

A Cross-Platform SpMV Framework on Many-Core Architectures¹

YUNQUAN ZHANG, State Key Laboratory of Computer Architecture, Institute of Computing Technologies, Chinese Academy of Sciences.

SHIGANG LI*, State Key Laboratory of Computer Architecture, Institute of Computing Technologies, Chinese Academy of Sciences.

SHENGEN YAN*, SenseTime Group Limited. Department of Information Engineering, Chinese University of Hong Kong.

HUIYANG ZHOU, Department of Electrical and Computer Engineering, North Carolina State University.

Sparse matrix-vector multiplication (SpMV) is a key operation in engineering and scientific computing. Although the previous work has shown impressive progress in optimizing SpMV on many-core architectures, load imbalance and high memory bandwidth remain the critical performance bottlenecks. We present our novel solutions to these problems, for both GPUs and Intel MIC many-core architectures. First, we devise a new SpMV format, called blocked compressed common coordinate (BCCOO). BCCOO extends the blocked common coordinate (COO) by using bit flags to store the row indices to alleviate the bandwidth problem. We further improve this format by partitioning the matrix into vertical slices for better data locality. Then, to address the load imbalance problem, we propose a highly efficient matrix-based segmented sum/scan algorithm for SpMV, which eliminates global synchronization. At last, we introduce an auto-tuning framework to choose optimization parameters. Experimental results show that our proposed framework has a significant advantage over the existing SpMV libraries. In single-precision, our proposed scheme outperforms clSpMV COCKTAIL format by 255% on average on AMD FirePro W8000, and outperforms CUSPARSE V7.0 by 73.7% on average and outperforms CSR5 by 53.6% on average on GeForce Titan X; in double-precision, our proposed scheme outperforms CUSPARSE V7.0 by 34.0% on average and outperforms CSR5 by 16.2% on average on Tesla K20, and has equivalent performance compared with CSR5 on Intel MIC.

CCS Concepts: • **Computing methodologies** → **Parallel computing methodologies**;

¹Extension of Conference Paper. The additional contributions of this manuscript over the previously published work of S. Yan et al at PPOPP-2014 include:

1. This paper extends our proposed BCCOO format for Intel Xeon Phi processors by introducing inner-block transpose (Section 3.2.2).
2. We propose a new segmented sum/scan strategy for Intel Xeon Phi processors to explore the potential performance of their 512-bit SIMD instructions (Section 3.2.2).
3. We present the experimental results on Intel Xeon Phi processor, and compare the performance with NVIDIA and AMD GPUs in the context of yaSpMV (Section 6).
4. We also extend the yaSpMV library to support double precision (Section 3.2.3) and present the experimental results in Section 6.

The new material is more than one-third of our PPOPP-2014 paper.

Author's addresses: Y. Zhang, State Key Laboratory of Computer Architecture, Institute of Computing Technologies, Chinese Academy of Sciences, Beijing 100190, China; S. Li (*corresponding author), State Key Laboratory of Computer Architecture, Institute of Computing Technologies, Chinese Academy of Sciences, Beijing 100190, China, Email address: shigangli.cs@gmail.com; S. Yan (*corresponding author), SenseTime Group Limited. Department of Information Engineering, The Chinese University of Hong Kong, Email address: yanshengen@gmail.com; H. Zhou, Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2016 ACM. 1544-3566/2016/08-ARTXXXX \$15.00

DOI: 0000001.0000001

Additional Key Words and Phrases: SpMV, Segmented Scan, BCCOO, OpenCL, CUDA, GPU, Intel MIC, Parallel Algorithms

ACM Reference Format:

Yunquan Zhang, Shigang Li, Shengen Yan and Huiyang Zhou, 2015. A Cross-Platform SpMV Framework on Many-Core Architectures. *ACM Trans. Architec. Code Optim.* x, x, Article XXXX (August 2016), 25 pages. DOI: 0000001.0000001

1. INTRODUCTION

Sparse matrix-vector multiplication (SpMV) is a key linear algebra algorithm and is heavily used in many important application domains. Many-core architectures feature high computational throughput and memory access bandwidth, which are the promising targets to accelerate the workloads like SpMV. According to the granularity of each core, many-core architectures can be divided into two categories. The first is based on massive light cores, like NVIDIA and AMD GPUs, and high throughput is achieved by massive fine-grained parallelism. The second is based on many heavy cores, in which both coarse-grained and fine-grained parallelism are supported. Intel Many Integrated Core (Intel MIC) architecture belongs to the second category. This paper aims at proposing a high performance SpMV framework for these two kinds of many-core architectures.

Although the sequential implementation of SpMV is straightforward, its parallel implementation is quite challenging, especially on many-core architectures. Firstly, since the non-zeros in a matrix may not be evenly distributed across different rows, the row-based parallelization usually suffers from the load imbalance problem. This problem is more severe on GPU architectures, since the threads operate in the single-instruction multiple-thread (SIMT) manner and the execution time is literally determined by the slowest thread. Secondly, SpMV puts high pressure on the memory hierarchy. The matrix data exhibit poor data reuse, as each non-zero element is only used once for computing the corresponding dot product. Besides, the access pattern of the multiplied vector is irregular, due to the discontinuous locations of the non-zeros in each row. On GPUs, memory coalescing [Ueng et al. 2008], namely all the threads in a warp access the consecutive memory address, is a key factor to achieve high memory bandwidth. However, irregular accesses will destroy memory coalescing, which makes the memory accesses serialized.

There has been a lot of research on accelerating SpMV by many-core processors. To reduce the memory footprint size and fully exploit the performance of the many-core architectures, researcher have proposed a bunch of many-core-oriented storage formats, such as COO [Bell and Garland 2009], ELLPACK [Bell and Garland 2009], ELL-R [Vázquez et al. 2011], SELL [Monakov et al. 2010], BCSR and BELL [Choi et al. 2010], ESB [Liu et al. 2013], and SELL-C- σ [Kreutzer et al. 2014]. On the other hand, the compressed sparse row (CSR) format is still dominant on traditional CPU architectures because of high performance and good compression. Recent research work [Daga and Greathouse 2015], [Liu and Schmidt 2015], [Greathouse and Daga 2014] proposed new algorithms for CSR-based SpMV, which attempted to make the CSR format also achieve high performance on many-core architectures. Furthermore, given the different features of target hardware platforms and different characteristics of sparse matrices, offline auto-tuning or benchmarking [Choi et al. 2010], [Li et al. 2015] is commonly used to improve the performance. Although previous work has achieved impressive performance improvement for SpMV, load imbalance and high memory bandwidth requirement remain the fundamental performance bottlenecks of SpMV.

In this paper, we propose our novel solution to SpMV. Since the proposed solution is yet another SpMV framework, we name it as yaSpMV. We first propose a new format for sparse matrices to alleviate the memory bandwidth pressure. 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 data locality of the multiplied vector, we partition the sparse matrix into vertical slices and align the slices in a top-down manner. Such vertically-partitioned BCCOO is referred to as the BCCOO+ format.

To address the load imbalance problem, we 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 performs sequential segmented scans/sums to generate partial sum results. Then, each workgroup/thread block will run the parallel segmented scan on the last partial sum results. When the final dot-product results require accumulating partial sums across multiple workgroups/thread blocks, adjacent synchronization [Yan et al. 2013] is used to eliminate the overhead of global synchronization. To further improve the performance of our SpMV kernel, we also introduce an auto-tuning framework to explore optimization parameters for different sparse matrices and different platforms. The parameters to be tuned form a large search space. We prune the search space of the parameters using some heuristics and reduce the auto-tuning time to a few seconds.

The yaSpMV framework is implemented based on OpenCL [Stone et al. 2010], which supports general purpose parallel programming on heterogeneous computing platforms, including GPUs and Intel MIC. Experimental results show that our proposed single format fits nearly all of the 20 sparse matrices used in the experiments. In single-precision, compared with the vendor-tuned library CUSPARSE V7.0, our proposed scheme achieves 73.7% on average on GeForce Titan X; compared with the clSpMV [Su and Keutzer 2012], which combines advantages of many existing formats, our proposed scheme achieves up to 195% and 70% on average on GTX680 GPUs, up to 2617% and 255% on average on AMD FirePro W8000 GPUs; compared with CSR5 [Liu and Vinter 2015], our proposed scheme achieves a performance gain of 53.6% on average on GeForce Titan X, and 14.9% on average on AMD FirePro W8000. In double-precision, our proposed scheme outperforms CUSPARSE V7.0 by 34.0% on average on Tesla K20; and outperforms CSR5 by 16.2% on average on Tesla K20, by 9.7% on average on AMD FirePro W8000. On Intel MIC, our proposed scheme has almost equivalent performance compared with CSR5.

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 (BCCOO) format builds upon the common coordinate (COO) format. In this section, we first present the COO format as the background, and then introduce the BCCOO format and its extension – BCCOO+. We will use the sparse matrix in Figure 1 as an example.

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 Figure 1 can be represented with a row index array, a column index array, and a data value array, as shown in Figure 2.

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

Fig. 1. An example of sparse matrix.

Row_index = [0 0 0 1 1 1 2 2 2 2 3 3 3 3 3]
Col_index = [2 6 7 2 3 6 4 5 6 7 0 1 4 5 6 7]
Value = [a b c d e f g h i j k l m n o p]

Fig. 2. The COO format of matrix A.

The parallelization strategy suitable with COO is segmented scan/reduction [Bell and Garland 2009]. As highlighted in [Bell and Garland 2009], [Su and Keutzer 2012], 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 element. Therefore, it has the worst memory footprint [Su and Keutzer 2012].

2.2. BCCOO Format

The BCCOO format extends the COO format in two ways. First, we combine the block-based design with the COO format. In block-based formats, such as blocked ELLPACK and blocked CSR [Choi et al. 2010], 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. Figure 3 shows the blocked COO (BCCOO) format of the matrix A in Figure 1 with the block size of 2×2 .

Row_index = [0 0 1 1 1]
Col_index = [1 3 0 2 3]
Value = $\begin{bmatrix} a & 0 & b & c & 0 & 0 & g & h & i & j \\ d & e & f & 0 & k & l & m & n & o & p \end{bmatrix}$

Fig. 3. The blocked COO format of matrix A with the block size of 2×2 .

From Figure 3, 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 2×2 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 2×2 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 3, we use two arrays rather than a single array to store the data value. For a block size with the height larger than 1, we put different rows in different 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 same as all the block-based formats, the BCCOO format may contain zero elements even in a non-zero block.

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

Fig. 4. The BCCOO format of matrix A with the block size of 2×2 .

Our key extension to the BCOO format is to use a bit flag array to compress the row index array. We first calculate the difference value of each pair of the adjacent elements in the row index array. If the difference value is not greater than 1, we set the corresponding bit in the bit flag array to the difference value. If the difference value is greater than 1, we set a corresponding number of 1s in the bit flag array. Then, we flip the bits of 1s and 0s in the bit flag array, such that a bit value of '0' represents a row stop, namely, the corresponding block is the last non-zero block in the row. A bit value of '1' represents that the corresponding block is not the last non-zero block in a row. When calculating the partial sums, this representation can eliminate the row stop check for each non-zero block (see Section 3.2 for details). The row index information can be reconstructed from the bit flag array by accumulating the number of row stops. Thus, the row index array is compressed in a lossless manner. We refer to this format as blocked compressed COO (BCCOO). For matrix A in Figure 1, the BCCOO format with the block size of 2×2 is shown in Figure 4.

Compared with the BCOO format shown in Figure 3, 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 of BCOO format, BCCOO achieves a compression ratio of 32 for the row index array. 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 the number non-zero blocks processed by a workgroup.

Similar to row-index arrays, we also try to reduce the data transmission overhead for the column index arrays using difference functions. Firstly, we logically partition the column index array into multiple segments, each of which is corresponding to the working set (i.e., the total non-zero blocks to be processed) of a thread. Then, we use a difference function on each segment of the column index array. In this way, there is no inter-thread dependency when reconstructing the column indices. The resulting difference value 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) The vertically sliced and rearranged matrix of matrix A.

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

(b) The bit flag, column index, and data value arrays.

Fig. 5. The BCCOO+ format of matrix A in Figure 1.

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. Next, we apply the BCCOO format on the vertically-sliced and rearranged matrix. However, the column index array is generated based on the block coordinates in the original matrix, rather than the transformed matrix. This is because we need the original column indices to locate the corresponding elements in the multiplied vector for dot-product operations. For matrix A in Figure 1, the vertically-sliced and rearranged matrix becomes matrix B in Figure 5(a), in which the number of slices is 2 and the slice width is 4. The BCCOO+ format of A is shown in Figure 5(b) with the block size of 2×2 . As shown in Figure 5, the bit flag array encodes that there is only one non-zero block in row 0, row 1, and row 2, and two non-zero blocks in row 3. The column indices of these blocks, however, are determined from matrix A rather than matrix B. Taking the 2×2 block $\begin{pmatrix} g & h \\ m & n \end{pmatrix}$ as an example, its column index value indicates that it resides at column 2 in matrix A.

The benefit of BCCOO+ format can be illustrated by the matrix-vector multiplication between matrix A and vector \vec{y} , namely, $A * \vec{y}$. Different rows in the same vertical slice use the same segment of \vec{y} to compute the dot-product. Thus, the temporal locality of vector \vec{y} is improved by BCCOO+. However, since the BCCOO+ format breaks the original matrix into slices, the intermediate results of each slice need to be combined to generate the final results. For matrix A in Figure 1, the computation of $A * \vec{y}$ based on the BCCOO+ format is shown in Figure 6. We can see that it is necessary to use a temporary buffer to store the intermediate results, and invoke an additional kernel to combine them. Suppose the original matrix is divided into s slices and the length of \vec{y} is l . The size of the temporary buffer is calculated by $s * l * \text{sizeof}(\text{datatype})$. Extra memory overhead hurts the performance. Thus, the BCCOO+ format is not always preferred over the BCCOO format. We resort to auto-tuning to determine either the BCCOO or BCCOO+ format should be used.

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

Fig. 6. Matrix-vector multiplication as a sum of the products between its vertical slices and the corresponding vector segments.

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, namely, the row index that the result belongs to. We use matrix C in Figure 7(a) as an example, in which each element represents a data block. To simplify the discussion, we assume the block size is $n \times 1$. 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 7(b). Assuming each thread processes 4 non-zero blocks, we will compute the row index of the first result generated by each thread. Such information can be computed by 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' . The first computation result, namely, $A' * y[0]$, is part of the final result for the dot-product between row 0 and the multiplied vector. Thus, the result entry for thread 0 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 result entry for thread 1 is also set to 0.

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

(a) Sparse Matrix C.

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']

(b) The BCCOO format of Matrix C.

Bit Flag = [1 1 1 1 0 1 0 1 1 0 1 1 1 1 1 0]
Result Entry: 0 0 2 3

(c) The location of the first result generated by each thread. Assume there are four threads and each thread processes four non-zero blocks.

Fig. 7. Auxiliary information for SpMV.

3. EFFICIENT MATRIX-BASED SEGMENTED SCAN/SUM FOR SPMV

For 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 customized matrix-based segmented scan/sum using the bit flag array; (3) combine the partial results, and write back the final results to global memory using the result entry array. 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 8. Its start flag array is generated from the bit flag array of the BCCOO format in Figure 7. The output of the inclusive segment scan is the *Result* array in Figure 8. Note that for SpMV, the complete segmented scan results are not necessary. Actually, we only need the last sum of each segment, which is marked with underscores in the *Result* array. Thus, a more lightweight segmented sum can be used for SpMV. However, considering that a warp is the minimum scheduling unit for GPUs, segmented sum has almost equal computation to segmented scan when there are few non-zeros in each row. Besides, for segmented sum, it has to check whether the corresponding bit for each non-zero value is a row stop or not, which brings overhead. Thus, we consider both segmented scan and segmented sum to implement SpMV.

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 8 0 4 2 6 9 2 2 4 4 5 8 9]

Fig. 8. An inclusive segmented scan with the start flags generated from the bit flag array in Figure 7(b).

Two main approaches have been proposed to parallelize the segmented scan primitive on GPUs. One is a tree-based approach [Blelloch 1989], which builds a binary tree through different processing stages. The tree-based approach suffers from the load imbalance problem. Furthermore, it requires workgroup-level synchronization between stages as discussed in [Dotsenko et al. 2008]. The other is a matrix-based approach [Dotsenko et al. 2008], 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. Customized Matrix-based Segmented Scan/Sum for SpMV

3.2.1. Per-thread and per-workgroup working sets. In our segmented sum/scan approach for SpMV, the non-zero blocks, the 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 9. For a thread-level tile, a single/few load(s) from the bit flag array (such as loading a single short type of data) will be sufficient to provide all the bit flag information. Compared with the previous approaches, which load the row index information for every non-zero, significant bandwidth will be saved.

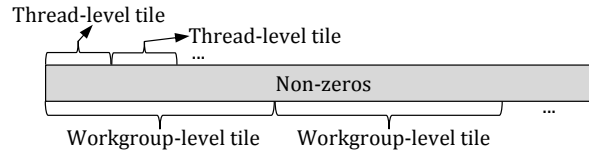


Fig. 9. Even workload distribution: each workgroup/thread block works on a workgroup-level tile; each thread works on a thread-level tile of non-zero blocks.

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*. Then, a parallel segmented scan [Sengupta et al. 2007] will be performed on this *last_partial_sums* array. The start flags of the *last_partial_sums* array are generated by each thread. We further 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. When the non-zeros in a row span multiple workgroups/thread blocks, we leverage the recently proposed adjacent synchronization [Yan et al. 2013] for inter-workgroup communication, which eliminates global synchronization.

3.2.2. Computing the partial sums and the final results. We design three strategies to compute the intra-workgroup partial sums and get the final results. The first and the second strategies are designed for GPUs, and the third one is proposed especially for Inte MIC.

Strategy (1): 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 (also called local memory in OpenCL), registers, or split between shared memory and registers. This strategy works well if most rows in a sparse matrix have very few non-zeros. For the matrix C in Figure 7(a), the computation of the intra-workgroup partial sums is illustrated in Figure 10, in which we assume that each thread-level tile contains 4 non-zero blocks and there are 4 threads in a workgroup. Each thread performs a sequential segmented scan, and stores the results in its *intermediate_sums* array. Each thread uses the last partial sum to update the corresponding entry of the *last_partial_sums* array, which locates in the shared memory and can be accessed 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 by thread 3 in Figure 10.

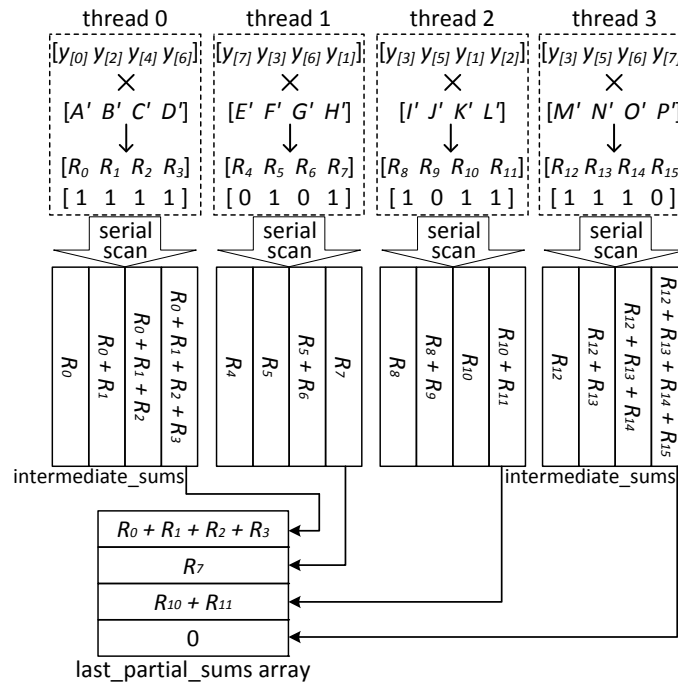


Fig. 10. Computing segmented scans: strategy (1), which uses per-thread buffers, namely, the *intermediate_sums* arrays to store the intermediate sum results.

Memory coalescing [Ueng et al. 2008] is the key factor to achieve high bandwidth when accessing the data value array. We view the data value array as a 2-dimension array with the width as the thread-level tile size. Then, with a transpose operation, the threads in a warp will access the data in a row-by-row manner, thereby satisfying the memory coalescing requirement. We do the same thing to the *Col.index* array. The transpose operation can be done either online or offline. With online 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. If non-zeros in a row are close to each other, online transpose may achieve better performance due to the improved

locality for the multiplied vector. We use online transpose for our first strategy. For offline transpose, the 2-dimension data value array is previously transposed during the format generation phase. Different from online transpose, offline transpose does not require a shared memory buffer for transposition. We will use offline transpose in strategy (2).

ALGORITHM 1: Combining the partial sums for strategy (1)

Input: *res_addr*: the first result location of the thread. *i*: the local thread id.
pre_partial_sum: the last partial sum of the previous workgroups.
bit_flag: the bit flag for the thread working set. *work_size*: the thread working set size.
Output: *results*[*i*]: the final results after combining the partial sums.

```

1 float tmp = 0.0;
2 for int n = 0; n < work_size; n++ do
3   if Is the n-th bit of bit_flag a row stop? then
4     if Is the first row stop in the current thread? then
5       if Is the first row stop in the current workgroup? then
6         | tmp = intermediat_sums[n] + last_partial_sums[i - 1] + pre_partial_sum;
7       else
8         | tmp = intermediat_sums[n] + last_partial_sums[i - 1];
9
10      else
11        | tmp = intermediat_sums[n];
12        results[res_addr++] = tmp;
13
```

Next, we need to combine the results in the per-thread *intermediate_sums* arrays, the results in the per-workgroup *last_partial_sums* array, and also the results from other workgroups to generate the final output of SpMV. ALGORITHM 1 illustrates how to generate the final output for strategy (1). Each thread will go through its *intermediate_sums* array. For each element, it checks whether the corresponding bit flag is a row stop (line 3). 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 (line 4). If not, it means the thread-level tile contains the complete segment and the corresponding result is the final result (line 11). In the example shown in Figure 10, for thread 1, the entry in its *intermediate_sums* array containing ($R5+R6$) is such a case. If a row stop is the first in a thread-level tile (such as the entry containing $R4$ for thread 1 in Figure 10), there are two possibilities. One is that the segment spans multiple threads within a workgroup (line 7-8). 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 ($R0+R1+R2+R3$) in the *last_partial_sums* array will be added to $R4$ of thread 1 in Figure 10. The other possibility is that the segment spans multiple threads across workgroups (line 5-6). In this case, we also need to accumulate the last partial sum results of the previous workgroups. We resort to adjacent synchronization to avoid global synchronization as discussed in Section 3.2.3. At last, the final results is written back to global memory (line 12).

Strategy (2): 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 data 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 result cache with the help of the result entry information generated along with the BCCOO format.

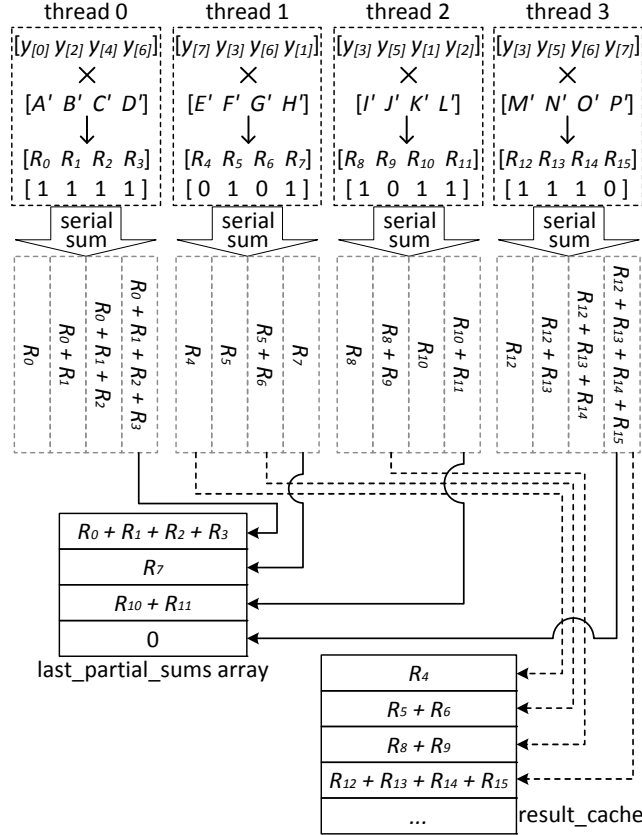


Fig. 11. Computing segmented sum: strategy (2), which uses a per-workgroup result cache to store segmented sums. The dashed blocks mean that the intermediate sums are not stored.

Using the matrix C in Figure 7(a) as an example, strategy (2) is illustrated in Figure 11. We assume that each thread-level tile contains 4 non-zero blocks and there are 4 threads in a workgroup. The result entry information shown in Figure 7 is used for updating the result cache. For example, as shown in Figure 7, the result entry for thread 1 and thread 2 is 0 and 2, respectively. Therefore, when thread 1 encounters the first row stop, it uses its current sum R_4 to update the entry 0 of the result cache. When thread 1 encounters the second row stop, it uses the sum R_5+R_6 to update the entry 1 of the result cache. In a sense, the result entry information partitions the result cache among different threads in a workgroup. When the number of row stops in a workgroup-level tile is larger than the result 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 start flags for the *last_partial_sums* array, in either strategy, each thread simply checks whether its bit flags contain a 0 (namely a row

stop). If so, its last partial sum should be a start for a segment in the *last_partial_sums* array. For the examples in Figure 10 and Figure 11, the start flags are [0, 1, 1, 1], since all threads except thread 0 process a tile containing a row stop. After updating the *last_partial_sums* array, all the threads in the workgroup perform a parallel segmented scan [Sengupta et al. 2007] on the *last_partial_sums* array using the start flags. In our example in Figure 10 or Figure 11, this parallel scan can be skipped since all the segment sizes are 1.

ALGORITHM 2: Combining the partial sums for strategy (2)

Input: *res_addr*: the first result location of the thread.

pre_partial_sum: the last partial sum of the previous workgroups.

b_res_addr: the first result location of the current workgroup.

cache_len: the length of the *result_cache*[]. *cache_end*: the number of the cached results.

workgroup_size: the number of threads in a workgroup.

Output: *results*[:]: the final results after combining the partial sums.

```

1 int i = local_thread_id;
2 if i == 0 then
3   | result_cache[0] += pre_partial_sum;
4   |
5   | workgroup-level-barrier();
6 if i != 0 then
7   | if Is there a row stop in the current thread? then
8     |   if res_addr - b_res_addr < cache_len then
9       |     | result_cache[res_addr - b_res_addr] += last_partial_sums[i - 1];
10      |   else
11        |     | results[res_addr] += last_partial_sums[i - 1];
12      |
13      |
14 while i < cache_end do
15   | results[i + b_res_addr] = result_cache[i];           // Write back in a coalesced way.
16   | i += workgroup_size;

```

ALGORITHM 2 illustrates how to generate the final output for strategy (2). In strategy (2), there are no per-thread intermediate sum arrays. Instead, there is a per-workgroup result cache. For thread 0, it updates the entry 0 of the result cache with the last partial sum from the previous workgroup (line 2-4). To avoid data race at the entry 0 of the result cache, a workgroup-level synchronization is used (line 5). Each thread except thread 0 first checks whether there are row stops in its thread-level tile (line 7). If so, it means that the thread has generated some partial sums corresponding to the row stops. Each thread only needs to process the partial sum at the first row stop (such as R_4 in the result cache in Figure 11). For subsequent row stops in the thread, the partial sums in the result cache are already the complete segment sums. Next, each thread except thread 0 adds the last partial sum from the previous thread to the partial sum at the first row stop, which is stored either in result cache (line 8-9) or in global memory (line 10-11). For example, $R_0+R_1+R_2+R_3$ from the *last_partial_sums* array is added to R_4 in the result cache in Figure 11). After the result cache is updated, it is written back to global memory in a memory-coalescing way (line 14-16).

Strategy (3): Intel MIC and GPUs have different hardware architectures and different performance optimization tricks. Thus, we propose the third strategy specifically for MIC, as illustrated in Figure 12. To achieve high throughput on MIC based on OpenCL, two key points should be noticed. Firstly, the local memory in OpenCL, which

is usually used as a high performance shared buffer, has no corresponding hardware implementation in MIC. The data in local memory is actually put in the global memory of MIC, with extra software overhead. As illustrated in Figure 12, the intermediate sums are not stored, and therefore the local memory consumption is avoided. All the segmented sums will be written back to the global memory directly with the help of the result entry information. Besides, the *last_partial_sums* array is also stored in the global memory.

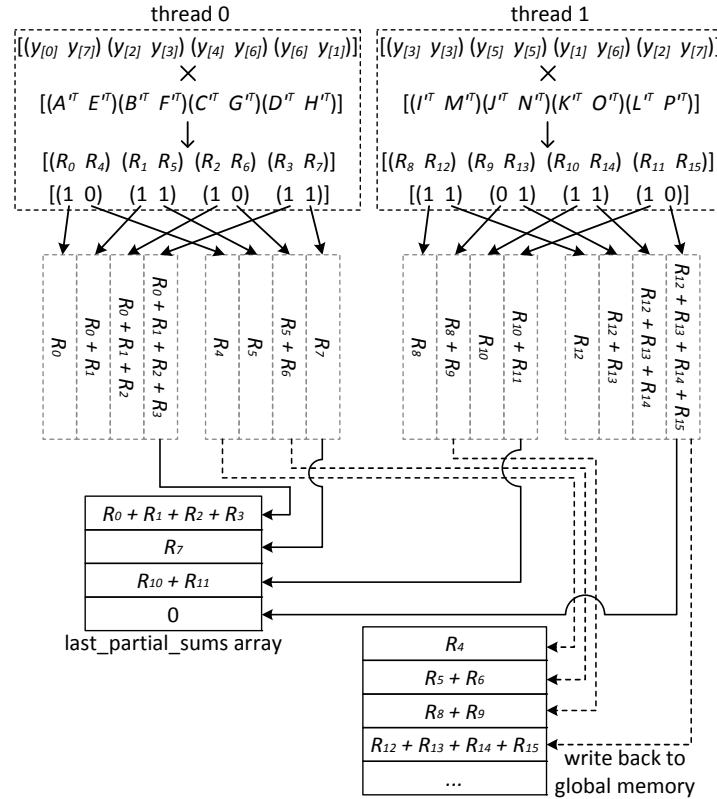


Fig. 12. Computing segmented sum: strategy (3), proposed only for Intel MIC. The dashed blocks mean that the intermediate sums are not stored.

Secondly, the 512-bit SIMD instruction in MIC should be fully utilized. To achieve this, besides the offline transpose for the data value array used in strategy (2), we conduct another inner-block transpose for the BCCOO format. In Figure 12, A'^T means the transposed block of A' . After the inner-block transpose, each thread can execute both the multiplication and addition operations in SpMV in a SIMD manner. The width of the SIMD instruction is decided by the width of each block (after transposition). When the width of each block is small, the potential performance of the 512-bit SIMD instruction cannot be fully exploited. However, enlarging the width of each block will bring more zero elements, which reduces the sparse matrix compression efficiency. To solve this dilemma, we propose a method of workload coalescing. As shown in Figure 12, the workload of two threads in Figure 11 is coalesced to be done by one thread. Take thread 0 in Figure 12 as an example, thread 0 processes two non-zero blocks (i.e., A'^T and E'^T) simultaneously, and thus the width of SIMD is doubled. The number of

threads, whose workload is coalesced to be done by one thread, is called *bunch_size* in the following content.

ALGORITHM 3: Combining the partial sums for strategy (3)

Input: *bunch_size*: the bunch size. *i*: the local thread id.
res_addr[:]: the first result locations for all *bunch_size* slices of the coalesced workload.
pre_partial_sum: the last partial sum of the previous workgroups.
Output: *results*[:]: the final results after combining the partial sums.

```

1 int i = local_thread_id;
2 if i == 0 then
3   | results[res_addr[0]] += pre_partial_sum;
4   |
5   | workgroup-level-barrier();
6 if i != 0 then
7   | if Is there a row stop in the 0-th slice of the coalesced workload? then
8   |   | results[res_addr[0]] += last_partial_sums[i - 1][bunch_size - 1];
9   |
10  | for int n = 1; n < bunch_size; n++ do
11  |   | if Is there a row stop in the n-th slice of the coalesced workload? then
12  |   |   | results[res_addr[n]] += last_partial_sums[i][n - 1];
13  |   |

```

ALGORITHM 3 illustrates how to generate the final output for strategy (3). In strategy (3), the intermediate results are written back to the global memory directly without being cached. Recall that there are total *bunch_size* slices of workload coalesced to be done by one thread. For thread 0, it updates the entry 0 of the result cache with the last partial sum from the previous workgroup (line 2-4). Each thread except thread 0 first checks whether there are row stops in its 0-th slice of the coalesced workload. If so, the thread accumulates the last partial sum (the partial sum of the (*bunch_size*-1)-th slice workload of the last thread) to the corresponding result entry in global memory (line 7-8). For example, $R_0+R_1+R_2+R_3$ from the *last_partial_sums* array is added to R_4 in the global memory in Figure 12. Then, the current thread checks whether there are row stops in the *n*-th slice workload of its own, where $n \in [1, \text{bunch_size}-1]$. If so, the current thread accumulates the partial sum of the (*n*-1)-th slice to the first result location of the *n*-th slice (line 11-12).

3.2.3. Accumulating partial sums across workgroups. As discussed in Section 3.2.2, for segments spanning multiple workgroups, the partial sums should be accumulated across the workgroups. The last workgroup, which contains the row stop, needs to accumulate the partial sums of previous workgroups. 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. The 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. We use a global memory array *Grp_sum* to accumulate partial sums across workgroups. We use another initialized *Sync_flag* array, whose elements have one-to-one correspondence to *Grp_sum* elements, for synchronization. Workgroup 0 updates the first entry *Grp_sum*[0] with its last partial sum, and then updates the corresponding element in the *Sync_flag* array. For a subsequent workgroup with id *X*, if it does not contain a row stop, it waits for the entry *Sync_flag*[*X*-1] to be 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 these chained updates and directly updates $Grp_sum[X]$ with its last partial sum. This approach is called adjacent synchronization [Yan et al. 2013].

However, to use the approach of adjacent synchronization on some platforms (such as AMD GPUs) with cache enabled by default, atomic primitives have to be used to guarantee that each workgroup can access the latest values of $Sync_flag[X-1]$ and $Grp_sum[X-1]$ once being updated by the previous workgroup. For single precision, we use the atomic primitive $atomic_xchg$. However, there is no corresponding atomic primitives which directly support double precision. By enabling $cl_khr_int64_base_atomics$, we can use the 64-bit atomic primitive $atom_xchg$ which only supports the types of $long$ and $ulong$. Using the function $as_ulong()$, we treat the values and variables of $double$ type as $ulong$ type, which can then be used in $atom_xchg$. So far, we can use the atomic primitive on double-precision variables and values for adjacent synchronization without losing precision.

4. AUTO-TUNING FRAMEWORK

As discussed in Sections 2 and 3, we propose BCCOO and BCCOO+ formats for sparse matrices, and three new strategies to compute segmented sums/scans for SpMV. To find the optimal solution for a sparse matrix, we build an offline auto-tuning framework to select the format, the computing strategy, and 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 in single precision on GPUs. 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 for GPUs and MIC are listed in Table I and Table II respectively. 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, Req_size and ShM_size . Besides, the strategy (3), which is proposed specifically for Intel MIC, includes $bunch_size$ as a tuning parameter.

As shown in Table I and Table II, there are many parameters to tune, which form a relatively large search space. Although the framework mainly aims at iterative SpMV, we try to minimize the overhead of offline auto-tuning using 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 different matrices. Third, we prune the search space using the following heuristics: (1) Since the memory footprint is highly dependent on block dimensions, we only select the block dimensions with the top 4 minimum memory footprints; (2) We always use the texture memory for the multiplied vector in single precision, and always use offline transpose; (3) We reduce the searching range of the result cache size for the second strategy; (4) We set the shared memory size as 0 for the per-thread intermediate sums array for the first strategy; (5) We use the BCCOO+ format only when the width of the sparse matrix is larger than the height; (6) We reduce the searching range of the thread-level tile size according to the dimensions of the sparse matrix. With these optimizations, our auto-tuning framework only runs 28 iterations of SpMV on average for the 20 sparse matrices in our study (shown in Table III). The average auto-tuning time is 3.6 seconds on average for the 20 sparse matrices on a machine with an Intel(R) Xeon(R) E5-2660 @ 2.20GHz (only one core is used) and an NVIDIA GTX680 GPU. Compared with the optimal results obtained from an exhaustive search of the parameters listed in Table I, our auto-tuning results are identical to the optimal

Table I. Tunable parameters of the auto-tuning framework on GPUs

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 (Reg_size)	0, 8, 16, 32
	Shared memory for the per-thread intermediate sums array (ShM_size)	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

Table II. Tunable parameters of the auto-tuning framework on MIC





















Parameter Name	Possible Values
Matrix format	BCCOO
Col_index compress	Yes, No
Block width	1, 2, 4
Block height	1, 2, 4
Data type for the bit flag array	Unsigned char, unsigned short, unsigned int
Workgroup size	1,2,4,8,16,32
Thread-level tile size	32~1024
Bunch size	1,2,4

ones on NVIDIA GTX680 and AMD W8000 GPUs. On NVIDIA GTX480, 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. As we can expect, a finer grain parameter selection may further improve the performance. In addition, we find that *block width*, *block height*, *thread-level tile size*, and *bunch size* are the parameters that top affect the performance among all the parameters.

5. EXPERIMENTAL METHODOLOGY

Our experiments have been performed on six different platforms - Nvidia GTX680, Nvidia GTX480 GPU, Tesla K20, GeForce Titan X, AMD FirePro W8000 and Intel MIC SE10P. We use a total of 20 sparse matrices for performance evaluation. Table III summarizes the information of the sparse matrices. These matrices have been widely used in previous works [Bell and Garland 2009], [Choi et al. 2010], [Monakov et al. 2010], [Su and Keutzer 2012], [Liu and Vinter 2015], [Williams et al. 2009]. All matrices except Dense are downloadable at the University of Florida Sparse Matrix Collection [Davis and Hu 2011]. In our experiments, we also use CUSPARSE V7.0 [NVIDIA 2014], CUSP [Bell and Garland 2009], clSpMV [Su and Keutzer 2012], and CSR5 [Liu and Vinter 2015] for performance comparisons. CUSPARSE supports three formats

Table III. 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

HYB, BCSR, and CSR. We manually searched the best performing configuration 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 also tested all the single formats and chose the best performing one for each matrix. Since the CUDA code is not supported on AMD platforms, we only compared our scheme with clSpMV on AMD FirePro W8000. On Intel MIC, we compared our scheme with CSR5 [Liu and Vinter 2015]. The code of our proposed framework is available at <http://code.google.com/p/yaspmv/>.

6. EXPERIMENTAL RESULTS

6.1. Memory footprint size comparison between different formats

We evaluate the impact of our proposed BCCOO/BCCOO+ format on memory bandwidth. In BCCOO/BCCOO+ format, all the information, including the bit flag array, the col_index array, the data value array, and the auxiliary information described in Section 2.4, is only read once. We assume that it is also the case for all the other formats for comparison. Therefore, we can simply use the total size the arrays to show the memory footprint of each format. The results are shown in Table IV. 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 IV). From Table IV, we find that BCCOO/BCCOO+ format significantly reduces the storage size of various sparse matrices. BCCOO/BCCOO+ format reduces the storage size by 40% on average compared with the COO format, 31% on average compared with the best single format among all the 9 formats in clSpMV, and 21% on average compared with the COCKTAIL format.

6.2. Performance on NVIDIA and AMD GPUs

We first 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,

Table IV. 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

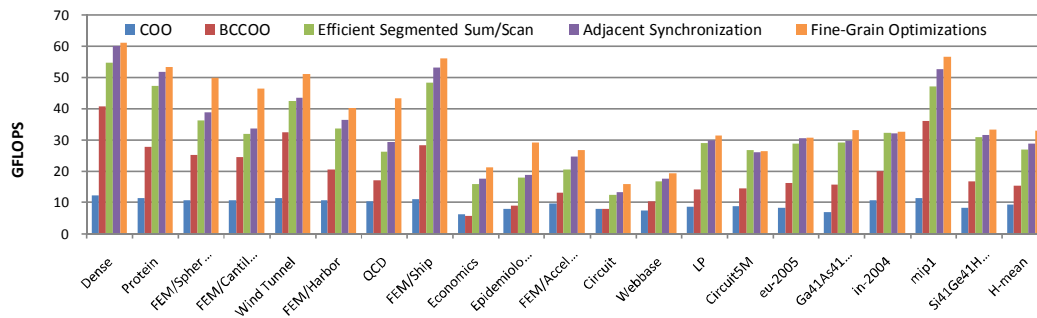


Fig. 13. Performance contributions from different optimization techniques on GTX680 (in single-precision).

which consist of (a) the use of the short data type for the *col_index* array and (b) skip the parallel scan on a *last_partial_sums* array if possible. The results are shown in Figure 13. 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'), in which the global synchronization is used to accumulate partial sums across workgroups. We then replace the global synchronization by the adjacent synchronization (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 sum/scan for SpMV.

Next, we compare the performance of our proposed scheme with the state-of-the-art techniques on GPUs. The single-precision performance on GTX680 are shown in Figure 14, in which our approach is labeled by 'yaSpMV'. We can see that yaSpMV outperforms the existing schemes for all the matrices except Dense. The Dense matrix prefers a block size of 2x8 in the BCSR format, which is selected as the best single format of clSpMV for Dense matrix. However, our auto-tuning framework limits the maxi-

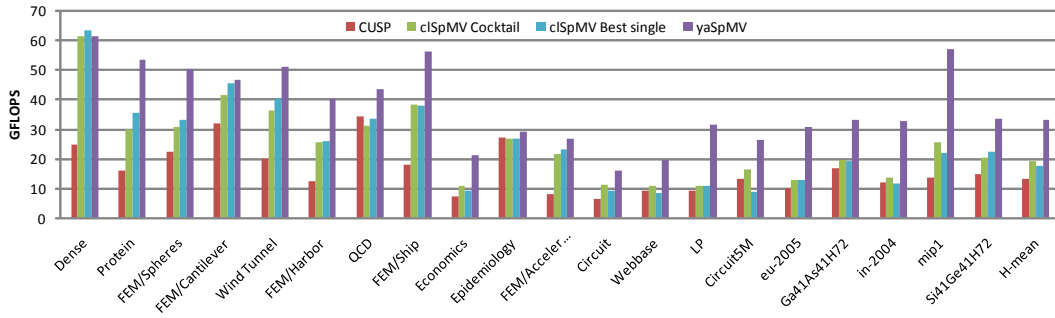


Fig. 14. Single-precision performance comparison between our proposed scheme (labeled 'yaSpMV') and CUSP, clSpMV-best single, and clSpMV-COCKTAIL on GTX680 GPUs.

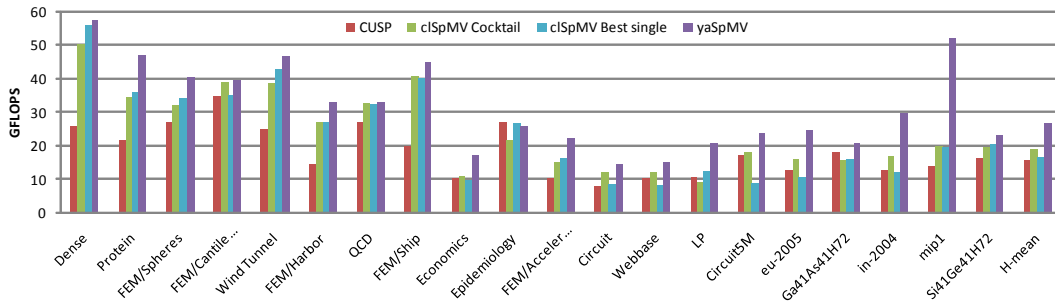


Fig. 15. Single-precision performance comparison between our proposed scheme (labeled 'yaSpMV') and CUSP, clSpMV-best single, and clSpMV-COCKTAIL on GTX480 GPUs.

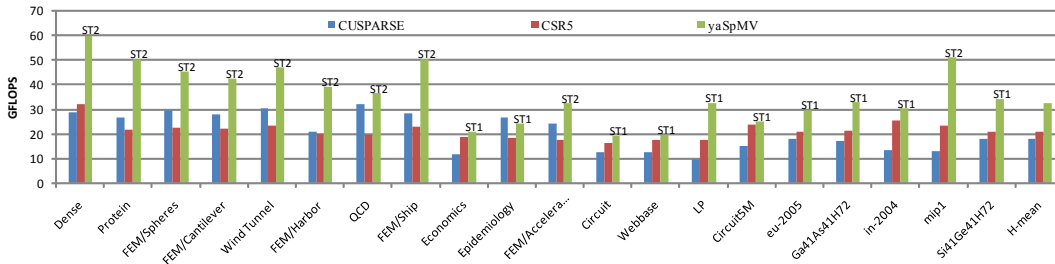


Fig. 16. Single-precision performance comparison between our proposed scheme (labeled 'yaSpMV') and CUSPARSE V7.0, and CSR5 on Nvidia Tesla K20. We also mark the performance winner between strategy (1) (labeled 'ST1') and strategy (2) (labeled 'ST2') for yaSpMV.

mal block height to 4, thereby achieving sub-optimal performance. Using the harmonic mean (H-mean) as the average throughput, 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 gain of yaSpMV over clSpMV COCKTAIL is achieved on matrix LP (195%). Compared with CUSPARSE, the highest performance gain of yaSpMV is achieved on matrix mip1 (229%).

We further evaluate the single-precision performance of SpMV on Nvidia GTX480 GPUs, Tesla K20, GeForce Titan X, and AMD FirePro W8000 GPUs. The results are shown in Figure 15, Figure 16, Figure 17 and Figure 18. On Nvidia GTX 480, our proposed yaSpMV achieve significantly higher performance than existing approaches (up to 162% better than clSpMV COCKTAIL and up to 150% better than CUSPARSE),

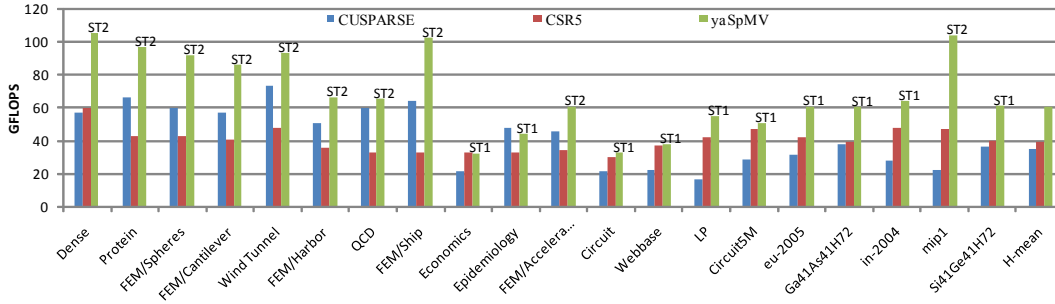


Fig. 17. Single-precision performance comparison between our proposed scheme (labeled 'yaSpMV') and CUSPARSE V7.0, and CSR5 on GeForce Titan X GPUs. We also mark the performance winner between strategy (1) (labeled 'ST1') and strategy (2) (labeled 'ST2') for yaSpMV.

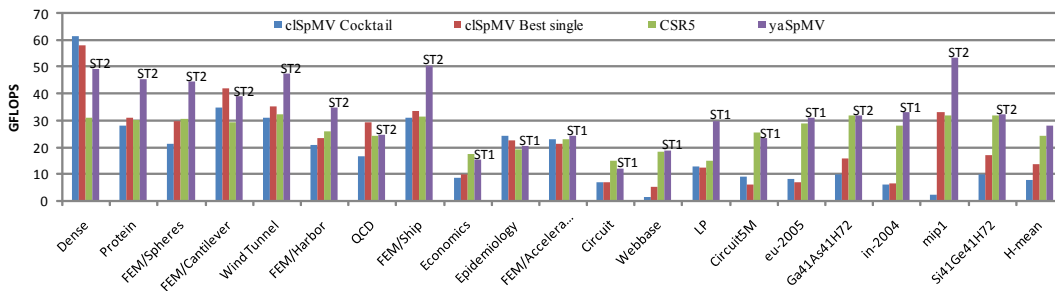


Fig. 18. Single-precision performance comparison between our proposed scheme (labeled 'yaSpMV') and clSpMV-best single, clSpMV-COCKTAIL, and CSR5 on AMD FirePro W8000 GPUs. We also mark the performance winner between strategy (1) (labeled 'ST1') and strategy (2) (labeled 'ST2') for yaSpMV.

as shown in Figure 15. The only exception is the Epidemiology matrix. It has 4 non-zeros on each row, which is a perfect fit for the ELL format. For this matrix, 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 H-mean, yaSpMV achieves a performance improvement of 40% over clSpMV COCKTAIL, 60% over clSpMV best single, 74% over CUSP, and 42% over CUSPARSE. As shown in Figure 16 and Figure 17, yaSpMV achieves the performance improvement of 65.8% on average on Tesla K20 and 73.7% on average on GeForce Titan X over CUSPARSE V7.0; and achieves the performance improvement of 56.4% on average on Tesla K20 and 53.6% on average on GeForce Titan X over the recently proposed format - CSR5.

As shown in Figure 18, on AMD FirePro W8000, yaSpMV performs better than clSpMV COCKTAIL format on most matrices, and the performance improvement is up to 2617% and on average 255%. Although there are only 9 matrices which our proposed yaSpMV performs better than the clSpMV best single format, yaSpMV also achieves a performance gain of 40% on average. Compared with CSR5, yaSpMV achieves a performance gain of 14.9% on average.

To further understand how strategy (1) and strategy (2) of yaSpMV perform on Tesla K20, GeForce Titan X, and AMD FirePro W8000, we mark the performance winner between these two strategies in Figures 16, 17, and 18. We find that both strategies are the potential winner for different sparse matrices, which demonstrates the necessity of the coexistence of the two strategies in our scheme.

We also evaluate the double-precision performance of SpMV on Tesla K20 and AMD FirePro W8000 GPUs. The results are shown in Figure 19, Figure 20, Figure 21 and

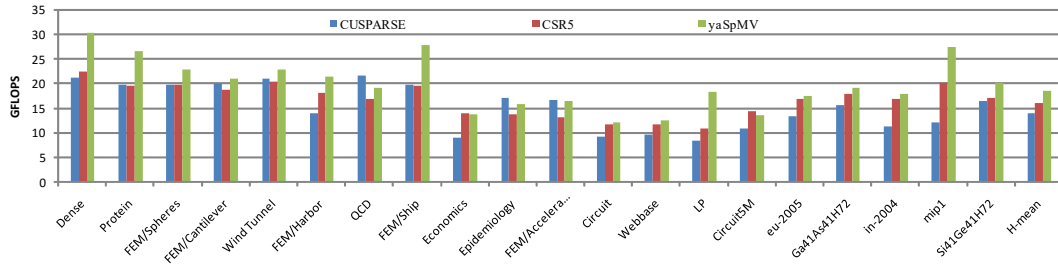


Fig. 19. Double-precision performance comparison between our proposed scheme (labeled 'yaSpMV') and CUSPARSE V7.0, and CSR5 on Nvidia Tesla K20.

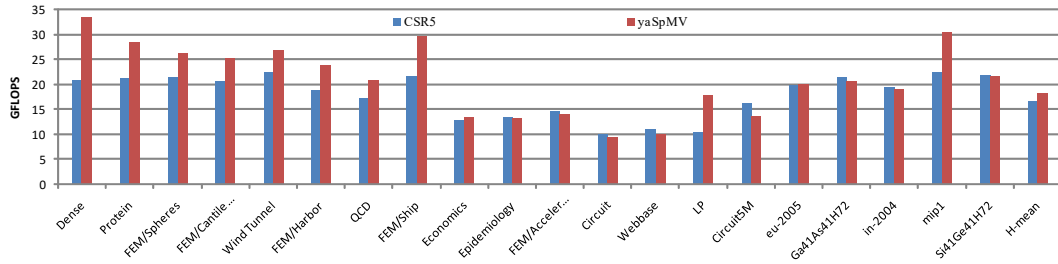


Fig. 20. Double-precision performance comparison between our proposed scheme (labeled 'yaSpMV') and clSpMV-best single, clSpMV-COCKTAIL and CSR5 on AMD FirePro W8000 GPUs.

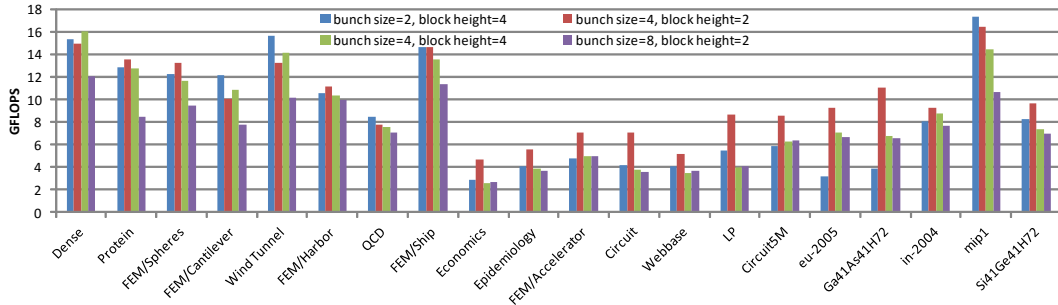


Fig. 21. Double-precision performance of our proposed scheme with different 'bunch size' and 'block height' on Intel MIC.

Figure 22. As shown in Figure 19, on Tesla K20, yaSpMV achieves the performance improvement of 34.0% on average over CUSPARSE V7.0 and 16.2% on average over CSR5. As shown in Figure 20, on AMD FirePro W8000, yaSpMV achieves the performance improvement of 9.7% on average over CSR5.

6.3. Performance on Intel MIC

Figure 21 presents the results of our proposed scheme with different 'bunch size' and 'block height' on Intel MIC. We can see that under different configurations of these two parameters, the performance varies largely. This is because the multiplication of these two parameters determines the width of SIMD instructions on Intel MIC, as discussed in Section 3.2.2. When it fits to the width of 512-bit with less zero elements (determined by the value of 'block height'), it can get the best performance. Thus, we need the auto-tuning framework to select the best configuration. As shown in Figure 22, the auto-

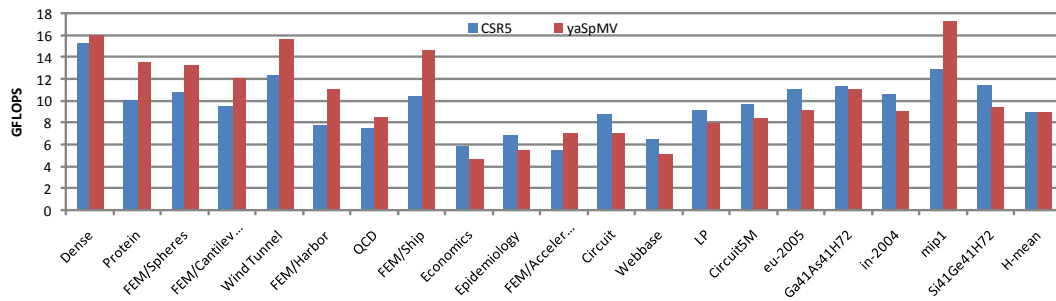


Fig. 22. Double-precision performance comparison between our proposed scheme (labeled 'yaSpMV') and CSR5 on Intel MIC.

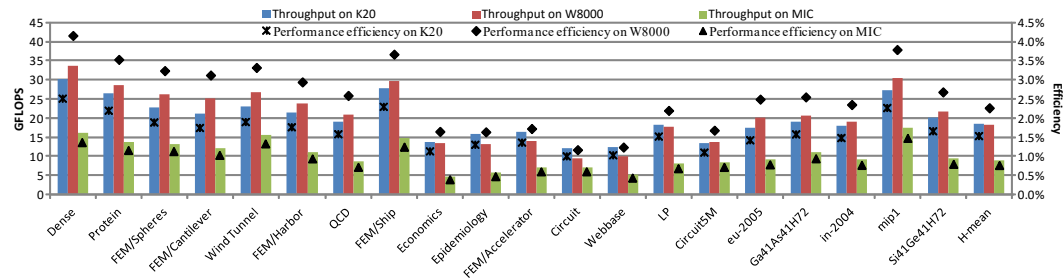


Fig. 23. Double-precision performance of yaSpMV on Nvidia Tesla K20, AMD FirePro W8000 and Intel MIC.

tuned yaSpMV gets an average of 8.97 Gflops for the sparse matrixes on Intel MIC in double precision, which is comparable with CSR5 (an average of 9.01 Gflops).

We also compare the performance between GPUs (Nvidia Tesla K20 and AMD FirePro W8000) and Intel MIC in the context of yaSpMV, and the results is presented in Figure 23. Note that the double-precision peak performance of Nvidia Tesla K20, AMD FirePro W8000 and Intel MIC is 1.17 Tflops, 806 Gflops and 1.011 Tflops, respectively. We find that both the throughput and the performance efficiency of GPUs are higher than Intel MIC. The average performance efficiency of yaSpMV on Nvidia K20, AMD W8000, and Intel MIC is 1.54%, 2.27%, and 0.82%, respectively. The lower efficiency of yaSpMV on Intel MIC is caused by the following reasons: (1) BCCOO is a block-based format to exploit the register reusing. However, register blocking leads to low SIMD efficiency on Intel MIC [Liu et al. 2013]. Although we coarsen the workload of each thread to improve the SIMD efficiency, it still brings some performance penalty; (2) Intel MIC is lack of texture cache, which is more useful for the irregular accesses of the multiplied vector. Thus, the load imbalance of each thread caused by irregular accesses is more severe on Intel MIC.

7. RELATED WORK

Bolz et al. first introduced the GPUs for SpMV [Bolz et al. 2003]. Bell and Garland [Bell and Garland 2009] implemented several well-known formats on Nvidia GPUs. These formats include DIA, ELL, CSR, COO and a hybrid format HYB, which combines the advantage of the ELL and COO formats. Su et al. [Su and Keutzer 2012] proposed the COCKTAIL format, which uses different formats to represent different partitions of a matrix. Vázquez et al. [Vázquez et al. 2011] proposed a derivative format of ELLPACK, ELL-R. They use an auxiliary array to store the row lengths. Alexander et al. [Monakov et al. 2010] proposed the Sliced ELL format (SELL). They horizontally

partition the original matrix into several slices to reduce the filling zeros. Compared with the ELL format, the ELL-R and SELL formats have less padding zeros. Kreutzer et al. [Kreutzer et al. 2014] proposed SELL-C- σ , which is a variant of SELL. To further reduce the padding overhead, the rows are sorted by the number of non-zero entries within a "sorting scope" σ . SELL-C- σ is SIMD-friendly, and suitable for different architectures, including GPU, Intel MIC, and CPU. Compared with SELL-C- σ , yaSpMV exploits register reusing and may have less padding zeros. However, the block-based design of yaSpMV makes it less suitable for Intel MIC than SELL-C- σ . We did not compare the performance of yaSpMV with SELL-C- σ , since the source code is not available to us yet. Liu et al. [Liu et al. 2013] proposed the ESB format for Intel MIC, which extends the ELLPACK format with finite-window sorting for high SIMD efficiency, a bit array to encode nonzero locations for lower padding overhead, and column blocking for good locality. The column blocking method used in ESB motivates us to propose the BCCOO+ format to improve the locality when accessing the multiplied vector.

Based on the CSR format, Kozaa et al. [Koza et al. 2012] 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. Choi et al. implemented the BCSR and BELL formats on GPUs [Choi et al. 2010] and proposed an auto-tuning framework. CSR-Adaptive [Greathouse and Daga 2014] is proposed to exploit the performance of SpMV with CSR format on GPUs. We tried to compare the performance of yaSpMV with CSR-Adaptive, which is implemented in ViennaCL. However, loading the sparse matrix from the disk to memory in ViennaCL is too time consuming (more than several days for some large matrices). Thus, we terminated the experiments on this format. Daga and Greathouse [Daga and Greathouse 2015] further improved CSR-Adaptive using novel reduction techniques and proposed a new SpMV algorithm for the irregular matrices with very long rows. Liu et al. presented CSR5 [Liu and Vinter 2015], which is insensitive to the sparsity structure of the input matrix and features fast format convention. Compared with CSR5, yaSpMV has almost equivalent performance on Intel MIC, and has an advantage on Nvidia and AMD GPUs. Liu and Schmidt [Liu and Schmidt 2015] proposed LightSpMV, which uses the standard CSR format and improves the performance by fine-grained dynamic distribution of matrix rows over warps and vectors.

There are some works focusing on compression and reordering techniques as well [Buluç et al. 2011], [Pichel et al. 2012]. 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. [Tang et al. 2013] 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.

Blelloch et al. [Blelloch et al. 1993] first introduced the segmented operations to SpMV on vector multiprocessors. Harris [Harris et al. 2007] 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 [Yan et al. 2013], the performance is limited. Baskaran et al. [Baskaran and Bordawekar 2008] implemented a more efficient segmented scan based SpMV using the matrix based scan [Dotsenko et al. 2008]. However, their scan-based implementation also is outperformed by their alternative implementations [Baskaran and Bordawekar 2008]. 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 propose the new BC-

COO/BCCOO+ format to drastically reduce the bandwidth requirement and efficient segmented sum/scan algorithms on different many-core architectures. 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 many-core architectures, including GPUs and Intel MIC. 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, for better locality of the accesses to the multiplied vector. Second, 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. Third, we propose an auto-tuning framework to further improve the performance with low overhead.

Our performance results from a set of 20 sparse matrices show that our proposed framework significantly advances the state-of-the-art SpMV schemes. In single-precision, yaSpMV outperforms CUSPARSE 7.0 by 65.8% on average on Tesla K20, by 73.7% on average on GeForce Titan X; outperforms clSpMV COCKTAIL format by 40% on average on GTX480 GPUs, by 70% on average on GTX680 GPUs, by 255% on average on AMD FirePro W8000 GPUs; and outperforms CSR5 by 56.4% on average on Tesla K20, by 53.6% on average on GeForce Titan X, by 14.9% on average on AMD FirePro W8000. In double-precision, yaSpMV outperforms CUSPARSE V7.0 by 34.0% on average on Tesla K20; and outperforms CSR5 by 16.2% on average on Tesla K20, by 9.7% on average on AMD FirePro W8000. On Intel MIC, yaSpMV has almost equivalent performance compared with CSR5.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China under Grant No. 61502450, Grant No. 61432018, Grant No. 61521092, and Grant No. 61272136; National Key Research and Development Program of China under Grant No.2016YFB0200803; NSF project 1216569; and a gift fund from AMD Inc. Shigang Li and Shengen Yan are the corresponding authors. The authors would like to thank the Supercomputing Center of CAS for providing free Intel MIC machines.

REFERENCES

- Muthu Manikandan Baskaran and Rajesh Bordawekar. 2008. Optimizing sparse matrix-vector multiplication on GPUs using compile-time and run-time strategies. *IBM Reserach Report, RC24704 (W0812-047)* (2008).
- Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 18.
- Guy E Blelloch. 1989. Scans as primitive parallel operations. *Computers, IEEE Transactions on* 38, 11 (1989), 1526–1538.
- Guy E Blelloch, Michael A Heroux, and Marco Zagha. 1993. *Segmented operations for sparse matrix computation on vector multiprocessors*. Technical Report. DTIC Document.
- Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. 2003. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In *ACM Transactions on Graphics (TOG)*, Vol. 22. ACM, 917–924.
- Aydın Buluç, Samuel Williams, Leonid Oliker, and James Demmel. 2011. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 721–733.
- Jee W Choi, Amik Singh, and Richard W Vuduc. 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *ACM Sigplan Notices*, Vol. 45. ACM, 115–126.

- Mayank Daga and Joseph L Greathouse. 2015. Structural Agnostic SpMV: Adapting CSR-Adaptive for Irregular Matrices. In *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*. IEEE, 64–74.
- Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1.
- Yuri Dotsenko, Naga K Govindaraju, Peter-Pike Sloan, Charles Boyd, and John Manferdelli. 2008. Fast scan algorithms on graphics processors. In *Proceedings of the 22nd annual international conference on Supercomputing*. ACM, 205–213.
- Joseph L Greathouse and Mayank Daga. 2014. Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 769–780.
- Mark Harris, John Owens, Shubho Sengupta, Yao Zhang, and Andrew Davidson. 2007. CUDPP: CUDA data parallel primitives library. (2007).
- Zbigniew Koza, Maciej Matyka, Sebastian Szkoła, and Łukasz Mirosław. 2012. *Compressed multiple-row storage format*. Technical Report.
- Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R Bishop. 2014. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units. *SIAM Journal on Scientific Computing* 36, 5 (2014), C401–C423.
- ShiGang Li, ChangJun Hu, JunChao Zhang, and YunQuan Zhang. 2015. Automatic tuning of sparse matrix-vector multiplication on multicore clusters. *Science China Information Sciences* 58, 9 (2015), 1–14.
- Weifeng Liu and Brian Vinter. 2015. Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 339–350.
- Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. 2013. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM, 273–282.
- Yongchao Liu and Bertil Schmidt. 2015. LightSpMV: faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 82–89.
- Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. 2010. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *High Performance Embedded Architectures and Compilers*. Springer, 111–125.
- CUDA NVIDIA. 2014. CUSPARSE library. *NVIDIA Corporation, Santa Clara, California* (2014).
- Juan C Pichel, Francisco F Rivera, Marcos Fernández, and Aurelio Rodríguez. 2012. Optimization of sparse matrix-vector multiplication using reordering techniques on GPUs. *Microprocessors and Microsystems* 36, 2 (2012), 65–77.
- Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D Owens. 2007. Scan primitives for GPU computing. In *Graphics hardware*, Vol. 2007. 97–106.
- John E Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering* 12, 1-3 (2010), 66–73.
- Bor-Yiing Su and Kurt Keutzer. 2012. clSpMV: A cross-platform OpenCL SpMV framework on GPUs. In *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 353–364.
- Wai Teng Tang, Wen Jun Tan, Rajarshi Ray, Yi Wen Wong, Weiguang Chen, Shyh-hao Kuo, Rick Siow Mong Goh, Stephen John Turner, and Weng-Fai Wong. 2013. Accelerating sparse matrix-vector multiplication on GPUs using bit-representation-optimized schemes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 26.
- Sain-Zee Ueng, Melvin Lathara, Sara S Baghsorkhi, and W Hwu Wen-mei. 2008. CUDA-lite: Reducing GPU programming complexity. In *Languages and Compilers for Parallel Computing*. Springer, 1–15.
- Francisco Vázquez, José-Jesús Fernández, and Ester M Garzón. 2011. A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurrency and Computation: Practice and Experience* 23, 8 (2011), 815–826.
- Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2009. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Comput.* 35, 3 (2009), 178–194.
- Shengen Yan, Guoping Long, and Yunquan Zhang. 2013. StreamScan: fast scan algorithms for GPUs without global barrier synchronization. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 229–238.