2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 978-1-6654-4066-0/20/$31.00 ©2021 IEEE | DOI: 10.1109/IPDPS49936.2021.00016

TileSpMV: A Tiled Algorithm for Sparse Matrix-Vector Multiplication on GPUs

Yuyao Niu[1], Zhengyang Lu[1], Meichen Dong[1], Zhou Jin[1], Weifeng Liu[1], Guangming Tan[2]
1. *Super Scientific Software Laboratory, China University of Petroleum-Beijing, China*
2. *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, China*
{2019211256, 2017010055, 2019011738}@student.cup.edu.cn, {jinzhou, weifeng.liu}@cup.edu.cn, tgm@ict.ac.cn

***Abstract*—With the extensive use of GPUs in modern supercomputers, accelerating sparse matrix-vector multiplication (SpMV) on GPUs received much attention in the last couple of decades. A number of techniques, such as increasing utilization of wide vector units, reducing load imbalance and selecting the best formats, have been developed. However, the 2D spatial sparsity structure has not been well exploited in the existing work for SpMV on GPUs.**

In this paper, we propose an efficient tiled algorithm called TileSpMV for optimizing SpMV on GPUs through exploiting 2D spatial structure of sparse matrices. We first implement seven warp-level SpMV methods for calculating sparse tiles stored in a variety of formats, and then design a selection method to find the best format and SpMV implementation for each tile. We also adaptively extract nonzeros in the very sparse tiles into a separate matrix to maximize the overall performance. The experimental results show that our method is faster than state-of-the-art SpMV methods such as Merge-SpMV, CSR5 and BSR in most matrices of the full SuiteSparse Matrix Collection and delivers up to 2.61x, 3.96x and 426.59x speedups, respectively.

***Index Terms*—sparse matrix-vector multiplication, tiling, GPU**

I. Introduction

Sparse matrix-vector multiplication (SpMV) multiplies a sparse matrix A and a dense vector x, and gives a resulting dense vector y. It plays a key role in sparse iterative solvers, such as conjugate gradient (CG) methods [1], and graph processing frameworks, such as GraphBLAS [2]–[5], and may be the most studied kernel of the level-2 sparse basic linear algebra subprograms (sparse BLAS) in the past decades [1], [6]–[29], [29]–[43].

The SpMV operation is in general both irregular and memory bandwidth bound, and thus is hard to optimize. To achieve high throughput SpMV on modern processors, researchers have proposed a number of techniques including reducing memory footprint of sparse matrix [6], [7], [11], [13], [15]–[17], [21], increasing data locality of accessing vector x [7]–[9], [11], [13], [14], [17], [18], [21], [22], [25], [26], [28], [44], utilizing wide vector units on modern architectures [12], [19], [20], [23], [27], [34], [37], [39], [42], [44], improving load balancing on massively parallel processors [12], [20], [21], [24], [27], [30], [32], [36], [38], [42], [45], and selecting the best format and algorithm through machine learning [1], [29], [33], [40], [41].

However, in spite of the efforts aforementioned, it should be noticed that parallel SpMV still faces a number of challenges to best use the modern parallel processors, in particular GPUs. The first one is that the recently optimized fundamental formats, such as compressed sparse row (CSR), ELLpack (ELL) and their variants, in general bring inadequate memory bandwidth utilization. The reason is that neither CSR nor ELL considered spatial structure of sparse matrix, and thus the reuse of x is often unsatisfactory. The second is that there lacks SpMV implementations optimized for very small sparse matrices that can be completely stored in the on-chip scratchpad memories. The third is that although the format and algorithm selection techniques using machine learning have been proven effective in SpMV, they have only been used for the whole matrix, and the micro-structures of a sparse matrix have not obtained benefits from the techniques.

To address the above challenges, we in this paper propose a method called TileSpMV. Its objectives include to exploit 2D sparse tile structures of sparse matrices and to implement and select the best formats and SpMV algorithms for a variety of tiles. Firstly, TileSpMV stores sparse matrices into regular sparse tiles of the same size (in our implementation, the size is always 16 by 16) to obtain in general better cache locality and higher bandwidth utilization. Secondly, the TileSpMV kernel now sees the tiles as the basic working units, but not rows or a group of nonzero elements in the existing methods, and we optimize the implementations of SpMV using seven typical formats (i.e., CSR, COO, ELL, HYB, dense, dense row and dense column) involved in the warp level on the CUDA platform. Thirdly, we design an adaptive selection method to find a best format and SpMV implementation for each sparse tile. Thus now the micro-structure could get benefits from tile-wise format and algorithm selection.

In our experiments, we compare the TileSpMV kernel with three state-of-the-art SpMV methods: the block compressed row (BSR)-SpMV in cuSPARSE v11.1 (using dense block of size 4x4), the CSR-SpMV in cuSPARSE v11.1 (i.e., an improved implementation of the Merge-SpMV [32]), and the CSR5-SpMV [27]. The test dataset includes all 2757 matrices in the SuiteSparse Matrix Collection [46], and the experimental platform contains a latest NVIDIA A100 (Ampere) GPU and an NVIDIA Titan RTX (Turing) GPU. The experimental results show that our method is faster than Merge-SpMV on 1813 matrices, faster than CSR5 on 2040 matrices, and faster than BSR on 1638 matrices, and achieves up to 2.61x, 3.96x and 426.59x speedups over them, respectively.

978-1-6654-4066-0/21/$31.00 ©2021 IEEE
DOI 10.1109/IPDPS49936.2021.00016

This work makes the following contributions:

- We propose an efficient tiled algorithm called TileSpMV for parallel SpMV on modern GPUs.
- We implement highly optimized warp-level SpMV kernels for small sparse matrices represented as sparse tiles.
- We develop an adaptive selection method to find the best storage format and kernel for each sparse tile.
- We achieve obvious speedups over state-of-the-art SpMV methods on the newest GPUs.

II. Background and Motivation

A. Parallel Sparse Matrix-Vector Multiplication (SpMV)

Sparse Matrix-Vector Multiplication (SpMV) operation multiplies a sparse matrix A with a dense vector x, and gets a dense vector y. Figure 1 shows a simple example of SpMV. In this procedure, y_i is computed by the dot product of a_{i*}, i.e., the ith row of A, and the vector x. It is easy to find that there is no dependency between rows throughout the execution process. So SpMV can be executed in parallel in rows. Algorithm 1 shows a pseudocode of parallel SpMV.

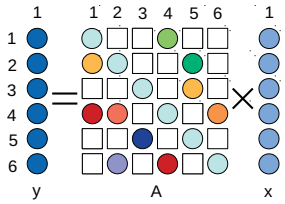


Fig. 1. An example of SpMV that multiplies a 6-by-6 sparse matrix with a dense vector x and gets a dense vector y.

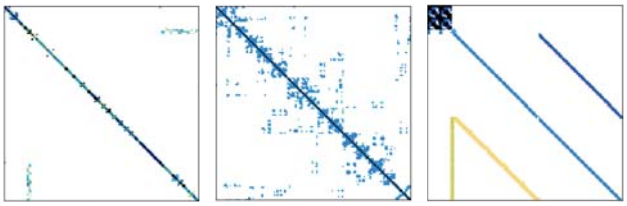
Algorithm 1 A pseudocode of parallel SpMV.

```
1: for each a_i* in the matrix A do
2:     y_i ← 0
3:     for each nonzero entry a_ij in a_i* in parallel do
4:         y_i ← y_i + a_ij × x_j
5:     end for
6: end for
```

B. Motivation

There have been a series of studies on exploiting small dense structures in sparse matrices generated from computational science and engineering problems such as finite element modeling. Figure 2 demonstrates three matrices that include obvious small dense block structures. To use the structures for accelerating SpMV on CPUs, Im et al. [7], [47], [48] developed the SPARSITY framework that could provide register level optimization for the small dense block structures, and Vuduc et al. [8], [49]–[51] developed the OSKI package [9] with a number of auto-tuning methods for register blocking and memory hierarchy optimizations.

However, such optimizations have only shown their effectiveness on CPU platforms. On the GPU part, even though the SpMV algorithms recently developed for GPUs resolved wide vectorization and load balancing problems to some extent,



(a) pwtk (wind tunnel) (b) pdb1HYS (protein) (c) TSOPF_FS_b162..

Fig. 2. Three representative sparse matrices with small block structures.

the advantages of utilizing small block structures have been largely ignored on GPUs.

Actually, it is not trivial to optimize the small block structures for parallel SpMV on GPUs for several reasons. The first one is that to use the wide vector programming model on GPUs, e.g., warp of 32 threads in CUDA, the blocks should be enough large to saturate GPU's wide SIMD units, i.e., should not be as small as the ones used for CPU register. The second reason is that when the blocks are large, they should not be saved in the dense form that may waste too much space for padding zeros and may offset the performance/space benefits. The third reason is that there is no one single sparse format and algorithm can always deliver the best performance for blocks of any sparsity structures, and a selection method is always needed. Therefore, how to design efficient GPU kernels for relatively large and sparse blocks, and to select the best formats and algorithms for them is in particular important for optimizing SpMV using block structures on modern GPU architectures.

This motivates us to design an efficient tiled SpMV algorithm for GPUs. The next section will introduce the details of the TileSpMV algorithm proposed by us and explain how we effectively address these challenges.

III. TileSpMV

A. Overview

Our TileSpMV algorithm first divides the whole input sparse matrix into a number of sparse tiles of the same and enough large size (always 16-by-16 in this work) to obtain better data locality and to saturate GPU SIMD units. We also deliver seven format options (i.e., CSR, COO, ELL, HYB, dense, dense row and dense column) for each sparse tile. Section III.B will introduce the storage structure for TileSpMV.

Then in order to better compute the sparsity structures of the tiles for SpMV, we develop seven corresponding warp-level SpMV algorithms for different structures. Since the sparse tiles are in general much smaller than a complete sparse matrix, the algorithms are required to be carefully designed. Section III.C will introduce the seven algorithms.

To make the algorithms more efficient, we also design a two-level selection method to automatically find the most suitable sparse format and algorithm for each sparse tile according to its sparsity structure, and to decide whether it is worth to extract very sparse tiles into a separate sparse matrix. Section III.D will introduce the selection method.

这里写错了吧

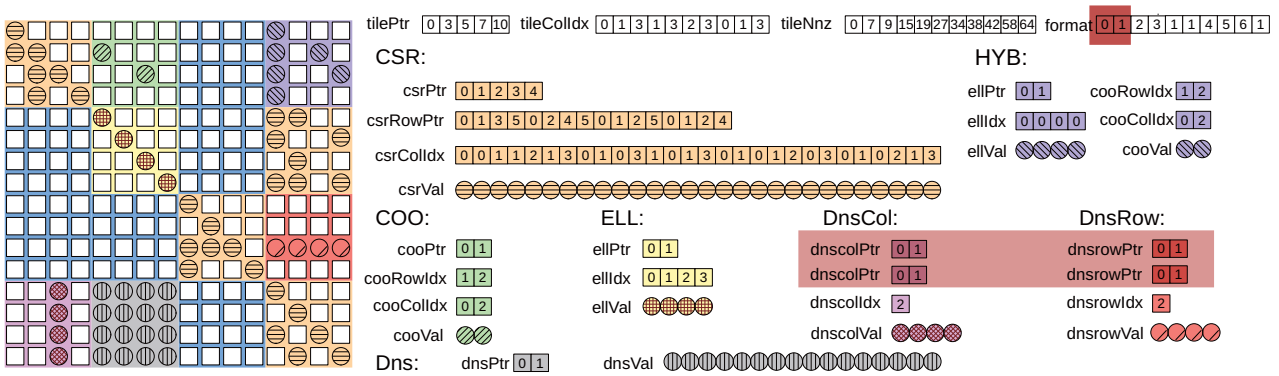


Fig. 3. An example matrix A of size 16-by-16 stored in 10 sparse tiles of size 4-by-4. The tile structure includes three arrays tilePtr, tileColIdx and tileNnz representing the memory offsets of tiles, tile column indices and the offsets for the number of nonzeros in sparse tiles. According to the format selection method, four of the 10 tiles keep the CSR format unchanged and use three arrays to store their information. The remaining six tiles are transformed to different formats, including COO, ELL, HYB, Dns, DnsRow and DnsCol. Each format has several corresponding arrays to store the nonzeros and their indices.

B. Two-Level Storage Structure

TileSpMV first divides an input matrix A into a number of sparse tiles of the same size (16-by-16 in this work), and uses a sparse tile as the basic working unit. After the partitioning, two levels of information represented as a group of arrays are generated for storing the sparse tiles. The two levels of the information store the tile structure of the matrix, and the internal information of each sparse tile, respectively. Figure 3 shows an example matrix of size 16-by-16. In this case, we divide the matrix into 10 sparse tiles of size 4-by-4 for explaining the storage structure proposed.

The tile structure of the matrix in *the first level* contains three arrays: (1) the `tilePtr` array of size $tilem A+1$, where $tilem A$ is the number of tile rows of the matrix, that stores the memory offsets of tiles in tile rows, (2) the `tileColIdx` array of size $numtile A$, where $numtile A$ is the number of sparse tiles in the matrix, that stores the column index of each tile, and (3) the `tileNnz` array of size $numtile A + 1$ that stores the memory offset of the number of nonzeros in the sparse tiles. The three arrays are plotted on the top area of Figure 3.

The second level stores the nonzero elements and their indices in each sparse tile in different formats. In this work, we have seven selections: CSR, COO, ELL, HYB, dense (Dns), dense row (DnsRow), and dense column (DnsCol).

For the CSR format, we create three arrays for saving the tile data: (1) the `csrVal` array of size $nnz A$ that stores values of all the nonzero entries in tile's order, (2) the `csrColIdx` array of size $nnz A$ that stores the column index of each nonzero. Note that due to the size of our sparse tile (i.e. 16-by-16), the column index in a tile only needs four bits and the column indices of two continuous entries are packed into one unsigned char of eight bits to further reduce space required, and (3) the `csrRowPtr` array of size $numtile A \times 16$ that stores 16 memory offsets for the nonzeros in the tile. Although the normal row pointer should contain 16+1 entries in the classical CSR, we only save 16 entries here for utilize the unsigned char data type since the second last row pointer value would not exceed 240. This means that unsigned char data type is enough to save all offsets in the row pointer, except the last value which is possibly 256. We will obtain the total number of the nonzeros, i.e., the last value of the row pointer array, from the above mentioned level-1 `tileNnz` array which stores the nonzeros number of the sparse tiles.

For the COO format, we set three arrays `cooVal`, `cooRowIdx` and `cooColIdx` to record the values, row indices and column indices of the nonzero entries respectively. Because the tile size is 16-by-16, we can find that four bits are enough for each row/column index. So we pack the 4-bit row index and 4-bit column index into an 8-bit unsigned char. The green tile in Figure 3 shows an example of the COO tile of two entries.

For the ELL format, we create two arrays `ellVal` and `ellColIdx` to store values and column indices of the nonzero entries. We also set the number of nonzero entries for each row to an equal number, called $tilewidth$ of this tile. The value of $tilewidth$ records the maximum number of nonzero entries in each row. For rows less than $tilewidth$, we fill the empty location with zero. For finding the corresponding $tilewidth$ to each tile in the ELL format, we need an extra array to store the $tilewidth$ information for each ELL tile. In Figure 3, the yellow tile is an example of ELL, and its $tilewidth$ value is 1, because the number of nonzero entries in each row is 1.

For the HYB format, a combination of ELL and COO formats is created. We use the ELL format to store the regular part of the tile, then the rest of the nonzero entries are stored in the COO format. To determine the $tilewidth$ of the ELL part, we adapt a method to gradually calculate the least memory space by setting the ELL width from the maximum to zero until the smallest memory space is find. Then the width we get is the $tilewidth$ of the ELL part. The purple tile in Figure 3 shows an example of the HYB format. It can be seen that four nonzeros in it are saved in the first column and will be stored in the ELL part, and the other two are saved into the

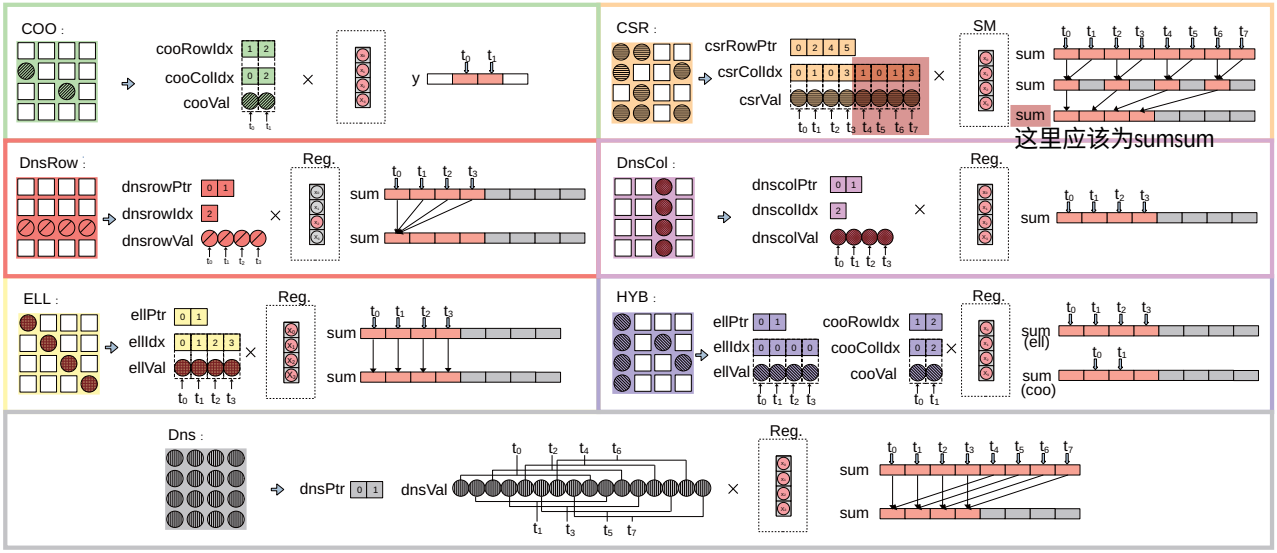


Fig. 4. Examples of seven warp-level SpMV algorithms corresponding to the formats on GPUs. For COO tile, each thread processes one element and the result of each thread will add to the corresponding location in an atomic way. For CSR tile, every 2 threads process one row, and then shuffle operations add results from these threads. For DnsRow tile, due to the four nonzero entries are distributed in the third row, we use four threads t_0 to t_3 to calculate by using x in registers. For DnsCol tile, there are four elements in the third column, so that the four threads can process them respectively in the register. For ELL tile, all threads can process its corresponding elements one by one at the same time until the calculation is over. Due to the four elements are the first element of each row, we assign four threads t_0 to t_3 to process them respectively. For HYB tile consisted of ELL part and COO part, the four threads t_0 to t_3 process the four elements stored in ELL data and then the two threads t_0 and t_1 process the two elements stored in COO data. For Dns format, eight threads process 16 elements and add the corresponding values to get result.

COO part. Here the $tilewidth$ of ELL is 1. The purple arrays on the right side show the six nonzero entries data in HYB format.

For the Dns format, we store very dense tiles such as the gray tile in the example matrix of Figure 3 in it. We only need one array `dnsVal` to stores values of all the nonzero entries in the column-major order. The gray array in the bottom of Figure 3 shows the Dns data.

For the DnsRow/DnsCol format, we create three arrays: (1) the `dnsRowVal`/`dnsColVal` array that stores values of all the nonzero entries in a natural order. The size of the array is $dnsrownnzA$/$dnscolnnzA$, where $dnsrownnzA$/$dnscolnnzA$ is the sum of nonzero entries in all tiles stored in the DnsRow/DnsCol format. (2) the `rowid`/`colid` array of size $numrow$/$numcol$ records the number of these rows/columns of all tiles store in DnsRow/DnsCol, and (3) the `dnsRowPtr`/`dnsColPtr` that saves the memory offsets for the numbers of dense rows/columns in the sparse tiles of size $numtileA + 1$. As shown in Figure 3, the four nonzero entries in the red sparse tile are all in the third rows. Thus it should be saved into the DnsRow format, and row index 3 is recorded in `rowid` of this tile. The pink one which is converted to DnsCol format is similar.

Now the high level storage structure and seven formats are introduced, and the next subsection will explain the warp-level SpMV implementations for the seven formats.

C. Tile-Wise SpMV Algorithms

Figure 4 shows examples of the seven warp-level SpMV algorithms corresponding to the formats. The following will explain these algorithms.

In the warp-level CSR-SpMV algorithm, a 32-thread warp always processes a tile with 16 rows, which means that every two consecutive threads process one row. Before the computations, we load the corresponding segment of 16 entris in vector x into the on-chip scratchpad shared memory for better and controllable data locality. After the calculation of the threads, the partial y are added together. There is an example in the CSR part of Figure 4. We assume that we have 8 threads (t_0–t_7) to process the 4-by-4 tile in the CSR format, and every two consecutive threads process one row. It should be noted that the third row only has one element, so that t_4 can calculate it alone, and t_5 does nothing. On the contrary, the fourth row has three elements, so t_6 needs to process two elements. Then we use `sum` to store the calculation result of every thread and `shuffle` will be used twice to add the results of two adjacent threads and transfer the result to fit `sum`, the operation is shown in the far right of the CSR part. Algorithm 2 shows its pseudocode.

In the warp-level COO-SpMV algorithm, very sparse tiles are calculated. The 32 threads in a warp are assigned to process all the nonzeros, and the resulting partial sums are added together in shared memory by using the `atomicAdd` operation. The COO part in Figure 4 shows an example. In this case, only two elements are in the COO tile, and two threads

Algorithm 2 A pseudocode of warp-level CSR-SpMV.

```
for ti = 0 to 31 in parallel do
    sum ← 0
    ri = ti/2
    vi = ti%2
    for j =csrRowPtr[ri]+vi to csrrowptr[ri+1] do
        csrcol ← csrColIdx[j]
        sum += s_x_warp[csrcol]×csrVal[j]
    end for
    sum += __SHFL_DOWN_SYNC(0xffffffff, sum, 1)
    sum += __SHFL_DOWN_SYNC(0xffffffff, sum, ti)
end for
```

(t_0 and t_1) process them at the same time. Algorithm 3 shows its pseudocode.

Algorithm 3 A pseudocode of warp-level COO-SpMV.

```
for i = 0 to tilennz of the tile in parallel do
    rowidx ←cooRowIdx[i]
    colidx ←cooColIdx[i]
    ATOMICADD(y_rowidx,cooVal[i] ×x_colidx)
end for
```

In the warp-level ELL-SpMV algorithm, a warp of 32 threads is used to process nonzero entries stored in the column-major. Each thread in a half warp of 16 threads is assigned to a row, and the computation completes when the ELL width is reached. The ELL part of Figure 4 shows an example. As the ELL data are stored in the column-major, the four elements are stored continuously, and the memory accesses will be aligned. For the four elements, we assign four threads (t_0–t_3) to process them respectively. For faster memory access to the vector x, the corresponding segment of x is loaded into registers, and accessed through register shuffle instructions. After the calculation, the results of each thread are stored into the corresponding `sum` as the final result of this tile. Algorithm 4 shows its pseudocode.

Algorithm 4 A pseudocode of warp-level ELL-SpMV.

```
for ti = 0 to 31 in parallel do
    sum ← 0
    elllen ← tilewidth × 16
    for j = ti to elllen do
        of ellcol ← ellIdx[j]
        x_gathered ← __SHFL_SYNC(0x0000ffff,x_ti, ellcol)
        sum += ellVal[j] ×x_gathered
    end for
end for
```

In the warp-level HYB-SpMV algorithm, two steps respectively calculating the ELL and COO parts are used. The purple part in Figure 4 explains the two steps specifically. The HYB tile is consisted by ELL part and COO part, so in the first step, four threads (t_0–t_3) process the four elements stored in ELL data and in the second step, two threads (t_0 and t_1) process the two elements stored in the COO data. Similar to the ELL-SpMV, the vector x and calculation processing are loaded into the register in advance.

In the warp-level Dns-SpMV algorithm, all elements of the tile are involved in the computation. A 32-thread warp need to process a dense tile of 16-by-16 and finish the work after eight rounds, and each thread processes 8 elements. After calculating, results are stored to `sum` for each thread and `shuffle` will be used to add the `sum` value of threads processing the same row. As can be seen in the Dns part of Figure 4, we assume there are eight threads (t_0–t_7) to calculate the gray tile of 4-by-4. In the first round, the eight threads process the elements in the first and second columns. In the second round, they process the elements in the last two columns. The gray array in the Dns part shows the detail of each thread work.

In the warp-level DnsCol-SpMV algorithm, the task assignment for threads is similar to the Dns-SpMV. The DnsCol part of Figure 4 shows an example. The usable elements of vector x are in the register now. Since there are four elements in the third column, the threads (t_0–t_3) process them independently but reuse the same entry in x in the register, as shown in the pink arrays.

In the warp-level DnsRow-SpMV algorithm, the reduction-sum operation is required, and the corresponding x should be loaded into the registers. In the DnsRow part in Figure 4, the elements are distributed in the third row. Four threads (t_0–t_3) will process each element and the results of the four threads will be added into a right result by reduction-sum implemented using `shuffle` as shown in the DnsRow part.

D. Format Selection Method

On top of the storage structure and basic warp-level SpMV kernels, we implement three TileSpMV algorithms to validate the effectiveness of utilizing the variety of formats: (1) the `TileSpMV_CSR` method that always stores all the sparse tiles with the CSR formats. (2) the `TileSpMV_ADPT` method that first inspects the sparsity structure of each tile and adaptively selects a format from the seven to store and calculate the tile, and (3) the `TileSpMV_DeferredCOO` method that defers the computations of the nonzeros should be stored in the COO form (i.e., in the COO format or in the COO part of the HYB format) by extracting them into a separate matrix and computing its own SpMV. This operation is like the HYB-SpMV that computes an ELL-SpMV and a CSR/COO-SpMV.

Because storing all tiles in the CSR format and computing CSR-SpMV is simple, we do not introduce the `TileSpMV_CSR` method in detail. As for the `TileSpMV_ADPT` method, we plot it in Figure 5 and construct the following steps:

For very sparse tiles, such as the tiles in which the number of nonzero entries is less than 12 and the nonzero entries are distributed not evenly among the rows, the COO format undoubtedly occupies the least memory space, and thus is selected.

When a sparse tile contains no less than 128 nonzeros, it is saved in the Dns format, i.e., in a pure dense pattern, and only values are recorded.

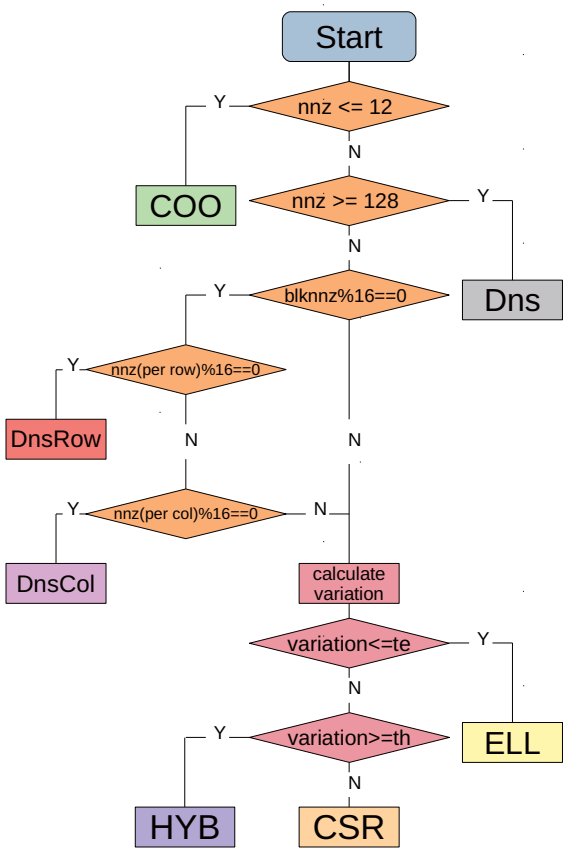


Fig. 5. The flow chart of our format selection method for `TileSpMV_ADPT`.

If all nonzeros of a tile are in certain rows/columns, and all the other rows/columns are empty, we select the DnsRow/DnsCol format to store the nonzeros of the tile.

After the above three steps, the selection of the first four formats (i.e., COO, Dns, DnsRow, DnsCol) is completed. For those tiles do not meet the above criteria, we use nonzero entries distribution as the second group of rules to select ELL, HYB or CSR format for them. We now set a parameter $variation$ (i.e., the ratio of standard deviation to average row length) to evaluate the balance of nonzero distribution. When the variation value goes down, the number of nonzero elements between rows is more balanced. For distinguishing between different format with individual nonzero entries distribution, we set two thresholds te and th to split variation range into three intervals, and each interval corresponds to a format.

The ELL format is selected when $variation$ stays between zero and te, meaning that the number of nonzero entries in rows are relatively balanced, so we can select ELL format to achieve better space cost.

When the variation is greater than th, meaning that the distribution of the nonzero entries is more irregular, the HYB format consisting ELL and COO parts is selected for possibly better efficiency.

The remaining sparse tiles will be stored in the CSR format, since $variation$ between te and th indicates that general pattern should give the best performance.

We in our method experimentally set te and th to 0.2 and 1.0, respectively, since we found the two thresholds in general give us the best performance.

Besides the SpMV implementations of the tile-wise formats, improving load balancing should also be considered. Although using the sparse tiles of a fixed size as the basic working unit already can naturally avoid load imbalance to some extent, we also need to split very long tile rows into small pieces for more even workload. In our implementation, we add a parameter named *tbalance* (always set to 8 in our code) and let a warp process no more than *tbalance* tiles. If the number of sparse tiles in one tile row is greater than *tbalance*, we divide the tile row and use multiple warps to deal with it together. Finally, the partial y generated by the warps belong to the same tile row are added by atomic addition. In this way, we can ensure that each warp has similar tasks to improve load balancing.

Moreover, even though the above selection and load balancing methods can achieve good performance for most matrices, the SpMV performance of very sparse matrices from graph problems maybe still unsatisfactory. Their most obvious structure is that COO tiles dominate the nonzero count. This fact motivates us to develop the third selection method called `TileSpMV_deferredCOO`. In this method, the tiles with COO data (including all tiles in the COO format and the COO part of the HYB format) are extracted to form a separate matrix stored in a normal CSR format and computed by the CSR5-SpMV method. That is to say, in the SpMV computation, two matrices will be calculated for together generating the final resulting vector y.

IV. Experimental Results

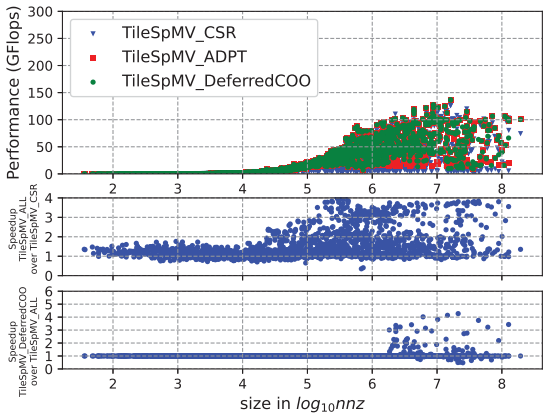
A. Experimental Setup

Our experimental platform includes two NVIDIA GPUs: a Geforce Titan RTX (Turing architecture) and an A100 (Ampere architecture). The GPU driver version is 455.23.05, and the CUDA version is 11.1.

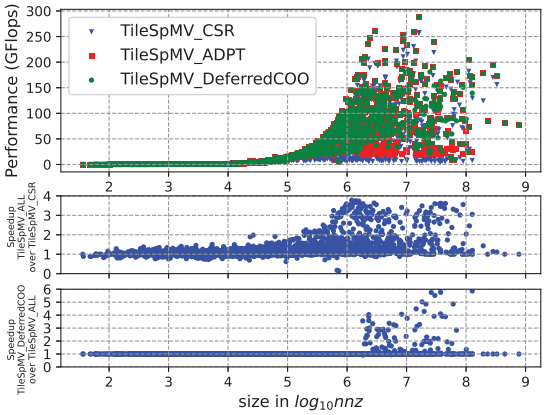
We compare our TileSpMV work with the latest cuSPARSE v11.1 kernel `cusparse?bsrmv()` using the BSR format, the Merge-SpMV algorithm[1] proposed by Merrill and Garland [32], and the CSR5-SpMV algorithm proposed by Liu and Vinter [27]. The specifications of the GPUs and the algorithms tested are listed in Table I. Besides, our experiments did not test several other open-source SpMV algorithms such as yaSpMV [21], HolaSpMV [36] and CSR-Adaptive [24], [30], since we tried our best to build them but still cannot let them run in the CUDA v11.1 environment and the newest GPUs.

The test dataset includes all 2757 sparse matrices in the SuiteSparse Matrix Collection [46].

[1]It is worth to note that the `cusparseSpMV()` using the CSR format with argument `CUSPARSE_CSRMV_ALG2` in cuSPARSE v11.1 is an official implementation of the Merge-SpMV work [32], but in most cases of our test delivers slower performance than the open-source implementation of Merge-SpMV. Thus we in this paper compare our work with the original Merge-SpMV open-source code.



(a) Double precision TileSpMV performance and speedups on Titan RTX



(b) Double precision TileSpMV performance and speedups on A100

Fig. 6. The two sub-figures on top show performance (in GFlops) of the basic `TileSpMV_CSR` method and the two optimization methods `TileSpMV_ADPT` and `TileSpMV_DeferredCOO` on the two GPUs. The four sub-figures on bottom respectively show the speedup of `TileSpMV_ADPT` to `TileSpMV_CSR` and the speedup of `TileSpMV_DeferredCOO` to `TileSpMV_ADPT`.

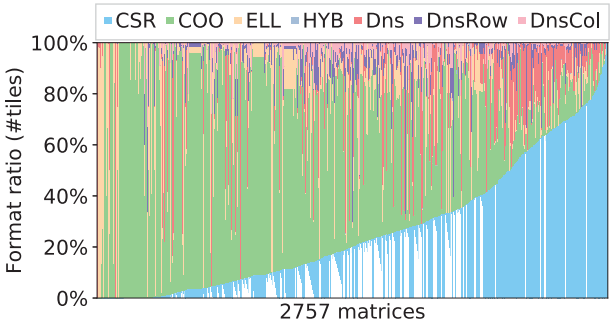
TABLE I
THE TWO GPUS AND FOUR ALGORITHMS EVALUATED.

Two NVIDIA GPUs	Four algorithms
(1) Titan RTX (Turing), 4608 CUDA cores @ 1770 MHz, 24 GB, B/W 672 GB/s	(1) cuSPARSE v11.1 BSR
(2) A100 (Ampere), 6912 CUDA cores @ 1410 MHz, 40 GB, B/W 1555 GB/s	(2) Merge-SpMV [32]
	(3) CSR5 [27]
	(4) TileSpMV (this work)

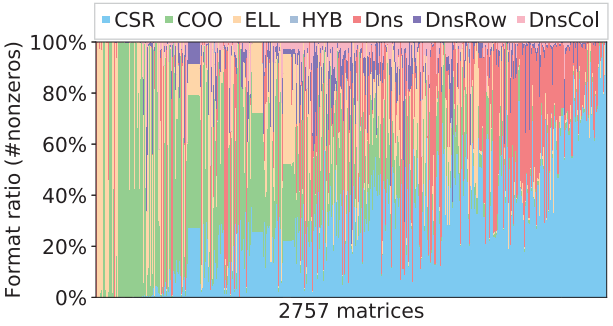
B. *Effectiveness of Adaptive Format Selection*

In the TileSpMV algorithm, the selection of formats and corresponding methods gives significant performance gain. To show its effectiveness, by benchmarking the 2757 matrices, we plot the ratio of the number of tiles in different formats to the total tiles, and the ratio of the number of nonzeros in different formats to the total nonzeros in Figures 7(a) and (b), respectively. In the figures, different color bars correspond to different formats. As can be seen, the green bars (representing the COO format) take up the largest area of the tile formats. Also, in Figure 7(b), even though there are many tiles in the COO format, the nonzero ratios of the COO format are not that high compared to the format ratio, because of the low density of the COO tiles.

We on the two GPUs test `TileSpMV_CSR` in which each tile is originally stored in the CSR format, `TileSpMV_ADPT` with adaptive format selection for each tile, and `TileSpMV_DeferredCOO` which chooses whether to split the COO tiles into a new matrix. The performance and speedup are shown in Figure 6. It can be seen that after the format selection, the performance of `TileSpMV_ADPT` can be up to 6.75x faster than `TileSpMV_CSR`, and the advantage becomes more obvious as the size of matrices increase. Moreover, when the size of matrices is relatively small, we still choose `TileSpMV_ADPT`. But when the size is larger than a certain size (1.8M in our work), the advantage of `TileSpMV_DeferredCOO` begins to become prominent. It further achieves up to 7.02x speedup over `TileSpMV_ADPT`, which indicates that our optimizations are quite effective.

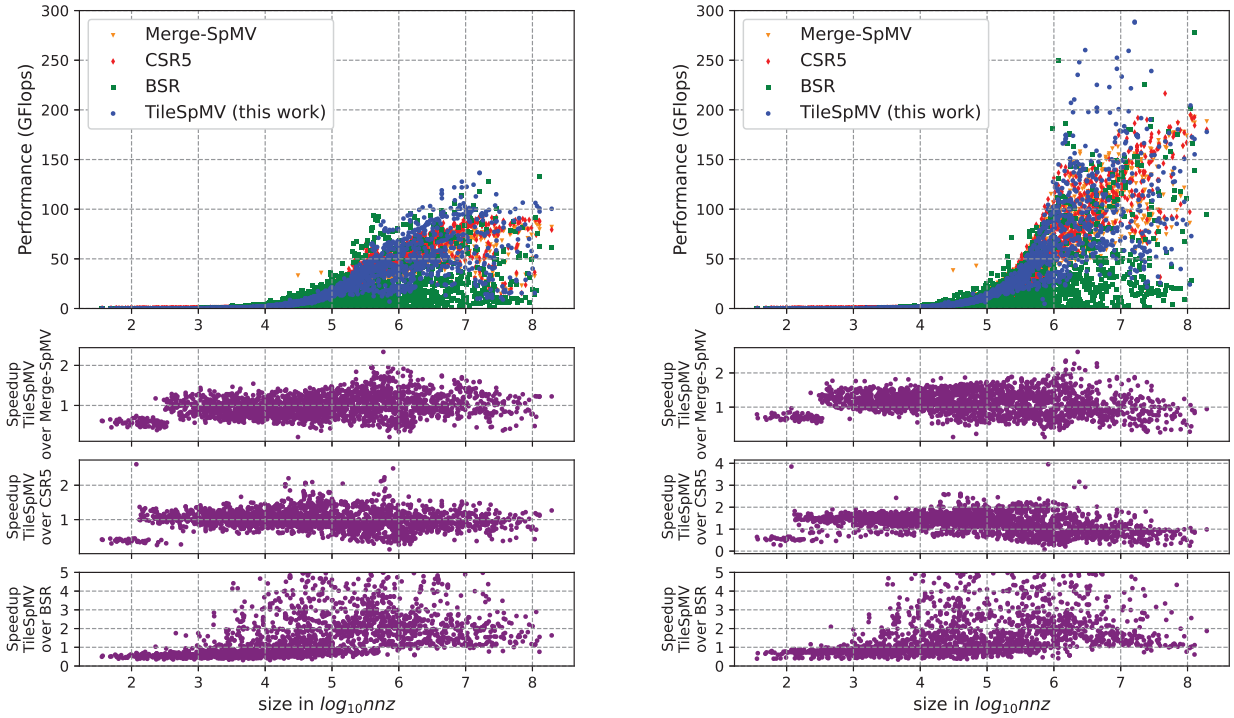


(a) The ratio of tiles in different formats



(b) The ratio of nonzero entries in different formats

Fig. 7. Two figures show the ratio of the number of tiles and nonzeros in different formats to the total tiles and nonzeros, respectively.



(a) Double precision SpMV performance and speedups on Titan RTX.

(b) Double precision SpMV performance and speedups on A100.

Fig. 8. The two sub-figures on top show performance (in GFlops) of the four SpMV methods on two GPUs. The six sub-figures on bottom show the speedups of our TileSpMV over the Merge-SpMV, CSR5 and BSR.

C. Performance Comparison over Existing SpMV Work

We compare our `TileSpMW_DeferredCOO` algorithm with Merge-SpMV, CSR5 and BSR work, and the performance comparison of the four methods on the two GPUs is shown in Figure 8. As can be seen, our method shows the best performance for most matrices on both Titan RTX and A100. Specifically, compared with the three methods, our method is faster than Merge-SpMV on 1813 matrices, faster than CSR5 on 2040 matrices, and faster than BSR on 1638 matrices, and achieves up to 2.61x, 3.96x and 426.59x speedups over them. The best speedups occur in matrices 'exdata_1', 'rel8' and 'lp_osa_60', respectively. Since the nonzero entries are concentrated in a certain area in 'exdata_1', the proportion of the Dns tiles in it has exceeded 80%. Thus it is more efficient to use the dense computation in our method than the sparse for these tiles. The matrix 'rel8' has a large number of tiles in the COO format extracted to the CSR5 part, and using the best combination of CSR5 and TileSpMV gives the matrix good performance. As for 'lp_osa_60', the lack of small dense structures makes BSR less efficient. Overall, it can be seen that our method has obvious greater advantages for many matrices, and our highest performance reaches nearly 300 GFlops (on matrix 'TSOPF_RS_b2383'). In contrast, the highest performance of Merge-SpMV does not exceed 200 GFlops, and CSR5 is merely around 210 GFlops.

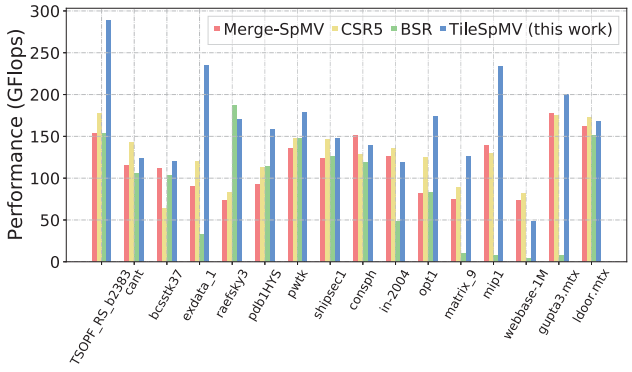


Fig. 9. Performance comparison of 16 representative matrices on A100 GPU.

To conduct a more detailed analysis, we list the performance comparison of 16 representative matrices (see Table II) on A100 in Figure 9. As can be seen, the matrix that achieves the highest performance in our method is 'TSOPF_RS_b2383',

TABLE II
INFORMATION OF THE 16 REPRESENTATIVE MATRICES.

Matrix	Plot	Size	nnz
TSOPF_RS_b2383		38K×38K	16.1M
cant		62K×62K	4M
bcsstk37		25K×25K	1,1M
exdata_1		6K×6K	2.2M
raefsky3		21K×21K	1.4M
pdb1HYS		36K×36K	4.3M
pwtk		217K×217K	11.5M
shipsec1		140K×140K	3.5M
consph		83K×83K	6M
in-2004		1.4M×1.4M	16.9M
opt1		15K×15K	1.9M
matrix_9		103K×103K	1.2M
mip1		66K×66K	10.4M
webbase-1M		1M×1M	3.1M
gupta3		16.8K×16.8K	9.3M
ldoor		952K×952K	42.5M

which can reach 288 GFlops and is 1.88x and 1.63x faster than Merge-SpMV and CSR5, respectively. This is because that Dns tiles occupy a large portion of all tiles. Besides, there are also 905 DnsRow tiles and 2885 DnsCol tiles in it. Through the analysis of these matrix structure, our method in general has performance advantage to handle the matrices with a large proportion of Dns or DnsRow/DnsCol tiles. But actually, for matrices with a moderate number of CSR and COO tiles, through our optimized algorithm, we can also achieve comparable performance to Merge-SpMV and CSR5. The matrix 'cant' is an example.

D. Space Cost Comparison

Figure 10 shows the space costs of the standard CSR format, our `TileSpMV_CSR` and the further optimized `TileSpMV_ADPT`. In order to get a clearer comparison, we use the largest 150 matrices in the dataset. As we can see, compared with the standard CSR format (red line), our `TileSpMV_CSR` (blue line) basically occupies the same or less memory space on most matrices. But for a small number of matrices, there are obvious higher memory consumption, this is because the tiles of these matrices are very sparse and a complete row pointer array of each tile is still allocated. Moreover, in `TileSpMV_ADPT` (green line), the memory footprint is overall improved, although some cases still occupy more space than the CSR format.

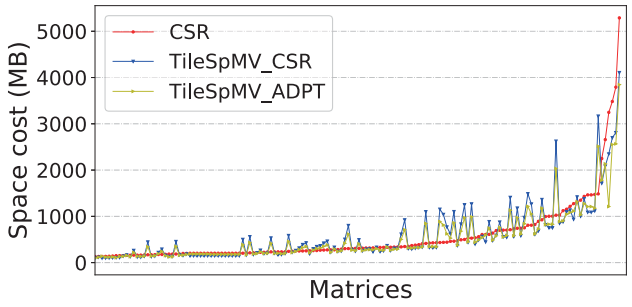


Fig. 10. A space cost comparison of the standard CSR format, `TileSpMV_CSR` and `TileSpMV_ADPT`.

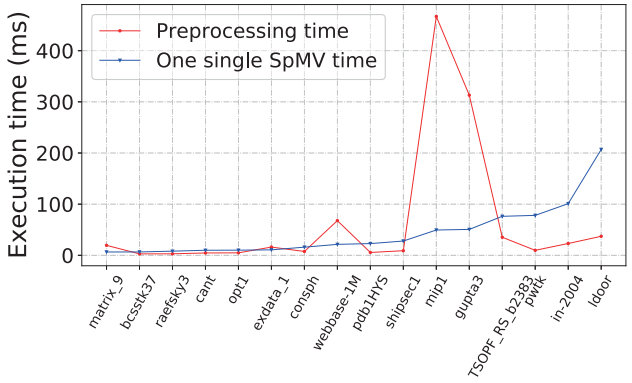


Fig. 11. Comparison of preprocessing time and a serial single SpMV time of the 16 representative matrices.

E. Preprocessing Overhead Analysis

We also record the preprocessing overhead of converting a basic CSR matrix to our tile form in the TileSpMV algorithm. Figure 11 shows an execution time comparison of the preprocessing and a serial single SpMV time on CPU. As can be seen, the preprocessing can take less than 10x more cost than a single SpMV (matrix 'mip1'), and can be even faster than an SpMV (e.g., matrix 'ldoor'), demonstrating that the overhead depends on the sparsity structure and the several adaptive optimizations for the matrix.

V. RELATED WORK

Accelerating SpMV through utilizing **small block structures** has been studied from various angles. Im et al. [7], [47], [48] and Vuduc et al. [8], [9], [49], [50] proposed a series of register level, memory hierarchy and auto-tuning optimizations for small dense blocks and developed the SPARSITY and OSKI packages. Buluç et al. [13], [17], [52], [53] proposed the CSB and DCSC formats that keep the sparsity structures in small sparse blocks and consider hyper sparse cases. Moreover, Buttari et al. [54] designed the BCSR format, and Martone [26] improved the CSB in a recursive formulation. Also, Yzelman and Bisseling [14], [18], [22] developed cache-oblivious methods for multi-cores. On GPUs, Choi et al. [15] modeled SpMV with blocked formats, and Yan et al. [21] developed the BCCOO format that stores dense 2D blocks.

Wang et al. [55] and Lu et al. [56] used block formats for sparse triangular solve kernel of Sunway processors and GPUs, respectively. Compared to the above work, the TileSpMV proposed in this paper divides a sparse matrix into sparse tiles of medium size and focuses performance optimizations on GPUs.

Designing **new formats and algorithms** is the most widely used method for parallel SpMV research. A number of new formats have been developed on top of the basic ELL [6] and CSR formats. Bell and Garland developed the HYB [12] format consisting of both ELL and CSR/COO parts, and Su and Keutzer [19] proposed the clSpMV framework that includes more formats. Other ELL variants have been proposed by Kreutzer et al [23], Liu et al. [20], Liang et al. [38], Ashari et al. [28], Anzt et al. [42], [57] and Xie et al. [44]. Some variants of the CSR format, such as CSX [16], ACSR [45], CSR-Adaptive [24], [30], CSR5 [27], Merge-SpMV [32] and HolaSpMV [36], also demonstrated low preprocessing cost and fast SpMV performance on GPUs. Liu and Vinter [58] developed a new speculative segmented sum primitive for SpMV. Yesil et al [59] split the input matrix into a dense and a sparse portion and stored the dense part in a new representation for better data locality. Elafrou et al [60] break the rows into multiple phases for conflict-free parallel execution. In contrast, our TileSpMV work considers seven formats including CSR, COO, ELL and HYB, and also proposes a method to adaptively select them.

Because there is no one single format can deliver the best SpMV performance for all kinds of sparse matrices, **machine learning techniques** are used for selecting the best format and SpMV method for a given matrix. Various machine learning tools have been used by Li et al. [41], Sedaghati et al. [29], Benatia et al. [33] and Tan et al. [61]. Recently Zhao et al. [1], [40] proposed new CNN and deep learning techniques and took preprocessing stage into consideration. Xie et al. [62] proposed MatNet for matrix structure analysis. Also, Guo and Lee [25] and Lehnert et al. [34] proposed modeling techniques for predicting SpMV performance on various platforms. In this work, we use a simple but effective heuristics method for selecting the best format and SpMV implementation for each sparse tile, and receive obvious performance gain.

Besides the research work listed above, a number of **performance evaluations and surveys** give valuable overview of parallel SpMV. Williams et al. [11] studied several key techniques for optimizing SpMV on multi-core CPUs. Goumas et al. [10], Elafrou et al. [37], [39] and Filippone et al. [35] evaluated the performance of SpMV on various CPU and GPU platforms. Langr and Tvrdík [31] proposed a group of evaluation criteria for sparse matrix formats. Li et al. [63], Tsai et al. [64], [65], Zhang et al. [66] evaluated sparse kernels on the latest CPUs, GPUs and APUs, respectively.

VI. Conclusion

In this work, we have proposed a tiled algorithm called TileSpMV for accelerating SpMV on GPUs through exploiting 2D spatial structures of sparse matrices. The algorithm optimized warp-level tile-wise SpMV and adaptively selects the best format and SpMV algorithm for each tile. The experimental results from testing the 2757 matrices in the SuiteSparse Matrix Collection show that our method is faster than Merge-SpMV on 1813 matrices, faster than CSR5 on 2040 matrices, and faster than BSR on 1638 matrices, and achieves up to 2.61x, 3.96x and 426.59x speedups over them, respectively.

Acknowledgement

We deeply appreciate the invaluable comments from all the reviewers. Zhou Jin is the corresponding author of this paper. This research was supported by the Science Challenge Project under Grant No. TZZT2016002, the National Natural Science Foundation of China under Grant No. 61972415, 61972377, 62032023, and the Science Foundation of China University of Petroleum, Beijing under Grant No. 2462019YJRC004, 2462020XKJS03, 2462020YXZZ024.

References

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

[2] A. Buluç and J. R. Gilbert, "The combinatorial blas: design, implementation, and applications," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011.

[3] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, C. Yang, J. D. Owens, M. Zalewski, T. Mattson, and J. Moreira, "Mathematical foundations of the graphblas," in *HPEC '16*, 2016, pp. 1–9.

[4] T. G. Mattson, C. Yang, S. McMillan, A. Buluç, and J. E. Moreira, "Graphblas c api: Ideas for future versions of the specification," in *HPEC '17*, 2017, pp. 1–6.

[5] C. Yang, A. Buluç, and J. D. Owens, "Implementing push-pull efficiently in graphblas," in *ICPP '18*, 2018, pp. 89:1–89:11.

[6] J. R. Rice and R. F. Boisvert, *Solving Elliptic Problems Using ELLPACK*. Springer-Verlag New York, Inc., 1984.

[7] E. Im, "Optimizing the performance of sparse matrix-vector multiplication," Ph.D. dissertation, University of California, Berkeley, 2000.

[8] V. R., "Automatic performance tuning of sparse matrix kernels," Ph.D. dissertation, University of California, Berkeley, 2003.

[9] R. Vuduc, J. Demmel, and K. Yelick, "Oski: A library of automatically tuned sparse matrix kernels," *Journal of Physics: Conference Series*, vol. 16, no. 1, p. 521, 2005.

[10] 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*, vol. 50, no. 1, pp. 36–77, 2009.

[11] 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*, vol. 35, no. 3, pp. 178–194, 2009.

[12] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *SC '09*, 2009, pp. 18:1–18:11.

[13] 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, pp. 233–244.

[14] A. N. Yzelman and R. H. Bisseling, "Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods," *SIAM Journal on Scientific Computing*, vol. 31, no. 4, pp. 3128–3154, 2009.

[15] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on gpus," in *PPoPP '10*, 2010, pp. 115–126.

[16] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris, "Csx: An extended compression format for spmv on shared memory systems," in *PPoPP '11*, 2011, pp. 247–256.

[17] A. Buluç, S. Williams, L. Oliker, and J. Demmel, "Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication," in *IPDPS '11*, 2011, pp. 721–733.

[18] A. N. Yzelman and R. H. Bisseling, "Two-dimensional cache-oblivious sparse matrix–vector multiplication," *Parallel Computing*, vol. 37, no. 12, pp. 806–819, 2011.

[19] B. Su and K. Keutzer, "clspmv: A cross-platform opencl spmv framework on gpus," in *ICS '12*, 2012, pp. 353–364.

[20] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *ICS '13*, 2013, pp. 273–282.

[21] S. Yan, C. Li, Y. Zhang, and H. Zhou, "yaspmv: Yet another spmv framework on gpus," in *PPoPP '14*, 2014, pp. 107–118.

[22] A. N. Yzelman and D. Roose, "High-level strategies for parallel shared-memory sparse matrix-vector multiplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 1, pp. 116–125, 2014.

[23] 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*, vol. 36, no. 5, pp. C401–C423, 2014.

[24] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on gpus using the csr storage format," in *SC '14*, 2014, pp. 769–780.

[25] P. Guo, L. Wang, and P. Chen, "A performance modeling and optimization analysis tool for sparse matrix-vector multiplication on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 5, pp. 1112–1123, 2014.

[26] M. Martone, "Efficient multithreaded untransposed, transposed or symmetric sparse matrix-vector multiplication with the recursive sparse blocks format," *Parallel Computing*, vol. 40, no. 7, pp. 251 – 270, 2014.

[27] W. Liu and B. Vinter, "Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *ICS '15*, 2015, pp. 339–350.

[28] 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*, vol. 76, pp. 3–15, 2015.

[29] N. Sedaghati, T. Mu, L. Pouchet, S. Parthasarathy, and P. Sadayappan, "Automatic selection of sparse matrix representation on gpus," in *ICS '15*, 2015, pp. 99–108.

[30] M. Daga and J. L. Greathouse, "Structural agnostic spmv: Adapting csr-adaptive for irregular matrices," in *HiPC '15*, 2015, pp. 64–74.

[31] D. Langr and P. Tvrdík, "Evaluation criteria for sparse matrix storage formats," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 2, pp. 428–440, 2016.

[32] D. Merrill and M. Garland, "Merge-based parallel sparse matrix-vector multiplication," in *SC '16*, 2016, pp. 678–689.

[33] A. Benatia, W. Ji, Y. Wang, and F. Shi, "Sparse matrix format selection with multiclass svm for spmv on gpu," in *ICPP '16*, 2016, pp. 496–505.

[34] C. Lehnert, R. Berrendorf, J. P. Ecker, and F. Mannuss, "Performance prediction and ranking of spmv kernels on gpu architectures," in *Euro-Par 2016: Parallel Processing*, 2016, pp. 90–102.

[35] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, "Sparse matrix-vector multiplication on gpgpus," *ACM Trans. Math. Softw.*, vol. 43, no. 4, pp. 30:1–30:49, 2017.

[36] M. Steinberger, R. Zayer, and H. Seidel, "Globally homogeneous, locally adaptive sparse matrix-vector multiplication on the gpu," in *ICS '17*, 2017, pp. 13:1–13:11.

[37] 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, pp. 292–301.

[38] Y. Liang, W. T. Tang, R. Zhao, M. Lu, H. P. Huynh, and R. S. M. Goh, "Scale-free sparse matrix-vector multiplication on many-core architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 12, pp. 2106–2119, 2017.

[39] 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 Trans. Math. Softw.*, vol. 44, no. 3, pp. 26:1–26:32, 2018.

[40] Y. Zhao, J. Li, C. Liao, and X. Shen, "Bridging the gap between deep learning and sparse matrix format selection," in *PPoPP '18*, 2018, pp. 94–108.

[41] J. Li, G. Tan, M. Chen, and N. Sun, "Smat: An input adaptive auto-tuner for sparse matrix-vector multiplication," in *PLDI '13*, 2013, pp. 117–126.

[42] 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 Trans. Parallel Comput.*, vol. 7, no. 1, 2020.

[43] W. Liu, "Parallel and scalable sparse basic linear algebra subprograms," Ph.D. dissertation, University of Copenhagen, 2015.

[44] 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, pp. 149–162.

[45] 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, pp. 781–792.

[46] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, 2011.

[47] E.-J. Im and K. Yelick, "Optimizing sparse matrix computations for register reuse in sparsity," in *ICCS '01*, 2001, pp. 127–136.

[48] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *The International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 135–158, 2004.

[49] 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, pp. 26–26.

[50] R. Vuduc, A. Gyulassy, J. W. Demmel, and K. A. Yelick, "Memory hierarchy optimizations and performance bounds for sparse atax," in *ICCS '03*, 2003, pp. 705–714.

[51] B. C. Lee, R. W. Vuduc, J. W. Demmel, and K. A. Yelick, "Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply," in *ICPP '04*, 2004, pp. 169–176 vol.1.

[52] A. Buluç and J. R. Gilbert, "On the representation and multiplication of hypersparse matrices," in *IPDPS '08*, 2008, pp. 1–11.

[53] ——, "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C170–C191, 2012.

[54] 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*, vol. 21, no. 4, pp. 467–484, 2007.

[55] X. Wang, W. Liu, W. Xue, and L. Wu, "swsptrsv: A fast sparse triangular solve with sparse level tile layout on sunway architectures," in *PPoPP '18*, 2018, pp. 338–353.

[56] Z. Lu, Y. Niu, and W. Liu, "Efficient block algorithms for parallel sparse triangular solve," in *ICPP '20*, 2020.

[57] 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," 2020.

[58] W. Liu and B. Vinter, "Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors," *Parallel Computing*, vol. 49, no. C, pp. 179–193, 2015.

[59] S. Yesil, A. Heidarshenas, A. Morrison, and J. Torrellas, "Speeding up spmv for power-law graph analytics by enhancing locality amp; vectorization," in *SC '20*, 2020.

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

[61] G. Tan, J. Liu, and J. Li, "Design and implementation of adaptive spmv library for multicore and many-core architecture," *ACM Trans. Math. Softw.*, vol. 44, no. 4, 2018.

[62] Z. Xie, G. Tan, W. Liu, and N. Sun, "Ia-spgemm: An input-aware auto-tuning framework for parallel sparse matrix-matrix multiplication," in *ICS '19*, 2019, p. 94–105.

[63] A. Li, W. Liu, M. R. B. Kristensen, B. Vinter, H. Wang, K. Hou, A. Marquez, and S. L. Song, "Exploring and analyzing the real impact of modern on-package memory on hpc scientific kernels," in *SC '17*, 2017, pp. 26:1–26:14.

[64] Y. M. Tsai, T. Cojean, and H. Anzt, "Sparse linear algebra on amd and nvidia gpus – the race is on," in *ISC '20*, 2020, pp. 309–327.

[65] ——, "Evaluating the performance of nvidia's a100 ampere gpu for sparse linear algebra computations," 2020.

[66] F. Zhang, W. Liu, N. Feng, J. Zhai, and X. Du, "Performance evaluation and analysis of sparse matrix and graph kernels on heterogeneous processors," *CCF Transactions on High Performance Computing*, p. 131–143, 2019.