-level control. Performance evaluation are conducted on Nvidia V100 and L40 GPUs. Compared with state-of-the-art approaches, Spaden demonstrates substantial speed advantage and memory efficiency, achieving, for example, a 1.63× speedup and 2.83× memory saving over cuSPARSE CSR on Nvidia L40.

KEYWORDS

SpMV, Tensor Cores, GPUs

ACM Reference Format:
YuAng Chen and Jeffery Xu Yu. 2024. Bitmap-Based Sparse Matrix-Vector Multiplication with Tensor Cores. In *The 53rd International Conference on Parallel Processing (ICPP '24), August 12–15, 2024, Gotland, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3673038.3673055

1 INTRODUCTION

Sparse matrix-vector multiplication (SpMV) serves as the foundational components for a wide range of scientific computing and engineering applications. For instance, graph algorithms (e.g., PageRank, BFS) are oftentimes converted into linear algebraic formulations, wherein the graph is represented as a sparse matrix multiplied with a vector [11, 27]. Additionally, SpMV finds extensive deployments in various machine learning algorithms, such as Support Vector Machines (SVMs) [32] and Collaborative Filtering (CF) [37]. Due to the importance of SpMV, extensive research has been conducted for its acceleration on CPUs [4, 28, 42] as well as GPUs [14, 24, 25].

The SpMV operation can be expressed as the equation $y = Ax$, where A is a sparse matrix and x and y are input and output dense vectors. Despite its seemingly simplicity, achieving efficient processing of SpMV on parallel computing platforms has proven to be a challenging task [22, 42, 44]. The difficulty essentially lies in the irregular access pattern of the matrices, which depends on the sparsity structure of the matrix [16]. To address these challenges, various optimization techniques have been developed, including vectorization [28], reordering [2], blocking [4] and reformatting [14].

Recent advancements in HPC hardware have introduced dense matrix accelerators, which are hardware units commonly integrated into CPUs or GPUs, including Nvidia's Tensor Cores, Apple's Matrix Coprocessor, AMD's Matrix Cores, and Google's TPU. These accelerators are specifically designed for general matrix multiplication (GEMM) of deep neural networks [26], where the inputs consist of two dense matrices.

Ongoing research is now extending the usage of the dense matrix units for sparse matrices. Applications include the sparse matrix-matrix (SpMM) [7] and the sparse general matrix-matrix (SpGEMM) [45], where the left-hand side matrix is sparse and the right-hand side may be either dense or sparse. Also, researchers are exploring the utilization of tensor cores to accelerate various algorithms, such as reduction and scan computations [9] and iterative solvers [17].

Deploying SpMV on the dense matrix unit poses greater challenges compared to the aforementioned works. Consequently, it received less attention in research. Take Nvidia's L40 as an example; its tensor cores strictly require inputs in the form of fixed-size 2D matrices (e.g., 16x16) flattened as 1D arrays. This conflicts with the format of sparse matrices, which are typically compressed in formats like Compressed Sparse Row (CSR). Moreover, the right-hand side input of SpMV is a vector, further complicating compatibility with the requirements of tensor cores.

Therefore, the primary challenge in leveraging tensor cores lies in efficiently loading sparse matrices and dense vectors onto the tensor core for computation. This paper offers insights into the previously undisclosed details of the register-level utilization of tensor cores. Through reverse engineering, we uncover the undocumented information about tensor cores, specifically deciphering the mapping between register indices and thread IDs on the tensor cores. The established mapping enables directly loading/storing to the (portion of) tensor cores, skipping the conventional data preparation overhead.

Building upon the insights outlined above, we propose *Spaden*, which utilizes tensor cores to processing SpMV. Firstly, Spaden converts the sparse matrix into a bitmap-based compressed format. This format not only saves more memory space compared to conventional compression formats but also aligns with the strict layout requirements of tensor cores. Secondly, an efficient kernel is devised to efficiently transfer the SpMV workloads to tensor cores

ICPP '24, August 12–15, 2024, Gotland, Sweden
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1793-2/24/08.
https://doi.org/10.1145/3673038.3673055

and retrieve the results from them. In summary, the contributions of this paper can be generalized as follows:

- We present a reverse engineering process that unveils an efficient approach to load and store data to the tensor cores.
- Based on the insights, we introduce Spaden, a novel approach that leverages tensor cores to accelerate SpMV computations. Spaden consists of two components: a tensor-friendly format and the corresponding computing kernel.
- A bitmap-based format is devised, which not only efficiently compresses sparse matrices but also facilitates their mapping onto tensor cores for processing.
- Paired with the customized format, a kernel is developed to efficiently manipulate the tensor core at the register level.
- In comparison with other SpMVs on different GPUs, Spaden demonstrates significant performance gains across a range of matrices. To better understand its effectiveness and efficiency, Spaden is decomposed to identify the contributing factors and assess its overhead.

This paper is organized as follows: Section 2 discusses background, Section 3 reverse engineers Nvidia tensor cores, Section 4 details the design of our proposal, Section 5 evaluates the performance, Section 6 provides related work and Section 7 concludes the paper.

2 BACKGROUND

2.1 SpMV with CSR

The Compressed Sparse Row (CSR) format provides a memory-efficient means of storing sparse matrices. It retains only the non-zero entries of a matrix, row by row, through the use of three arrays: `row_pointers`, `col_indices`, and `values`. The `row_pointers` array denotes the starting points of each row within the `col_indices` and `values` arrays. The difference between consecutive elements in `row_pointers` (i.e., `row_pointers[i+1] - row_pointers[i]`) signifies the number of nonzero entries per row. Meanwhile, the `col_indices` and `values` arrays keeps the column indices and corresponding values of the non-zero entries, respectively.

Algorithm 1: Standard CSR SpMV

```
for i = 0 to nRows-1 do                      ▷ allow parallelization
    sum = 0
    for j= row_pointers[i] to row_pointers[i+1] do
        sum += values[j] * x[col_indices[j]]
    y[i] = sum
```

The SpMV operation, represented as $\mathbf{y} = \mathbf{Ax}$, involves a sparse matrix $\mathbf{A}$ and input/output dense vectors $\mathbf{x}$ and $\mathbf{y}$. Using CSR format, SpMV is formalized as Algorithm 1. Accesses to arrays `row_pointers`, `values`, and `y` are entirely sequential, while access to `x` is determined randomly by the elements of `col_indices`. Because there are no write conflicts for `y`'s elements, CSR SpMV can be easily parallelized by rows, making it a standard choice for SpMV across various sparse mathematics libraries [10, 11, 27].

SpMV on GPUs. Various optimization techniques and storage formats are developed for improving the performance and efficiency of SpMV on GPUs [14]. These include CSR for its row-wise storage efficiency, ELL for its fixed number of non-zero entries per row, HYB to combine the advantages of CSR and ELL, COO for its simplicity in representing sparse matrices, and DIA for matrices with diagonal patterns. Each technique aims to optimize different aspects of SpMV, such as memory usage, parallelism, and computational overhead, making them suitable for different types of sparse matrices and computational environments.

LightSpMV [24] proposes an optimized CSR-based SpMV by introducing fine-grained dynamic row distribution strategies to addresses the issue of imbalanced parallelization. Gunrock [40] implements SpMV as message passing on graph edges, where each node pulls the data from its in-neighbors. DASP [25] is the first attempt that leverages tensor core to enhance the compute part of SpMV, categorizing rows into long, medium, and short for tailored processing.

2.2 Nvidia Tensor Cores

The significant progress in deep learning leads to the development of accelerators for dense matrix multiplication, which is the central operation of various deep learning algorithms. Here, we focus on Nvidia tensor cores [20] for discussion and experimentation because of their popularity in contemporary works.

Nvidia GPUs provide two types of processing units: CUDA cores and tensor cores. CUDA cores handle general-purpose operations such as addition, division, and logic operations like branching. In contrast, tensor cores are specifically designed for Matrix Multiply-Accumulate (MMA) operations, expressed as $D_{M\times N} = A_{M\times K} \times B_{K\times N} + C_{M\times N}$. The computation precision is typically mixed, with inputs in 16-bit half floating-point format and outputs in 32-bit floating-point format. The mixed precision reduces memory usage and thus accelerates computation, without impacting the result's

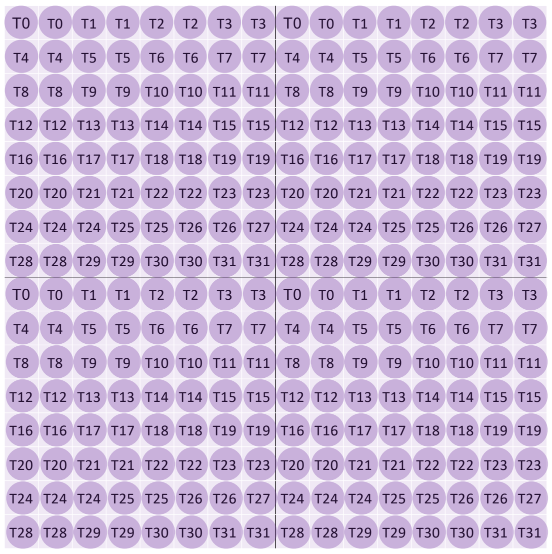

T0 T0 T1 T1 T2 T2 T3 T3 T0 T0 T1 T1 T2 T2 T3 T3
T4 T4 T5 T5 T6 T6 T7 T7 T4 T4 T5 T5 T6 T6 T7 T7
T8 T8 T9 T9 T10 T10 T11 T11 T8 T8 T9 T9 T10 T10 T11 T11
T12 T12 T13 T13 T14 T14 T15 T15 T12 T12 T13 T13 T14 T14 T15 T15
T16 T16 T17 T17 T18 T18 T19 T19 T16 T16 T17 T17 T18 T18 T19 T19
T20 T20 T21 T21 T22 T22 T23 T23 T20 T20 T21 T21 T22 T22 T23 T23
T24 T24 T24 T25 T25 T26 T26 T27 T24 T24 T24 T25 T25 T26 T26 T27
T28 T28 T29 T29 T30 T30 T31 T31 T28 T28 T29 T29 T30 T30 T31 T31
T0 T0 T1 T1 T2 T2 T3 T3 T0 T0 T1 T1 T2 T2 T3 T3
T4 T4 T5 T5 T6 T6 T7 T7 T4 T4 T5 T5 T6 T6 T7 T7
T8 T8 T9 T9 T10 T10 T11 T11 T8 T8 T9 T9 T10 T10 T11 T11
T12 T12 T13 T13 T14 T14 T15 T15 T12 T12 T13 T13 T14 T14 T15 T15
T16 T16 T17 T17 T18 T18 T19 T19 T16 T16 T17 T17 T18 T18 T19 T19
T20 T20 T21 T21 T22 T22 T23 T23 T20 T20 T21 T21 T22 T22 T23 T23
T24 T24 T24 T25 T25 T26 T26 T27 T24 T24 T24 T25 T25 T26 T26 T27
T28 T28 T29 T29 T30 T30 T31 T31 T28 T28 T29 T29 T30 T30 T31 T31

Figure 1: The thread layout of a 16×16 `fragment` **on the tensor core**

final accuracy. Further, the matrix dimensions <M, K, N> are fixed, such as <16, 16, 16>.

Nvidia GPUs organize threads into warps of 32 threads, such that threads within a warp are executed in a lockstep. These threads can be assigned to either 32 CUDA cores or the MMA units of the tensor cores. Figure 1 plots the layout of a 16×16 `fragment` (i.e., tensor core's buffer) held by a warp of threads. The fragment consists 4 repeated 8×8 portions. Within each portion, one thread controls two consecutive elements. When the warp executes the MMA operation, each thread handling 8 elements from the 4 portions.

To utilize the tensor cores effectively, developers can employ the Warp Matrix Multiply-Accumulate (WMMA) API, which offers high-level C++ functions for efficient coding. The MMA processing on tensor cores consists of three steps: data loading to the fragment (i.e., tensor core's buffer), tensor computation, and result storage back to memory, facilitated by `wmma::load`, `wmma::mma`, and `wmma::store`, respectively.

3 REVERSE ENGINEERING OF TENSOR CORES

This section outlines the process of reverse engineering the tensor cores, enabling higher efficiency and flexibility compared to the official documented APIs.

The `fragment` refers to the specific data buffer for the tensor core, which can be initialized using the `wmma::fragment` API. Conventionally, to load or store data with fragments, users must first allocate shared memory, transfer data from global memory to shared memory, align shared memory with the fragment, and then load data from shared memory to the fragment via `wmma::load`. The use of shared memory introduces an additional level of indirection, which can burden computations.

In addition to the standard method, an alternative approach to populate the tensor core is directly writing to the fragment's registers, e.g., `fragment.x[i] = value`. Each individual register holding data in the fragment can be accessed via `fragment.x[i]`. However, this approach is seldom utilized due to the absence of documentation regarding the internal layout of the registers, specifically the mapping between the registers and elements of the fragment, as well as their placement in the threads.

To establish the mapping relation, we assign values to the fragment register in each thread: `fragment.x[i]=i`, where `i` ranges from 0 to 15, and observe the resulting data layout. Surprisingly, the valid register indices of the fragment only range from 0 to 7, as depicted in Figure 2. Considering the thread layout in Figure 1, we can also divide the fragment into 4 portions. The top-left portion of 64 elements corresponds to `fragment.x[0,1]`, collectively handled by a warp of 32 threads. Each thread controls two consecutive elements, precisely mapped to its local registers `fragment.x[0,1]`. Similarly, by manipulating the `fragment.x[4,5]`, `fragment.x[2,3]` and `fragment.x[6,7]` inside every thread, we can directly modify the other three portions of the fragment.

The reverse engineering process applies to all of Nvidia's tensor cores. Discovering the mapping between register indices and thread IDs enhances understanding of the underlying hardware architecture. Furthermore, it facilitates efficient data movement for our solution, as detailed in the subsequent sections.

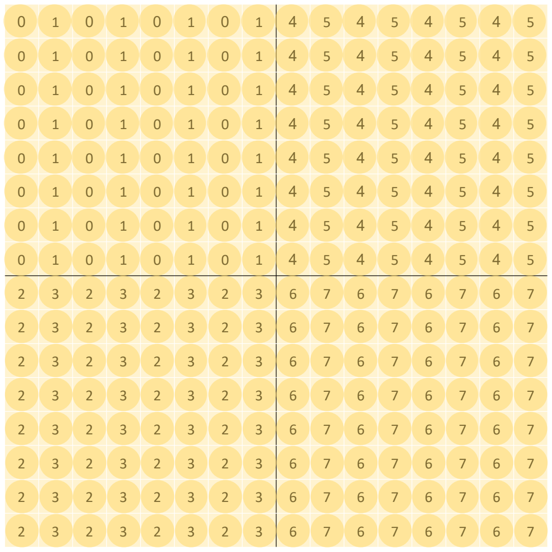


Figure 2: The data layout of a 16×16 fragment by assigning `fragment.x[i]=i` in each thread

4 DESIGN

In this section, we introduce our solution, *Spaden*, for optimizing SpMV with tensor cores. We firstly outline the abstraction of our design, and then details the two main optimizations of Spaden: a highly efficient matrix representation and the pairing SpMV execution kernel.

4.1 Abstraction of Spaden

The conceptual design of Spaden is depicted in Figure 3. Given the fixed data layout of tensor cores, Spaden initially converts the conventional sparse CSR format into a tensor-friendly format. For this purpose, blocked CSR (BSR) format is adopted, where each block represents a submatrix from the original matrix, aligning the block dimensions with the tensor core. To reduce the memory footprint of BSR, Spaden further compresses each block into a bitmap, resulting in an efficient sparse matrix representation termed *bitBSR*.

Subsequently, Spaden loads the bitBSR onto the tensor cores, decoding the bitmap to recover the original matrix elements. Transfer of data between the GPU's memory and the tensor cores is optimized by substituting data loading with calculation.

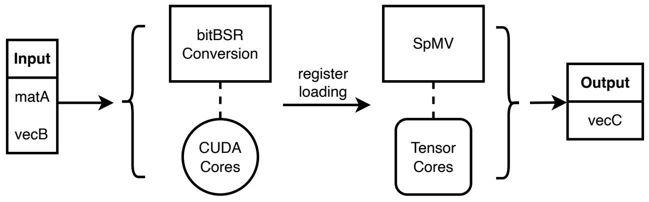
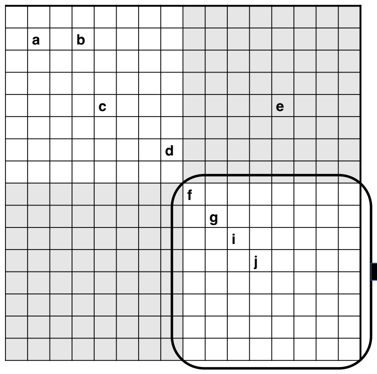


Figure 3: Conceptual design of Spaden



BitBSR
Row_Pointers = [0, 2, 3]
Column_Indices = [0, 1, 1]

Values = [f, g, i, j]
Offset = 5
Bitmap =
0x0000 0000 0804 0201 =>
Row 0: 00000001 = 0x01
Row 1: 00000010 = 0x02
Row 2: 00000100 = 0x04
Row 3: 00001000 = 0x08

Figure 4: BitBSR conversion of a sparse matrix

4.2 Sparse Matrix Representation

CSR provides efficient storage for sparse matrices by eliminating zero elements. However, it results in the loss of the rectangular structure of the matrix, which is crucial for tensor cores. To deploy the sparse matrix on a tensor core, Spaden divides the sparse matrix into multiple blocks with fixed dimensions that aligns with the tensor core's requirements. These non-empty blocks' positions are encoded in CSR; each block is formatted as a dense matrix, where every element, whether zero or nonzero, is recorded. In essence, Blocked CSR (BSR) represents a CSR with dense blocks of fixed size rather than individual scalar elements.

BSR enables the sparse matrix deployment on tensor cores but substantially increases memory usage due to storing zero elements. To address this, Spaden compresses the blocks using a bitmap (e.g., an unsigned integer) to avoid allocating extra memory for zeros. In the bitmap, each bit corresponds to a block element, with 1 indicating a nonzero and 0 indicating a zero. The floating point values of the nonzero elements and their counts per block are stored, whereas the zero element is expressed by a single 0 bit, considerably decreasing the memory space. This bitmap, with a length equal to the block size, encodes the rectangular shape of the matrix. Efficient decoding is conducted to recover block data during the execution of SpMV, as explained in later sections.

The choice of block size requires careful consideration of various factors. Firstly, it should be compatible with the fragment's size. Secondly, it relies on the available data types of integers (e.g., 32-bit int or 64-bit long) usable as a bitmap on GPUs. Thirdly, the block size affects the compression rate, as larger sizes will retain more zero bits within the blocks. Taking all factors into account, the block size is determined to be 8×8, which can be efficiently represented by a 64-bit unsigned long integer. With such block size, a single fragment of 16×16 on the tensor core can accommodate two blocks positioned diagonally. This design choice strikes a balance between programmability and performance.

The bitmap-based BSR is named bitBSR, which is visualized in Figure 4 for in-depth explanation. The original sparse matrix of size 24×24 is divided into 9 blocks of size 8×8. The positions of non-empty blocks are encoded as CSR. The elements of a block is mapped to a single bit in the 64-bit integer, where the least and most significant bits correspond to the top-left and bottom-right elements of the block, respectively.

In the highlighted block, row0 contains 8 elements (corresponding to 8 bits), but only the first element `f` is nonzero, so row0 is represented by `0x01`. The floating-point values of the nonzero elements (e.g., `f,g,i,j`) are stored consecutively in an array, along with the values of other blocks. The count of nonzero elements in each block is recorded and computed with exclusive scan to determine the `offset`. It enables the quick location of the starting index of each block in the value array.

BitBSR achieves compression through the bitmap, where 64 bits can encode the positions of up to 64 elements in a block. Assuming the element positions are conventionally represented as row and column indices (i.e., COO), both of which are typed as 32-bit integers. The number of elements in a non-empty block can vary from 1 to 64, thus occupying between 64 to 4096 bits. Consequently, the compression rate, calculated as `sizeof(COO)/sizeof(bitmap)`, achieved by the bitmap ranges from 1 to 64.

It is straightforward to infer that bitBSR is expected to be effective for high-degree matrices. As for low-degree matrices, the utilization of bitmaps may become burdensome, e.g., if one block only contains a single nonzero element. Additionally, the bitmap technique is widely used to represent sparse matrices [5, 6, 45]. This paper introduces an effective integration of this technique with tensor cores to enhance SpMV, as detailed in the following sections.

4.3 SpMV kernel

Spaden's SpMV kernel takes bitBSR and a dense vector **x** as input and produces a dense vector **y** as output. Each row of blocks (i.e., block-row) is computed with a segment of **x** with length 8, producing a segment of **y** with length 8. Since one tensor accommodates two blocks, two segments of **y** is generated simultaneously. In other words, during each iteration of SpMV, a tensor core generates 16 results. Despite not fully occupying the buffer space of the tensor core, Spaden is able to efficiently perform SpMV via directly accessing the tensor cores (detailed later), thus delivering high performance.

Figure 5 illustrates the placement of data on the tensor cores' `fragment` buffer, which has a size of 16×16. In `Frag A`, two blocks from the bitBSR are decoded, and then placed diagonally. In `Frag B`, two segments of vector **x** are initially loaded from global memory to the local register of threads, and then broadcast to the fragment. The vector is arranged vertically (in column-major order) in the tensor cores, as indicated by the shadowed vector in `Frag B`. The same vector is repeated eight times vertically to compose a block. Consequently, `Frag B` forms two blocks, combined with `Frag A` to execute MMA operations on tensor cores. Finally, the MMA outputs

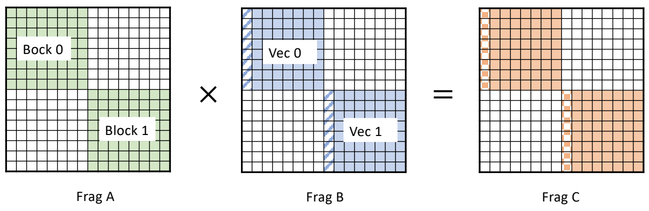


Figure 5: Data placement on tensor cores

a `Frag C` containing two valid blocks. The first columns (in shadow) of the resultant blocks are extracted, yielding the segments of vector `y`. In such sense, 16 rows from the original matrix are processed in parallel by every tensor core, producing 16 meaningful results, which is a double of DASP's throughput [25].

To perform the described SpMV process on tensor cores, three steps are executed, which are elaborated in the following sections:

(1) Decoding bitBSR to fetch data from the input matrix and vector.
(2) Loading blocks and vectors to tensor for computation.
(3) Extracting the resultant vector from the tensor.

4.3.1 bitBSR Decoding. To reconstruct the block data from the bitmap and accurately position it on the `fragment`, a decoding procedure is applied to bitBSR. A tensor core is scheduled collectively by a warp of 32 threads. For a block containing 64 elements, 2 consecutive elements are retrieved by every thread. More specifically, with bitBSR, two bits are accessed from the bitmap of a block. Only the floating-point data corresponding to the nonzero bits are fetched from global memory. The zero elements are calculated instead of loading from memory, thus effectively avoiding redundant data movement.

Algorithm 2: BitBSR Decoding

```
   Data: bitmap, A_values ← bitBSR; B_values ← input
         vector; lid ← lane id per warp; A_idx, B_idx
         ← block/segment index for matrix/vector
   Result: A_val1,A_val2, B_val1, B_val2
   // matrix decoding
1  lid_offset = uint64_t(lid) « 1;
2  bit1 = 1 « lid_offset;
3  bit2 = 2 « lid_offset;
4  bmp = bitmap[A_idx];
5  A_val1 = bmp & bit1 > 0? load(A_values, lid): 0;
6  A_val2 = bmp & bit2 > 0? load(A_values, lid+1): 0;
   // vector decoding
7  B_pos1 = (lid & 3) « 1;
8  B_pos2 = B_pos1 + 1;
   // BLOCK_DIM = 8
9  B_val1 = B_values[B_idx * BLOCK_DIM + B_pos1];
10 B_val2 = B_values[B_idx * BLOCK_DIM + B_pos2];
```

Algo. 2 provides the algorithmic description of the decoding procedure, including the corresponding segment of input vector. Firstly, the positions of two consecutive bits are calculated using bitwise operations (lines 1-3). The positions are used to extract the corresponding bits from the 64-bit bitmap `bmp` (line 4-6). Then, a ternary operation is incurred: if the extracted bit is zero, the result is assigned 0; otherwise, the value of the matrix element is loaded from the global memory. For the input vector, a warp of threads fetches the elements from the global memory in a repetitive pattern, where each thread accesses two consecutive positions with a spacing of 4.

4.3.2 Tensor Core Computing. The overall computation procedure for SpMV on tensor cores is similar to the CSR SpMV described in Algo.1, with the distinction that the processed data unit is a block rather than a scalar element.

Algorithm 3: Tensor Core Computing

```
   Data: rowptr, colidx, ← bitBSR of matrix;
         A_val1,A_val2, B_val1, B_val2
   Result: C_vals
   // initialize fragments
1  a_frag, b_frag, c_frag ← 0;
   // the left-top block
2  for A_idx = rowptr[i]; A_idx < rowptr[i]; A_idx++ do
3      B_idx = colidx[A_idx];
4      A_val1, A_val2 ← Matrix_Decoding(A_idx);
5      B_val1, B_val2 ← Vector_Fetching(B_idx);
6      a_frag.x[0] = A_val1, a_frag.x[1] = A_val2;
7      b_frag.x[0] = B_val1, b_frag.x[1] = B_val2;
8      wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);
9  c_vals ← extract(c_frag);
```

First, the tensor core fragments are initialized (line 1), followed by a loop that resembles CSR SpMV, but at the granularity of blocks. The indices `A_idx` and `B_idx` control the positions of the fetched block and vector (lines 2-3). The corresponding matrix and vector are decoded as previously described in Algorithm 2 (line 4-5). The decoded data are then written directly to the fragments' registers, with specific indices `[0]` and `[1]`, filling the left-top portion of the fragment (lines 6-7). In `a_frag`, 64 elements from the block are scattered; while in `b_frag`, 8 elements from the vector are distributed row-wise, with each column being identical.

It's important to note that filling the bottom-right portion of the fragment follows a similar process, albeit with different register indices `[6]` and `[7]`. The corresponding code is omitted due to limited paper space. These indices are determined through reverse engineering of the tensor core, as discussed in the previous Section 3. Finally, after filling the top-left and bottom-right portions of the fragment, the tensor core executes the MMA operation by calling the WMMA API (line 8).

4.3.3 Vector Extracting. The tensor core completes matrix multiplication and generates a matrix as a result, whereas the SpMV expects the vector as output. Hence, only a part of the matrix is needed. Algo. 4 outlines the extraction of a vector from the matrix. The left-most element of every row is retrieved from both the left-top (line 2) and right-bottom (line 3) portions of the resultant fragment, each contributing 8 elements. To extract the vector elements, 8 threads are utilized (line 4). With correctly calculated indexes (line 5,6), the extracted elements are copied to the output vector (line 7,8).

Advantages. Direct access to the tensor core via the fragment's register in Algo. 3 and Algo. 4 yields two-fold benefits to Spaden's performance. Firstly, as illustrated in Figure 5, Spaden occupies only the two diagonal portions of the fragment, totaling 128 in size. With the insights into the fragment's layout, Spaden can precisely control the small portions of the fragment, overcoming the limitation of the WMMA API. Secondly, the register-level controls of the fragment enables the direct writing of computed zeros (lines 5-6 in Algo. 2) to the tensor core. Only the valid values from the non-zero elements

Algorithm 4: Vector Extracting

```
Data: acc_frag
Result: C_values ← output vector
if lid % 4 == 0 then
    C_offset1 = row1 * BLOCK_DIM + lid / 4;
    C_offset2 = row2 * BLOCK_DIM + lid / 4;
    C_values[C_offset1] = acc_frag.x[0];
    C_values[C_offset2] = acc_frag.x[6];
```

are loaded. This eliminates redundant overhead, such as preparing a data buffer of size 256 in shared memory to fit into the fragment as in the conventional approach. Consequently, the memory efficiency is substantially improved.

5 EVALUATION

5.1 Experimental Setting

The performance evaluation is conducted on two Nvidia GPUs: L40, featuring 568 4th-generation tensor cores, and V100, equipped with 640 1st-generation tensor cores. The host and CUDA compilers are gcc 12.1 and nvcc 11.6. The code[1] follows the C++20 standard. The precision of the evaluated output is 32-bit floating point.

The performance of Spaden is compared with cuSPARSE's CSR and BSR SpMV routines [29], which are widely used as benchmarks. The optimal block size for cuSPARSE BSR varies significantly with different graphs. For simplicity and a fair comparison, the block size for both cuSPARSE BSR and Spaden is set to 8×8. Further, the performance comparison includes LightSpMV [24], Gunrock's SpMV kernel [40] and DASP [25], representing state-of-the-art work. Both Spaden and DASP leverage tensor cores for computation, whereas other methods utilize CUDA cores.

[1]https://github.com/yuang-chen/Spaden-ICPP24

Table 1: Matrix Dataset Information. ***nrow*** **and** ***nnz*** **mean the dimension of and the nonzero count in the original (square) matrices.** ***Bnrow*** **and** ***Bnnz*** **mean the dimension of and the block count in the matrix converted in bitBSR format. The two bottom matrices does NOT meet our selection criteria.**

Matrix	$nrow$	nnz	$Bnrow$	$Bnnz$
raefsky3	21,200	1,488,768	2,650	23,262
conf5	49,152	1,916,928	6,144	108,544
rma10	46,835	2,374,001	5,855	99,267
cant	62,451	4,007,383	7,807	180,069
pdb1HYS	36,417	4,344,765	4,553	140,833
consph	83,334	6,010,480	10,417	272,897
shipsec1	140,874	7,813,404	17,610	355,376
pwtk	217,918	11,634,424	27,240	357,758
Si41Ge41H72	185,639	15,011,265	23,205	1,557,151
TSOPF	38,120	16,171,169	4,765	294,897
Ga41As41H72	268,096	18,488,476	33,512	2,030,502
F1	343,791	26,837,113	42,974	2,253,370
scircuit	170,998	958,936	21,375	260,036
webbase1M	1,000,005	3,105,536	125,001	550,745

The performance of SpMV approaches heavily depends on the sparsity structures of matrices. Therefore, various matrix formats and corresponding SpMV kernels have been devised [14], including cuSPARSE's CSR and BSR. Similarly, the effectiveness of Spaden is influenced by the structures of matrices. When the matrix is excessively sparse (e.g., only one element per row), the nnz per block becomes too small to effectively utilize the tensor cores. We suggest considering our approach for matrices with $nrow > 10,000$ and $nnz/nrow > 32$, or when cuSPARSE BSR is preferred over cuSPARSE CSR.

In our experiments, we showcase 12 matrices from SuiteSparse [12] with $nnz/nrow > 32$. The selected matrices are commonly evaluated in contemporary SpMV works [4, 28]. Additionally, two random matrices with $nnz/nrow < 6$ are used for comparison. Their information before and after conversion is detailed in Table 1.

5.2 Performance

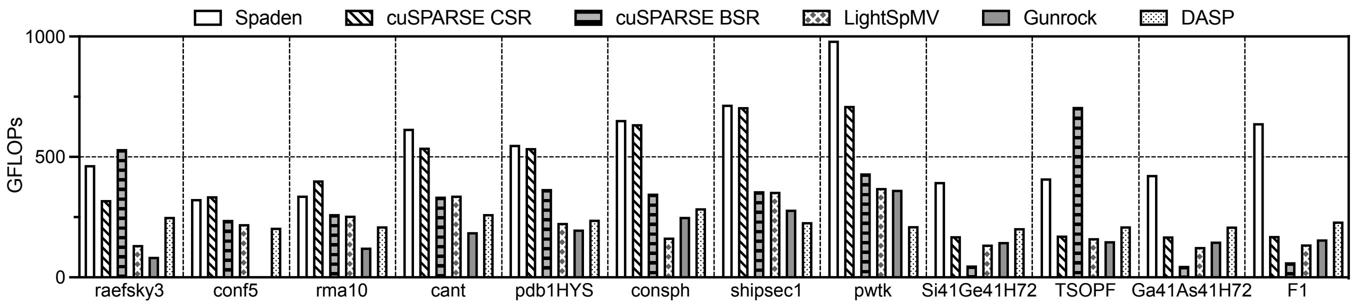
Figure 6 plots the performance comparison among Spaden and other SpMV methods on two Nvidia GPUs, L40 and V100. We firstly focus on the 12 matrices that meet with our selection criteria. On the L40, Spaden exhibits significant speedups over cuSPARSE's CSR and BSR SpMVs, and LightSpMV, Gunrock, and DASP, achieving 1.63×, 3.37×, 2.68×, 2.82×, and 2.32×, respectively. On the V100, Spaden still outperforms the competing methods with speedups of 1.30×, 2.21×, 1.86×, 2.58×, 1.20×.

cuSPARSE's CSR SpMV ranks as the second fastest SpMV method on average. cuSPARSE BSR SpMV demonstrates optimal performance on raefsky3 and TSOPF, two matrices predominantly composed of dense blocks (further discussed in Section 5.4). Other tested matrices lack sufficient nonzeros to fill the blocks of the BSR format. In principle, CSR SpMV is suitable for sparser matrices, while BSR SpMV is more suitable for denser structures.

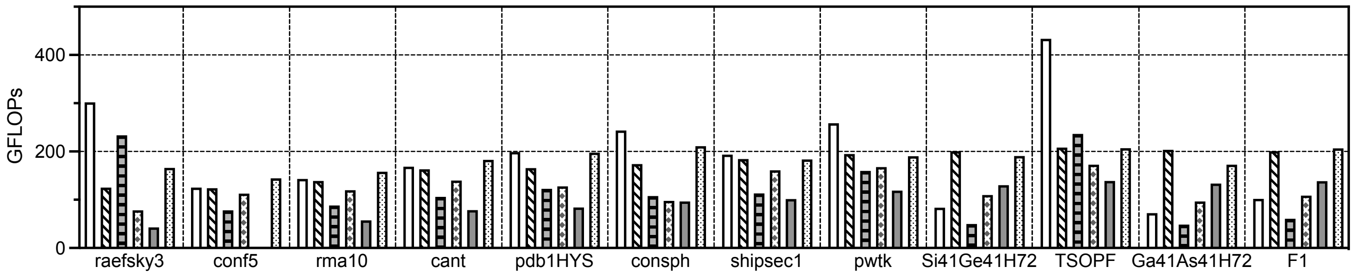
LightSpMV, an attempt at optimizing CSR SpMV on GPUs is surpassed by the modern version of cuSPARSE CSR from CUDA toolkits v11.6. Gunrock provides versatile APIs for expressing graph algorithms. However, its SpMV implementation, which translates SpMV as message passing along the edges of the graph, is less performant than specific sparse matrix libraries.

DASP originally does not support 32-bit floating point output. We modified DASP to support 32-bit floating points on L40 but encountered compatibility issues on V100. On L40, it exhibited suboptimal performance with 32-bit precision. On V100, DASP's 16-bit precision achieved lightly better performance to cuSPARSE's 32-bit precision. DASP achieves higher performance on V100 than L40 because it operates with lower precision. Additionally, its key tensor core operation `mma.sync.m8n8k4` is optimized for the architecture of V100, which may suffer from substantially reduced performance on other architectures according to the official CUDA document [33].

When running on the low-degree matrices scircuit and webbase-1M, all SpMV algorithms exhibit remarkably low throughput. This is because these matrices are too sparse ($nnz/nrow < 6$) to effectively utilize the cache hierarchy and massive parallelism of the GPU. Especially, for Spaden, the majority of fragments are written with zeros, significantly wasting computing resources of tensor cores.

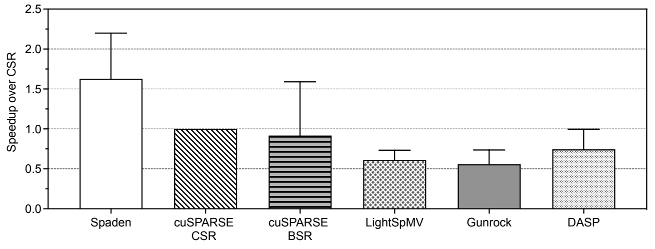


(a) Performance on Nvidia L40

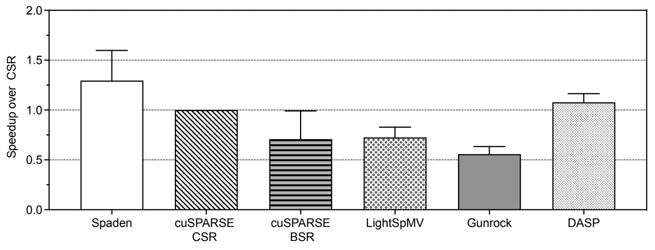


(b) Performance on Nvidia V100

Figure 6: Performance of different SpMV methods



(a) Speedups on Nvidia L40



(b) Speedups on Nvidia V100

Figure 7: Speedup of different SpMVs on over cuSPARSE CSR

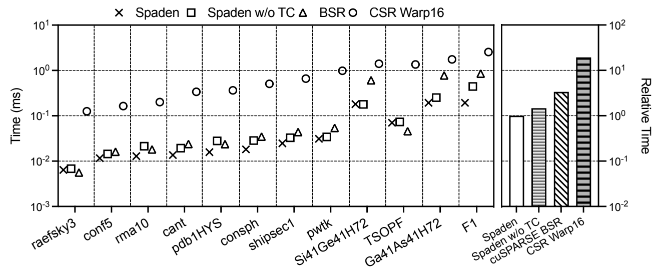


Figure 8: Speedup breakdown of Spaden on Nvidia L40

Consequently, it achieves only 41% of the through of cuSPARSE CSR. In following sections, unless otherwise stated, we do not include the two matrices for experimentation, since they fall outside of Spaden's effective scope.

5.3 Speedup Breakdown

The speedup of Spaden results from two notable factors: (1) The efficiency of bitBSR and (2) the computing capacity of tensor cores. To understand the contributions of these factors, we further implemented two variants of Spaden. One operates bitBSR on CUDA cores instead of tensor cores, denoted as Spaden w/o TC. The other, denoted as CSR Warp16, utilizes CSR on CUDA cores with 16 rows processed per warp, identical to the original Spaden. For better clarity, we include cuSPARSE BSR in the discussion once again. Due to limited space, we solely discuss the performances on L40.

As depicted in Figure 8, Spaden delivers speedups over Spaden w/o TC, cuSPARSE BSR and CSR Warp16 by 1.47×, 3,37×, 23.18×, respectively. CSR Warp16 exhibits the poorest performance due to uncoalesced memory accesses by the threads within a warp. In this method, each thread processes two rows independently, resulting in neighboring threads loading non-consecutive elements from global memory, thus disrupting the coalesced memory access pattern. On the other hand, cuSPARSE BSR and Spaden (and its variant w/o TC) process data block by block, enabling consecutive access within each block and thus facilitating coalesced accesses for a warp of threads. The performance contrast between uncoalesced and coalesced memory accesses highlights the significant impact of memory access pattern on GPU's performance.

However, the abundance of zero elements in the BSR format leads to redundant data movement, thereby limiting the performance of cuSPARSE BSR. Spaden w/o TC enhances BSR's efficiency through the use of bitBSR. Instead of storing a zero with 32-bit float, bitBSR efficiently store it as a single bit, which is decoded in the kernel via bitwise calculation. This optimization significantly accelerates SpMV execution, leading to a speedup over cuSPARSE BSR by 2.29×. Spaden further boosts the performance with a speedup of 1.47× over Spaden w/o TC by shifting computation to tensor cores. In essence, Spaden achieves high performance with advantage in coalesced memory access pattern, elimination of redundant zero loading, and the computing capacity of tensor cores.

5.4 Impact of Matrix Structure

This section explores the correlation between Spaden and matrix structure. After converting the matrix into bitBSR format, the blocks of size 8×8 are categorized into three groups based on *nnz* in each block. The *sparse* blocks contain $nnz \leq 32$, the *medium* blocks contain *nnz* in the range of 33 to 48, and the *dense* blocks have $nnz > 48$.

Figure 9a illustrates the ratio of the three types of blocks for different matrix datasets. Matrices raefsky3 and TSOPF are primarily composed of dense blocks, pwtk exhibit an even distribution across the three types, while the remaining matrices mainly consist of sparse blocks.

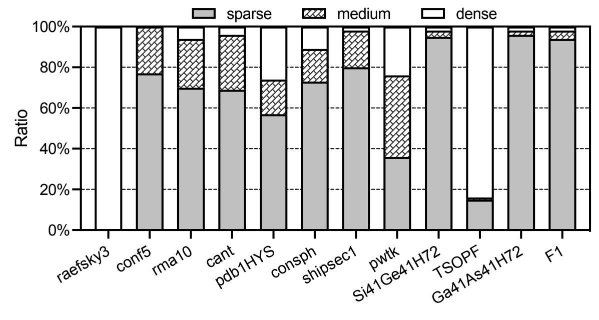
To profile the relationship between the performance of Spaden and the sparsity of the matrices, we sort the matrices by the ratio of sparse blocks and correspond them with Spaden's speedups over cuSPARSE BSR. Figure 9b illustrates the correlation between the two metrics.

It is observed that the more sparse blocks in a matrix, the faster the Spaden compared to cuSPARSE BSR. For matrices like raefsky3 and TSOPF, characterized by a high ratio of dense blocks, BSR outperforms Spaden, achieving speedups of 1.2× and 1.5×, respectively. In contrast, matrices with a high ratio of sparse blocks (e.g., Si41Ge41H72 and Ga41As41H72) demonstrate significantly better performance with Spaden. On these matrices, Spaden achieves speedups of 4.0× and 4.2× over BSR, respectively. The increasing speedup with the higher ratio of sparse blocks demonstrates the effectiveness of bitBSR in addressing the sparsity issue encountered in existing BSR methods.

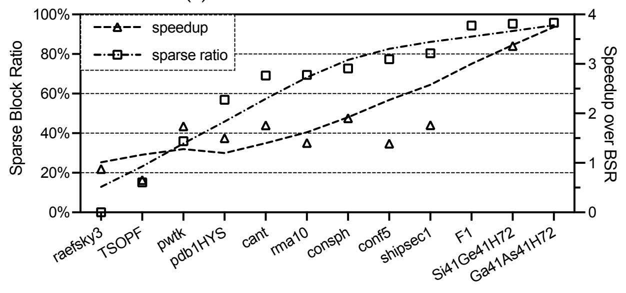
5.5 Conversion Overhead

The time and memory costs of converting CSR to bitBSR are examined in comparison with cuSPARSE's BSR and DASP. Both the absolute results and the normalized results (divided by the *nnz* of matrices) are offered. These results indicate whether the conversion cost is more related to the matrix dimensions or to the number of nonzero elements within the matrix. It's worth noting that although cuSPARSE CSR does not involve format conversion, it requires preprocessing for partitioning and buffer allocation, the cost of which is also shown in Figure 10 for reference.

cuSPARSE BSR exhibits the lowest preprocessing time but the highest memory usage. On average, it requires 1.21 ns and 13.63 Bytes per *nnz*. Also, as shown in the lower part of Figure 10b, BSR's normalized memory footprint varies significantly across different datasets, while other methods maintain a nearly constant memory

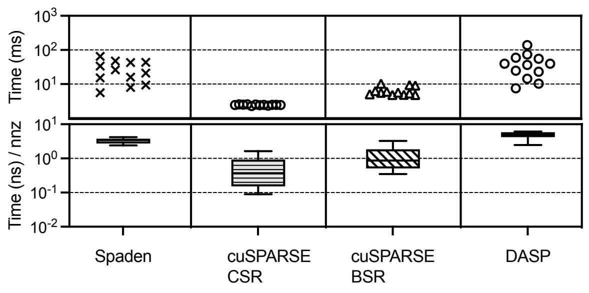


(a) Block distribution ratio

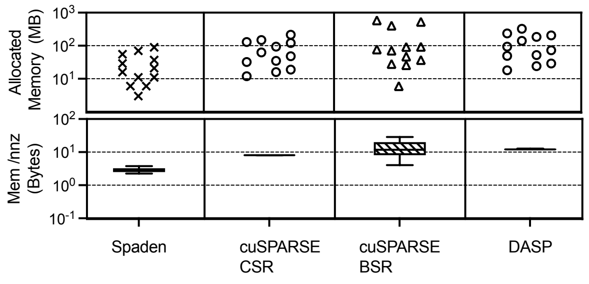


(b) Speedup with varied sparse block ratios

Figure 9: Time and memory costs of different methods on Nvidia L40



(a) Preprocessing time



(b) Memory space

Figure 10: Time and memory costs of different methods on Nvidia L40

footprint per *nnz*. cuSPARSE CSR also demonstrates high memory usage at 8.06 Bytes per *nnz* due to buffer allocation, but it maintains a relatively constant and low preprocessing time across different datasets, as illustrated in the upper part of Figure 10a.

The time and memory costs of Spaden and DASP are proportional to *nnz*. Spaden's preprocessing time is at 3.31 ns per *nnz*, yet it incurs the lowest memory consumption of 2.85 Bytes per *nnz*. DASP yields the highest preprocessing time of 4.95 ns per *nnz*, alongside a substantial memory footprint of 12.25 Bytes per *nnz*.

Compared to cuSPARSE CSR, BSR, and DASP, Spaden achieves preprocessing speedups[2] of 0.17×, 0.36×, and 1.5×, respectively. Meanwhile, Spaden provides memory savings of 2.83×, 4.70×, and 4.32×. Given that the conversion is performed only once and significantly benefits memory-constrained GPUs, we consider it an acceptable cost.

6 RELATED WORK

Accelerating SpMV. Significant efforts have been made to accelerate SpMV with various strategies such as vectorization [38, 42], reordering [1, 2], blocking [4, 18], and reformatting [14]. These strategies are oftentimes combined to achieve optimal performance.

Vectorization refers to optimizing SpMV with vector instructions (e.g., Intel AVX512) on modern CPUs. VHCC [38], for instance, transforms the CSR to a 2D jagged partition to fit with the width of vector lanes. CSR5 [22] improves the VHCC, converting the CSR into a compact, sparsity-insensitive 2D tiles. CVR [42] further enhances the vectorized SpMV with locality improvement.

Blocking SpMV enhances cache efficiency by segmenting matrices into submatrices sized to cache capacities [18, 30]. Building upon it, binning SpMV regularizes data access within blocks for sequential data access pattern [3, 4]. LAV [44] integrates cache blocking with vectorization, organizing matrix portions into cache-sized blocks for vectorization.

Reordering algorithms re-number the rows and columns of a sparse matrix (of a graph) to reduce cache misses and enhance parallelism. Gorder [41] investigates into the internal structure of graphs. It analyzes the connectivity of vertices, such that consecutive IDs are relabeled to vertices that share common neighbors. Rabbit [1] explores the hierarchic clustering stucture in parallel, to enhance both runtime efficiency and analytical outcomes. Since graph structure analysis can be time-consuming, simpler reordering algorithms based solely on vertex degree have been developed [2, 13].

Graphs and Matrices. Exploiting the duality between graphs and matrices [19, 37], graph algorithms can be reformulated into linear algebraic forms. This paradigm is demonstrated by several important systems. GraphBLAS [11] translates graph operations into a set of matrix and vector kernels. It outperforms conventional graph framework with the highly optimized mathematical routines. LAGraph [27] is a library that builds on top of GraphBLAS to offer a comprehensive suite of graph algorithms formulated as linear algebra operations. It standardizes graph algorithm implementations, based on the foundation of GraphBLAS. GraphBLAST [43] leverages the massive parallelism of GPUs to speed up linear algebraic operations central to graph processing. However, GraphBLAST was developed for compatibility with CUDA 9.0 and has since not received further support; therefore, it has not been included in the testing benchmarks of this paper.

Further, deep learning frameworks, such as DGL [39], enable Graph Neural Networks (GNNs) by leveraging sparse matrix operations for graph-structured data. DGL efficiently abstracts node aggration and message passing on the graphs into sparse matrix operations. The abstraction allows direct application of neural network models on graphs, facilitating the implementation and acceleration of GNNs.

Applications on tensor cores. Tensor cores are primarily used for accelerating GEMM in the field of deep learning. Recent research efforts have extended the utility of tensor cores to handle more irregular workloads in deep learning, such as GNN [7, 21]. These efforts have focused on two specific operations: SpMM and SDDMM, where only one of the inputs is sparse, while the others are dense matrices. Meanwhile, studies have explored the deployment of SpGEMM [45], taking two sparse matrices as inputs, and SpMV [25], where the input is sparse matrix and vector, on tensor cores to accelerate a wider range of mathematical applications. The presence of dense matrix in SpMM and SDDMM simplifies the adaptation of tensor cores compared to SpGEMM and SpMV.

Also, there are attempts to leverage tensor cores for addressing various scientific problems, including stencil computation [23], and fourier transforms [35] and molecular dynamics [15], quantum annealing [8], epistasis detection [31], and compact fractals [36]. Additionally, the extension library is developed to enhance the programmability of tensor cores [34].

7 CONCLUSION

In this paper, we introduce *Spaden*, a method leverages tensor cores to accelerate SpMV. To efficiently adapt irregular sparse workloads to tensor cores, we first reveal the mapping between the fragment's registers and threads through reverse engineering. These insights enable precise operations on a portion of the fragments with direct accesses. Then, an efficient bitmap-based format is devised, named *bitBSR*, which significantly reduces the memory storage of sparse matrices. This format is paired with a performant SpMV kernel that manipulates the tensor core via register-level control. We evaluate Spaden on Nvidia L40 GPU and V100, demonstrating its effectiveness in speedup and memory saving compared to state-of-the-art approaches. Also, the effective scope of Spaden is explained, and its performance factors are analyzed.

In future work, we plan to extend the bitmap-based blocking technique to support additional sparse matrix formats, such as COO. Also, we aim to explore the adaptation of bitBSR for other sparse operations on dense matrix units, including SpMM and SDDMM. Furthermore, a sparse math library centered around the bitmap & blocking can be developed, incorporating support for various formats and operations.

ACKNOWLEDGMENTS

This work is supported by the National Key Research and Development Program of China (No. 2018YFB1003505).

[2]A value less than 1 indicates a slowdown.

REFERENCES

[1] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. 2016. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 22–31.

[2] V. Balaji and B. Lucia. 2018. When is Graph Reordering an Optimization? Studying the Effect of Lightweight Graph Reordering Across Applications and Input Graphs. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 203–214.

[3] Scott Beamer, Krste Asanović, and David Patterson. 2017. Reducing pagerank communication via propagation blocking. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 820–831.

[4] Daniele Buono, Fabrizio Petrini, Fabio Checconi, Xing Liu, Xinyu Que, Chris Long, and Tai-Ching Tuan. 2016. Optimizing sparse matrix-vector multiplication for large-scale data analytics. In *Proceedings of the 2016 International Conference on Supercomputing*. 1–12.

[5] Jou-An Chen, Hsin-Hsuan Sung, Xipeng Shen, Sutanay Choudhury, and Ang Li. 2023. BitGNN: Unleashing the Performance Potential of Binary Graph Neural Networks on GPUs. In *Proceedings of the 37th International Conference on Supercomputing*. 264–276.

[6] Jou-An Chen, Hsin-Hsuan Sung, Xipeng Shen, Nathan Tallent, Kevin Barker, and Ang Li. 2022. Bit-graphblas: Bit-level optimizations of matrix-centric graph processing on gpu. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 515–525.

[7] Zhaodong Chen, Zheng Qu, Liu Liu, Yufei Ding, and Yuan Xie. 2021. Efficient tensor core-based gpu kernels for structured sparsity under reduced precision. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[8] Yi-Hua Chung, Cheng-Jhih Shih, and Shih-Hao Hung. 2022. Accelerating simulated quantum annealing with GPU and tensor cores. In *International Conference on High Performance Computing*. Springer, 174–191.

[9] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. 2019. Accelerating reduction and scan using tensor core units. In *Proceedings of the ACM International Conference on Supercomputing*. 46–57.

[10] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. 2014. Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. http://cusplibrary.github.io/. Accessed: 2024-01-05.

[11] Timothy A Davis. 2018. SuiteSparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software (TOMS)* (2018).

[12] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[13] Priyank Faldu, Jeff Diamond, and Boris Grot. 2019. A closer look at lightweight graph reordering. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 1–13.

[14] Salvatore Filippone, Valeria Cardellini, Davide Barbieri, and Alessandro Fanfarillo. 2017. Sparse matrix-vector multiplication on GPGPUs. *ACM Transactions on Mathematical Software (TOMS)* 43, 4 (2017), 1–49.

[15] Joshua Finkelstein, Justin S Smith, Susan M Mniszewski, Kipton Barros, Christian FA Negre, Emanuel H Rubensson, and Anders MN Niklasson. 2021. Quantum-based molecular dynamics simulations using tensor cores. *Journal of Chemical Theory and Computation* 17, 10 (2021), 6180–6192.

[16] Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis, and Nectarios Koziris. 2009. Performance evaluation of the sparse matrix-vector multiplication on modern architectures. *The Journal of Supercomputing* 50 (2009), 36–77.

[17] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J Higham. 2018. Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 603–613.

[18] Eun-Jin Im. 2000. *Optimizing the performance of sparse matrix-vector multiplication*. University of California, Berkeley.

[19] Jeremy Kepner and John Gilbert. 2011. *Graph algorithms in the language of linear algebra*. SIAM.

[20] Ronny Krashinsky, Olivier Giroux, Stephen Jones, Nick Stam, and Sridhar Ramaswamy. 2020. NVIDIA Ampere Architecture In-Depth. https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth. Accessed: 2024-01-15.

[21] Shigang Li, Kazuki Osawa, and Torsten Hoefler. 2022. Efficient quantized sparse matrix operations on tensor cores. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[22] Weifeng Liu and Brian Vinter. 2015. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 339–350.

[23] Xiaoyan Liu, Yi Liu, Hailong Yang, Jianjin Liao, Mingzhen Li, Zhongzhi Luan, and Depei Qian. 2022. Toward accelerated stencil computation by adapting tensor core unit on gpu. In *Proceedings of the 36th ACM International Conference on Supercomputing*. 1–12.

[24] Yongchao Liu and Bertil Schmidt. 2015. LightSpMV: Faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 82–89.

[25] Yuechen Lu and Weifeng Liu. 2023. DASP: Specific Dense Matrix Multiply-Accumulate Units Accelerated General Sparse Matrix-Vector Multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[26] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. 2018. Nvidia tensor core programmability, performance & precision. In *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 522–531.

[27] Tim Mattson, Timothy A Davis, Manoj Kumar, Aydin Buluc, Scott McMillan, José Moreira, and Carl Yang. 2019. LAGraph: A community effort to collect graph algorithms built on top of the GraphBLAS. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 276–284.

[28] Naveen Namashivavam, Sanyam Mehta, and Pen-Chung Yew. 2021. Variable-sized blocks for locality-aware SpMV. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 211–221.

[29] Maxim Naumov, L Chien, Philippe Vandermersch, and Ujval Kapasi. 2010. Cusparse library. In *GPU Technology Conference*.

[30] Rajesh Nishtala, Richard W Vuduc, James W Demmel, and Katherine A Yelick. 2007. When cache blocking of sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication and Computing* 18, 3 (2007), 297–311.

[31] Ricardo Nobre, Aleksandar Ilic, Sergio Santander-Jiménez, and Leonel Sousa. 2020. Exploring the binary precision capabilities of tensor cores for epistasis detection. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 338–347.

[32] Eriko Nurvitadhi, Asit Mishra, and Debbie Marr. 2015. A sparse matrix vector multiply accelerator for support vector machine. In *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. IEEE, 109–116.

[33] NVIDIA. 2020. CUDA Parallel Thread Execution ISA Version 8.4. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html. Accessed: 2024-01-15.

[34] Hiroyuki Ootomo and Rio Yokota. 2023. Reducing shared memory footprint to leverage high throughput on Tensor Cores and its flexible API extension library. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. 1–8.

[35] Louis Pisha and Łukasz Ligowski. 2021. Accelerating non-power-of-2 size Fourier transforms with GPU tensor cores. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 507–516.

[36] Felipe A Quezada, Cristóbal A Navarro, Nancy Hitschfeld, and Benjamin Bustos. 2022. Squeeze: Efficient compact fractals for tensor core gpus. *Future Generation Computer Systems* 135 (2022), 10–19.

[37] Narayanan Sundaram, Nadathur Rajagopalan Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. Graphmat: High performance graph analytics made productive. *arXiv preprint arXiv:1503.07241* (2015).

[38] Wai Teng Tang, Ruizhe Zhao, Mian Lu, Yun Liang, Huynh Phung Huyng, Xibai Li, and Rick Siow Mong Goh. 2015. Optimizing and auto-tuning scale-free sparse matrix-vector multiplication on Intel Xeon Phi. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 136–145.

[39] Minjie Yu Wang. 2019. Deep graph library: Towards efficient and scalable deep learning on graphs. In *ICLR workshop on representation learning on graphs and manifolds*.

[40] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*. 1–12.

[41] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*. 1813–1828.

[42] Biwei Xie, Jianfeng Zhan, Xu Liu, Wanling Gao, Zhen Jia, Xiwen He, and Lixin Zhang. 2018. Cvr: Efficient vectorization of spmv on x86 processors. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 149–162.

[43] Carl Yang, Aydın Buluç, and John D Owens. 2022. GraphBLAST: A high-performance linear algebra-based graph framework on the GPU. *ACM Transactions on Mathematical Software (TOMS)* 48, 1 (2022), 1–51.

[44] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. 2020. Speeding up spmv for power-law graph analytics by enhancing locality & vectorization. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[45] Orestis Zachariadis, Nitin Satpute, Juan Gómez-Luna, and Joaquín Olivares. 2020. Accelerating sparse matrix–matrix multiplication with GPU Tensor Cores. *Computers & Electrical Engineering* 88 (2020), 106848.