CSR2: A New Format for SIMD-accelerated SpMV

Haodong Bian, Jianqiang Huang, Runting Dong, Lingbin Liu and Xiaoying Wang

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

March 3, 2020
Abstract—SpMV (Sparse matrix-vector multiplication) has attracted the attention of researchers in related fields at home and abroad. Of course, improving SpMV performance has also been a research hot spot for researchers in related fields. In this paper, we propose a new sparse matrix storage format CSR2 (Compressed Sparse Row 2) suitable for SIMD (Single Instruction Multiple Data)-accelerated SpMV. First, the format operation of CSR2 is easy to implement and has a low overhead of conversion. Second, CSR2 is a new single format and suitable for use on processor platforms with SIMD vectorization. We compare the SpMV algorithm based on CSR2 with the one based on the current most advanced single format CSR5 (Compressed Sparse Row 5) on two mainstream high-performance processors: Intel Core i7-7700HQ CPU and Intel Xeon CPU E5-2670 v3. We choose 10 sets of regular matrices and 3 sets of irregular matrices to be used as benchmark suit. Experiments show that for the 13 sets of regular and irregular matrices in the benchmark suite, CSR2 has an average performance improvement of more than 50% compared to CSR5 (up to 125% on Intel Core i7-7700HQ CPU and 303% on Intel Xeon CPU E5-2670 v3). For applications with multiple iterations, in reality, using our CSR2 can bring low-overhead format conversion and high-throughput computing performance.

Index Terms—SpMV, SIMD Vectorization, CSR2, CSR5, Storage Formats, CPU

I. INTRODUCTION

The use of sparse matrix-vector multiplication (SpMV) has been pervasive. In the field of graph computing, it is vital to have an architecture for large-scale graph analysis, and many graph computing algorithms are inseparable from the support of SpMV [1]-[2]. In the field of deep learning, we can also see the existence of SpMV, which is used in the optimization of some neural network algorithms such as DNN (Deep Neural Networks) [3]-[4]. In the field of linear algebra, some scholars have developed high-performance processing systems specifically for linear algebra, so SpMV is also used very frequently [5]-[6]. As for solving eigenvalue problems, the emergence of SpMV is inevitable, because in some complex formulas, the performance of SpMV often directly affects the overall performance of the problem solving [7]-[8].

Moreover, a series of research fields that involve complex computing and are closely related to SpMV, such as fluid dynamics, require high-performance solution systems and methods [9]-[10]. Therefore, in most cases, SpMV often occupies a pivotal position as long as it is needed. The performance of SpMV has always been a research focus for researchers in related fields. For example, in the PageRank algorithm, the core that affects program performance is SpMV. However, even with the rapid development of hardware performance today, people still cannot stand only to improve single-core performance to meet the needs of program performance. Therefore, for SpMV of fixed complexity, most users use multi-core CPU and GPU and other processor architectures to accelerate SpMV calculations in parallel to obtain better performance.

In the rapid development of the multi-core era, there are also many types of processors. Of course, different processors have different effects on program performance. On the SW26010 platform, parallelizing SpMV by efficiently compressing compressed sparse row (CSR) can achieve considerable performance improvements [11]-[13]. On the multi-core and multi-threaded high-performance processor owned by Intel Xeon Phi, the performance improvement of SpMV or SpMM can be achieved by saturating the memory bandwidth [14]-[15],[29]. On the NVIDIA GPUs platform, SpMV is accelerated based on the ELLPACK-R format, and a significant performance speedup is obtained [16]-[20]. On the FPGA platform, partitioning effectively solves the limited buffer under large-scale data, which can accelerate SpMV [21]. On the AMD GPU platform, the PageRank ranking algorithm is effectively accelerated and has a perfect effect [22]-[23].

Secondly, the parallel effects of SpMV calculation of different data sets in different formats are also very different. Currently, there are two types of sparse matrix storage methods: single format and mixed format. The operation of a single format is easier to implement and understand than the mixed format. Compressed sparse row 5 (CSR5) [24] is one of the rare formats in existence that can achieve not only higher calculations but also load balancing under low time overhead. Among the many benchmarks for comparing SpMV performance, in addition to the common CSR, Coordinate (COO), ELLPACK (ELL), Diagonal (DIA), and other formats, researchers often use CSR5 as one of the benchmarks for performance comparison. In addition to CSR5, there are also some single formats such as CSX [25], BCCOO [26], which have good performance improvements. The use of mixed formats can achieve more effective and balanced performance improvement compared with the single formats. For example, ELLPACK-RP [27] can be used to reduce load imbalance to improve the performance of SpMV.

1The source code of this work is downloadable at https://github.com/nulidangxueshen/CSR2
Moreover, the ability to give full play to processor performance is also the key to SpMV calculation speed improvement. In addition to using OpenMP in multiple threads and MPI in multi-machine communication, the optimization of the instruction set is also an indispensable part of the overall program performance improvement. At present, most processors support extended 256bit, extended 512bit, or even higher single instruction multiple data (SIMD) vectorization technology, which allows programs to achieve a certain degree of performance improvement based on multi-threading and multi-processing. For example, the AVX2 and AVX512 instruction set technologies under the Intel architecture can bring good performance improvements to programs [28]-[30].

In order to make full use of SIMD vectorization to accelerate SpMV, we have designed compressed sparse row 2 (CSR2), a sparse matrix compression format suitable for SIMD-enabled platforms to accelerate SpMV. This is a new storage format extended from CSR. As for CSR2, three arrays are retained based on CSR, while the values of the corresponding elements may change, which is also designed to take full advantage of SIMD vectorization. Of course, CSR2 occupies more storage space than CSR generally. However, the parameter control can make CSR2 expand little space-based on CSR with better performance improvement. From the test results, we found that CSR2 occupies a storage space that is equivalent to or less than that of CSR5. At the same time, the implementation of CSR2 is convenient and straightforward, and the conversion overhead is equivalent to or less than that of CSR5.

In this article, we have made the following three contributions:

1) We propose a new format CSR2, which is suitable for SIMD-enabled processors to accelerate SpMV. This format has lower conversion overhead and smaller space occupation and can make full use of SIMD vectorization to achieve parallel execution with better performance.

2) We analyze the principle of CSR2 in detail and give a parallel implementation algorithm of SpMV based on CSR2.

3) On Intel Xeon CPU E5-2670 v3 and Intel Core i7-7700HQ CPU platforms, we use the performance of the SpMV parallel program implemented based on CSR5 format as a test benchmark to compare the one based on the CSR2 format we designed. A benchmark suite consisting of 13 sets of data was selected as performance test data. Finally, we made a detailed analysis of the experimental results.

By using the Benchmark suite consisting of 10 sets of regular matrices and 3 sets of irregular matrices, we find that CSR2 has better performance than CSR5 on the parallel implementation of SpMV, and its conversion overhead is also equivalent to or less than CSR5. During the actual running process of the program test, we found that the physical memory space occupied by CSR2 is equivalent to or less than CSR5 by using the RES parameter under the Htop test tool. At the same time, the implementation of CSR2 is convenient and easy to understand, and the conversion overhead is equivalent to or less than that of CSR5.

II. BACKGROUND

A. PageRank

PageRank is a page ranking algorithm which is used by Google to reflect the importance of web pages. The core idea has two points: (1) If many other pages link a web page, it means that the web page is essential, so the PageRank value will be higher. (2) If the PageRank value of web pages is relatively high, then the PageRank value of the webpage to which it is linked will also increase to a certain extent. During the calculation of the algorithm, due to multiple iterations, the PageRank values of all linked webpages are 0s (the termination point problem), or the PageRank value of one of the linked webpages is 1, and the remains are 0s (trap problems) (Figure 1).

Fig. 1: End point and trap problem.

This algorithm has also undergone a series of improvements and has become a well-known classic algorithm. Equation (1) is the principle formula for calculating the PageRank value, where \( d \) represents the probability of viewing the current web page, \( p_i \) represents the \( i \)-th web page, \( N \) represents the total number of the linked web pages, \( L(p_j) \) represents the number of links from the \( i \)-th web page to others, and \( p_j \in M(p_j) \) represents the \( i \)-th web page belonging to the entire web page collection.

\[
PageRank(p_i) = \frac{1 - d}{N} + d \sum_{p_j \in M(p_j)} \frac{PageRank(p_j)}{L(p_j)} \tag{1}
\]

Equation (2) is a formula used in the real solution process, where \( V_i \) represents the vector obtained by the \( i \)-th iteration, \( \alpha \) represents the probability of viewing the current web page, \( M \) represents the proportion of links between webpages, and \( e \) represents the unit vector.

\[
V_i = \alpha MV_{i-1} + (1 - \alpha) e \tag{2}
\]

B. Sparse matrix vector multiplication

Matrix vector multiplication is one of the knowledge points in linear algebra, which is often used to solve the system of linear equations. Its mathematical expression is:

\[
A = BX \tag{3}
\]

In Equation (3), \( B \) represents a matrix, \( X \) represents a vector, and \( A \) represents the result of multiplying a matrix by a vector. Assuming that \( B \) is a \( N \times M \) matrix and \( X \) is a \( M \times 1 \) matrix, the resulting \( A \) should be a \( N \times 1 \) vector. The calculation
process of matrix-vector multiplication is shown in Figure 2 below. In the figure, a $4 \times 4$ matrix is multiplied by a $4 \times 1$ vector:

The process of the sparse matrix-vector multiplication algorithm in the common matrix format is shown in Algorithm 1, where matrix represents a $n \times m$ matrix, vector represents a vector of $m \times 1$, and ans represents the result of the product, which is a $n \times 1$ vector.

### Algorithm 1: Sparse matrix vector multiplication

1. for $i = 0$ to $n - 1$
2.   for $j = 0$ to $m - 1$
3.     ans[$i$] = ans[$i$] + matrix[$i$ * $n + j$] * vector[$j$]
4.   end for
5. end for

![Fig. 2: Sparse matrix vector multiplication.](image)

In Figure 2, we can find that there are a large number of zero elements in the matrix, and the appearance of these zero elements is worthless; on the contrary, it will also bring the extra calculation. SpMV is the core of the calculation of the PageRank algorithm. Since most of the matrices, in reality, are sparse, compressing the sparse matrix reasonably has become the key to reduce the complexity of the entire SpMV algorithm, and it is also the key to speed up the entire PageRank algorithm.

### III. RELATED WORK

A. The CSR storage format and SpMV algorithm

CSR is a matrix compression format with high storage efficiency based on behavioral order. This format consists of three arrays that store data elements:

1. Row offset pointer array: It is used to mark the offset distance between the first address and the first address of each row. For the N-row matrix, the row offset pointer array has a total of N + 1 elements;
2. Column index array: It is used to mark the sequence number of the column corresponding to each non-zero element, and the number of array elements (M) is equal to the number of non-zero elements in the matrix (Nonzero);
3. Value array: It is used to store the value of each non-zero element, and the number of array elements is equal to Nonzero;

The CSR format is shown in Figure 3.

Algorithm 2 is an $N \times M$ sparse matrix multiplied by an $M \times 1$ dense vector. The value array represents the matrix, the vec array represents the vector, and the answer array is the final result vector. It is assumed that the number of non-zero elements in the sparse matrix is $nzz$, so the time complexity of the algorithm is $O(nzz)$, which is much smaller than the time complexity of the ordinary matrix operations.

B. The CSR5 storage format and SpMV algorithm

The CSR5 [24] storage format is based on the serial CSR prefix summation algorithm and adds the idea of matrix partitioning to obtain a balanced load. As a matter of course, CSR5 that gets a balanced load has an excellent performance in both regular and irregular matrices. It stores the matrix value array and column index value array in CSR format in the main column order on the pre-divided block matrix. Besides, each matrix will have a corresponding auxiliary matrix. The bit_flag array is used to mark whether the current element is the first element of the block or the first element in the original matrix, which can easily find the value of the segmentation sum. y_offset is used to mark the position of the first T element of each column of the current block in row_ptr.

### Algorithm 3: Core of the CSR5-based SpMV for the tidth tile

1. for $i = 0$ to $w - 1$ in parallel do
2.   sum = 0
3.   for $j = 0$ to $\sigma - 1$ do
4.     ptr = tid * $\omega * \sigma + j * \omega + i$
5.     sum = sum + val[ptr] * x[col_val[ptr]]
6.     if */Tmp part*/ then
7.       tmp[i - 1] = sum
8.       sum = 0
9.     else if */Full part*/ then
10.      y[tile_ptr[tid] + y_offset[i]] = sum
11.      y_offset[i] = y_offset[i] + 1
12.     sum = 0
13.   end if
14. end for
15. last_tmp[i] = sum
16. end for

Furthermore, if there are no T elements, the y_offset value of the column is the sum of the number of T elements in the...
first few columns. **seg_offset** is used to mark whether there is a column in the block that is as same as a row in the matrix, which is set to facilitate CSR5 to use SIMD vectorization to process elements during processing blocks. The purpose of setting the **empty_offset** array is to use this array instead of the **y_offset** array value to ensure correct processing results when there is a situation where the current block spans a row or more than zero. Figure 4 shows the CSR5 format.

![CSR5 format](image)

Algorithm 3 is a part of the SpMV algorithm based on CSR5, where \( w \) represents the number of columns of the block, and \( \sigma \) represents the number of rows of the block. If the **bit_flag** of the first element of each column is F, the sum of the elements that are between the element at that position and the first T element will be stored in the array **tmp**, which is labeled with Tmp part in Algorithm 3. The sum of the elements that are between the first T element and the last element will be stored in the array **last_tmp**. For a certain column element in the block, if a complete row in the matrix appears, we use the Full part labeled in Algorithm 3. Algorithm 3 only shows the block product summation part of the SpMV algorithm based on CSR5.

### IV. The CSR2 Storage Format

#### A. Format Information

CSR2 is a new format improved on CSR and is suitable for the processors with SIMD function to accelerate SpMV. The CSR2 format has the following two modifications to the original CSR format:

1. We will divide the two arrays of value and **col_idx** by selecting appropriate block sizes, and add an appropriate amount of redundancy (zero elements) during the filling process to facilitate the full use of the SIMD vectorization function of the processor so that the computer can achieve better performance.

2. For the expanded value and **col_idx**, the corresponding transformation is performed according to the block size; the value of the element corresponding to each position of the **row_ptr** row offset may change; the total number of elements does not change; the number of rows is added 1. Figure 5 shows the CSR2 conversion operation.

Figure (a) is a sparse matrix and a dense vector. We will use the CSR2 format to compress it. In order to help understand the process of designing CSR2, some necessary steps are shown in Figure (b), although this does not exist in the format conversion. Figures (c) and (d) are elements contained in the expanded CSR value and **col_idx** arrays. Figure (e) is the **Row_ptr** row offset array in CSR2 format. The value and **col_idx** in Figures (f) and (g) are defined as **m256d** type, a data type of the avx2 instruction set. These two arrays are stored by column. For example, using the AVX2 instruction set to define the value value[0] in the first column of value, the storage content of value[0] is \{1, 6, 1, 3\}. Similarly, using the AVX2 instruction set to define the value of col_idx[1] in the first column, the content of col_idx[0] is \{1, 2, 8, 2\}. In the **col_idx** array, we start counting with the value 1 for the first element of each row in the original matrix. This is to make the value of the array at position 0 be 0 so that the expanded zero elements can use the value of the array with position 0 to mark their current values uniformly. Similarly, the corresponding vector storage also stores elements starting from position 1, and the value of the empty position 0 is marked as zero, which facilitates the calculation operation of the SpMV algorithm.

#### B. Block Size Selection

Assume that the sparse element matrix is \( n \times m \); the number of non-zero elements is \( n_{zz} \); the block width is \( \text{mtx\_width} \); the height is \( \text{mtx\_high} \), so the minimum value and **col_idx** of the expanded value are \( n_{zz} \) and the maximum is \( n_{zz} + (\text{mtx\_width} - 1) \times n \).

Therefore, the space occupied by the entire CSR2 format depends on the value of **mtx\_width**. For the automatic value operation of the **mtx\_width** value, we show it through the process shown in Figure 6.

The algorithm for automatically adjusting the **mtx\_width** value is shown in Algorithm 4. In the algorithm, **nzz\_size** represents the number of non-zero elements in the matrix, **row\_size** represents the number of rows in the matrix, \( x \) is...
the average number of non-zero elements in each row, and cnt_ite represents the leftward shift times.

![Image](a) Matrix
![Image](b) Value and Col_idx
![Image](c) Value array expanded based on CSR format
![Image](d) Col_idx array expanded based on CSR format
![Image](e) Row offset array Row_ptr in CSR2 format
![Image](f) Value array in CSR2 format
![Image](g) Col_idx array in CSR2 format

Fig. 5: The CSR2 storage format of sparse matrix of size 8×8.

C. Algorithm for converting CSR to CSR2 format

```
Algorithm 4 Get mtx_width automatically.
1: x = INT(nzz_size * 1.0 / row_size + 0.5)
2: if x > 16 then
3:   if x ≥ 128 then
4:     cnt_ite = 0
5:     while x ≥ 128 do
6:       if x & 1 then
7:         x++
8:       end if
9:       x = x >> 1
10:   end if
11:   end while
12: else
13:   if x & 1 then
14:     if x % 10 ≥ 5 then
15:       x = (x / 10 + 1) * 10
16:     else
17:       x = x / 10 * 10
18:     end if
19:   end if
20: end while
21: if x & 1 then
22:   if x % 10 ≥ 5 then
23:     x = (x / 10 + 1) * 10
24:   else
25:     x = x / 10 * 10
26:   end if
27: end if
28: end if
29: end if
30: mtx_width = x
```

x = nzz/row
The loop determines whether x is an odd number. If x is odd, x is incremented by one. Then x moves 1 bit to the left, and the number of iterations is increased by one. If the number of iterations is more than five times, it will automatically jump out of the loop; otherwise, it will only jump out of the loop if x is less than 128.

x ≥ 128
x ≥ 64
x ≥ 32
x ≥ 16
x > 16
x < 64
x < 32
x < 16
The loop determines whether x is an odd number. If x is odd, x is incremented by one. Then x moves 1 bit to the left, and the number of iterations is increased by one. If the number of iterations is more than three times, it will automatically jump out of the loop; otherwise, it can only jump out of the loop if x is less than 32.

The selection of the mtx_high value should depend on the type of data. Assume that the maximum bit width of the instruction set supported by the current device is x bits, and a single data can be represented by y bits in binary, so the value of mtx_high should be x/y. For example, x/64 should be selected as mtx_high for double-precision floating-point and 64-bit integer, while x/32 for single-precision floating-point and 32-bit integer.
Algorithm 5 The CSR storage format convert CSR2 storage format

1: for i = 0 to row_num + 1 in parallel do
2:   CSR2_row_ptr[i] = CSR_row_ptr[i] / mtx_width
3: end for
4: block_size = mtx_high * mtx_width
5: for i = 0 to nonzero_num in parallel do
6:   x = (i / block_size) * mtx_width + i % mtx_width
7:   y = (i % block_size) / mtx_width
8:   CSR2_mtx_val[x][y] = mtx_val[i]
9: end for

As shown in Algorithm 5, transform CSR to CSR2 requires two steps. The first step is to convert row_ptr in parallel. The CSR2 row pointer is the result of dividing each element by the block width mtx_width based on the expanded CSR row pointer. The first step is to convert the value in parallel. The CSR2_mtx_val in the algorithm is not presented as a two-dimensional array in a real program, but it appears as a one-dimensional array in the array defined by the instruction set, and the x-th element in the array consists of y elements. This step can complete the conversion of value. For Col_idx, we do not take conversion. Here we will put mtx_width processing once in the specific calculation operation, which can reduce storage overhead. After format conversion, the storage space occupied by row_ptr and the value of the original CSR can be released.

V. THE CSR2-BASED SPMV ALGORITHM

A. CSR2 block multiply and sum algorithm

As shown in Figure 7, it is based on the operation of multiplying and accumulating elements in the corresponding position of the array under the AVX2 instruction set. The data type used in the figure is a double-precision floating-point format. This operation is the core part of the CSR2 block multiply and sum algorithm, and it is also the only place in the entire algorithm where SIMD calculation instructions are used.

\[ D = \text{mun256}_y \text{fmadd}_y (A, B, C) \]

![Fig. 7: multiply and sum in AVX2.](image)

The block multiply and sum algorithm is the core algorithm of SpMV parallel computing in the CSR2 storage format. It can take full advantage of OpenMP parallel language and SIMD instruction set (Algorithm 6 uses the AVX2 instruction set) vectorization with SpMV and perform dot product and sum operation in the first stage, which is also the key to improving the performance of the entire algorithm. Assuming that the device supports x threads, the SIMD instruction set can be multiplied and added to y data at one time. So, in theory, at the same time, the computer can multiply and add \( x \times y \) numbers. Of course, in actual practice, we also gained a significant performance improvement.

As shown in Figure 8, the three-block matrices, (a), (c), and (e) three-block matrices are executed by thread 1, thread 2, and thread 3, respectively. The fmaddd operation can be performed 4 times in each thread. The three pictures, (b), (d), and (f) are the specific execution flow in each thread.

![Fig. 8: Detailed steps to parallelize block multiply and sum with OpenMP and AVX2.](image)

In Algorithm 6, Nonzero_num represents the number of stored elements after CSR expansion, tile_size represents the number of elements contained in each matrix and is equal to the value of mtx_width \( \times \) mtx_high. The CSR2_mid_val array is used to store the final multiplication result, and the CSR2_col_val array is used to store the values of the elements.
in the vector corresponding to each element of the original matrix.

**Algorithm 6** The block multiply and sum

1: i_end = nonzero_num / block_size
2: for i = 0 to i_end in parallel do
3: CSR2_mid_val[i] = _mm256_setzero_pd()
4: xx = i * mtx_width
5: yy = xx * mtx_high
6: for j = 0 to mtx_width do
7: _mm256d CSR2_col_val
8: YY = yy + j
9: CSR2_col_val = _mm256_set_pd(
10: vec_val[Col_idx[yy + 3 * mtx_width]],
11: vec_val[Col_idx[yy + 2 * mtx_width]],
12: vec_val[Col_idx[yy + mtx_width]],
13: vec_val[Col_idx[yy]])
14: CSR2_mid_val[i] = _mm256_fma_pd(
15: CSR2_mtx_val[xx + j],
16: CSR2_col_val,
17: CSR2_mid_val[i])
18: end for
19: end for

**B. CSR2 block accumulation sum algorithm**

For the intermediate value CSR2_mid_val obtained by the CSR2 block multiply and sum algorithm, we use OpenMP multithreading to perform a summation operation according to the defined block offset pointer CSR2_row_ptr, and obtain the final result mtx_ans. The specific operation process of the block accumulation sum algorithm is shown in Figure 9.

As shown in table I above, We selected Intel Core i7-7700HQ CPU and Intel Xeon CPU E5-2670 v3 to test SpMV in CSR2 and CSR5 formats. The details of these two platforms are shown in the table above. We use Intel Core i7-7700HQ CPU and Intel Xeon CPU E5-2670 v3 of Ubuntu 18.04 LTS system, and compile our SpMV program in CSR2 and the CSR5 program downloaded from Github through Intel C / C ++ compiler 2019 compiler; two platforms we selected both support the use of the AVX2 instruction set and the OpenMP parallel language. The data precision uses double-precision uniformly.

**B. Benchmark suite**

As shown in table II above, we selected 10 sets of regular matrices and 3 sets of irregular matrices to gather a total of 13 sets of sparse matrix data as our benchmark suite for this experiment. The diagram shows the dimensions and sparseness of these data. In the selection of the matrix, we have selected the hot test data of many scholars who have studied SpMV [24],[29]. Related data sets can be downloaded from the University of Florida Sparse Matrix Collection official website.

**C. SpMV performance comparison**

As shown in figure 10 above, this is the execution time of 13 sets of the sparse matrix for 100 and 1000 iterations on the Intel Core i7-7700HQ platform. We compared our proposed CSR2 with the current most advanced format CSR5. In the figure, we can find that in the first ten sets of regular matrices, nd24k, Dense, and crankseg_2 all have performance improvements of nearly 1x or even more. This is because the number of non-zero elements in each row

---

**Algorithm 7** The block accumulation sum

1: memset(mtx_ans, 0, sizeof(double) * row_num)
2: for i = 0 to row_num in parallel do
3: for j = CSR2_row_ptr[i] to CSR2_row_ptr[i + 1] do
4: mtx_ans[i] = mtx_ans[i] +
5: CSR2_mid_val[j / mtx_high][j % mtx_high]
6: end for
7: end for

---

VI. PERFORMANCE EVALUATION

**A. Execution platform**

**TABLE I: Execution platform**

<table>
<thead>
<tr>
<th>Execution platform</th>
<th>Dual Socket Intel Xeon CPU E5-2670 v3</th>
<th>Intel Core i7-7700HQ CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Haswell, 2*12 <a href="mailto:cores@2.3GHz">cores@2.3GHz</a>, L3 cache: 6 MB, 1766.4SPFlops, 883.2DPFlops, L1d cache: 32K, L1i cache: 32K, 1720.4SPFlops, 851.2DPFlops, L1d cache: 32K, L1i cache: 32K, L1d cache: 256K, L1i cache: 256K, 30720K, 8GB RAM)</td>
<td>Intel Core i7-7700HQ CPU</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Kaby Lake, 1*4 <a href="mailto:Cores@2.8GHz">Cores@2.8GHz</a>, thread nums: 8, 358.4SPFlops, 179.2DPFlops, L1d cache: 32K, L1i cache: 32K, 256K, 32K, 128GB RAM)</td>
<td>Intel Core i7-7700HQ CPU</td>
</tr>
</tbody>
</table>

---

**Table II: Benchmark suite**

<table>
<thead>
<tr>
<th>Benchmark suite</th>
<th>Regular matrix</th>
<th>Irregular matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10 sets</td>
<td>3 sets</td>
</tr>
<tr>
<td></td>
<td>100 iterations</td>
<td>1000 iterations</td>
</tr>
</tbody>
</table>

Fig. 9: Detailed steps to parallelize block accumulation sum with OpenMP.

The SpMV algorithm based on the CSR2 format consists of a block multiply and sum algorithm and a block accumulation sum algorithm. The block multiply and sum algorithm is also the key to the acceleration of the entire SpMV algorithm. It can fully utilize the fmadd function in Figure 7 for SIMD vectorization, and the performance of the program has been dramatically improved coupled with multi-threaded parallelism. Moreover, the block multiply and sum algorithm and the block accumulation sum algorithm are efficient and straightforward, and easy to read.

The block accumulation sum algorithm is shown in Algorithm 7.
<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Dim</th>
<th>nnz</th>
<th>nnz per row</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>nd24k</td>
<td>70K×70K</td>
<td>14393817</td>
<td>200</td>
</tr>
<tr>
<td>2</td>
<td>Si41Ge41H72</td>
<td>181K×181K</td>
<td>7598452</td>
<td>41</td>
</tr>
<tr>
<td>3</td>
<td>Dense</td>
<td>2K×2K</td>
<td>4194304</td>
<td>2048</td>
</tr>
<tr>
<td>4</td>
<td>Ga41As41H72</td>
<td>262K×262K</td>
<td>9378286</td>
<td>35</td>
</tr>
<tr>
<td>5</td>
<td>pwdk</td>
<td>213K×213K</td>
<td>5926171</td>
<td>27</td>
</tr>
<tr>
<td>6</td>
<td>pdbhYS</td>
<td>36K×36K</td>
<td>2190591</td>
<td>60</td>
</tr>
<tr>
<td>7</td>
<td>cant</td>
<td>61K×61K</td>
<td>2034917</td>
<td>32</td>
</tr>
<tr>
<td>8</td>
<td>crankedg_2</td>
<td>62K×62K</td>
<td>7106348</td>
<td>111</td>
</tr>
<tr>
<td>9</td>
<td>QCD</td>
<td>48K×48K</td>
<td>1916928</td>
<td>39</td>
</tr>
<tr>
<td>10</td>
<td>shipsec1</td>
<td>137K×137K</td>
<td>3977139</td>
<td>28</td>
</tr>
<tr>
<td>11</td>
<td>ins2</td>
<td>302K×302K</td>
<td>1530448</td>
<td>5</td>
</tr>
<tr>
<td>12</td>
<td>mlp1</td>
<td>65K×65K</td>
<td>5209641</td>
<td>78</td>
</tr>
<tr>
<td>13</td>
<td>rail4284</td>
<td>4K×1.1M</td>
<td>11284032</td>
<td>2634</td>
</tr>
</tbody>
</table>

Note: 1K = 1024, 1M = 1024 × 1024

is relatively large and the distribution is relatively uniform. The conversion from CSR storage format to CSR2 storage format requires less redundancy, which is more conducive to the calculation of SpMV based on CSR2 storage format. The advantages. At the same time, more non-zero elements participating in the calculation can fully take advantage of SIMD vectorization. The data QCD and CSR5 are equivalent, and the remaining six groups of data have varying degrees of performance improvement. For the last three sets of irregular matrices, we find good performance improvements for both 100 iterations and 1000 iterations. Among them, mlp1 has a larger performance improvement, which also fully illustrates that CSR2 has achieved a good performance improvement for both regular and irregular matrices compared to CSR5 on this platform. Besides, we can also intuitively see that there is no obvious performance difference between CSR2 for 100 iterations and 1000 iterations, which also shows that the SpMV performance of CSR2 on this platform is slightly affected by the number of iterations.

Figure 11 shows the execution time of 13 sets of sparse matrix iterations for 100 and 1000 iterations on the Intel Xeon CPU E5-2670 v3 platform. We still compare our proposed CSR2 with the format CSR5. First, we can find that the time consumption is significantly reduced, which means that the performance of the program with 24 threads is better than 8 threads by 2-3 or even more times. Here we can find that for the ten sets of regular matrices, the two sets of data, Dense and shipsec1, have more than 1x performance improvement; pdbhYS, cant also achieve nearly 1x improvement, and the performance of the remaining six sets of data have varying degrees improvement. As for the irregular matrices, mlp1 and ins2 both have more than 1x performance improvement. The performance improvement of rail4284 is very satisfactory in the case of 100 iterations, but it does not work well in the case of 1000 iterations. CSR2 maintains a low execution time for both 100 and 1000 iterations, while the execution of CSR5 is relatively unstable. Comparing the performance of the ten sets of regular matrices on the i7-7700HQ platform, we can find that although the improvements are varying, the overall improvement trend is very considerable. For the three sets of irregular matrices, we also find that the performance improvement of the test on the Intel Xeon CPU E5-2670 v3 platform is better than on the i7-7700HQ platform.

Comparing the implementation of SpMV in the CSR2 and CSR5 formats on different platforms, in general, our proposed CSR2 has a better performance improvement, which is also our intention to give full play to the SIMD function. In order to more intuitively show that SpMV based on
TABLE III: Impact of platforms and iterations on SpMV performance.

<table>
<thead>
<tr>
<th>ID</th>
<th>Regular Matrix</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Irregular Matrix</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SpeedUp (ite=100)</td>
<td>SpeedUp (ite=1000)</td>
<td>SpeedUp (ite=100)</td>
<td>SpeedUp (ite=1000)</td>
<td>SpeedUp (ite=100)</td>
<td>SpeedUp (ite=1000)</td>
<td>SpeedUp (ite=100)</td>
<td>SpeedUp (ite=1000)</td>
<td>SpeedUp (ite=100)</td>
<td>SpeedUp (ite=1000)</td>
</tr>
<tr>
<td>1</td>
<td>1.56x</td>
<td>1.32x</td>
<td>1.82x</td>
<td>1.81x</td>
<td>2.24x</td>
<td>2.20x</td>
<td>1.35x</td>
<td>1.29x</td>
<td>1.82x</td>
<td>1.81x</td>
</tr>
<tr>
<td>2</td>
<td>1.27x</td>
<td>1.17x</td>
<td>1.57x</td>
<td>1.59x</td>
<td>2.25x</td>
<td>2.24x</td>
<td>1.72x</td>
<td>1.70x</td>
<td>1.57x</td>
<td>1.59x</td>
</tr>
<tr>
<td>3</td>
<td>4.03x</td>
<td>3.61x</td>
<td>4.75x</td>
<td>4.73x</td>
<td>2.25x</td>
<td>2.24x</td>
<td>1.72x</td>
<td>1.70x</td>
<td>1.57x</td>
<td>1.59x</td>
</tr>
<tr>
<td>4</td>
<td>1.17x</td>
<td>1.12x</td>
<td>1.57x</td>
<td>1.55x</td>
<td>2.25x</td>
<td>2.24x</td>
<td>1.72x</td>
<td>1.70x</td>
<td>1.57x</td>
<td>1.59x</td>
</tr>
<tr>
<td>5</td>
<td>1.60x</td>
<td>1.50x</td>
<td>1.47x</td>
<td>1.47x</td>
<td>2.25x</td>
<td>2.24x</td>
<td>1.72x</td>
<td>1.70x</td>
<td>1.57x</td>
<td>1.59x</td>
</tr>
<tr>
<td>6</td>
<td>1.90x</td>
<td>1.75x</td>
<td>1.69x</td>
<td>1.73x</td>
<td>2.25x</td>
<td>2.24x</td>
<td>1.72x</td>
<td>1.70x</td>
<td>1.57x</td>
<td>1.59x</td>
</tr>
<tr>
<td>7</td>
<td>1.78x</td>
<td>1.75x</td>
<td>1.72x</td>
<td>1.61x</td>
<td>2.25x</td>
<td>2.24x</td>
<td>1.72x</td>
<td>1.70x</td>
<td>1.57x</td>
<td>1.59x</td>
</tr>
<tr>
<td>8</td>
<td>1.43x</td>
<td>1.39x</td>
<td>1.81x</td>
<td>1.79x</td>
<td>2.25x</td>
<td>2.24x</td>
<td>1.72x</td>
<td>1.70x</td>
<td>1.57x</td>
<td>1.59x</td>
</tr>
<tr>
<td>9</td>
<td>1.21x</td>
<td>1.15x</td>
<td>1.08x</td>
<td>1.13x</td>
<td>2.25x</td>
<td>2.24x</td>
<td>1.72x</td>
<td>1.70x</td>
<td>1.57x</td>
<td>1.59x</td>
</tr>
<tr>
<td>10</td>
<td>2.85x</td>
<td>2.45x</td>
<td>1.58x</td>
<td>1.58x</td>
<td>2.25x</td>
<td>2.24x</td>
<td>1.72x</td>
<td>1.70x</td>
<td>1.57x</td>
<td>1.59x</td>
</tr>
</tbody>
</table>

Note: ite means iteration

The CSR2 format has a better performance improvement than CSR5, we list the 13 sets of data in the table above. Each set is iterated 100 times and 1000 times on the Intel Xeon CPU E5-2670 v3 and Intel Core i7-7700HQ platform. According to Table III, we can find that performing 100 iterations on the Intel Xeon CPU E5-2670 v3 platform can achieve a minimum performance of 17% faster than CSR5 and maximum performance improvement of 303%. For the vast performance improvement of the Dense matrix, we found in tests that under the same conditions, iterating 100 or 1000 times, using CSR2 can get better and more stable performance, while the performance of CSR5 fluctuates greatly, of course, occasionally CSR5 will also have Good performance, so we can only take the average of 10 measurements as the final evaluation standard. We calculated the geometric average of 13 sets of performance data. The performance improvement of SpMV can reach more than 70% for 100 iterations, and it can reach more than 50% performance for 1000 iterations. Although the performance is reduced compared with 100 iterations, however, the reduction rate is still lower than its speedup. On the Intel Core i7-7700HQ platform, the performance speed fluctuation caused by the increase of the number of iterations is minimal. We can find that the average performance of 13 groups of data is improved by more than 50% for 100 and 1000 iterations. Although the SpMV speedup of rail4284 data is only about 10%, it still maintains the performance equivalent to CSR5.

For the calculation method of SpeedUp in the table, we get the corresponding results by the following Equation 4.

\[
\text{SpeedUp} = \frac{\text{CSR5SpMV time}}{\text{CSR2SpMV time}}
\]  

(4)

In the process of data testing, we turned on the Htop monitor and collected the physical memory size that was stably displayed under RES. By comparing the data memory space in Figure 12, we can find that for the 13 sets of test data, CSR2 requires the same or less memory space than CSR5 under the same SpMV operation, which is also the advantage of CSR2 that occupies less data memory in the real running of the program.

In the process of real sparse matrix compression, the overhead of format conversion often has a significant impact on the overall performance of SpMV. The format conversion overhead of CSR5 has taken a significant advantage over other formats. Moreover, we also made statistics on the format conversion cost during the experimental comparison process. As shown in Figure 13, the conversion overhead of CSR2 for the 13 sets of test data is equal to or even less than CSR5. Of course, for the case where part of the data overhead exceeds CSR5, which is minimal for dozens or even hundreds of iterations, CSR2 will also fill the gap in several SpMV iterations. Therefore, the overall performance of CSR2 is superior to CSR5 in regular and irregular matrices and SpMV iteration scenarios of dozens or even hundreds of times.

VII. CONCLUSION

We use OpenMP parallel language and AVX2 instruction set on Intel Xeon CPU E5-2670 v3 and Intel Core i7-7700HQ to compare the performance of SpMV based on CSR5 format to that of the one based on CSR5. First, the data storage space occupied by the CSR2 format is lower than CSR5; second, CSR2 has comparable or lower conversion overhead than CSR5; furthermore, the CSR2 format is suitable for use on platforms with SIMD functions, because it can make full use of its SIMD vectorization utilization rate. Besides, the
conversion process of CSR2 format and the SpMV solution process are simple and easy to understand. Of course, in the final block sum of SpMV, SIMD vectorization can be further used to optimize the instruction set for this, which is also where further improvements are needed in related work in the future.

ACKNOWLEDGMENT

The authors are grateful to the reviewers for valuable comments that have greatly improved the paper. This paper is partially supported by the National Natural Science Foundation of China (No.61762074, No.61962051), National Natural Science Foundation of Qinghai Province (No.2019-ZJ-7034), "Qinghai Province High-end Innovative Thousand Talents Program - Leading Talents" Project Support. The Open Project of State Key Laboratory of Plateau Ecology and Agriculture, Qinghai University (2020-ZZ-03), National College Students Innovation and Entrepreneurship Training Program (No.201910743002).

REFERENCES