

# yaSpMV: Yet Another SpMV Framework on GPUs

Shengen Yan

Institute of Software, Chinese  
Academy of Sciences  
University of Chinese Academy  
of Sciences Beijing, China  
North Carolina State University  
Raleigh, NC  
yanshengen@gmail.com

Chao Li

North Carolina State University  
Raleigh, NC  
cli17@ncsu.edu

Yunquan Zhang

State Key Lab of Computer  
Architecture, Institute of  
Computing Technology, Chinese  
Academy of Sciences  
Institute of Software, Chinese  
Academy of Sciences  
Beijing, China  
zyq@ict.ac.cn

Huiyang Zhou

North Carolina State University  
Raleigh, NC  
hzhou@ncsu.edu

## Abstract

SpMV is a key linear algebra algorithm and has been widely used in many important application domains. As a result, numerous attempts have been made to optimize SpMV on GPUs to leverage their massive computational throughput. Although the previous work has shown impressive progress, load imbalance and high memory bandwidth remain the critical performance bottlenecks for SpMV. In this paper, we present our novel solutions to these problems. First, we devise a new SpMV format, called blocked compressed common coordinate (BCCOO), which uses bit flags to store the row indices in a blocked common coordinate (COO) format so as to alleviate the bandwidth problem. We further improve this format by partitioning the matrix into vertical slices to enhance the cache hit rates when accessing the vector to be multiplied. Second, we revisit the segmented scan approach for SpMV to address the load imbalance problem. We propose a highly efficient matrix-based segmented sum/scan for SpMV and further improve it by eliminating global synchronization. Then, we introduce an auto-tuning framework to choose optimization parameters based on the characteristics of input sparse matrices and target hardware platforms. Our experimental results on GTX680 GPUs and GTX480 GPUs show that our proposed framework achieves significant performance improvement over the vendor tuned CUSPARSE V5.0 (up to 229% and 65% on average on GTX680 GPUs, up to 150% and 42% on average on GTX480 GPUs) and some most recently proposed schemes (e.g., up to 195% and 70% on average over clSpMV on GTX680 GPUs, up to 162% and 40% on average over clSpMV on GTX480 GPUs).

**Categories and Subject Descriptors** D.1.3 [Concurrent Programming]: Parallel programming

**Keywords** SpMV, Segmented Scan, BCCOO, OpenCL, CUDA, GPU, Parallel algorithms

## 1. Introduction

Sparse matrix vector multiplication (SpMV) is a key linear algebra algorithm and is heavily used in many important application domains. As state-of-art many-core GPUs feature remarkably high computational throughput and memory access bandwidth, there has been strong interest in GPU-accelerated SpMV [1][6][7][12][14][15][16][17][21].

Although the sequential implementation of SpMV is fairly straightforward, its parallel implementation is challenging for two main reasons. First, the row-based parallelization, i.e., assigning one thread to compute the dot-product between one row of the matrix and the multiplied vector, although making logical sense, suffers from the load imbalance problem as non-zeros in a matrix may not be evenly distributed across different rows. Such a load imbalance problem is more severe in GPU architectures since the threads in a warp operate in the single-instruction multiple-data (SIMD) manner. Load imbalance among threads in a warp will result in control divergence and the execution time of all the threads in a warp will be forced to be equal to the longest running one. Second, SpMV puts high pressure on the memory hierarchy. The matrix data have low reuse as each non-zero element is only used once for computing the corresponding dot product. On the other hand, although the multiplied vector is reused as each non-empty row of the matrix will use it to compute a dot-product, the access pattern is irregular due to irregular locations of non-zeros in different rows. Such irregular accesses do not meet the GPU memory coalescing requirement, which means that different threads in a warp need to access the data in the same block, to achieve high memory access bandwidth.

Many approaches have been proposed to optimizing SpMV on multi-core CPUs and many-core GPUs. To reduce the memory footprint of sparse matrices, different formats have been proposed to leverage different characteristics of sparse matrices. It has been shown in [16] that among the existing formats, no single format can achieve the best performance and a cocktail format is proposed to combine the strengths of existing formats by partitioning a sparse matrix and applying different formats to different partitions. Given the different features of target hardware platforms and different characteristics of sparse matrices, offline auto-tuning or benchmarking is commonly used to improve the

performance. Although previous work has achieved impressive performance improvement for SpMV, the load imbalance problem and the high memory bandwidth requirement remain the fundamental performance bottlenecks for SpMV. In this paper, we propose our novel solution to SpMV.

We first propose a new format for sparse matrices to alleviate the high memory bandwidth requirement of SpMV. Our new format is referred to as blocked compressed common coordinate (BCCOO) as it is built upon the common coordinate (COO) format. The BCCOO format extends the COO format with blocking to reduce the size for both row and column index arrays. Then, it uses bit-flags to drastically reduce the size of the row index array. To improve the cache hit rate for accessing the multiplied vector, we partition a sparse matrix into vertical slices and align the slices in a top-down manner before applying the BCCOO format. Such vertical partition-based BCCOO is referred to as the BCCOO+ format.

To address the load imbalance problem, we revisit the matrix-based segmented scan and design a new highly optimized segmented scan/sum kernel for SpMV. In our approach, each thread processes the same number of consecutive non-zero blocks and it performs *sequential* segmented scans/sums to generate partial sum results. This way, it avoids the workload imbalance problem and reduces the memory requirement on the row information associated with each thread. Then, each workgroup/thread block will run the parallel segmented scan on the last partial sum results computed from each of its threads. When the final dot-product results require accumulating partial sums across multiple workgroups/thread blocks, adjacent synchronization [24] is used to eliminate the overhead of global synchronization.

To further improve the performance of our SpMV kernel, we introduce an auto-tuning framework to explore optimization parameters for different sparse matrices and different platforms. Such optimization parameters include whether to use texture cache for multiplied vector, whether to perform transpose online or offline, the suitable block sizes for our proposed BCCOO/BCCOO+ format, the number of non-zero blocks to be processed by each thread, the number of threads in a workgroup, the size of shared memory (also called local memory in OpenCL [19]) or registers to be used for intermediate partial sums, etc. As these parameters form a large search space, we introduce a set of accelerations to reduce the auto-tuning time to a few seconds.

Our experiments on a set of 20 sparse matrices show that our proposed single format fits nearly all of the sparse matrices under our study. Compared to the vendor-tuned library CUSPARSE V5.0, our proposed scheme achieves performance improvement by up to 150% and 42% on average on GTX480 GPUs, up to 229% and 65% on average

on GTX680 GPUs. Compared to the clSpMV [16], which combines advantages of many existing formats, our proposed scheme achieves a performance gain of up to 162% and 40% on average on GTX480 GPUs, up to 195% and 70% on average on GTX680 GPUs.

The remainder of this paper is organized as follows. Section 2 presents our proposed BCCOO/BCCOO+ format for sparse matrices. Section 3 details our proposed customized matrix-based segmented scan/sum approach for SpMV. Section 4 summarizes our auto-tuning framework. The experimental methodology and the results are discussed in Sections 5 and 6, respectively. Section 7 addresses the related work. Section 8 concludes the paper.

## 2. The Block-based Compressed Common Coordinate (BCCOO) Format

Our proposed block-based compressed common coordinate format builds upon the common coordinate (COO) format. In this section, we first present the COO format as the background and then introduce our BCCOO format and its extension BCCOO+ format. For illustration, we use the matrix in Eq. 1 as a running example.

$$A = \begin{bmatrix} 0 & 0 & a & 0 & 0 & 0 & b & c \\ 0 & 0 & d & e & 0 & 0 & f & 0 \\ 0 & 0 & 0 & 0 & g & h & i & j \\ k & l & 0 & 0 & m & n & o & p \end{bmatrix} \quad \text{Eq. 1}$$

### 2.1 COO Format

The COO format is a widely used format for sparse matrices. It has explicit storage for the column and row indices for all non-zeros in a sparse matrix. For example, the matrix in Eq.1 can be represented with a row index array, a column index array, and a data value array, as shown in Figure 1.

$$\begin{aligned} \text{Row\_index} &= [0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 2 \ 2 \ 2 \ 2 \ 3 \ 3 \ 3 \ 3 \ 3] \\ \text{Col\_index} &= [2 \ 6 \ 7 \ 2 \ 3 \ 6 \ 4 \ 5 \ 6 \ 7 \ 0 \ 1 \ 4 \ 5 \ 6 \ 7] \\ \text{Value} &= [a \ b \ c \ d \ e \ f \ g \ h \ i \ j \ k \ l \ m \ n \ o \ p] \end{aligned}$$

Figure 1. The COO format of matrix A.

The parallelization strategy suitable with COO, as shown in previous work [1], is segmented scan/reduction. As highlighted in [1][16], the advantage of the COO format is that it does not suffer from the load imbalance problem and can achieve consistent performance over different types of sparse matrices. However, the key problem of the COO format is that it needs to explicitly store both the row index and the column index for every non-zero data element. Therefore, it has the worst memory footprint [16].

### 2.2 BCCOO Format

Our proposed BCCOO format extends the COO format in two ways. First, we incorporate the block-based format to the COO format. In block-based formats such as blocked ELLPACK and blocked CSR [7], a non-zero block is stored consecutively. This way, one block of data values will share the same row index and the same column index. Therefore,

the storage overhead of the row index array and the column index array can be significantly reduced. For matrix A in Eq. 1, if a block size of 2x2 is used, the blocked COO (BCOO) format has the index arrays and the data value array shown in Figure 2.

$$\begin{aligned} \text{Row\_index} &= [0 \ 0 \ 1 \ 1 \ 1] \\ \text{Col\_index} &= [1 \ 3 \ 0 \ 2 \ 3] \\ \text{Value} &= \begin{pmatrix} [a \ 0 \ b \ c \ 0 \ 0 \ g \ h \ i \ j] \\ [d \ e \ f \ 0 \ k \ l \ m \ n \ o \ p] \end{pmatrix} \end{aligned}$$

**Figure 2. The blocked COO format of matrix A with the block size of 2x2.**

From Figure 2, we can see that there are 5 non-zero blocks. Both the row index array and the column index array have been reduced significantly. The first non-zero 2x2 block is  $\begin{pmatrix} a & 0 \\ d & e \end{pmatrix}$  and its block-based row index and column index are 0 and 1, respectively. The next non-zero 2x2 block is  $\begin{pmatrix} b & c \\ f & 0 \end{pmatrix}$  and its blocked-based row index and column index are 0 and 3, respectively. Note that in Figure 2, we use two data value arrays rather than a single array in Figure 1. The reason is that for a block size with the height larger than 1, we put different rows in different data value arrays such that both the row index and column index can be used directly to index the data in each of the value arrays. Such data arrangement is also helpful for contiguous memory accesses. The overhead of the BCOO format, which is shared among all block-based formats, is the zeros in the data value array when a non-zero block contains zeros.

$$\begin{aligned} \text{Bit Flag} &= [1 \ 0 \ 1 \ 1 \ 0] \\ \text{Col\_index} &= [1 \ 3 \ 0 \ 2 \ 3] \\ \text{Value} &= \begin{pmatrix} [a \ 0 \ b \ c \ 0 \ 0 \ g \ h \ i \ j] \\ [d \ e \ f \ 0 \ k \ l \ m \ n \ o \ p] \end{pmatrix} \end{aligned}$$

**Figure 3. The BCCOO format of matrix A with the block size of 2x2.**

Our key extension to the COO format is to use a bit flag array to compress the row index array in a lossless manner. The bit flag array can be viewed simply as the result of a difference function being applied to the row index array. For a difference value larger than 1, we replace it with multiple 1s. Then, we flip 1s and 0s such that a bit value of ‘0’ in the bit flag array represents a row stop, i.e., the corresponding value is the last non-zero in a row. A bit value of ‘1’ represents that it is not the last non-zero in a row. The reason for such representation is that when we compute the partial sums for dot-product result, using the value ‘0’ eliminates the condition check on the next non-zero for the end of a row (see Section 3.2). As our bit flag array provides lossless compression on the row index array, the row index information can be reconstructed from the bit flag array by accumulating the number of row stops. We refer to this format as blocked compressed COO (BCCOO). For matrix

A in Eq. 1, the BCCOO format is shown in Figure 3 with the block size of 2x2.

Compared to the BCOO format shown in Figure 2, the column index array and the data value arrays remain the same. The row index array becomes a bit vector of 5 bits. Assuming that integers are used for row indices, a compression ratio of 32 is achieved for the row index array.

In our implementation, in order to remove the control flow to check the end of the bit flag array, we pad it with bit ‘1’ such that the length of the bit flag array is a multiple of the working set (i.e., number of non-zero blocks to be processed) of a workgroup.

Similar to row-index arrays, we can also try to reduce data transmission required for column index arrays using difference functions. In our approach, we first apply a segmented difference function on a column index array with each segment being the working set of each thread. This way, there is no inter-thread dependency when reconstructing the column indices. The resulting difference array is stored using the short data type instead of the regular integer type. If a difference value is beyond the range of a signed short, we replace it with a fixed value -1, which means that the original column index array needs to be accessed for this particular index.

$$B = \begin{bmatrix} 0 & 0 & a & 0 \\ 0 & 0 & d & e \\ 0 & 0 & 0 & 0 \\ k & l & 0 & 0 \\ 0 & 0 & b & c \\ 0 & 0 & f & 0 \\ g & h & i & j \\ m & n & o & p \end{bmatrix}$$

(a)

$$\begin{aligned} \text{Bit Flag} &= [0 \ 0 \ 0 \ 1 \ 0] \\ \text{Col\_index} &= [1 \ 0 \ 3 \ 2 \ 3] \quad (\text{uncompressed}) \\ \text{Value} &= \begin{pmatrix} [a \ 0 \ 0 \ 0 \ b \ c \ g \ h \ i \ j] \\ [d \ e \ k \ l \ f \ 0 \ m \ n \ o \ p] \end{pmatrix} \end{aligned}$$

(b)

**Figure 4. The BCCOO+ format of matrix A in Eq. 1. (a) The vertically sliced and rearranged matrix of matrix A. (b) The bit flag array, the column index array, and the data value arrays.**

### 2.3 BCCOO+ Format

We also propose an extension to our BCCOO format to improve the locality of the accesses to the multiplied vector, referred to as the BCCOO+ format. In this format, we first partition a sparse matrix into vertical slices and then align the slices in a top-down manner. Then, we apply the BCCOO format on the vertically sliced and rearranged matrix with an exception on column indices. The column index array is generated based on the block coordinates in the original matrix rather than the transformed matrix as we need original column indices to locate the corresponding elements in the multiplied vector for dot-product operations.

For matrix A in Eq. 1, the vertically sliced and rearranged matrix becomes matrix B in Figure 4a if the number of slice is 2 and the slice width is 4. The BCCOO+ format of A is shown in Figure 4b when the block size 2x2 is used.

As shown in Figure 4, the bit flag array encodes that there is only one non-zero block in row 0, row 1, and row 2. Row 3, in contrast, contains 2 non-zero blocks. The column indices of these blocks, however, are determined from matrix A rather than matrix B. Taking the 2x2 block  $\begin{pmatrix} g & h \\ m & n \end{pmatrix}$  as an example, it resides at column 2 in matrix A, which is why its column index value is 2 as shown in Figure 4b.

The benefit of BCCOO+ format can be illustrated with matrix-vector multiplication between matrix A and vector y, i.e.,  $A * \vec{y}$ . Different rows in the same vertical slice, e.g., slice 0, will all use  $y[0] \sim y[3]$ . Similarly, all the rows in slice 1 will use  $y[4] \sim y[7]$  to compute the dot-product. As the block  $\begin{pmatrix} g & h \\ m & n \end{pmatrix}$  is in slice 1, it needs to use  $y[4] \sim y[7]$ , with the block size of 2x2, its column index of 2 provides the necessary information for indexing  $y[4]$  and  $y[5]$  from the vector  $\vec{y}$ .

$$A * \vec{y} = \begin{bmatrix} 0 & 0 & a & 0 \\ 0 & 0 & d & e \\ 0 & 0 & 0 & 0 \\ k & l & 0 & 0 \end{bmatrix} * \begin{bmatrix} y[0] \\ y[1] \\ y[2] \\ y[3] \end{bmatrix} + \begin{bmatrix} 0 & 0 & b & c \\ 0 & 0 & f & 0 \\ g & h & i & j \\ m & n & o & p \end{bmatrix} * \begin{bmatrix} y[4] \\ y[5] \\ y[6] \\ y[7] \end{bmatrix}$$

**Figure 5. Matrix-vector multiplication as a sum of the products between its vertical slices and the corresponding vector segments.**

Since the BCCOO+ format breaks the original matrix into slices, after performing the matrix-vector multiplication on each slice, the intermediate results need to be combined to generate the final results. Using our running example of matrix A in Eq. 1, the derivation of  $A * \vec{y}$  is shown in Figure 5. Therefore, when using the BCCOO+ format, it is necessary to use a temporary buffer to store the intermediate results and to invoke an additional kernel to combine them. Depending on the number of slices, the size of the temporary buffer can be large, thereby hurting the performance. As a result, the BCCOO+ format is not always preferred over the BCCOO format and we resort to auto-tuning to determine either the BCCOO or BCCOO+ format should be used.

#### 2.4 Auxiliary Information for SpMV

To facilitate the computation of SpMV, the following information is computed and stored along with the BCCOO/BCCOO+ format. First, based on the number of non-zeros that each thread will process, we compute the location of the first result generated by each thread, i.e., the row index that the result belongs to. Using matrix C in Eq. 2

as an example, in which each element is a block of data. To simplify the discussion, we assume the block size as  $n \times 1$ . As discussed in Section 2.2, for a block size with the height larger than 1, each row will be stored in a separate value array. The BCCOO format of matrix C is shown in Figure 6a. As there are 16 non-zero data blocks, assuming each thread will process 4 non-zero blocks, we will compute the row index that the first result generated by each thread belongs to. Such information can be computed with a scan operation on the bitwise inverse of the bit flag array in the BCCOO format. In this example, thread 0 processes the first 4 non-zero data blocks A', B', C', and D' and its first computation result, i.e.,  $A' * y$ , is part of the final result for the dot-product between row 0 and the multiplied vector. So, the result entry is set to 0. Similarly, thread 1 processes the next four non-zero blocks E', F', G', and H'. As block E' still belongs to row 0, the entry for the first result of thread 1 is set as 0.

$$C = \begin{bmatrix} A' & 0 & B' & 0 & C' & 0 & D' & E' \\ 0 & 0 & 0 & F' & 0 & 0 & G' & 0 \\ 0 & H' & 0 & I' & 0 & J' & 0 & 0 \\ 0 & K' & L' & M' & 0 & N' & O' & P' \end{bmatrix} \quad \text{Eq.2}$$

Bit Flag = [1 1 1 1 0 1 0 1 1 0 1 1 1 1 1 0]

Col\_index = [0 2 4 6 7 3 6 1 3 5 1 2 3 5 6 7] (uncompressed)

Value =

[A' B' C' D' E' F' G' H' I' J' K' L' M' N' O' P']

(a)

Bit Flag: 1 1 1 1 0 1 0 1 1 0 1 1 1 1 1 0

Result Entry: 0 0 2 3

(b)

**Figure 6. (a) The BCCOO format of Matrix C in Eq.2. (b) The example of compute the location of the first result generated by each thread, assuming that there are four threads and each thread processes four non-zero blocks.**

Second, we perform a quick check to see whether we can skip the parallel segmented scan operation at the workgroup level. It is the case when each thread in a workgroup encounters a row stop, which results in the segment size being 1 for the parallel segmented scan.

### 3. An Efficient Matrix-based Segmented Sum/Scan for SpMV

With a sparse matrix stored in our BCCOO/BCCOO+ format, SpMV can be implemented in three logical steps: (1) read the data value arrays and multiply them with the corresponding vector values indexed by the *Col\_index* array; (2) perform a segmented scan using the bit flag array from our BCCOO/BCCOO+ format; (3) write back the results to global memory. In our proposed scheme, all these three steps are implemented in a single kernel so as to minimize the kernel invocation overhead.

### 3.1 Segmented Scans

The segmented scan primitive scans multiple data segments that are stored together. A start flag array is typically used to identify the first element of a segment. We show an example of the inclusive segmented scan in Figure 7. Its start flag array is generated from the bit-flag array of the BCCOO format in Figure 6. The output of the inclusive segment scan is the ‘Result’ array in Figure 7. Note that for SpMV, the complete segmented scan results are not necessary. Instead, the last sum of each segment is sufficient, as marked with underscores in the ‘Result’ array. In other words, for SpMV, the segmented reduction/sum primitive can be used rather than the segmented scan primitive.

Input = [3 2 0 2 1 0 4 2 4 3 2 2 0 1 3 1]
Bit Flag = [1 1 1 1 0 1 0 1 1 0 1 1 1 1 1 0]
Start Flag = [1 0 0 0 0 1 0 1 0 0 1 0 0 0 0 0]
Result = [3 5 5 7 <u>8</u> 0 <u>4</u> 2 6 <u>9</u> 2 4 4 5 8 <u>9</u> ]

Figure 7. An inclusive segmented scan with the start flags generated from the bit flag array in Figure 6(a).

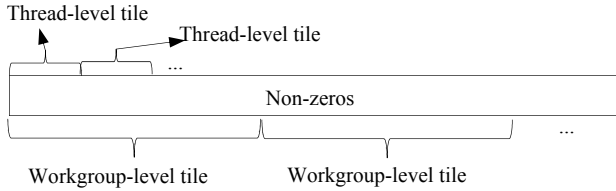


Figure 8. Even workload distribution: each workgroup/thread block works on a workgroup-level tile; each thread works on a thread-level tile of non-zero blocks.

Two main approaches have been proposed to parallelize the segmented scan primitive on GPUs. One is a tree-based approach [5], which builds a binary tree through different processing stages. The tree-based approach suffers from the load imbalance problem as different numbers of threads will be idle in different processing stages. Furthermore, it requires workgroup-level synchronization between stages as discussed in [8]. The other is a matrix-based approach, which is proposed to improve memory efficiency and overcome the load imbalance problem. Our proposed BCCOO/BCCOO+ format suits better with the matrix-based segmented scan and we further customize it for SpMV.

### 3.2 A Customized Matrix-based Segmented Sum/Scan for SpMV

#### 3.2.1 Per-thread and per-workgroup working sets

In our segmented sum/scan approach for SpMV, the input non-zero blocks as well as the corresponding bit-flag array and the column index array are divided evenly among workgroups. The working set of each workgroup is referred to as a workgroup-level tile, which in turn will be divided evenly among the threads within the workgroup. The working set of a thread is referred to as a thread-level tile, as shown in Figure 8. The benefits of using a single thread to

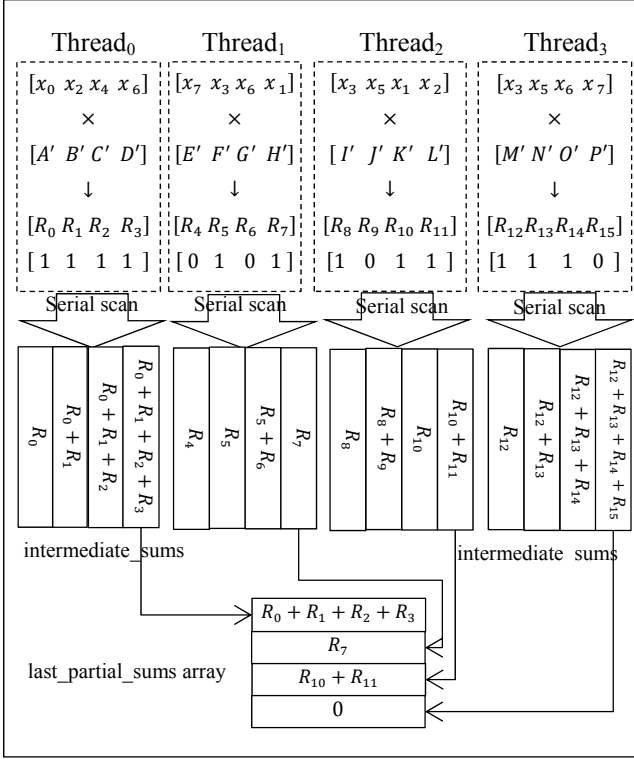
process multiple consecutive non-zero blocks (e.g., 16) are two-folds. First, a single/few load(s) from the bit flag array (e.g., loading a single short type of data) will be sufficient to provide all the bit flag information. Compared to the previous approaches, which load the row index information for every non-zero, significant bandwidth will be saved. Second, each thread will perform the segmented scan in a sequential manner and may use a segmented sum instead of a segmented scan, which has fewer intermediate results to keep. Also, note that the bit flags in our BCCOO/BCCOO+ format are different from the start flags that are used in typical segmented scans as shown in Figure 7. Although the start flags can be derived from the bit flags, we choose to use the bit flags since it is straightforward to tell whether a segment ends from the bit flags. If the start flags were used, one needs to search for the next start to find the end of the current segment. It would be more complex when the non-zeros in a row span across multiple thread-level or workgroup-level tiles.

Since a thread-level tile may contain row stops, each thread will write its last partial sum into a temporary array, called ‘last\_partial\_sums’, based on its thread identifier (tid) within the workgroup. Then, a parallel scan will be performed on this last\_partial\_sums array. The start flags of the last\_partial\_sums array are generated by each thread as well. To handle the case when the non-zeros in a row span multiple workgroups/thread blocks, we leverage the recently proposed adjacent synchronization [24] to enable inter-workgroup communication while eliminating global synchronization.

#### 3.2.2 Computing per-thread and per-workgroup partial sums

We design two strategies to compute intra-workgroup partial sums from a workgroup-level tile. Either suits for different types of sparse matrices. In the first strategy, each thread has an array, called ‘intermediate\_sums’, to keep all the intermediate sums of its thread-level tile. This intermediate\_sums array can be stored in shared memory, registers, or split between shared memory and registers. This strategy works well if the lengths of the segments are very small, meaning that many rows in a sparse matrix have very small numbers of non-zeros. For matrix C in Eq. 2, assuming that each thread-level tile contains 4 non-zero blocks and there are 4 threads in a workgroup, the computation is illustrated in Figure 9. From the figure, we can see that each thread performs a sequential segmented scan, stores the results in its intermediate\_sums array, and uses the last partial sum to update the corresponding entry of the last\_partial\_sums array, which locates in shared memory and is accessible by all the threads in a workgroup. If the last element of a thread-level tile is a row stop, the last partial sum of this thread is 0, as shown in thread 3 in Figure 9.

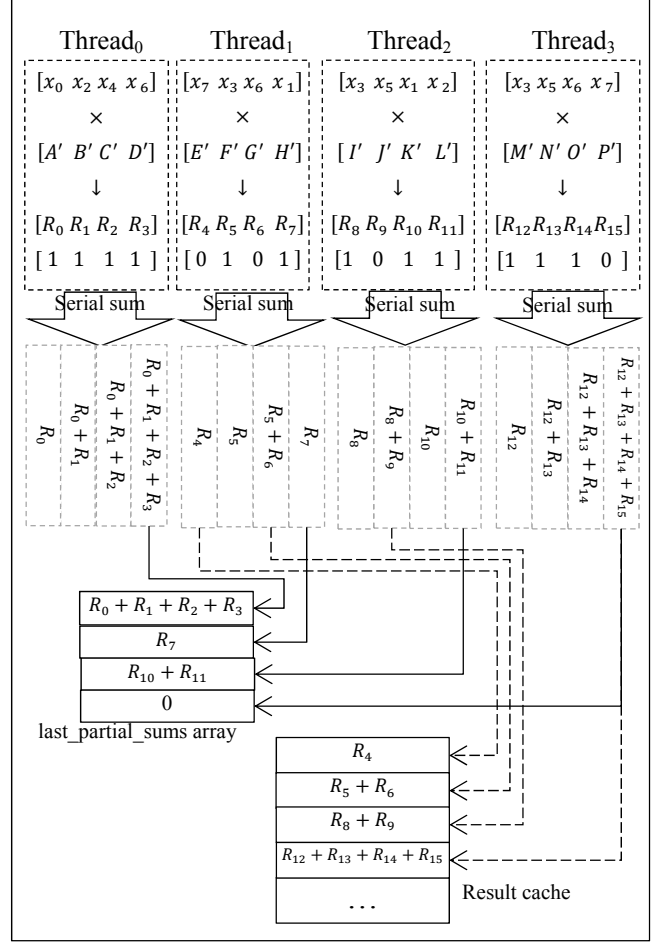
To facilitate memory accesses to the data value array, we can view it as a 2-dimension array with the width as the



**Figure 9. Computing segmented scans: strategy 1, which uses per-thread buffers, i.e., ‘intermediate\_sums’ arrays to store intermediate sum results.**

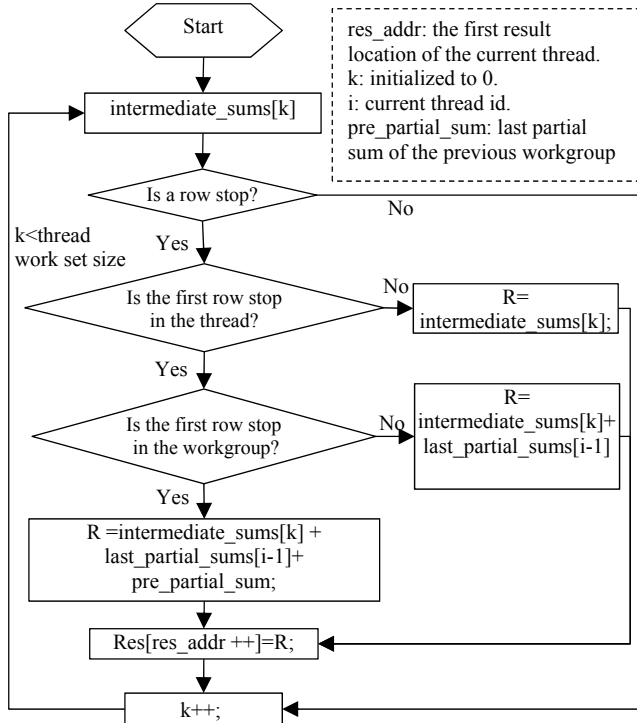
thread-level tile size. Then, with a transpose operation, which can be done either on-line or offline, the threads in a warp will access the data in a row-by-row manner, thereby satisfying the memory coalescing requirement. The same also applies to the `col_index` array. Offline transpose removes the need for a share memory buffer which is required for transpose. In comparison, with the on-line approach, the threads in a warp read one tile at a time in a coalesced manner and multiply with the corresponding vector elements, then store the results in a shared memory buffer still in the row-based manner. Later on, when performing the segmented scan, the threads read the buffer in a column-based manner. This way, better performance may be achieved due to improved locality from accesses to the multiplied vector if non-zeros in a row are close to each other.

In our second strategy, we allocate a result cache in shared memory to only store the sum of each segment. This strategy works better for long segments and also benefits from efficient memory writes as we can store the result cache to global memory in a coalesced way. With this strategy, the offline transpose is used to ensure coalesced memory reads from the value array and the `col_index` array. After performing the multiplication with vector elements, each thread carries out a segmented sum sequentially on its thread-level tile, using the bit flag array as the mask for the segments. All the segmented sums will be written to the



**Figure 10. Computing segmented scan: strategy 2, which uses a per-workgroup result cache to store segmented sums. The dashed blocks mean that the intermediate sums are *not* stored.**

result cache with the help of the first-result-entry information generated along with the BCCOO format. The process is illustrated in Figure 10 for matrix C in Eq. 2 assuming that the thread-level tile size is 4 and there are 4 threads in a workgroup. The first-result-entry information shown in Figure 6 is used for updating the result cache. For example, as shown in Figure 6 the first-result-entry for thread 1 and thread 2 is 0 and 2, respectively. Therefore, when thread 1 encounters the first row stop, i.e., the end of the first segment, it uses its current sum R4 to update the results cache entry 0. When thread 1 encounters the second row stop, it uses the sum R5+R6 to update the result cache entry 1. In a sense, the first-result-entry information computed along the BCCOO format partitions the result cache among different threads in a workgroup. In the case when the number of row stops in a workgroup-level tile is larger than the results cache size, the extra segmented sums will be stored in the result array in global memory, which will be re-accessed later to generate the final outputs. The same as the first strategy, each thread also writes its last partial sum to the `last_partial_sums` array. To generate the



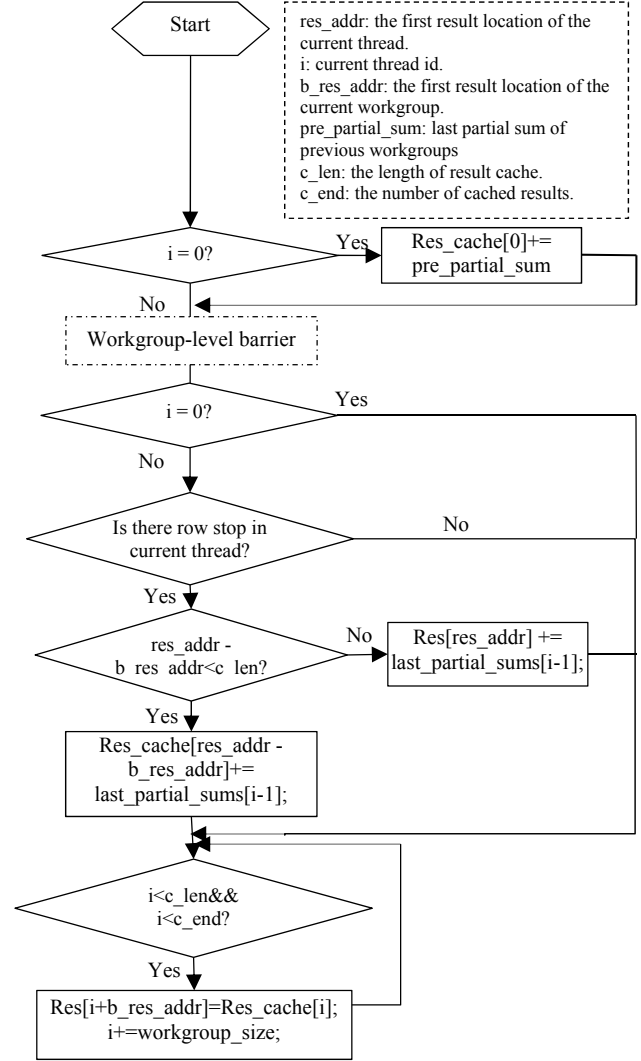
**Figure 11. A flow chart of combining the partial sums from threads and workgroups: strategy 1.**

start flags for the last\_partial\_sums array, in either strategy, each thread performs the simple check on whether its bit flags contain a 0. In other words, each thread checks whether there is a row stop in its thread-level tile. If so, its last partial sum should be a start for a segment in the last\_partial\_sums array. For the example in Figure 9 and Figure 10, the start flags are [0, 1, 1, 1] since all threads except thread 0 process a tile containing a row stop. After all threads in a workgroup update its last partial sum in the last\_partial\_sums array and generate the start flags, which is signaled with a workgroup-level synchronization or syncthreads(), the threads in the workgroup perform a parallel segmented scan using the scan algorithm in [18] and the results are also stored in the same last\_partial\_sums array. In our example in Figure 9 or Figure 10, this parallel scan can be skipped as all the segment sizes are 1.

### 3.2.3 Combining per-thread and per-workgroup partial sums

Next, we need to combine the results in the per-thread intermediate\_sums arrays, the scanned result for the per-workgroup last\_partial\_sums array, and also the results from other workgroups to generate the final output of SpMV.

For our first strategy, each thread will go through its intermediate\_sums array. For each element, it checks whether the corresponding bit flag is a row stop. If not, it means the corresponding result has already been incorporated into the sum of the segment. For a row stop, a thread further checks whether it is the first stop in its thread-level tile. If not, it means the thread-level tile contains the



**Figure 12. A flow chart of combining the partial sums from threads and workgroups: strategy 2**

complete segment and the corresponding result is the final result. In the example shown in Figure 9, for thread 1, the entry in its intermediate\_sums array containing ( $R_5 + R_6$ ) is such a case. If a row stop is the first in a thread-level tile (e.g., the entry containing  $R_4$  for thread 1 in Figure 9), there are two possibilities. One is that the segment spans multiple threads within a workgroup. Then, the last\_partial\_sums array of the workgroup will be used to retrieve the last partial sum of the previous threads. For example, the entry containing ( $R_0 + R_1 + R_2 + R_3$ ) in the last\_partial\_sums array will be added to  $R_4$  of thread 1 in Figure 9. As the last\_partial\_sums array contains the scanned result of the original last partial sums of each thread, the entry *last\_partial\_sums[i-1]* (*i* the current thread id) already accumulates the partial sums of multiple threads for segments spanning multiple threads. The other possibility is that the segment spans multiple threads across workgroups. In this case, we also need to accumulate previous workgroups' last partial sum results. We resort to adjacent

synchronization to avoid global synchronization as discussed in Section 3.2.4. In Figure 11, a flow chart of the kernel program is presented to illustrate the process discussed above.

For our second strategy, there are no per-thread intermediate sum arrays. Instead, there is a per-workgroup result cache and therefore threads in a workgroup rely on the first result location, which is produced along the BCCOO/BCCOO+ format, to process the result cache and the flow chart of the kernel code is presented in Figure 12. Each thread except thread 0 first checks whether there are row stops in its thread-level tile. If so, it means that the thread has generated some partial sums corresponding to the row stops. Here, each thread only needs to process the partial sum at the first row stop since it may be a part of a long segment spanning multiple thread-level tiles (e.g.,  $R_4$  in the result cache in Figure 10). For subsequent row stops in the thread, the partial sums in the result cache are already complete segment sums (e.g.,  $R_5+R_6$  in the result cache in Figure 10). Then, each thread except thread 0 checks whether its first partial sum is written in the result cache or in global memory depending on its first result position and the result cache size. As the last partial sum corresponding to the previous thread in the `last_partial_sums` array already accumulates the partial sums of multiple threads for segments spanning multiple threads, it is added to the result cache entry (e.g.,  $R_0+R_1+R_2+R_3$  from the `last_partial_sums` array is added to  $R_4$  in the result cache in Figure 10). For thread 0, it updates result cache entry 0 with the last partial sum from the previous workgroup. To avoid data race at result cache entry 0, a workgroup-level synchronization is added after thread 0 processes the result cache entry 0. After the result cache is processed, it is written to global memory in a memory coalesced way by all threads together in a workgroup.

### 3.2.4 Accumulating partial sums across workgroups

As discussed in Section 3.2.3, for segments spanning multiple workgroups, the last workgroup, which contains the row stop, needs to accumulate previous workgroups partial sums. Here, we make an implicit assumption that the workgroup-level tiles are distributed to workgroups in-order. In other words, workgroup 0 processes the first tile; workgroup 1 processes the second tile; etc. Current GPUs dispatch workgroups in-order. Therefore, we can directly use the workgroup ids in the kernel. If a GPU dispatches workgroups out-of-order, workgroups can get such ‘logic’ workgroup ids from global memory using atomic fetch-and-add operations. This approach incurs small performance overhead, less than 2% in our experiments. To accumulate partial sums across workgroups, we use a global memory array ‘`Grp_sum`’. The array is initialized to a special value (e.g., maximal floating-point number). This array is updated in a sequential manner. Workgroup 0 updates the first entry ‘`Grp_sum[0]`’ with its last partial sum. For a subsequent workgroup with id  $X$ , if it does not contain a row stop, it waits for the entry ‘`Grp_sum[X-1]`’ to be changed from the

initial value, i.e., updated by workgroup  $(X-1)$ , and then updates ‘`Grp_sum[X]`’ with the sum of its last partial sum and ‘`Grp_sum[X-1]`’. If a workgroup contains a row stop, it breaks such chained updates and directly updates ‘`Grp_sum[X]`’ with its last partial sum. This approach is called adjacent synchronization in [24].

## 4. Auto-Tuning Framework

As discussed in Sections 2 and 3, we propose a new format BCCOO and its variant BCCOO+ for sparse matrices, and two new strategies to compute segmented sums/scans for SpMV. To find the optimal solution for a sparse matrix, we build an auto-tuning framework to select the format, the computing strategy, as well as their associated parameters. Then, the OpenCL code is generated according to the selected parameters from this auto-tuning framework. We also use this framework to exploit the texture cache for the multiplied vector. Another optimization is that we use the ‘unsigned short’ data type for the `col_index` array if the width of a sparse matrix is less than 65535. In this case, there is no need to further compress the `col_index` array using the approach discussed in Section 2.2. The parameters that this framework explores are listed in Table 1. Note that when strategy 1 is used to compute the segmented scan, the thread-level tile size is the size of the `immediate_sums` array, which is the sum of the parameters, `Reg_size` and `ShM_size`.

**Table 1. Tunable parameters of the auto-tuning framework.**





















Parameter Name		Possible Values
Matrix format		BCCOO, BCCOO+
Col_index compress		Yes, No
Block width		1, 2, 4
Block height		1, 2, 3, 4
Data type for the bit flag array		Unsigned char, unsigned short, unsigned int
Vertical slice number		1, 2, 4, 8, 16, 32
Transpose		Offline, online
Texture memory for multiplied vector		Yes, No
Workgroup size		64, 128, 256, 512
Strategy 1	Registers for the per-thread intermediate sums array ( <code>Reg_size</code> )	0, 8, 16, 32
	Shared memory for the per-thread intermediate sums array ( <code>ShM_size</code> )	0, 8, 16, 32
Strategy 2	Thread-level tile size	8,16,24,32,40,64,96,128
	Result cache size (multiple of the workgroup size)	1,2,3,4

As shown in Table 1, there are many parameters to tune, which form a relatively large search space for a sparse matrix on a particular hardware platform. In order to accelerate auto-tuning, we perform the following optimizations. First, we use GPUs to accelerate the translation from the COO format to the BCCOO/BCCOO+ format. Second, we cache compiled kernels in a hash table so that they can be reused for difference matrices. Third, we prune the search space using the following heuristics: since the memory footprint is highly dependent on block dimensions, we only need to select the block dimensions corresponding to the 4 smallest memory footprints. Fourth, we further



reduce the search space by: always using the texture memory for the multiplied vector, always using offline transpose, limiting the result cache size to 1 and 2 for strategy 2, and setting the shared memory size as 0 for the per-thread intermediate sums array for strategy 1. With these optimizations, the average auto-tuning time is 12.8 seconds among the 20 matrices in our study, running on a desktop machine with an Intel(R) Core2 Quad CPU Q9650 @ 3.00GHz and an NVIDIA GTX680 GPU. Compared to the optimal results obtained from an exhaustive search of the parameters listed in Table 1, our auto-tuning results are identical to the optimal ones on GTX 680 GPUs. On GTX480 GPUs, however, the optimal configurations show 10.5% better performance for the matrix Epidemiology, which prefers no texture memory usage, and 11.1% better performance for the matrix Circuit, which prefers online transpose. Furthermore, a finer grain parameter selection may further improve performance. For example, a Thread-level tile size of 40 yields 5% better performance for the matrix Dense than our auto-tuning results on GTX480 GPUs.

**Table 2. The sparse matrices used in the experiments.**

Spyplot	Name	Size	Non-zeros (NNZ)	NNZ/Row
	Dense	2K * 2K	4000000	2000
	Protein	36K * 36K	4344765	119
	FEM/Spheres	83K * 83K	6010480	72
	FEM/Cantilever	62K * 62K	4007383	65
	Wind Tunnel	218K*218K	11634424	53
	FEM/Harbor	47K * 47K	2374001	59
	QCD	49K * 49K	1916928	39
	FEM/Ship	141K*141K	7813404	28
	Economics	207K*207K	1273389	6
	Epidemiology	526K*526K	2100225	4
	FEM/Accelerator	121K*121K	2620000	22
	Circuit	171K*171K	958936	6
	Webbase	1M * 1M	3105536	3
	LP	4K * 1.1M	11279748	2825
	Circuit5M	5.56M* 5.56M	59524291	11
	eu-2005	863K*863K	19235140	22
	Ga41As41H72	268K*268K	18488476	67
	in-2004	1.38M* 1.38M	16917053	12
	mip1	66K * 66K	10352819	152
	Si41Ge41H72	186K*186K	15011265	81

## 5. Experimental Methodology

We implemented our proposed scheme in OpenCL[19]. Our experiments have been performed on both an Nvidia GTX680 GPU and an Nvidia GTX480 GPU.

We use a total of 20 sparse matrices, 14 of them are from [23] and 6 of them are from [16]. Table 2 summarizes the

information of the sparse matrices, including the size, total number of non-zeros, and number of non-zeros per row. These matrices have been widely used in previous works [1][7][12][16][23].

In our experiments, we also use CUSPARSE V5.0 [13], CUSP [1], and clSpMV [16] for performance comparisons. CUSPARSE supports three formats HYB, BCSR, and CSR. As the HYB format is a hybrid format combining the advantages of the ELL and COO formats, the row length of the ELL part is configurable. We manually searched the row length in a wide range and use the best performing one for each matrix. For the BCSR format in CUSPARSE, we also searched the block size for the best performance. For clSpMV, besides the COCKTAIL format, which uses different formats for different partitions of a matrix, we tested all the single formats and chose the best performing single format for each matrix. The same performance testing framework is used as in [16]. The code of our proposed framework is available at <http://code.google.com/p/yaspmv/>.

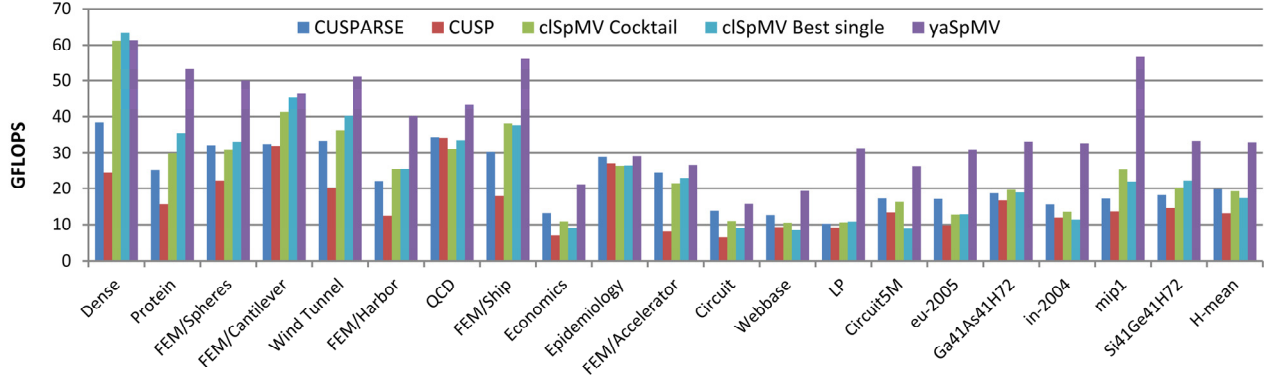
## 6. Experimental Results

In our first experiment, we evaluate the impact of our proposed BCCOO/BCCOO+ format on memory bandwidth. Since in our BCCOO/BCCOO+ format, all the information, including the bit flag array, the col\_index array, the data value array, as well as the auxiliary information described in Section 2.4, is only read once, we assume that it is also the case for all the formats in comparison. Therefore, we can simply use the sum of the array sizes to show the memory footprint of each format. The results are shown in Table 3. As our auto-tuning framework selects the BCCOO+ format only for the matrix LP, we do not separate the BCCOO and the BCCOO+ format. For some sparse matrices, due to the high variance in the number of non-zeros in different row, the ELL format is not applicable (labeled 'N/A' in Table 3).

**Table 3. The memory footprint size (MB) of different formats.**

Name	COO	ELL	Cocktail	Best Single	BCCOO
Dense	48	32	17	17	17
Protein	52	59	40	34	21
FEM/Spheres	72	54	52	51	31
FEM/Cantilever	48	39	25	25	21
Wind Tunnel	140	314	78	78	65
FEM/Harbor	28	54	24	24	14
QCD	23	15	15	15	9
FEM/Ship	94	115	56	59	34
Economics	15	73	14	28	8
Epidemiology	25	17	17	17	14
FEM/Accelerator	31	79	26	25	17
Circuit	12	483	9	23	6
Webbase	37	N/A	29	138	27
LP	135	1927	91	91	85
Circuit5M	714	N/A	578	714	516
eu-2005	231	N/A	248	209	159
Ga41As41H72	222	1505	139	170	136
in-2004	203	N/A	209	203	132
mip1	124	N/A	66	54	51
Si41Ge41H72	180	983	118	135	105
Average	122	N/A	93	106	73

From Table 3, we can see that our proposed BCCOO/BCCOO+ format significantly reduces the storage size of various sparse matrices. On average, our proposed



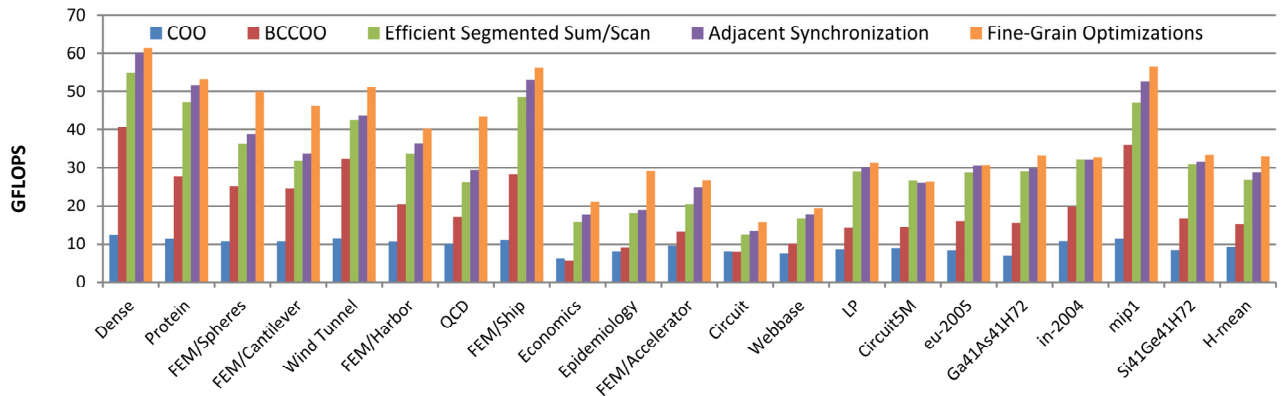
**Figure 13. Performance comparison between our proposed scheme (labeled 'yaSpMV') and CUSPARSE 5.0, CUSP, clSpMV-best single, and clSpMV-COCKTAIL on GTX680 GPUs.**

BCCOO/BCCOO+ format reduces the storage size by 40% compared to the COO format, 31% compared to the best single format among all the 9 formats included in clSpMV, and 21% compared to the COCKTAIL format.

In the second experiment, we compare the performance of our proposed scheme to the state-of-art techniques. The results of GTX680 are shown in Figure 13 and the results of our proposed approach are labeled 'yaSpMV' in the figure. From the figure, we can see that our proposed approach outperforms the existing schemes for all the matrices except Dense. The Dense matrix prefers a block size of 2x8 as used in the BCSR format from the 'clSpMV best single' results. However, our auto-tuning framework limits the maximal block height is limited to 4, thereby achieving sub-optimal performance. Using the harmonic mean (H-mean) as the average throughput, our yaSpMV achieves an average performance improvement of 65% over CUSPARSE, 70% over clSpMV COCKTAIL, 88% over clSpMV best single, and 150% over CUSP. The highest performance improvement of yaSpMV achieved over clSpMV COCKTAIL is on matrix LP (195%). Compared to CUSPARSE, the highest performance gain of yaSpMV is

from the matrix mip1 (229%).

In the third experiment, we examine the performance contributions from different optimizations in our approach, including memory footprint reduction, efficient segmented sum/scan, adjacent synchronization to remove global synchronization, and fine-grain optimizations, which consist of (a) the use of the short data type for the col\_index array and (b) early check to skip the parallel scan on a last\_partial\_sums array if each thread-level tile in a workgroup-level tile contains a row stop. The results are shown in Figure 14. We start with the COO format with a tree-based segment sum (labeled 'COO'). Then, we replace the COO format with our BCCOO/BCCOO+ format (labeled 'BCCOO'). Next, we replace the tree-based segmented sum with our proposed efficient matrix-based segment sum/scan (labeled 'Efficient segmented sum/scan') while using another kernel to accumulate partial sums across workgroups. We then use adjacent synchronization to replace this kernel (labeled 'adjacent synchronization') and add the fine-grain optimizations (labeled 'fine-grain optimizations'). From the figure, we can see that the main performance gains are from our proposed BCCOO/BCCOO+ format and our efficient segmented



**Figure 14. Performance Contributions from different optimization techniques (GTX680)**

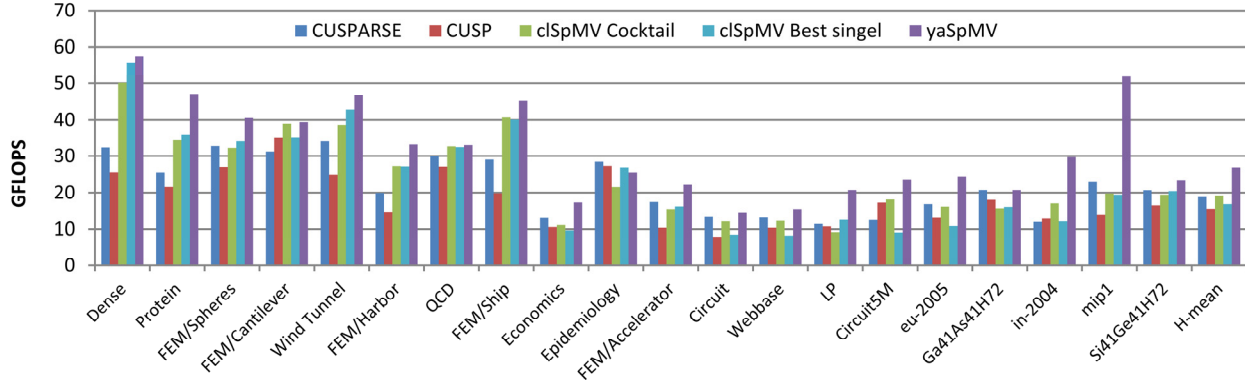


Figure 15. Performance comparison between our proposed scheme (labeled 'yaSpMV') and CUSPARSE 5.0, CUSP, clSpMV-best single, and clSpMV-COCKTAIL on GTX480 GPUs.

sum/scan for SpMV.

We also evaluate the performance of SpMV on Nvidia GTX480 GPUs. The results are shown in Figure 15. Among the 20 sparse matrices, our proposed yaSpMV achieve significantly higher performance than existing approaches, up to 162% better than clSpMV COCKTAIL and up to 150% better than CUSPARSE. The only exception is the matrix Epidemiology. It has 4 non-zeros on each row and therefore is a perfect fit for the ELL format. For this matrix, our yaSpMV has a suboptimal performance of 25.5 GFLOPS. The best performing approach for this matrix, CUSPARSE, has a throughput of 28.5 GFLOPS. On average using the harmonic mean, our proposed yaSpMV achieves a performance improvement of 40% than clSpMV COCKTAIL, 60% over clSpMV best single, 74% over CUSP, and 42% over CUSPARSE.

## 7. Related Work

Sparse matrix-vector multiplication (SpMV) is so important that there have been numerous works optimizing its performance. We only discuss the most relevant ones here. Williams et al. present several optimizations for multicore platforms [23]. Kourtis et al. [11] proposed an Extended Compression Format (CSX) on shared memory systems. OSKI [22] is a library collection which provides low-level primitives for automatically tuned kernels on sparse matrices. Aydın Buluc et al. [3] introduced a compressed sparse blocks (CSB). Among the research works leveraging GPUs for SpMV, Bolz et al. first introduced the GPU for SpMV [6]. Bell and Garland implemented several well-known formats on Nvidia GPUs [1]. These formats include DIA, ELL, CSR, COO and a new hybrid format HYB, which combines the advantage of the ELL and COO formats. Vázquez et al. proposed a derivative format of ELLPACK, ELL-R [21]. They use an auxiliary array to store the row lengths. Alexander et al. proposed the Sliced ELL format (SELL) [12]. They horizontally partition the original matrix into several slices and different slices use different ELL padding lengths to reduce the filling zeros. Compared to the

ELL format, the ELL-R and SELL formats have less padding zeros while the workload may be imbalanced. Based on the CSR format, Kozaa et al. [10] proposed a Compressed Multiple-Row Storage Format for SpMV on GPUs. The advantage of this format is that the adjacent rows may be processed by the same thread, so the multiplied vector data could be reused. Sun et al. [17] proposed a CRSD format for diagonal sparse matrices. Choi et al. implemented the BCSR and BELL formats on GPUs [7]. A performance model driven framework is also proposed in [7] for performance auto-tuning of SpMV on GPUs. Su et al. [16] proposed the COCKTAIL format, which uses different formats to represent different partitions of a matrix. There are some works focusing on compression and reordering techniques as well [2][14]. The challenge of compression technique is the complexity of the decompression algorithm. The problem with the reordering technique is that it changes the inherent locality of the original matrix. A recent work by Tang et al. [20] studies bit-representations to compress index arrays. Similar to our work, a difference function is applied to index arrays. The difference from our proposed formats is that a bit packing scheme is then used to encode the delta values, which makes their decompression scheme more complicated than ours and also does not exploit the row stop information, when compressing row index arrays.

Blelloch et al. [4] first introduced the segmented operations to SpMV on vector multiprocessors. Harris [9] implemented the segmented scan based SpMV in the library CUDPP. Because they used a tree based scan algorithm, which has been shown to be inefficient [24], the performance is limited. Baskaran et al. [15] implemented a more efficient segmented scan based SpMV using the matrix based scan [8]. However, their scan-based implementation also is outperformed by their alternative implementations [15]. Bell and Garland implemented their COO format use the segmented reduction (scan) algorithm. However, due to the disadvantage of the COO format and the two-kernel implementation, the performance is not highly competitive.

Different from the previous works, we design the new BCCOO/BCCOO+ format to drastically reduce the bandwidth requirement. We also propose an efficient matrix-based segmented sum/scan for SpMV to maximize the benefit from our new BCCOO/BCCOO+ format on GPUs. Our algorithm only needs one kernel and explores a number of optimization techniques.

## 8. Conclusions

In this paper, we present yet another framework for SpMV on GPUs. First, we propose a new format, called blocked compressed common coordinate (BCCOO), for sparse matrices. The key idea is to extend the COO format with blocking and to use a bit flag array to replace the row index array. We also propose to vertically partition a sparse matrix before using the BCCOO format so as to improve the locality for accesses to the multiplied vector. Second, we revisit segmented scans for SpMV. We propose a highly efficient matrix-based segmented sum/scan for SpMV. Our matrix-based segmented sum/scan is closely coupled to our BCCOO/BCCOO+ format to reduce the memory bandwidth and achieve load balance. Our performance results from a set of 20 sparse matrices show that our proposed framework significantly advances the state-of-art of the highly important SpMV algorithm. It outperforms the vendor tuned CUSPARSE by up to 150% and 42% on average on GTX480 GPUs, by up to 229% and 65% on average on GTX680 GPUs. Compared to the clSpMV, our proposed scheme achieves a performance gain of up to 162% and 40% on average on GTX480 GPUs, up to 195% and 70% on average on GTX680 GPUs.

## Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments. This paper is supported in part by the National High-tech R&D Program of China (No.2012AA010902), an NSF project CCF-1216569, an NSF CAREER award CCF-0968667, and NSFC (No. 61272136, No.60921002, No. 61100072).

## References

- [1] N. Bell and M. Garland. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. SC, 2009.
- [2] A. Buluç, S. Williams, L. Oliker and J. Demmel. Reduced-Bandwidth Multithreaded Algorithms for Sparse Matrix-Vector Multiplication. IPDPS, 2011.
- [3] 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. SPAA, 2009.
- [4] G. E. Blelloch, M. A. Heroux and M. Zagha. Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors. Tech. Rep. CMU-CS-93-173, School of Computer Science, Carnegie Mellon University, Aug 1993.
- [5] G. E. Blelloch. Scans as Primitive Parallel Operations. IEEE Transactions on Computers, 1989.
- [6] J. Bolz, I. Farmer, E. Grinspun and P. Schröder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. ACM Transactions on Graphics (TOG), July 2003.
- [7] J. W. Choi, A. Singh and R. W. Vuduc. Model-driven Autotuning of Sparse Matrix-Vector Multiply on GPUs. PPOPP, 2010.
- [8] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd and J. Manferdelli. Fast Scan Algorithms on Graphics Processors. ICS, 2008.
- [9] M. Harris, S. Sengupta, and J. D. Owens. CUDPP: CUDA Data Parallel Primitives Library. <http://gpgpu.org/developer/cudpp>
- [10] Z. Koza, M. Matyka, S. Szkodra and L. Miroslaw. Compressed Multiple-Row Storage Format. CoRR 2008.
- [11] K. Kourtis, V. Karakasis, G. Goumas and N. Koziris. CSX: An Extended Compression Format for SpMV on Shared Memory Systems. PPOPP, 2011.
- [12] A. Monakov, A. Lokhmotov and A. Avetisyan. Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. HiPEAC, 2010.
- [13] Nvidia. CUSPARSE. <https://developer.nvidia.com/cusparserepository>. <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>
- [14] J. C. Pichel, F. F. Rivera, M. Fernández and A. Rodríguez. Optimization of sparse matrix-vector multiplication using reordering techniques on GPUs. Microprocessors and Microsystems, 36(2), 65–77, Mar 2012.
- [15] M. M. Baskaran and R. Bordawekar. Optimizing Sparse Matrix-Vector Multiplication on GPUs using Compile-time and Run-time Strategies. Technical Report RC24704 (W0812-047), IBM, Dec 2008.
- [16] B.-Y. Su and K. Keutzer. clSpMV: A Cross-Platform OpenCL SpMV Framework on GPUs. ICS, 2012.
- [17] X. Sun, Y. Zhang, T. Wang, X. Zhang, L. Yuan and L. Rao. Optimizing SpMV for Diagonal Sparse Matrices on GPU. ICCP, 2011.
- [18] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In Graphics Hardware 2007.
- [19] The Khronos OpenCL Working Group OpenCL. The Open Standard for Parallel Programming of Heterogeneous Systems. <http://www.khronos.org/opencl/>
- [20] W. Tang et al., Accelerating sparse matrix-vector multiplication on GPUs using bit-representation-optimized schemes, SC 2013.
- [21] F. Vázquez, J. J. Fernández and E. M. Garzón. A new approach for sparse matrix vector product on NVIDIA GPUs. Concurrency Computat.: Pract. Exper. Sep 2010.
- [22] R. Vuduc, J. W. Demmel and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. SciDAC 2005.
- [23] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. A. Yelick and J. W. Demmel. Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. SC, 2007.
- [24] S. Yan, G. Long and Y. Zhang. StreamScan: Fast Scan Algorithms for GPUs without Global Barrier, PPOPP, 2013.