DASP: Specific Dense Matrix Multiply-Accumulate Units Accelerated General Sparse Matrix-Vector Multiplication

Yuechen Lu
yuechenlu@student.cup.edu.cn
Super Scientific Software Laboratory,
China University of Petroleum-Beijing
Beijing, China

Weifeng Liu
weifeng.liu@cup.edu.cn
Super Scientific Software Laboratory,
China University of Petroleum-Beijing
Beijing, China

ABSTRACT

Sparse matrix-vector multiplication (SpMV) plays a key role in computational science and engineering, graph processing, and machine learning applications. Much work on SpMV was devoted to resolving problems such as random access to the vector x and unbalanced load. However, we have experimentally found that the computation of inner products still occupies much overhead in the SpMV operation, which has been largely ignored in existing work.

In this paper, we propose DASP, a new algorithm using specific dense MMA units for accelerating the compute part of general SpMV. We analyze the row-wise distribution of nonzeros and group the rows into three categories containing long, medium, and short rows, respectively. We then organize them into small blocks of proper sizes to meet the requirement of MMA computation. For the three categories, DASP offers different strategies to complete SpMV by efficiently utilizing the MMA units.

The experimental results on two newest NVIDIA GPUs A100 and H800 show that our DASP in FP64 precision outperforms five latest SpMV methods CSR5, TileSpMV, LSRB-CSR, cuSPARSE BSR format and cuSPARSE CSR format by a factor of on average 1.46x, 2.09x, 3.29x, 2.08x and 1.52x (up to 12.64x, 17.48x, 90.59x, 283.92x and 6.94x) on A100, respectively. As for SpMV in FP16 precision, our DASP outperforms cuSPARSE by a factor of on average 1.70x and 1.75x (up to 26.47x and 65.94x) on A100 and H800, respectively.

CCS CONCEPTS

• **Mathematics of computing** → **Mathematical software performance**; • **Computing methodologies** → **Shared memory algorithms**; **Vector / streaming algorithms**.

KEYWORDS

GPU, tensor core, matrix multiply-accumulate, sparse matrix-vector multiplication

SC '23, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0109-2/23/11...$15.00
https://doi.org/10.1145/3581784.3607051

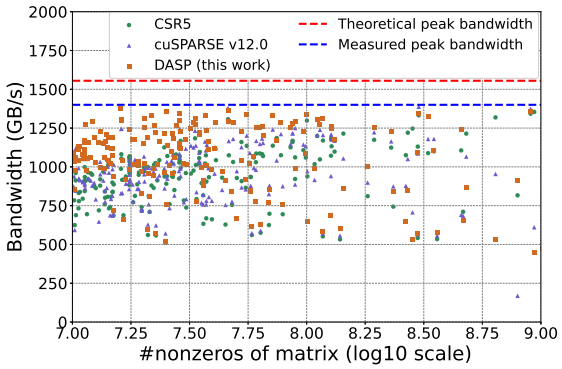


Figure 1: Bandwidth throughput of three double-precision SpMV approaches, CSR5, cuSPARSE and DASP (this work), running the largest 202 matrices with no less than 10^7 nonzeros in the SuiteSparse Matrix Collection on an NVIDIA A100 GPU. The red and blue dash lines are theoretical and measured (STREAM-like Triad) peak bandwidths, respectively.

ACM Reference Format:
Yuechen Lu and Weifeng Liu. 2023. DASP: Specific Dense Matrix Multiply-Accumulate Units Accelerated General Sparse Matrix-Vector Multiplication. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3581784.3607051

1 INTRODUCTION

Sparse matrix operations are one of the most fundamental cornerstones in computational science and engineering. Since the positions of the nonzeros of a sparse matrix can be very irregular, compared to dense matrix computations, there are more problems, such as poor memory locality and unbalanced load, to resolve [23, 33]. Sparse matrix-vector multiplication (SpMV) may be the most studied kernel among sparse matrix operations. Much research has focused on improving its memory access through vertical-slicing [37, 51] and 2D-tiling [76, 98] the nonzeros in a sparse matrix, and on balancing workload by reconstructing a nearly even-sized basic working unit [39, 64, 73].

However, even though such existing work already demonstrated promising improvements, the performance achieved is still unsatisfactory. Figure 1 shows the bandwidth throughput (GB/s) of three double-precision SpMV approaches, i.e., CSR5 [64] (probably the

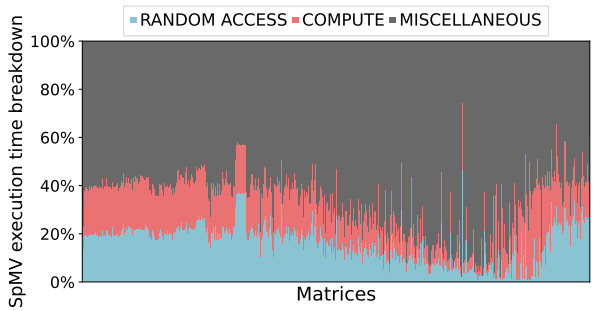


Figure 2: Execution time breakdown of running a standard CSR SpMV algorithm on an NVIDIA A100 GPU.

most tested open-source baseline in existing SpMV work), cuSPARSE v12.0 (the newest vendor-supported library), and DASP (our algorithm proposed in this paper) running on an NVIDIA A100 40GB PCIe GPU by using the largest 202 sparse matrices (with no less than 10 million nonzeros) in the SuiteSparse Matrix Collection [24]. As can be seen, neither CSR5 nor cuSPARSE can bring the bandwidth achieved to near peak bandwidth (measured with a STREAM-like Triad test [72]). This indicates that SpMV algorithms may still have room to improve.

To understand the performance behavior of SpMV, we further breakdown the costs of the standard CSR SpMV (written with CUDA and run on the NVIDIA A100) into three parts: (1) `RANDOM ACCESS` on the vector x, (2) `COMPUTE` of inner products of nonzeros in A and the corresponding components of x loaded, and (3) `MISCELLANEOUS` only including reading and writing arrays such as $row\_pointer$ and y, and plot the results of all the 2893 matrices from the SuiteSparse Matrix Collection [24] in Figure 2. It can be seen that, although SpMV is a well-known memory-bound kernel, its `COMPUTE` part still occupies much overhead. However, to the best of our knowledge, existing work largely ignored optimizing this part and leads to the suboptimal performance (recall the gap between the bandwidth achieved and the peak shown in Figures 1).

Fortunately, recent parallel processors, in particular GPUs, include dedicated matrix multiply-accumulate (MMA) units to significantly accelerate small dense general matrix multiplication (GEMM). NVIDIA Tensor Cores, AMD Matrix Cores, Apple Matrix Co-processors (AMX), as well as Intel Xe Matrix Extensions (XMX) and Advanced Matrix Extensions (AMX) are representatives of such MMA units. In addition to GEMM, in recent years, the performance of a number of other fundamental algorithms, such as reduction and scan [22], stencil [66], FFT [28, 56, 80], deep neural networks [55, 94], molecular dynamics [34], block iterative solvers [11, 40], sparse matrix-dense matrix multiplication (SpMM) [60, 87], has also been improved through the use of MMA units.

However, exploiting the computational power of the MMA units for SpMV is not trivial. The main reason is that, on one side, the distribution of nonzeros of the matrix in SpMV can be very irregular, but on the other, the MMA units need strict regular data layout to fully utilize the hardware. To address this problem, we in this paper propose DASP, a new algorithm using specific dense MMA units for accelerating general SpMV. We first analyze the row-wise distribution of nonzeros and group the rows into three categories to contain long, medium, and short rows, respectively. Then we further divide each long row into chunks of proper sizes to meet the requirement of the MMA computation, and aggregate short rows together to form the regular layout for MMA. For rows of medium sizes, we further distinguish them into regular and irregular parts and use different computational units accordingly. On top of the MMA-friendly data structure, different CUDA kernels utilizing the MMA units are developed for the three groups of rows.

We evaluate DASP by using all the 2893 matrices from the SuiteSparse Matrix Collection [24] on two latest NVIDIA GPUs A100 (Ampere) and H800 (Hopper) with hardware-supported FP64 and FP16 MMA units. Figure 1 demonstrates that by utilizing the MMA units, our DASP algorithm produces a much higher overall performance and brings the bandwidths of more matrices closer to the measured peak. More complete experimental results listed in Section 4 show that compared to five latest SpMV methods in FP64 precision, our DASP is faster than CSR5 [64] on 2403 matrices, faster than TileSpMV [76] on 2579 matrices, faster than LSRB-CSR [63] on 2251 matrices, faster than cuSPARSE v12.0 BSR format SpMV on 2340 matrices, and faster than cuSPARSE v12.0 CSR format SpMV on 2344 matrices, and achieves on average (geometric mean) 1.46x, 2.09x, 3.29x, 2.08x and 1.52x (up to 12.64x, 17.48x, 90.59x, 283.92x and 6.94x) speedups over them on A100, respectively. As for SpMV in FP16 precision, our DASP is faster than cuSPARSE CSR format SpMV on 2578 and 2576 matrices, and achieves on average 1.70x and 1.75x (up to 26.47x and 65.95x) speedups over cuSPARSE on A100 and H800, respectively.

This work makes the following contributions:

- We identify that the compute part could be a critical performance bottleneck of SpMV, and MMA as a novel hardware unit could bring higher performance;
- We propose the DASP algorithm that makes the irregular data layout in sparse matrices regular for efficiently exploiting the MMA units;
- We demonstrate that our approach brings most matrices higher performance than state-of-the-art SpMV work on the latest NVIDIA Ampere and Hopper GPUs.

2 BACKGROUND

2.1 SpMV and its Performance Analysis

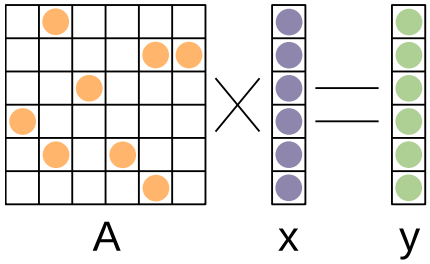


Figure 3: An example of SpMV that multiplies a 6-by-6 sparse matrix A with a vector x and gets a vector y.

Algorithm 1 A pseudocode of parallel CSR SpMV.

```
1: for i = 0 to m in parallel do
2:     sum ← 0
3:     for j = RowPtr[i] to RowPtr[i+1] do
4:         valx ← x[ColIdx[j]]     // RANDOM ACCESS x
5:         sum ← sum + valx × Val[j]     // COMPUTE
6:     end for
7:     y[i] ← sum
8: end for
```

The SpMV operation multiplies a sparse matrix A with a dense vector x to obtain a dense vector y as the result. Figure 3 shows a simple example of SpMV. The Compressed Sparse Row (CSR) format is currently the most commonly used storage format for the sparse matrix A in this operation and the basis for the data structure of this work. In the CSR format, the row coordinate information of nonzeros is compressed, and it uses three dense arrays to represent a matrix: (1) `Val`, stores the value of all nonzeros in the matrix, the size is *nnzA*, where *nnzA* represents the number of nonzeros of matrix A. (2) `ColIdx`, which stores the column index of the corresponding element in the array `Val`, the size is *nnzA*. (3) `RowPtr`, which stores the memory offset of the first element in each row of the sparse matrix, that is, the subscript of `Val`. `RowPtr`[i+1]-`RowPtr`[i] calculates the number of nonzeros in the ith row of the matrix. Its size is *rowA* + 1, where *rowA* is the number of rows in the matrix A.

There is no dependency between rows in the SpMV operation, so it can easily implement row-to-row parallel execution. Algorithm 1 shows a pseudocode of parallel SpMV using the CSR format. For each row of matrix A, it first creates a variable *sum* as the accumulator of that row, traverses each nonzero of that row, and obtains the corresponding x according to the column index of the nonzero (line 4 in Algorithm 1). It then accumulates the product of the x and the current nonzero to the variable *sum* (line 5 in Algorithm 1), and finally writes the value of the accumulator *sum* back to y. Recall that we in Section 1 divided the SpMV operation into three parts: `RANDOM ACCESS`, `COMPUTE` and `MISCELLANEOUS`, and Figure 2 shows the proportion of the three parts in the SpMV execution time for all 2893 matrices. In addition to the `RANDOM ACCESS` part, the overhead of the `COMPUTE` part also accounts for a large portion, and the average proportions of `RANDOM ACCESS`, `COMPUTE` and `MISCELLANEOUS` parts in the entire execution time of SpMV are 25.1%, 21.1% and 53.8%, respectively.

2.2 Specific MMA Units

In recent years, the rapid growth of artificial intelligence has prompted many processors to incorporate specialized matrix multiply-accumulate units to improve the performance of their most time-consuming arithmetic operations, such as dense GEMM and convolution. Examples of such units include NVIDIA Tensor Cores, AMD Matrix Cores, etc.

Taking tensor core for example, it is an ASIC specifically designed for small GEMM which can multiply two 4x4 matrices and add the result to another 4x4 matrix in a single clock cycle. The relevant instructions take the operands of 32 threads (a warp) to complete the GEMM operation in a tensor core. By the third generation of tensor core, it has been able to support integer, half, single, and double precision floating point data types. NVIDIA provided a CUDA C++ Warp Matrix Multiply-Accumulate (WMMA) API to program the tensor cores, but in order to support SpMV operation more flexibly, we call the mma instructions provided in PTX to complete the matrix multiply-accumulate operation. Figure 4 shows the layout of the fragments held by different threads of the FP64 precision `mma_m8n8k4` instruction, where the *MMA_M*, *MMA_N* and *MMA_K* are 8, 8 and 4, respectively. The warp that calls this instruction will multiply an 8-by-4 matrix A with a 4-by-8 matrix B and get an 8-by-8 dense matrix C. In this operation, the three fragments that make up these three matrices are distributed among the 32 threads of the warp. Listing 1 shows the corresponding PTX codes of the instruction. In this paper, the `COMPUTE` part in DASP is mostly done by calling the mma instructions.

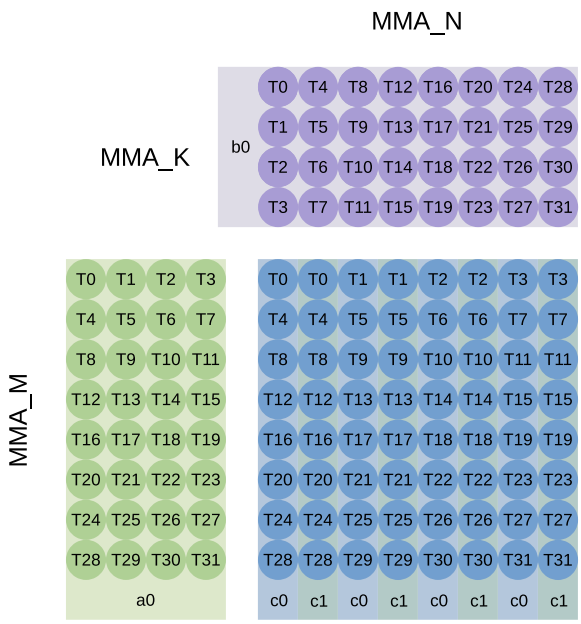


Figure 4: The layout of the fragments held by different threads of the FP64 precision `mma_m8n8k4` instruction.

Listing 1 The FP64 precision `mma_m8n8k4` instruction.

```
% __device__ __forceinline__
void mma_m8n8k4(double *acc, double &frag_a,
                double &frag_b){
    asm volatile(
        "mma.sync.aligned.m8n8k4.
         row.col.f64.f64.f64.f64"
        " { %0, %1 }, "
        " { %2 }, "
        " { %3 }, "
        " { %0, %1 };"
        : "+d"(acc[0]), "+d"(acc[1]):
        "d"(frag_a), "d"(frag_b));
}
```

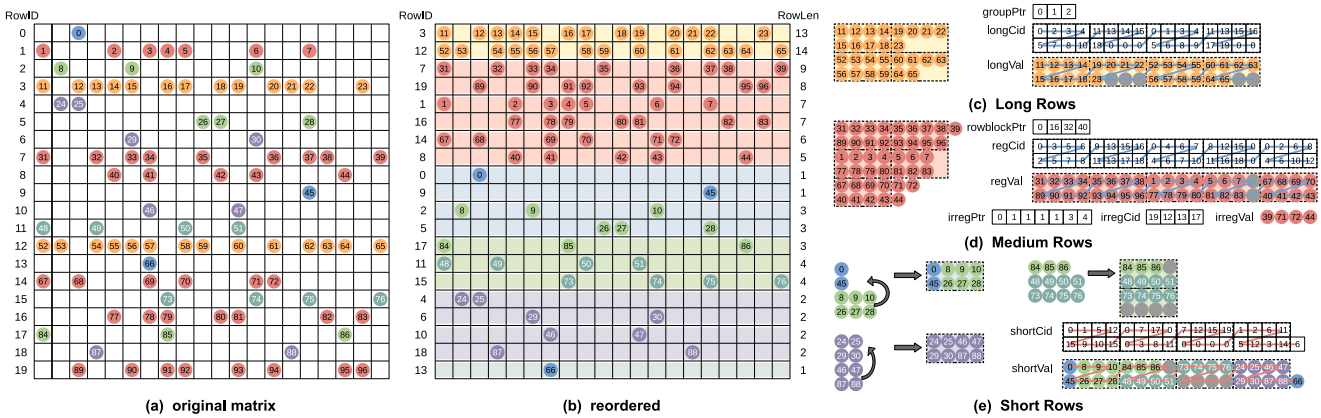


Figure 5: An example matrix A of size 20-by-20 stored in several blocks of size 2-by-4 (assuming $MMA\_M = 2$, $MMA\_K = 4$). The matrix A is divided into three categories after rearrangement: long rows, medium rows, and short rows. The nonzeros in each category are given several 2-by-4 blocks by some variation and supplementary zero elements, and the data related to all elements in one category are stored by several arrays.

3 DASP

3.1 Overview

DASP is a new algorithm using specific dense MMA units for accelerating general SpMV. It consists of two main components: a new data structure that can be efficiently used for the compute pattern of the MMA units, and the corresponding SpMV algorithms.

Due to the strict requirements of the MMA units on data layout, we first need to convert the sparse matrix A from the basic CSR format to a blocked format of a certain size (determined by the MMA_M, MMA_N and MMA_K corresponding to the mma instructions), so that we can directly use the mma instructions for computing in the GPU kernel. For the sparse matrix A, we classify it based on its row length and store the nonzeros of each part using different formats. Section 3.2 will introduce the DASP data structure in detail.

For nonzeros from different parts, we use different allocation methods to distribute them to the MMA units for calculation. Then by calling the CUDA shuffle instructions, we extract the results we needed from the accumulator obtained by the mma instruction. Finally, the results are written back to the vector y. In GPU kernels, we also employ optimization techniques such as cache bypass and adaptive workload allocation, which will be introduced in Section 3.3.

3.2 Data Structure

We analyze the distribution of nonzeros in each row of the sparse matrix A, and group all rows into three categories according to the number of nonzeros in each row (i.e. Row_len):

- Long Rows: $Row\_len > MAX\_LEN$;
- Medium Rows: $4 < Row\_len \leq MAX\_LEN$;
- Short Rows: $Row\_len \leq 4$.

The MAX_LEN is an adjustable parameter that represents the maximum length of medium rows, here we set its value to 256 (this is the value that is just right for the workload of a thread block, the details will be described in Section 3.3). In the actual calculation, the minimum layout supported by the mma instruction is m8n8k4 ($MMA\_M = 8, MMA\_N = 8, MMA\_K = 4$), so the block size in the actual data structure is 8×4. Figure 5 shows an example matrix of size 20-by-20. In this figure, we assume that the pattern supported by the mma instruction is m2n2k4, so the nonzero block size in the example is 2×4.

For the Long rows part, the nonzeros in each row will be divided into several groups, and the number of nonzeros in each group is $2 \times MMA\_M \times MMA\_K$, that is, 64. If the number of nonzeros in a row is not sufficient to be a multiple of the group, it will be padded with zero to reach a number of nonzeros that can be evenly divided by the group size. Therefore, the data of the long rows part are stored in three arrays: (1) `longVal`: stores the value of the elements in the long rows, including the filled zero elements. These zero elements are appended after the end of nonzeros in each row, and the size of this array is nnz_long_new. That is, the sum of the number of nonzeros in the long rows part and the number of filled zero elements; (2) `longCid`: stores the column index of the corresponding element in the array `longVal`. The column index of zero elements is set to 0, and the array size is nnz_long_new; (3) `groupPtr`: stores the offset of the first group of each row, which represents the position of the first group of each row in all groups, and its size is $row\_long + 1$ (row_long is the total number of rows in the long rows part). The yellow part in Figure 5 represents an example of the long rows part, where it is assumed that each group has 16 elements, so two long rows are divided into 2 groups.

For the medium rows part, all rows are sorted in a stable descending order. After that, each MMA_M rows is regarded as a row-block, and each row-block is divided into several blocks of size $MMA\_M \times MMA\_K$ according to $threshold$. The $threshold$ is a customized parameter that is set to 0.75 in DASP. When the number of nonzeros in an $MMA\_M \times MMA\_K$ space exceeds $MMA\_M \times MMA\_K \times threshold$, we recognize it as a block, and the empty positions are filled by zero elements, which we call these blocks the

regular part of the medium rows; otherwise, the nonzeros that are not considered blocks belong to the irregular part of the medium rows. We store the elements of the regular part and the irregular part separately, so the medium rows part uses six arrays to store its related data: (1) `regVal`: stores the values of the elements in the regular part, including the nonzeros in the original matrix and the filled zero elements. This array uses intra-block row-major layout, and its size is *nnz_reg_new*; (2) `regCid`: stores the column index of the corresponding elements in the array `regVal`, and its size is *nnz_reg_new*; (3) `rowblockPtr`: stores the memory offset of the first element belonging to the regular part in each row-block, and its size is *rowblock_num*+1 (*rowblock_num* is the number of row-blocks contained in the medium rows); (4) `irregVal`: stores the value of the nonzeros in the irregular part, which size is *nnz_irreg*; (5) `irregCid`: stores the column index of the corresponding nonzeros in the array `irregVal`, and its size is *nnz_irreg*; (6) `irregPtr`: stores the memory offset of the first nonzero of each row in the irregular part, and its size is *row_medium* + 1 (*row_medium* is the number of rows in the medium rows). The red part in Figure 5 shows the storage format of the medium rows.

For the short rows part, we adopt the strategy of piecing to blocks to enhance the utilization of the MMA units. The rows of *row_len* = 1 and *row_len* = 3 are pieced together to obtain rows of *row_len* = 4, and the rows of *row_len* = 2 are pieced together to get rows of *row_len* = 4. For rows of *row_len* = 3 remaining after piecing with rows of *row_len* = 1 (not enough to form a block), they are treated as rows of *row_len* = 4 by filling in one zero element. For the rows of *row_len* = 1 remaining after piecing with rows of *row_len* = 3, we place them at the end of all short rows elements. Therefore, all the data in the short rows part are divided into four categories: 1&3 pieced rows, rows of *row_len* = 4 (including those filled with a zero element), 2&2 pieced rows and the rows of *row_len* = 1. The data related to the short rows part are stored using two arrays: (1) `shortVal`: stores the value of all elements in the short rows part, and the size of this array is *nnz_short_new*; (2) `shortCid`: stores the column index of the corresponding element in the array `shortVal`, the index of zero element is set to 0, the size of this array is *nnz_short_new*. As the row lengths of the four categories in short rows are fixed, there is no need for redundant arrays to store element memory offsets. The cool-toned part of Figure 5 offers an example of the short rows part.

3.3 Algorithm Description

Our DASP offers different computation strategies for different categories of rows, and we will introduce them in the following. We adopt the bypass cache method in the following algorithms to improve the hit rate of x in the cache as much as possible.

3.3.1 Long Rows. As mentioned in the Section 3.2, each long row will be divided into several groups of $2 \times MMA\_M \times MMA\_K$ elements. Algorithm 2 shows the pseudocode of computation of the long rows part. Before each call to the mma instruction, a block ($MMA\_M \times MMA\_K$) of data is temporarily stored in the $fragA$ and $fragX$ registers of each thread. Then the mma instruction is called to make 32 threads in one warp work together to compute a GEMM of size m8n8k4. Upon completing two MMA computations, eight meaningful results are produced and distributed among the

Algorithm 2 A pseudocode of warp-level Long-Rows SpMV.

```
1:  for laneid = 0 to 31 in parallel do
2:      fragY[2], fragA, fragX ← 0
3:      idx = (3 & laneid) + (laenid >> 2)× MMA_K
4:      for i = 0 to 1 do
5:          fragA ← longVal[offsetA + idx]
6:          fragX ← valX[longCid[offsetA + idx]]
7:          mma_m8n8k4(fragY, fragA, fragX)
8:          idx += MMA_M × MMA_K
9:      end for
10:     fragY[0] += __shfl_down_sync(0xffffffff, fragY[0], 9)
11:     fragY[0] += __shfl_down_sync(0xffffffff, fragY[0], 18)
12:     fragY[1] += __shfl_down_sync(0xffffffff, fragY[1], 9)
13:     fragY[1] += __shfl_down_sync(0xffffffff, fragY[1], 18)
14:     fragY[0] += __shfl_sync(0xffffffff, fragY[1], 4)
15:     if laneid == 0 then
16:         warpVal[warpid] ← fragY[0]
17:     end if
18:     threadVal ← 0
19:     for i = laneid to row_warp_len step WARP_SIZE do
20:         threadVal += warpVal[offsetW + i]
21:     end for
22:     threadVal = warpReduceSum(threadVal)
23:     if laneid == 0 then
24:         valY[warpid] ← threadVal
25:     end if
26: end for
```

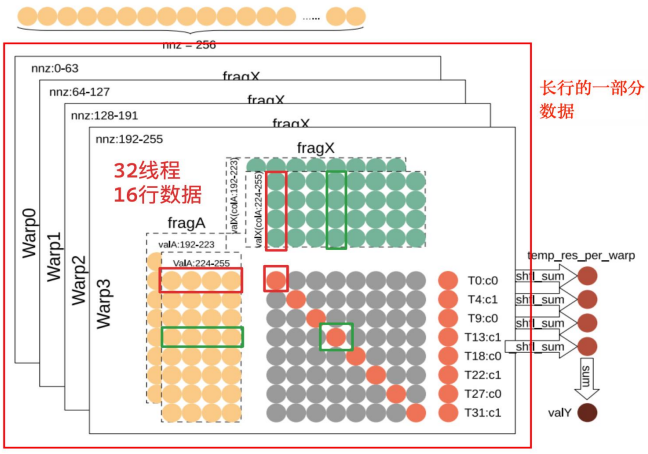


Figure 6: An example of DASP algorithm for long rows. The *Row_len* of this row is 256, and the calculation is completed by four warps.

32 threads in the register $fragY$. The next step is to call the CUDA Shuffle instructions to sum these 8 values and store the result in the register of the first thread. The result is then written to the pre-allocated global memory array `warpVal`. After all warps have been calculated and their results written in `warpVal`, the final value y is obtained by performing a warp-level summation of all values in `warpVal` generated in one row. It is set that there are 4 warps in a thread block, and each warp calculates 2 data blocks in the long rows, so a thread block will calculate 256 elements in one row. Therefore, the value of *MAX_LEN* is said to be exactly the

workload of a thread block. Figure 6 illustrates the computation of a long row with size 256.

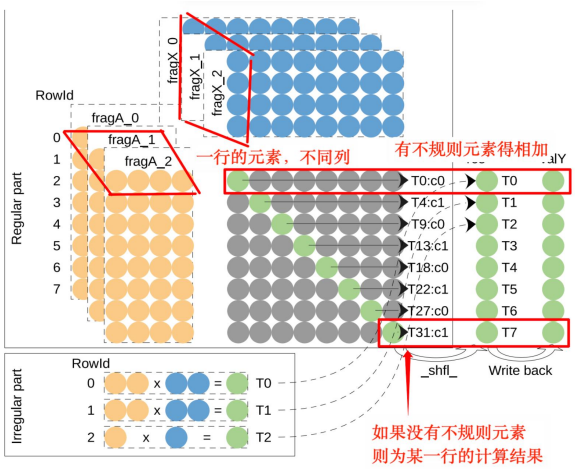


Figure 7: An example of DASP algorithm for medium rows, and it is a row-block which is divided to regular part with three blocks and a irregular part.

Algorithm 3 A pseudocode of warp-level Medium-Rows SpMV.

```
for laneid = 0 to 31 in parallel do
    for i = 0 to LOOP_NUM do
        fragY[2], fragA, fragX, res ← 0
        idx = (3 & laneid) + (laenid >> 2)× MMA_K
        bid = wid× LOOP_NUM + i
        len = rowblockPtr[bid + 1]−rowblockPtr[bid]
        for j = 0 to len step MMA_K do
            fragA ← regVal[offsetA + idx]
            fragX ← valX[regCid[offsetA + idx]]
            mma_m8n8k4(fragY, fragA, fragX)
            idx += MMA_M × MMA_K
        end for
        target = ((laneid − i × 8) >> 1) × 9
        fragY[0] = __SHFL_SYNC(0xffffffff, fragY[0], target)
        fragY[1] = __SHFL_SYNC(0xffffffff, fragY[1], target + 4)
        if (laneid >> 3) == i then
            res = (1 & laneid) == 0 ? fragY[0] : fragY[1]
        end if
    end for
    if (laneid >> 3) < LOOP_NUM then
        cur_row = wid× LOOP_NUM × MMA_M + laneid
        for i =irregPtr[cur_row] to irregPtr[cur_row + 1] do
            res += irregVal[i]× valX[irregCid[i]]
        end for
        valY[cur_row] ← res
    end if
end for
```

3.3.2 Medium Rows. A new parameter *LOOP_NUM* is introduced when performing the medium rows calculation, which indicates the number of row-blocks to be calculated by a warp. When *row_medium* is less than 59990, the value of *LOOP_NUM* is 1; when *row_medium* is greater than or equal to 59990 and less than 400000, the value of *LOOP_NUM* is 2; and the value of *LOOP_NUM* is 4 when *row_medium* is greater than or equal to 400000. Algorithm 3 shows the pseudocode of the medium rows' calculation. Similarly, the data of the corresponding block are loaded into the local register before the calculation, and then the mma instruction is called to perform the calculation. In general, a row-block will consist of multiple blocks, so the number of loop calculations is controlled by the length of the row block, and the products of multiple calculations of one row-block are accumulated in the register $fragY$. After extracting the value in the register $fragY$ to the corresponding location of the register *res* by the CUDA Shuffle instructions, the calculation of the regular part of the medium rows is basically completed. For the elements in the irregular part, each thread is responsible for one row in parallel, and the computation results are accumulated into the corresponding register *res* and finally written back to the array `valY` together. Figure 7 shows the computation process of the medium rows when $LOOP\_NUM = 1$. The row-block in the figure consists of three blocks, requiring the execution of three MMA computations.

Algorithm 4 A pseudocode of warp-level Short-1&3-Rows SpMV.

```
for laneid = 0 to 31 in parallel do
    fragA, fragX, res ← 0
    idx = (3 & laneid) + (laenid >> 2)× MMA_K
    for i = 0 to 3 do
        fragY[2] ← 0
        cidA ← shortCid[offsetA + idx]
        if 1 & i == 0 then
            fragA ← shortVal[offsetA + idx]
            fragX = 3 & laneid == 0 ? valX[cidA] : 0
        else
            fragX = 3 & laneid == 0 ? 0 : valX[cidA]
            idx += MMA_M × MMA_K
        end if
        mma_m8n8k4(fragY, fragA, fragX)
        target = ((laneid − i × 8) >> 1) × 9
        fragY[0] = __SHFL_SYNC(0xffffffff, fragY[0], target)
        fragY[1] = __SHFL_SYNC(0xffffffff, fragY[1], target + 4)
        if (laneid >> 3) == i then
            res = (1 & laneid) == 0 ? fragY[0] : fragY[1]
        end if
    end for
    valY[offsetY + laneid] ← res
end for
```

3.3.3 Short Rows. Because the short rows part are further grouped to four categories, they also correspond to four algorithms.

Algorithm 4 shows the pseudocode of the computation process of short rows by 1&3 piecing together. To optimize the utilization of compute resources and MMA units, one warp will call the mma instruction four times to finish the calculation of two blocks. This way, a warp can precisely compute 32 consecutive values of y. Before calling the mma instruction for the computation, the data of the relevant block must be stored in the registers $fragA$ and $fragX$. However, instead of two complete loads of data and MMA computations, each block only needs to load the value of matrix A once

and the value of x twice: the first time is to load the value of x corresponding to the first column of each block (the rest empty positions in $fragY$ are set to zero), and the second time is to load the value of x corresponding to the last three columns of each block. The results of the MMA computation are distributed among the 32 threads in the register $fragY$. Using the CUDA Shuffle instructions, these results are stored in the register res in order and then written back to the array `valY`. Figure 8 shows the calculation of 1&3 pieced short rows within a warp. Similarly, the short rows by 2&2 piecing together take almost the same computation strategy. When loading the value of x in a block into the register $fragX$, the first two columns are loaded first, followed by the last two columns. The method of piecing together effectively reduces the data transfer overhead and improves the efficiency of the MMA units.

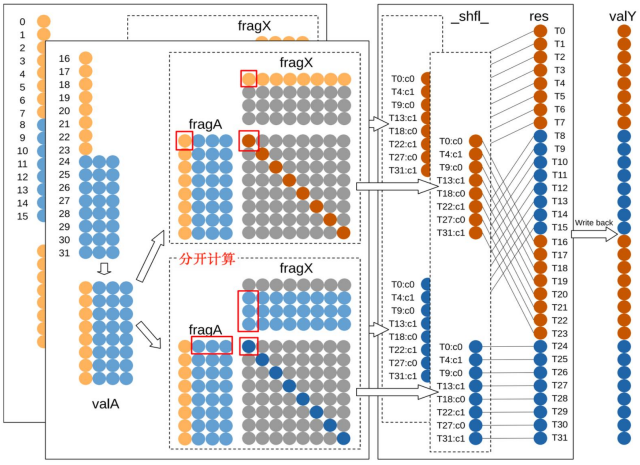


Figure 8: An example of DASP algorithm for short rows by 1&3 piecing together.

For rows of $row\_len = 4$ (including the rows made up of zero-filled elements), a warp also calls the mma instruction four times, and completes the computation of four blocks. Before each calculation, a whole block of data is loaded into registers $fragA$ and $fragX$, and then the mma instruction is called to perform the calculation. The eight meaningful products obtained from each MMA computation are stored in the registers of eight consecutive threads by the CUDA Shuffle instructions, and the result is written back to the array `valY` immediately after all four calculations are completed.

For rows with only one nonzero, we use the basic compute unit for calculation. Algorithm 5 shows the pseudocode of the computation process for such rows. For each row, a thread is assigned to compute one row, and the result is written back to the `valY` corresponding to that row.

Algorithm 5 A pseudocode of Short-1-Rows SpMV.

```
for tid = 0 to short_row_1 in parrallel do
    vala ← shortVal[offsetA + tid]
    valx ← valX[shortCid[offsetA + tid]]
    valY[offsetY + tid] = vala × valx
end for
```

4 EXPERIMENTAL RESULTS

4.1 Experimental Setup

Our experimental platform includes two NVIDIA GPUs: an A100 (Ampere architecture) and an H800 (Hopper architecture). The GPU driver version is 525.85.12, and the CUDA version is 12.0.

Table 1: The two GPUs and four algorithms evaluated.

Two NVIDIA GPUs	Four algorithms
(1) A100 (Ampere) PCIe, FP64 tensor cores with 19.5 TFlops, FP16 tensor cores with 312 TFlops, 40 GB, B/W 1555 GB/s	(1) CSR5 [64]
	(2) TileSpMV [76]
	(3) LSRB-CSR [63]
(2) H800 (Hopper) PCIe, FP16 tensor cores with 756 TFlops, 80 GB, B/W 2048 GB/s	(4) cuSPARSE-BSR
	(5) cuSPARSE-CSR
	(6) DASP (this work)

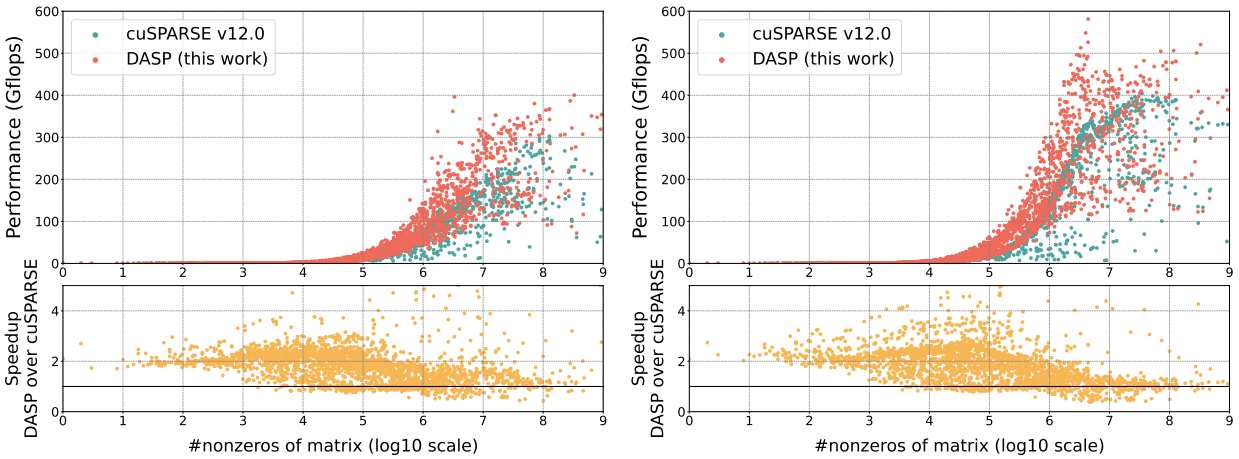
We evaluate DASP in both FP64 and FP16 precisions with hardware support on the two GPUs. For DASP with FP64 double precision, we perform the tests on A100 and compare with the CSR5 algorithm proposed by Liu and Vinter [64], the TileSpMV algorithm proposed by Niu et al. [76], the LSRB-CSR algorithm proposed by Liu et al. [63], the routines `cusparseSpMV()` (using CSR format) and `cusparse?bsrmv()` (using BSR format) in the latest cuSPARSE v12.0. For DASP with FP16 half precision, we do experiments on both A100 and H800 GPUs, and compare our work with the routine `cusparseSpMV()` (using CSR format) cuSPARSE v12.0 (since all CSR5, TileSpMV, LSRB-CSR, and `cusparse?bsrmv()` do not support half precision). The specifications of the two GPUs and the six algorithms tested are listed in Table 1. Note that we do not compare DASP with the latest SpMV work AlphaSparse [27], since it only supports single precision and takes way too much time for testing (preprocessing even a single sparse matrix needs several hours).

Our experimental dataset includes all 2893 matrices in the SuiteSparse Matrix Collection [24]. We also list 21 representative matrices widely tested in existing SpMV work [27, 64, 65, 73, 76, 95] in Table 2 to analyze the performance of our DASP more deeply.

4.2 Performance Comparison over Existing SpMV work

We compare our work DASP with CSR5, TileSpMV, LSRB-CSR, cuSPARSE v12.0 using the BSR and CSR formats, where we set the BSR block sizes to 2x2/4x4/8x8 and chose the best of the three settings as the final performance of cuSPARSE BSR method. The performance of these six methods on the A100 GPU for FP64 precision is shown in Figure 10, and the performance of DASP and cuSPARSE v12.0 using the CSR format for FP16 precision on the two GPUs is shown in Figure 9.

For the comparison of FP64 precision performance, it can be seen from the sub-figure on top of Figure 10 that our method shows the best performance on A100 GPU for the majority of matrices. Specifically, compared to these four methods, our method is faster than CSR5 on 2403 matrices, faster than TileSpMV on 2579 matrices, faster than LSRB-CSR on 2251 matrices, faster than cuSPARSE BSR on 2340 matrices, and faster than cuSPARSE CSR on 2344



(a) Half precision DASP performance and speedups on NVIDIA A100 GPU

(b) Half precision DASP performance and speedups on NVIDIA H800 GPU

Figure 9: The two sub-figures on top show performance (in GFlops) of the two SpMV methods on two GPUs with FP16 half precision. The two sub-figures at the bottom show the speedups of our DASP over the cuSPARSE v12.0, respectively.

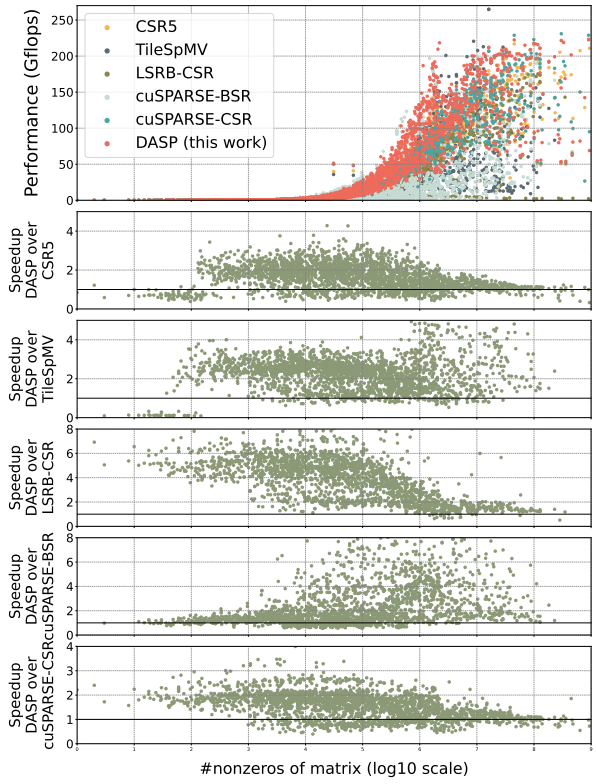


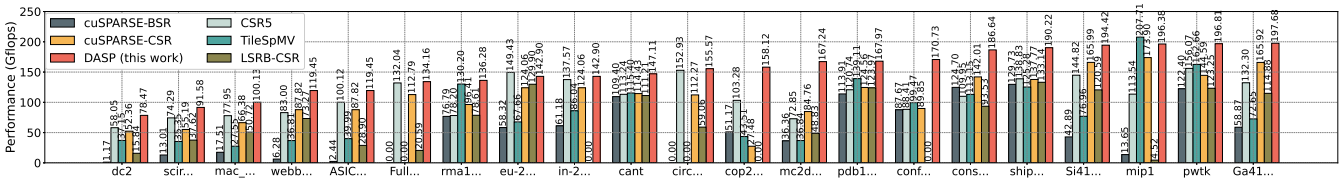
Figure 10: The sub-figure on top shows performance (in GFlops) of the six SpMV methods on an NVIDIA A100 GPU in FP64 double precision. The five sub-figures at the bottom show the speedups of our DASP over the CSR5, TileSpMV, LSRB-CSR, cuSPARSE v12.0 BSR and CSR format routines, respectively.

matrices, and achieves on average (geometric mean) 1.46x, 2.09x, 3.29x, 2.08x, and 1.52x (up to 12.64x, 17.48x, 90.59x, 283.92x, and 6.94x) speedups over them. The best speedups are in the matrices 'rel19', 'kron_g500-logn20', 'mycielskian18', 'lp_osa_60' and 'wiki-Talk', respectively. The matrix 'rel19' has very short rows, so all rows of this matrix belong to the short rows category in DASP, and the zero element fill rate of this matrix is 0.85%, which means that only a few numbers of zeros are filled to make up the suitable size. The matrix 'kron_g500-logn20' lacks a particularly obvious block structure, which can be unfriendly to the TileSpMV method. The distribution of nonzeros in matrix 'wiki-Talk' is rather irregular, with the few rows occupying the most of nonzeros, and the method for the long rows category in DASP copes with this situation precisely.

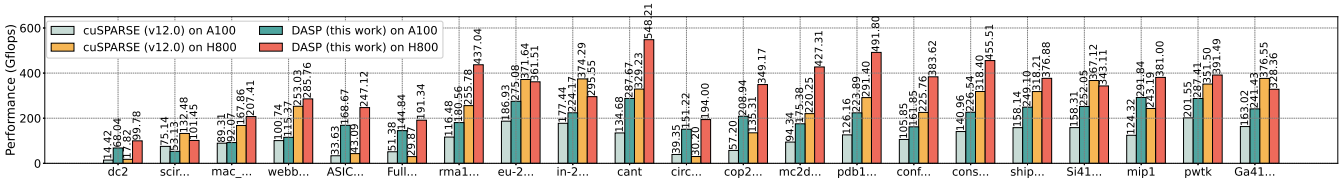
For the comparison of FP16 precision performance, our method also shows the better performance on both A100 and H800 GPUs for most matrices. Compared to cuSPARSE, our method is faster than it on 2578 and 2576 matrices on A100 and H800 GPUs, respectively. In addition, our method achieves on average 1.70x and 1.75x (up to 26.47x and 65.95x) speedups over the cuSPARSE on A100 and H800, respectively. The best speedups are both in the matrix 'bibd_20_10', a rectangular matrix, which has many nonzeros in each row, so all rows of this matrix belong to the long rows category in DASP. Overall, our method has a notable advantage for most matrices.

4.3 Effectiveness of Data Structure

To conduct a more detailed analysis, we list the FP64 and FP16 precision performance comparison of 21 representative matrices (in Table 2) on A100 and H800 GPUs in Figure 11. Meanwhile, the different categories and their corresponding methods in DASP make it possible to have suitable task assignments to obtain good performance for matrices of various patterns. To demonstrate the

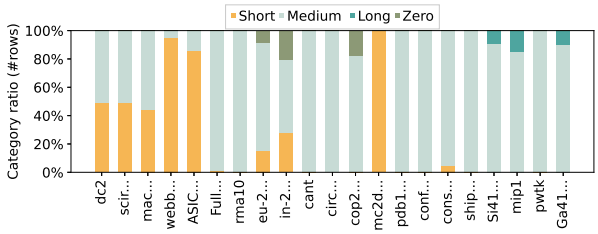


(a) FP64 double precision SpMV performance comparison of 21 representative matrices on an NVIDIA A100 GPU.

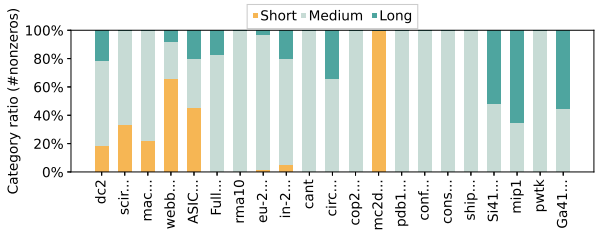


(b) FP16 half precision SpMV performance comparison of 21 representative matrices on NVIDIA A100 and H800 GPUs.

Figure 11: The sub-figure on top shows the performance comparison of performing double precision SpMV operation of the 21 representative matrices on NVIDIA A100 GPU. The bottom sub-figure shows the performance comparison of performing half precision SpMV operation of the 21 representative matrices on NVIDIA A100 and NVIDIA H800 GPUs. The '0.00' on the bar areas indicate that the corresponding algorithm fails to perform its SpMV operation on the matrix.



(a) The ratio of rows in different categories.



(b) The ratio of nonzeros in different categories.

Figure 12: Two figures show the ratio of the number of rows and nonzeros in different categories to the total number of rows and nonzeros, respectively.

effectiveness of the DASP data structure, by analyzing 21 representative matrices, we plot the ratio of the number of rows in different categories to the total rows, and the ratio of the number of nonzeros in different categories to the total nonzeros in Figure 12.

Combining Figures 11 and 12, it can be seen that most of the matrices dominated by each category are able to have a more desirable performance compared to the competitors. For matrices where short rows account for most of all rows, such as the matrices 'webbase-1M', 'ASIC_680k' and 'mc2depi', the FP64 and FP16 precision performance of these matrices can completely outperform the comparison methods on both GPUs. Taking 'mc2depi' for example, all rows of this matrix belong to the short rows category, and the FP64 precision performance of this matrix can achieve 2.29x, 4.54x, 3.42x, 4.60x and 1.97x speedups over CSR5, TileSpMV, LSRB-CSR, cuSPARSE-BSR and cuSPARSE-CSR on A100 GPU; the FP16 precision performance of this matrix can attain 1.85x and 1.94x speedups over cuSPARSE on A100 and H800 GPUs, respectively. For matrices consisting almost entirely of the medium rows category, such as the matrices 'rma10', 'cant', 'cop20k_A', 'pdb1HYS', 'conf5_4-8x8-10', 'consph', 'shipsec1' and 'pwtk', their FP64 and FP16 precision performance can still be better than the other methods. Taking 'cop20k_A' for example, it consists of 99843 medium rows and 21349 empty rows (rows without a nonzero), and its FP64 precision performance achieves 1.53x, 3.63x, 3.08x and 5.75x speedups over CSR5, TileSpMV, cuSPARSE-BSR and cuSPARSE-CSR on A100 GPU; the FP16 precision performance of this matrix can gain 3.65x and 2.58x speedups over cuSPARSE on A100 and H800 GPUs, respectively. For matrices consisting mainly of long rows category, such as matrices 'Si41Ge41H72', 'mip1' and 'Ga41As41H72', even though the computation method corresponding to the long row category will have more reduction operation compared to the methods of the other two categories, it can still show competitive performance on these matrices. These matrices in Figure 12(a) appear to have very few rows in the long rows category, but the lengths of long rows are usually very large, causing these matrices to still be matrices with a large portion of long rows, as shown in Figure 12(b).

Matrices which consist of at least two categories also do not suffer from performance discounts due to method differences, such as matrices 'FullChip', 'circuit5M' and 'dc2'. Taking a larger matrix 'circuit5M' and a moderately sized matrix 'dc2' as examples: the

Table 2: Information of the 21 representative matrices.

Matrix	Plot	Size	*nnz*
pwtk		217K×217K	11.5M
FullChip		2.9M×2.9M	26.6M
mip1		66K×66K	10.4M
mc2depi		525K×525K	2M
webbase-1M		1M×1M	3.1M
circuit5M		5.5M×5.5M	59M
Si41Ge41H72		185K×185K	15M
Ga41As41H72		268K×268K	18.4M
in-2004		1.4M×1.4M	16.9M
eu-2005		862K×862K	19M
shipsec1		140K×140K	7.8M
mac_econ_fwd500		206K×206K	1.2M
scircuit		170K×170K	959K
pdb1HYS		36K×36K	4.3M
consph		83K×83K	6M
cant		62K×62K	4M
cop20k_A		121K×121K	2.6M
dc2		116K×116K	766K
rma10		46.8K×46.8K	2.3M
conf5_4-8x8-10		49K×49K	1.9M
ASIC_680k		682K×682K	3.8M

FP64 performance of 'circuit5M' achieves 1.02x, 2.63x and 1.39x speedup over CSR5, LSRB-CSR and cuSPARSE on A100, and the FP16 performance of this matrix gains 3.84x and 6.42x speedups over cuSPARSE on A100 and H800, respectively; the FP64 performance of 'dc2' gains 1.35x, 2.11x, 4.95x, 66.89x and 1.50x speedup over CSR5, TileSpMV, LSRB-CSR, cuSPARSE-BSR and cuSPARSE-CSR on A100 GPU, and the FP16 performance of this matrix achieves 4.72x and 5.60x speedups over cuSPARSE on A100 and H800, respectively.

Therefore, DASP can efficiently exploit the MMA units to accelerate SpMV by using different methods in the three categories. Overall, DASP is a generalized SpMV method, this method does not tend to compute matrices with a certain morphology and characteristics, and almost arbitrary matrices can obtain excellent performance using the DASP method. We also find that some matrices show different performance on Ampere and Hopper architectures:

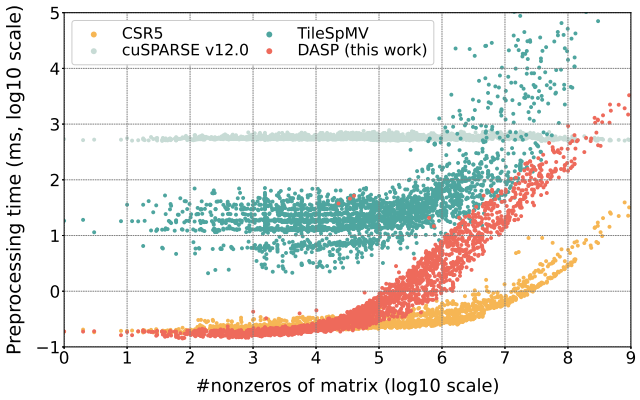


Figure 13: Comparison of preprocessing costs (converting the basic CSR format to a new data structure) of SpMV methods.

for example, the performance of the matrix 'Ga41As41H72' using the DASP method is better than cuSPARSE on the A100, while on the H800, the performance is better using cuSPARSE. We speculate that this difference is due to hardware vendors' architectural upgrades, or perhaps optimizations in the CUDA cores that result in better performance for some matrices on the H800.

4.4 Preprocessing Overhead Comparison

We also compare the preprocessing costs for converting the CSR format to our new data structure. Figure 13 shows the preprocessing time of CSR5, TileSpMV, cuSPARSE v12.0 and our DASP. As can be seen, the preprocessing of DASP is almost always faster than that of TileSpMV and cuSPARSE, and faster than CSR5 (converting the CSR data format in-place on GPU) when the number of nonzeros in the matrix is less than about $10^{4.5}$. Even though the preprocessing time of DASP can be longer than CSR5 when the matrix is large, it is deemed acceptable if more SpMV kernel calls are needed in an iterative solver.

5 RELATED WORK

There has been much work focusing on **accelerating SpMV** through a variety of methods such as balancing workload [2, 4, 6, 8, 10, 21, 39, 61, 64, 73, 74, 84, 99, 100], exploiting data locality [1, 3, 29–31, 48, 50, 51, 62, 67, 95, 96, 103–105, 107] and using machine learning for algorithm selection or format generation [9, 27, 58, 82, 88, 101, 102, 108, 109]. Anzt et al. [5] and Li et al. [59] analyzed the performance and energy efficiency of SpMV. Goumas et al. [38], Langr and Tvrdík [52], and Filippone et al. [33] surveyed the research on SpMV. Among the studies, **exploiting block structures** received much attention. Early work by Im et al. [44, 45], Vuduc et al. [91–93] and Demmel et al. [25] utilized CPU registers for optimizing very small blocks generated from some problems, such as finite element methods. Buttari et al. [14], Choi et al. [16] and Ashari et al. [7] studied the modeling of the block SpMV methods. Buluç et al. [12, 13] designed the CSB format for utilizing both block layout and the cache locality. Martone [71] developed a recursive blocking method for SpMV. Yan

et al. [98] stored dense blocks and tuned their sizes for higher performance. Recently, Niu et al. [76] developed a 2D block method and stored them in various sparse formats on GPUs, while this 2D block method was also applied in sparse matrix-sparse vector multiplication [46], sparse triangular solve [68] and sparse matrix-matrix multiplication (SpGEMM) [77]. Gao et al. [36] proposed a hybrid compression format TaiChi for diagonal dominant binary sparse matrices based on dividing one matrix into multiple dense and sparse blocks. Although these efforts divided the sparse matrix into small matrix blocks, they cannot be consumed directly by the emerging MMA units originally designed for small dense GEMM. Unlike the existing work, our DASP proposed in this paper reconstructs a general sparse row-wise data layout to utilize the specific MMA units, largely reduces the costs on the compute side and brings obvious higher overall performance.

As the MMA units can bring modern processors much higher computational power, their **real performance and potential problems** were evaluated from various aspects. Martineau et al. [70] conducted detailed benchmarks on the V100 GPUs. Choquette et al. [17] introduced the architecture of the A100 GPUs and its innovation compared to previous generations. Sun et al. [85] discussed the throughput and latency of tensor core programming. Domke et al. [26] identified the practical benefits for HPC and machine learning applications by having access to matrix engines. Chowdhury et al. [18, 19] proposed a computational model called TCU to give corresponding algorithms by capturing the main characteristics of the tensor units. Tukanov et al. [90] showed that a large number of hardware features of different matrix engines can be unified using delayed parting models. Markidis et al. [69] quantified the loss of precision in GEMM and proposed a way to reduce this loss at the expense of increasing the amount of computations. In addition, GEMM with different precision (e.g., half precision [97], mixed precision [75], and recovery precision [79]) can be greatly improved, if the MMA units are used properly. Furthermore, tensor cores have been better designed through architectural support for sparsity, duplicated memory accesses [49], redundant on-chip memory hierarchies [53], and working together with CUDA cores [41].

Besides GEMM and its use in machine learning and deep learning related operations [32, 35, 42, 43, 47, 49, 54, 55, 57, 83, 86, 94], **more algorithms can be accelerated by the MMA Units** as well. Several basic operators, such as scan and reduction [22], stencil computation [66], and FFT [28, 56, 80], have been improved by using tensor cores. When small dense or near-dense block structures can be found in sparse matrices, tensor cores can also give benefits to tile-wise SpGEMM [106]. Also, when a dense matrix is used for multiplying a sparse matrix, the key routine can be accelerated through tensor cores by Chen et al. [15], Sun et al. [87] and Li et al. [60]. In addition, tensor cores also have been used in some other applications, e.g., linear solver [11, 40, 89], molecular dynamics [34], quantum annealing simulation [20], epistasis detection [78], and compact fractals [81]. Compared to the above work using tensor cores for non-GEMM problems, this paper demonstrates that more irregular general SpMV (the input sparse matrix is not necessarily with small dense blocks) could also be accelerated by using regular specific MMA units.

6 CONCLUSION

We in this paper have proposed DASP, a new algorithm using specific dense MMA units for accelerating general SpMV. We identified that the compute part could be a critical performance bottleneck of SpMV and proposed MMA-friendly regular sparse data structures for using the MMA units. The experimental results on the latest NVIDIA Ampere and Hopper GPUs show that our DASP brought significant speedups over state-of-the-art SpMV work.

ACKNOWLEDGMENTS

We greatly appreciate the invaluable comments of all reviewers. We are also very grateful to NVIDIA China for their support in the experimental hardware, and to Haocheng Lian for the help in the preprocessing optimization. Weifeng Liu is the corresponding author of this paper. This research was supported by the National Natural Science Foundation of China under Grant No.61972415 and No.62372467.

REFERENCES

[1] C. Alappat, A. Basermann, A. R. Bishop, H. Fehske, G. Hager, O. Schenk, J. Thies, and G. Wellein. A recursive algebraic coloring technique for hardware-efficient symmetric sparse matrix-vector multiplication. *ACM Transactions on Parallel Computing*, 7(3), 2020.

[2] J. I. Aliaga, H. Anzt, T. Grützmacher, E. S. Quintana-Ortí, and A. E. Tomás. Compression and load balancing for efficient sparse matrix-vector product on multicore processors and graphics processing units. *Concurrency and Computation: Practice and Experience*, 34(14), 2022.

[3] H. Anzt, T. Cojean, G. Flegar, F. Göbel, T. Grützmacher, P. Nayak, T. Ribizel, Y. M. Tsai, and E. S. Quintana-Ortí. Ginkgo: A modern linear operator algebra framework for high performance computing. *ACM Transactions on Mathematical Software*, 48(1), 2022.

[4] H. Anzt, T. Cojean, C. Yen-Chen, J. Dongarra, G. Flegar, P. Nayak, S. Tomov, Y. M. Tsai, and W. Wang. Load-balancing sparse matrix vector product kernels on gpus. *ACM Transactions on Parallel Computing*, 7(1), 2020.

[5] H. Anzt, S. Tomov, and J. Dongarra. On the performance and energy efficiency of sparse linear algebra on gpus. *The International Journal of High Performance Computing Applications*, 31(5), 2017.

[6] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarath, and P. Sadayappan. Fast sparse matrix-vector multiplication on gpus for graph applications. In *SC '14*, 2014.

[7] A. Ashari, N. Sedaghati, J. Eisenlohr, and P. Sadayappan. A model-driven blocking strategy for load balanced sparse matrix-vector multiplication on gpus. *Journal of Parallel and Distributed Computing*, 76, 2015.

[8] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09*, 2009.

[9] A. Benatia, W. Ji, Y. Wang, and F. Shi. Sparse matrix format selection with multiclass svm for spmv on gpu. In *ICPP '16*, 2016.

[10] H. Bian, J. Huang, L. Liu, D. Huang, and X. Wang. Albus: A method for efficiently processing spmv using simd and load balancing. *Future Generation Computer Systems*, 116, 2021.

[11] P. Blanchard, N. J. Higham, F. Lopez, T. Mary, and S. Pranesh. Mixed precision block fused multiply-add: Error analysis and application to gpu tensor cores. *SIAM Journal on Scientific Computing*, 42(3), 2020.

[12] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *SPAA '09*, 2009.

[13] A. Buluç, S. Williams, L. Oliker, and J. Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *IPDPS '11*, 2011.

[14] A. Buttari, V. Eijkhout, J. Langou, and S. Filippone. Performance optimization and modeling of blocked sparse kernels. *The International Journal of High Performance Computing Applications*, 21(4), 2007.

[15] Z. Chen, Z. Qu, L. Liu, Y. Ding, and Y. Xie. Efficient tensor core-based gpu kernels for structured sparsity under reduced precision. In *SC '21*, 2021.

[16] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *PPoPP '10*, 2010.

[17] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro*, 41(2), 2021.

[18] R. Chowdhury, F. Silvestri, and F. Vella. A computational model for tensor core units. In *SPAA '20*, 2020.

[19] R. Chowdhury, F. Silvestri, and F. Vella. Algorithm design for tensor units. In *Euro-Par '21*, 2021.

[20] Y.-H. Chung, C.-J. Shih, and S.-H. Hung. Accelerating simulated quantum annealing with gpu and tensor cores. In *ISC '22*, 2022.

[21] M. Daga and J. L. Greathouse. Structural agnostic spmv: Adapting csr-adaptive for irregular matrices. In *HiPC '15*, 2015.

[22] A. Dakkak, C. Li, J. Xiong, I. Gelado, and W.-m. Hwu. Accelerating reduction and scan using tensor core units. In *ICS '19*, 2019.

[23] S. Dalton, L. Olson, and N. Bell. Optimizing sparse matrix-matrix multiplication for the gpu. *ACM Transactions on Mathematical Software*, 41(4), 2015.

[24] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1), 2011.

[25] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley, and K. Yelick. Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2), 2005.

[26] J. Domke, E. Vatai, A. Drozd, P. ChenT, Y. Oyama, L. Zhang, S. Salaria, D. Mukunoki, A. Podobas, M. WahibT, and S. Matsuoka. Matrix engines for high performance computing: A paragon of performance or grasping at straws? In *IPDPS '21*, 2021.

[27] Z. Du, J. Li, Y. Wang, X. Li, G. Tan, and N. Sun. Alphasparse: Generating high performance spmv codes directly from sparse matrices. In *SC '22*, 2022.

[28] S. Durrani, M. S. Chughtai, M. Hidayetoglu, R. Tahir, A. Dakkak, L. Rauchwerger, F. Zaffar, and W.-m. Hwu. Accelerating fourier and number theoretic transforms using tensor cores and warp shuffles. In *PACT '21*, 2021.

[29] A. Elafrou, G. Goumas, and N. Koziris. Performance analysis and optimization of sparse matrix-vector multiplication on modern multi- and many-core processors. In *ICPP '17*, 2017.

[30] A. Elafrou, G. Goumas, and N. Koziris. Conflict-free symmetric sparse matrix-vector multiplication on multicore architectures. In *SC '19*, 2019.

[31] A. Elafrou, V. Karakasis, T. Gkountouvas, K. Kourtis, G. Goumas, and N. Koziris. Sparsex: A library for high-performance sparse matrix-vector multiplication on multicore platforms. *ACM Transactions on Mathematical Software*, 44(3), 2018.

[32] B. Feng, Y. Wang, T. Geng, A. Li, and Y. Ding. Apnn-tc: Accelerating arbitrary precision neural networks on ampere gpu tensor cores. In *SC '21*, 2021.

[33] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo. Sparse matrix-vector multiplication on gpgpus. *ACM Transactions on Mathematical Software*, 43(4), 2017.

[34] J. Finkelstein, J. S. Smith, S. M. Mniszewski, K. Barros, C. F. A. Negre, E. H. Rubensson, and A. M. N. Niklasson. Quantum-based molecular dynamics simulations using tensor cores. *Journal of Chemical Theory and Computation*, 17(10), 2021.

[35] J. S. Firoz, A. Li, J. Li, and K. Barker. On the feasibility of using reduced-precision tensor core operations for graph analytics. In *HPEC '20*, 2020.

[36] J. Gao, W. Ji, Z. Tan, Y. Wang, and F. Shi. Taichi: A hybrid compression format for binary sparse matrix-vector multiplication on gpu. *IEEE Transactions on Parallel and Distributed Systems*, 33(12), 2022.

[37] C. Gómez, F. Mantovani, E. Focht, and M. Casas. Efficiently running spmv on long vector architectures. In *PPoPP '21*, 2021.

[38] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris. Performance evaluation of the sparse matrix-vector multiplication on modern architectures. *The Journal of Supercomputing*, 50(1), 2009.

[39] J. L. Greathouse and M. Daga. Efficient sparse matrix-vector multiplication on gpus using the csr storage format. In *SC '14*, 2014.

[40] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham. Harnessing gpu tensor cores for fast fp16 arithmetic to speed up mixed-precision iterative refinement solvers. In *SC '18*, 2018.

[41] K. Ho, H. Zhao, A. Jog, and S. Mohanty. Improving gpu throughput through parallel execution using tensor cores and cuda cores. In *ISVLSI '22*, 2022.

[42] N.-M. Ho and W.-F. Wong. Tensorox: Accelerating gpu applications via neural approximation on unused tensor cores. *IEEE Transactions on Parallel and Distributed Systems*, 33(2), 2022.

[43] G. Huang, H. Li, M. Qin, F. Sun, Y. Ding, and Y. Xie. Shfl-bw: Accelerating deep neural network inference with tensor-core aware weight pruning. In *DAC '22*, 2022.

[44] E.-J. Im and K. Yelick. Optimizing sparse matrix computations for register reuse in sparsity. In *ICCS '01*, 2001.

[45] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *The International Journal of High Performance Computing Applications*, 18(1), 2004.

[46] H. Ji, H. Song, S. Lu, Z. Jin, G. Tan, and W. Liu. Tilespmspv: A tiled algorithm for sparse matrix-sparse vector multiplication on gpus. In *ICPP '22*, 2022.

[47] Z. Ji and C.-L. Wang. Efficient exact k-nearest neighbor graph construction for billion-scale datasets using gpus with tensor cores. In *ICS '22*, 2022.

[48] E. Karimi, N. B. Agostini, S. Dong, and D. Kaeli. Vcsr: An efficient gpu memory-aware sparse format. *IEEE Transactions on Parallel and Distributed Systems*, 33(12), 2022.

[49] H. Kim, S. Ahn, Y. Oh, B. Kim, W. W. Ro, and W. J. Song. Duplo: Lifting redundant memory accesses of deep neural networks for gpu tensor cores. In *MICRO '20*, 2020.

[50] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris. Csx: An extended compression format for spmv on shared memory systems. In *PPoPP '11*, 2011.

[51] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units. *SIAM Journal on Scientific Computing*, 36(5), 2014.

[52] D. Langr and P. Tvrdík. Evaluation criteria for sparse matrix storage formats. *IEEE Transactions on Parallel and Distributed Systems*, 27(2), 2016.

[53] S. Lee, S. Hwang, M. J. Kim, J. Choi, and J. H. Ahn. Future scaling of memory hierarchy for tensor cores and eliminating redundant shared memory traffic using inter-warp multicasting. *IEEE Transactions on Computers*, 71(12), 2022.

[54] A. Li, T. Geng, T. Wang, M. Herbordt, S. L. Song, and K. Barker. Bstc: A novel binarized-soft-tensor-core design for accelerating bit-based approximated neural nets. In *SC '19*, 2019.

[55] A. Li and S. Su. Accelerating binarized neural networks via bit-tensor-cores in turing gpus. *IEEE Transactions on Parallel and Distributed Systems*, 32(7), 2021.

[56] B. Li, S. Cheng, and J. Lin. tcfft: A fast half-precision fft library for nvidia tensor cores. In *CLUSTER '21*, 2021.

[57] G. Li, J. Xue, L. Liu, X. Wang, X. Ma, X. Dong, J. Li, and X. Feng. Unleashing the low-precision computation potential of tensor cores on gpus. In *CGO '21*, 2021.

[58] J. Li, G. Tan, M. Chen, and N. Sun. Smat: An input adaptive auto-tuner for sparse matrix-vector multiplication. In *PLDI '13*, 2013.

[59] K. Li, W. Yang, and K. Li. Performance analysis and optimization for spmv on gpu using probabilistic modeling. *IEEE Transactions on Parallel and Distributed Systems*, 26(1), 2014.

[60] S. Li, K. Osawa, and T. Hoefler. Efficient quantized sparse matrix operations on tensor cores. In *SC '22*, 2022.

[61] W. Li, H. Cheng, Z. Lu, y. Lu, and W. Liu. Haspmv: Heterogeneity-aware sparse matrix-vector multiplication on modern asymmetric multicore processors. In *CLUSTER '23*, 2023.

[62] C. Liu, B. Xie, X. Liu, W. Xue, H. Yang, and X. Liu. Towards efficient spmv on sunway manycore architectures. In *ICS '18*, 2018.

[63] L. Liu, M. Liu, C. Wang, and J. Wang. Lsrb-csr: A low overhead storage format for spmv on the gpu systems. In *ICPADS '15*, 2015.

[64] W. Liu and B. Vinter. Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *ICS '15*, 2015.

[65] W. Liu and B. Vinter. Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors. *Parallel Computing*, 49(C), 2015.

[66] X. Liu, Y. Liu, H. Yang, J. Liao, M. Li, Z. Luan, and D. Qian. Toward accelerated stencil computation by adapting tensor core unit on gpu. In *ICS '22*, 2022.

[67] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *ICS '13*, 2013.

[68] Z. Lu and W. Liu. Tilesptrsv: a tiled algorithm for parallel sparse triangular solve on gpus. *CCF Transactions on High Performance Computing*, 5, 2023.

[69] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter. Nvidia tensor core programmability, performance & precision. In *IPDPSW '18*, 2018.

[70] M. Martineau, P. Atkinson, and S. McIntosh-Smith. Benchmarking the nvidia v100 gpu and tensor cores. In *Euro-Par '18*, 2019.

[71] M. Martone. Efficient multithreaded untransposed, transposed or symmetric sparse matrix-vector multiplication with the recursive sparse blocks format. *Parallel Computing*, 40(7), 2014.

[72] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, 2(19-25), 1995.

[73] D. Merrill and M. Garland. Merge-based parallel sparse matrix-vector multiplication. In *SC '16*, 2016.

[74] H. Mi, X. Yu, X. Yu, S. Wu, and W. Liu. Balancing computation and communication in distributed sparse matrix-vector multiplication. In *CCGrid '23*, 2023.

[75] D. Mukunoki, K. Ozaki, T. Ogita, and T. Imamura. Dgemm using tensor cores, and its accurate and reproducible versions. In *ISC '20*, 2020.

[76] Y. Niu, Z. Lu, M. Dong, Z. Jin, W. Liu, and G. Tan. Tilespmv: A tiled algorithm for sparse matrix-vector multiplication on gpus. In *IPDPS '21*, 2021.

[77] Y. Niu, Z. Lu, H. Ji, S. Song, Z. Jin, and W. Liu. Tilespgemm: A tiled algorithm for parallel sparse general matrix-matrix multiplication on gpus. In *PPoPP '22*, 2022.

[78] R. Nobre, A. Ilic, S. Santander-Jiménez, and L. Sousa. Exploring the binary precision capabilities of tensor cores for epistasis detection. In *IPDPS '20*, 2020.

[79] H. Ootomo and R. Yokota. Recovering single precision accuracy from tensor cores while surpassing the fp32 theoretical peak performance. *The International Journal of High Performance Computing Applications*, 36(4), 2022.

[80] L. Pisha and . Ligowski. Accelerating non-power-of-2 size fourier transforms with gpu tensor cores. In *IPDPS '21*, 2021.

[81] F. A. Quezada, C. A. Navarro, N. Hitschfeld, and B. Bustos. Squeeze: Efficient compact fractals for tensor core gpus. *Future Generation Computer Systems*, 135, 2022.

[82] N. Sedaghati, T. Mu, L. Pouchet, S. Parthasarathy, and P. Sadayappan. Automatic selection of sparse matrix representation on gpus. In *ICS '15*, 2015.

[83] Z. Song, J. Wang, T. Li, L. Jiang, J. Ke, X. Liang, and N. Jing. Gpnpu: Enabling efficient hardware-based direct convolution with multi-precision support in gpu tensor cores. In *DAC '20*, 2020.

[84] M. Steinberger, R. Zayer, and H. Seidel. Globally homogeneous, locally adaptive sparse matrix-vector multiplication on the gpu. In *ICS '17*, 2017.

[85] W. Sun, A. Li, T. Geng, S. Stuijk, and H. Corporaal. Dissecting tensor cores via microbenchmarks: Latency, throughput and numeric behaviors. *IEEE Transactions on Parallel and Distributed Systems*, 34(1), 2023.

[86] W. Sun, S. Sioutas, S. Stuijk, A. Nelson, and H. Corporaal. Efficient tensor cores support in tvm for low-latency deep learning. In *DATE '21*, 2021.

[87] Y. Sun, L. Zheng, Q. Wang, X. Ye, Y. Huang, P. Yao, X. Liao, and H. Jin. Accelerating sparse deep neural network inference using gpu tensor cores. In *HPEC '22*, 2022.

[88] G. Tan, J. Liu, and J. Li. Design and implementation of adaptive spmv library for multicore and many-core architecture. *ACM Transactions on Mathematical Software*, 44(4), 2018.

[89] J. Tu, M. A. Clark, C. Jung, and R. D. Mawhinney. Solving dwf dirac equation using multi-splitting preconditioned conjugate gradient with tensor cores on nvidia gpus. In *PASC '21*, 2021.

[90] N. Tukanov, R. Srinivasaraghavan, J. E. Moreira, and T. M. Low. Modeling matrix engines for portability and performance. In *IPDPS '22*, 2022.

[91] R. Vuduc, J. W. Demmel, and K. A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1), 2005.

[92] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *SC '02*, 2002.

[93] R. W. Vuduc and H.-J. Moon. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *HPCC '05*, 2005.

[94] Y. Wang, B. Feng, and Y. Ding. Qgtc: Accelerating quantized graph neural networks via gpu tensor core. In *PPoPP '22*, 2022.

[95] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3), 2009.

[96] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, and L. Zhang. Cvr: Efficient vectorization of spmv on x86 processors. In *CGO '18*, 2018.

[97] D. Yan, W. Wang, and X. Chu. Demystifying tensor cores to optimize half-precision matrix multiply. In *IPDPS '20*, 2020.

[98] S. Yan, C. Li, Y. Zhang, and H. Zhou. yaspmv: Yet another spmv framework on gpus. In *PPoPP '14*, 2014.

[99] W. Yang, K. Li, Z. Mo, and K. Li. Performance optimization using partitioned spmv on gpus and multicore cpus. *IEEE Transactions on Computers*, 64(9), 2014.

[100] X. Yang, S. Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on gpus: Implications for graph mining. *Proceedings of the VLDB Endowment*, 4(4), 2011.

[101] S. Yesil, A. Heidarshenas, A. Morrison, and J. Torrellas. Wise: Predicting the performance of sparse matrix vector multiplication with machine learning. In *PPoPP '23*, 2023.

[102] X. You, C. Liu, H. Yang, P. Wang, Z. Luan, and D. Qian. Vectorizing spmv by exploiting dynamic regular patterns. In *ICPP '22*, 2022.

[103] A. N. Yzelman and R. H. Bisseling. Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods. *SIAM Journal on Scientific Computing*, 31(4), 2009.

[104] A. N. Yzelman and R. H. Bisseling. Two-dimensional cache-oblivious sparse matrix-vector multiplication. *Parallel Computing*, 37(12), 2011.

[105] A. N. Yzelman and D. Roose. High-level strategies for parallel shared-memory sparse matrix-vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 25(1), 2014.

[106] O. Zachariadis, N. Satpute, J. Gómez-Luna, and J. Olivares. Accelerating sparse matrix-matrix multiplication with gpu tensor cores. *Computers & Electrical Engineering*, 88, 2020.

[107] Y. Zhang, S. Li, F. Yuan, D. Dong, X. Yang, T. Li, and Z. Wang. Memory-aware optimization for sequences of sparse matrix-vector multiplications. In *IPDPS '23*, 2023.

[108] Y. Zhao, J. Li, C. Liao, and X. Shen. Bridging the gap between deep learning and sparse matrix format selection. In *PPoPP '18*, 2018.

[109] Y. Zhao, W. Zhou, X. Shen, and G. Yiu. Overhead-conscious format selection for spmv-based applications. In *IPDPS '18*, 2018.

Appendix: Artifact Description/Artifact Evaluation

ARTIFACT DOI

https://doi.org/10.5281/zenodo.8084940

ARTIFACT IDENTIFICATION

DASP is a new algorithm using specific dense MMA units for accelerating general SpMV operation. We analyze the row-wise distribution of nonzeros and group the rows into three categories containing long, medium and short rows, respectively. We then organize them into small blocks of proper sizes to meet the requirement of MMA computation. For the three categories, DASP offers different strategies to complete SpMV by efficiently utilizing the MMA units.

We ran DASP on NVIDIA A100 GPU and H800 GPU, and record the execution time of an average of 1000 times. The GPU driver version is 525.85.12, and the CUDA version is 12.0. This artifact contains the proprecessing C code and the CUDA code for the kernels proposed in DASP. Besides, it also contains the python script for downloading matrices and the shell scripts for reproducing the major experiments.

More details of the artifact are listed as below.

- Relevant hardware details: Intel Xeon Silver 4210 CPU, NVIDIA A100 GPU and NVIDIA H800 GPU
- Operating systems and versions: CentOS Linux release 8.5.2111
- Compilers and versions: NVIDIA nvcc v12.0, GNU gcc v12.2
- Libraries and versions: NVIDIA cuSPARSE v12.0, CSR5 and TileSpMV
- Key algorithms: parallel sparse matrix-vector multiplication
- Input datasets and versions: all matrices in the SuiteSparse Matrix Collection

REPRODUCIBILITY OF EXPERIMENTS

- **Installation and Compile (3 minutes)**
  Firstly, clone the DASP code to the local machine. Then, use GNU make to build the executable:
  Compile DASP of FP64 precision: **$ make double**
  Compile DASP of FP16 precision: **$ make half**
  After that, one will get two executable called *spmv_double* and *spmv_half*. Note that one should configure the file *Makefile* according to the GPU that use. After that, in the folder `<DASP_dir>/test/`, execute the following command to compute the matrix 'cop20k_A':
  $bash run_test.sh
- **Matrices Downloading(35 hours or more)**
  In the folder `<DASP_dir>/script/`, execute the python script:
  $python3 matrix_download.py
  Then, one will get all matrices in the folder `<DASP_dir>/MM/`.
- **Large-scale Dataset Evaluation (10 hours)**
  Execute the shell scripts **run_spmv.sh** to get the preprocessing time and the performance of this dataset which are recorded in files `a100_f64_record.csv` and `a100_f16_record.csv` in folder `data/`.
  After that, execute the python scripts `script/f64_scatter.py`, `script/f16_scatter.py`, `script/f64_bar.py` and `script/f16_bar.py` and get the performance figure in this paper.

ARTIFACT DEPENDENCIES REQUIREMENTS

- **Description of the hardware resources**
  GPU: NVIDIA A100 GPU(FP64 Tensor Core, FP16 Tensor Core) and NVIDIA H800 GPU(FP64 Tensor Core). Overall, the GPU being used should be one or two NVIDIA GPUs with FP64 Tensor Core and FP16 Tensor Core.
  CPU: Intel Xeon Silver 4210 CPU. Please use multicore CPU.
  Disk Space: at least 750GB (to store the experiment input dataset).
- **Description of the operating systems**
  Any Linux system that can support CUDA v12.0 or above and GCC v12.0 or above.
- **Software libraries**
  NVIDIA CUDA Toolkit v12.0 or above; GCC v12.0 or above; Python 3.9.
  Python libraries: Pandas, Matplotlib, Numpy, Scipy.
- **Input dataset**
  Our experimental dataset includes all 2893 matrices in the SuiteSparse Matrix Collection which is publicly available(https://sparse.tamu.edu/about). The SuiteSparse Matrix Collection is so far the best collection that resolves the sparse matrix test problems [Duff et al., Sparse Matrix Test Problems, ACM TOMS, 1989]. SuiteSparse includes matrices from 264 application domains, and has been cited more than 4,300 times. Using matrices from SuiteSparse as input dataset of DASP will make our work more understandable and reproducible.

ARTIFACT INSTALLATION DEPLOYMENT PROCESS

- **Installation and compilation (3 minutes)**
  Download our artifacts from the link(https://doi.org/10.5281/zenodo.8084940). Once downloaded locally, use the command to unzip it.
  $unzip DASP.zip
  DASP requires both NVCC version and GCC version 12.0 and above. If the default compiler version of the current environment does not meet the requirements, users need to manually change the compiler path in the DASP/Makefile file:
  `line2: NVCC = /Your_CUDA_path/bin/nvcc`
  `line4: GCC = /Your_GCC_path/bin/gcc`

If you are using a machine other than the A100, change the arch number at line6. After that, in the artifact directory `DASP/`, compile the project.
$make
Then you can get the executable files `spmv_double` and `spmv_half`.
Run the following commands:
$cd test
$bash run_test.sh
This test demonstrates that the artifacts are available and verifies the correctness of the results with the matrix 'cop20k_A' as the input data.

- **Preparation for Dataset (35 hours or more)**
  Our experimental dataset includes all 2893 matrices in the SuiteSparse Matrix Collection. These matrices need to be downloaded locally via the script provided in the artifact. This process will take approximately 35 hours (at a download speed of 6MB/s, the total size is estimated to be 750GB) or longer.
  If users accept the simplified dataset, we have also prepared a small-scale dataset of 309 matrices obtained by random sampling from the SuiteSparse Matrix Collection. The download of the simplified dataset will take approximately 2.5 hours and this dataset is estimated to be 51GB in total size. Using the simplified dataset requires some simple modifications to scripts:
  In matrix_download.py:
  `line8 matrix_list1.csv -> matrix_list2.csv`
  In run_spmv_all.sh:
  `line4 matrix_list1.csv -> matrix_list2.csv`
  (Optional) Users also can install and use 'axel' tool to download matrices in parallel. In script/matrix_download.py, use line 25 code and comment out code of line 26.
  Dataset download command:
  $python3 matrix_download.py
  All matrices will be stored in the directory `MM/`. If you changed the dataset path, you need to modify the *MM_path* at line 3 in the script `run_spmv_all.sh`.
- **Preparation for Comparative SpMV Methods**
  In this paper, for DASP with FP64 double precision, we compared with the cuSPARSE(v12.0), the CSR5 (https://github.com/weifengliu-ssslab/Benchmark_SpMV_using_CSR5) and the TileSpMV (https://github.com/SuperScientificSoftwareLaboratory/TileSpMV). For DASP with FP16 half precision, wen just compared with the cuSPARSE(v12.0). The CSR5 and the TileSpMV code have been included in the current artifact.
  Compile the CSR5. You can change the compiler path at line 11 in CSR5 Makefile:
  $cd spmv_code/CSR5_cuda/
  $make
  Then, compile the TileSpMV. Note the important information we got from the authors of the TileSpMV: it requires that it must be compiled with CUDA v11.1, otherwise many of the matrices will be miscalculated leading to questionable performance records. If the user does not have CUDA v11.1, it is OK to compile directly with the current CUDA version, but the final comparison with DASP will be invalid. You can change the compiler path at line 11 in TileSpMV Makefile.
  Compile command:
  $cd ../TileSpMV/
  $make
- **Deployment (20 hours)**
  After the dataset and the other methods are ready, in the directory `script/`, run this command:
  $bash run.sh
  This script includes tests of double-precision and half-precision SpMV of the target dataset matrices, and will generate a result file and some performance plots similar to those in the paper at the end of the tests.
  The file `result.txt` in `DASP/` is the performance analysis results.
  The `a100_f64_scatter.pdf` and the `a100_f16_scatter.pdf` correspond to Figure10 and Figure9(a) in the paper, respectively.
  The `a100_f64_bar.pdf` and the `a100_f16_bar.pdf` correspond to Figure11 in the paper.
  The `preprocessing_time.pdf` corresponds to Figure 13 of the paper.