EasyChair Preprint

№ 8763

CSR&RV: an Efficient Value Compression Format for Sparse Matrix-Vector Multiplication

Junjun Yan, Xinhai Chen and Jie Liu

June 15, 2023

CSR&RV: An Efficient Value Compression Format for Sparse Matrix-Vector Multiplication

1st Junjun Yan
Science and Technology on Parallel and Distributed Processing Laboratory
National University of Defense Technology
Changsha, 410073 China
yanjunjun@nudt.edu.cn

2nd Xinhai Chen
Science and Technology on Parallel and Distributed Processing Laboratory
National University of Defense Technology
Changsha, 410073 China
chenxinhai16@nudt.edu.cn

3rd Jie Liu *
Laboratory of Software Engineering for Complex System
National University of Defense Technology
Changsha, 410073 China
liujie@nudt.edu.cn

***Abstract*—Sparse Matrix-Vector Multiplication (SpMV) is kernel arithmetic in many areas of science and engineering applications. The storage space of value array in general sparse matrixes is the highest. However, the existing compressed formats normally compress the row and column arrays but failed to compress the value array. To address this issue, we propose an efficient value compression format called Compressed Sparse Row and Repetition Value (CSR&RV). This format saves each different value once and uses the indexes array to save the position of values, which reduces the storage space by compressing the value array. We conduct a series of experiments on an Intel Xeon processor and compare the CSR&RV format with five other formats in 30 real-world matrixes. Experimental results show that CSR&RV can achieve a speedup up to 3.86× (1.66× on average) and a speedup up to 12.42× (3.12× on average) for single-core and multi-core throughput, respectively. Meanwhile, our format can reduce the memory space by 48.57% on average.**

Keywords—sparse matrix, SpMV, value compression, storage format

I. Introduction

Sparse Matrix-Vector Multiplication (SpMV) is a basic operation in many important fields, such as parallel computing, scientific computation, and machine learning [1-4]. Fig. 1 shows the process of SpMV, the expression of SpMV is $Y \leftarrow A * X$, where A is a sparse matrix and both X and Y are dense vectors. The sparse matrix is a matrix composed of a large number of zero elements and a small number of non-zero elements (*nnz*). In general, the number of *nnz* in a sparse matrix accounts for less than 5% of the total elements. In some typical applications, the scale of sparse matrixes is very large. To improve the storage and computing efficiency, some classic sparse matrix storage formats have been proposed.

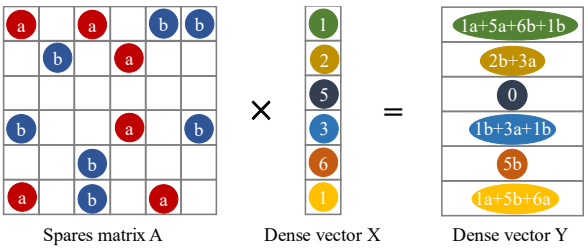


Fig. 1. The process of SpMV.

The most classic formats are Compressed Sparse Row (CSR) [5] and Coordinates (COO). The COO format uses the triple form (*row, col, value*) to save all nonzero elements and the CSR format compresses the row array to the *row_ptr* array, which only saves the start and end index of each row. In recent years, the number of cores and threads in processors is increased, the capacity of caches is enhanced, and the vector processing capability of Single Instruction Multi Data (SIMD) units is improved. How to redesign and optimize the classic formats by using those new features has become the main research direction in this field [6]. Some formats use the SIMD instructions to make full use of vectorization and use blocking algorithms to improve the data locality [7-10]. Other formats compress the storage space to reduce the memory access [11, 12] because SpMV is a memory-bound application (each element gathered to processors only executes twice floating arithmetic).

Compressing the memory space of sparse matrixes is one major approach to accelerating SpMV. The ways to compress the memory space is depended on the feature of matrixes [12]. In many applications, the sparse matrixes only have a small number of elements recurrent. This feature can be used to instruct the design of sparse matrix storage format, by compressing the repeated elements in the value array to reduce memory access times and improve the SpMV efficiency. However, the studies about compression and optimization of the value array are rare, and SpMV algorithms based on state-of-art formats are still not efficient enough because of loading the repetitive values. Since the value array is the type of *double* and the other two arrays are the type of *int*, the value array has the largest memory overhead among the triples. Therefore, the compression of the value array is necessary.

In this paper, we propose CSR&RV, an efficient value compression storage format. To solve the problem that current formats store too many repetitive values, CSR&RV just stores it the first time. For the other repetitive values, use the index array to store the position in the value array and load the value indirectly. Thus, CSR&RV compresses the original value array to a non-repetition value array and an index array. Because the number of values is much less than *nnz*, the memory space of the value array is much less than the index array. Normally the index array uses *int* type but the value array uses *double* type, so the space is compressed. By using *int8*, *int16* to store the indexes, we can feather compress the space of the index array. Thus, the SpMV can benefit a lot by compressing the value array.

We compare the CSR&RV format to five other formats with different evaluating criteria. Our format has the highest throughput and the lowest memory space overload. Compared to the MKL-CSR [23], our format can get an average of 1.66× (maximum of 3.86×) speedup and an average of 3.21× (maximum of 12.42×) speedup in single-core and multi-core throughput, respectively. Compared to the best state-of-art formats, our format can get an average of 1.36× (maximum of 2.65×) speedup and an average of 1.87× (maximum of 7.92×) speedup in single-core and multi-core throughput, respectively. By using index compression, our format can reduce an average of 48.57% (maximum of 58.13%) memory space on matrixes saved in CSR format.

This paper accepted by the 19th IFIP International Conference on Network and Parallel Computing, NPC 2022

In short, our contributions can be concluded as follow:

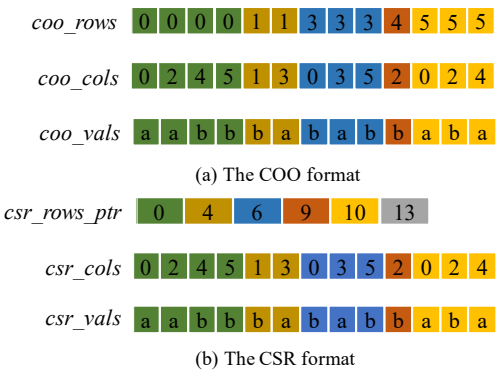
- We propose a new value compression storage format called CSR&RV, which saves each different value once and uses the indexes array to save the position of values.
- We employ the index compression method in the index arrays which can further reduce the memory space and improve the SpMV performance.
- We evaluate the throughput, memory space, scalability, and pre-processing overload in CSR&RV and other 5 formats.

The remainder paper is organized as follows. In Section II, we introduce the background. In Section III, we illustrate the CSR&RV format and the index compression method. In Section IV, we describe our two major methods: the converting method and the SpMV algorithm. In Section V and Section VI, we show our experimental environment and evaluate our experimental results. In Section VII, we give the related work. Finally, in Section VIII, we report the conclusion.

II. Background

A. The COO Format and the CSR Format

The COO format is the most intuitive sparse matrix storage format, as shown in Fig. 2 (a). It stores all nonzero elements in the form of the triple (*row, col, val*). The *coo_rows* array saves the *x* coordinate of each nonzero element, while the *coo_cols* array and *coo_vals* array save the corresponding *y* and *value*, respectively. The COO format is simple and easy to program. However, the memory overload in this format is high. Considering that in a large sparse matrix, the number of *nnz* is much smaller than the number of rows, the CSR format is proposed. Which compresses the *coo_rows* array in COO format, as shown in Fig. 2 (b). The elements in CSR are sorted by rows. For elements in the same row, they are saved consecutively and only the starting positions in the value array corresponding to the row are recorded. Thus, the size of the row array of the matrix is compressed from the original *nnz* to *numRow*+1. CSR is one of the most widely used formats. The memory cost in CSR is lower than COO, but it still cannot utilize the new features in modern computer architectures. Therefore, it is not efficient on modern platforms.



(a) The COO format

(b) The CSR format

Fig. 2. The COO format and the CSR format.

B. CSR Based SpMV

The elements in CSR are sorted by the row ranks. Thus, the SpMV can compute each row one by one. Algorithm 1 shows the CSR-based SpMV, which has a two-layers loop. The exterior loop uses variable *i* to traverse each row in the *row_ptr* array. The inner loop uses variable *j* to traverse the nonzero elements in each row and calculates results which will write back to the output vector *y* according to the row ranks. Noticing that each row only writes back to *y* once and it is consecutive, the memory access of *y* has a decent locality. Although the CSR format has the disadvantages of poor locality in the access of *x* vector and imbalance overload in multi-thread, many modern formats use CSR as the basic storage format [13]. By using the sum segment method, blocking algorithm, and vectorization instructions, the modern CSR-based formats can overcome those disadvantages to certain degrees, like CSR5 [8], CVR [10] et al. Our value compression format is also based on the CSR format.

Algorithm 1: Scalar-SpMV with the CSR format.

```
1:  for i = 0 to numRows - 1 do
2:      sum ← 0
3:      for j = csr_rows_ptr[i] to csr_rows_ptr[i+1] - 1 do
4:          sum ← sum + csr_vals[j] × x[csr_cols[j]]
5:      end for
6:      y[i] ← sum
7:  end for
```

III. The Compressed Sparse Row and Repetition Value Format

A. CSR&RV Representation

Many sparse matrixes have a large number of repetitive elements, such features can be seen in TABLE I. The table shows some detail of real-world matrixes. The *numValues* column represents the number of values in a matrix, and the *nnz* column represents the number of nonzero elements. For example, the *atmosmodl* matrix (NO.3) has 10.32M nonzero elements but the number of values is four, which means those four elements appear in this matrix over and over again. Considering this feature, the CSR-based storage format is not efficient enough. Because the value array saves too many duplicative values and the SpMV will access the repetitive values many times, which leads to the precious memory bandwidths wasted.

To overcome this disadvantage, we propose the CSR&RV format, as shown in Fig. 3. By compressing the repetitive values, our format can reduce the memory overload in *csr_vals* array specifically. In CSR&RV, the *csrv_values* array only saves each unequal value once. And for every element, we use an index array called *csrv_vals_idx* to point to the location of values in the *csrv_values*. Thus, the memory access for values is indirectly by using *csrv_vals_idx* as indexes. Compared to the *csrv_vals_idx*, the memory cost of *csrv_values* array can be ignored because the number of values is much small than *nnz*. Hence, the *csr_vals* can be compressed to a same-length array *csrv_vals_idx* and a negligible array *csrv_values*. That operation can save about half the size of *csr_vals* array since *csr_vals* uses *double* type while *csrv_vals_idx* uses *int* type. What's more, the *csrv_vals_idx* can be compressed further through index compression, which is introduced in the next section.

B. Index Compression

In the last section, we assume the type of *csrv_vals_idx* array as *int*. Using *int* to save the index array is not always efficient because the length of the value array is usually small. For example, in the *atmosmodj* matrix, the *numValues* is four.

It means the range of the index array is zero to three and can be expressed by two bits. Therefore, we can use *int8* or *int16* instead of *int32* as the type of index array, the operation called index compression in our paper. As shown in (1), if the number of value array is less than 256, we can use *uint8_t* (unsigned char, one byte) as the type of index array. If the number of values is much than 256 but less than 65536 (2^{16}), we can use *uint16_t* (unsigned short, two bytes). Otherwise, we use *int* type. Index compression can save the storage space, which improves the SpMV efficiency by reducing the memory access.

$$type(vals\_idx)=\begin{cases} uint8\_t,\ 0 < numValues \le 256 \\ uint16\_t,\ 256 < numValues \le 65536 \\ int,\ 65536 < numValues \le INT\_MAX \end{cases} \quad (1)$$

To analyze the storage efficiency, we compare our format with the CSR format. We define an indicator named compressed rate to represent the space reduced by the CSR&RV format, shown in (2):

$$Compressed\ Rate = \frac{Size_{CSR}-Size_{CSR\&RV}}{Size_{CSR}} = 1-\frac{Size_{CSR\&RV}}{Size_{CSR}} \quad (2)$$

The compressed rate is defined as the reduced storage size divided by the original storage size, the higher compressed rate means the higher reduced space. The storage size of CSR and CSR&RV can be computed by (3) and (4), respectively:

$$Size_{CSR} = 4\times numRows + 4 \times nnz + 8\times nnz \quad (3)$$

$$Size_{CSR\&RV} = 4\times numRows + 4\times nnz + 8\times numValues + sizeof(Type(vals\_idx))\times nnz \quad (4)$$

The size of CSR is composed of three parts: the first part is *csr_rows_ptr*, it uses *int* type (four bytes) and the length is *numRows*; the second part is *csr_cols*, it also uses *int* type and the length is *nnz*; the last part is *csr_vals*, it uses *double* type (eight bytes) and the length is *nnz*. As for CSR&RV, the first and the second parts are the same as CSR, but the third part is replaced by *csrv_values* and *csrv_vals_idx*. The former uses *double* type and the length is the number of values; the type of the latter is dependent on the number of values, which is defined in (1).

In sparse matrixes, the number of rows is much less than *nnz*; and the number of values is also much less than *nnz* when the matrixes have the feature of many repeated values. Those conditions can be concluded as (5):

$$\textbf{S.T.}\ \begin{cases} numRows \ll nnz \\ numValues \ll nnz \\ sizeof(double) = 8Bytes \\ sizeof(int) = 4Bytes \\ sizeof(uint16\_t) = 2Bytes \\ sizeof(uint8\_t) = 1Byte \end{cases} \quad (5)$$

When (5) is fitted and *csrv_vals_idx* uses the *int* type, the compressed rate can be computed by (6). Because the storage space of *csr_rows_ptr* can be ignored compared to the *csr_cols* and *csr_vals*, the size of CSR is 12*nnz*. In the same way, the *csrv_rows_ptr* and *csrv_values* can be ignored compared to the *csrv_cols* and *csrv_vals_idx*. Therefore, the size of CSR&RV is 8*nnz* and the maximum compressed rate is about 0.33 in theory.

$$Compressed\ Rate = 1-\frac{8nnz+4numRow+8numValues}{12nnz+4numRow} \approx 1-\frac{8nnz}{12nnz} \approx 0.33 \quad (6)$$

When we use *int8* to save the *csrv_als_idx*, we can get a higher compressed rate, because the memory size of *csrv_vals_idx* is less. According to (7), we can get the maximum compressed rate of 0.58 by using *int8* theoretically.

$$Compressed\ Rate = 1-\frac{5nnz+4numRow+8numValues}{12nnz+4numRow} \approx 1-\frac{5nnz}{12nnz} \approx 0.58 \quad (7)$$

Without the index compression method, the CSR&RV format can get a maximum of about 33% compressed rate in theory; with index compression, CSR&RV can improve the maximum compressed rate to about 58%. In Section VI, we can see that the experimental results in our benchmark are accordant with the theoretical computation.

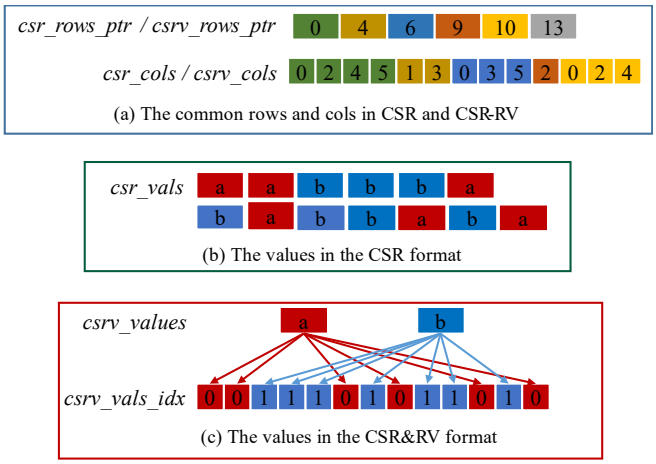


Fig. 3. The CSR&RV format.

IV. CSR&RV Implementation and Approach

A. The Conversion Method

Cause most applications save matrixes in the CSR format, the SpMV algorithms based on other formats must convert to corresponding formats from CSR, which is called pre-processing. In this section, we introduce how to convert a matrix from CSR to the CSR&RV format. As shown in Algorithm 2, the conversion method is convenient. CSR&RV is origin from CSR, the row and col arrays are the same in both formats, the only difference is the value array. Therefore, the kernel operation is converting the *csr_vals* to the *csrv_values* and *csrv_vals_idx*. The converting method has three major steps:

1) Step1: The first step is traversing the *csr_vals* to remove the duplicative elements in the value array (line 3), and count the number of values (line 5). To achieve this purpose, we need a data structure called HashMap, which encapsulated as *unordered_map* in C++ STL (Standard Template Library).

2) Step2: The second step is using the number of values counted in step1 to create the memory space of *csrv_vals* (line 6). Then, we traverse the *unordered_map* and put the values into *csrv_vals*, which is used for saving all the non-reductive elements in CSR&RV (line 9). We can use this traverse to

save the indexes into *unordered_map* meanwhile, it can look up corresponding indexes through inputting values (line 10), which is useful in step3.

3) Step3: In the last step, we traverse the *csr_vals* again and convert it to the *csrv_vals_idx* array by looking up the indexes in the *unordered_map* generated by step2 (line 14).

Algorithm 2: Converting from CSR to CSR&RV

```
1:  std::unordered_map<floatType,valPtrType>
    value_hashmap
2:  for i = 0 to m-1 do                                  Step 1
3:      value_hashmap.insert(csr_vals[i], 0)
4:  end for
5:  numVals ← value_hashmap.size
6:  MALLOC(*csrv_vals, numVals)                          Step 2
7:  i←0
8:  for it = value_hashmap.begin to
            value_hashmap.end   do
9:      csrv_vals[i]  ← it.key
10:     it.val ← i
11:     i ← i+1
12: end for
13: for i=0 to nnz do                                    Step 3
14:     csrv_vals_idx[i] ←
        value_hashmap.find(csrs_vals[i])
15: end for
```

B. The simple SpMV Algorithm Based on CSR&RV

The simple SpMV algorithm in CSR&RV format is similar to the CSR-based SpMV, the only difference is CSR can access the values using *csr_vals* directly, but CSR&RV needs to use the index array (*csrv_vals_idx*) to access the memory indirectly. As shown in Algorithm 3, the difference is marked by red color. Although CSR&RV needs extra access, the cost is small because the type of the index array may be *int8* or *int16*. Compared to the extra cost, the CSR&RV format can get more benefits from the *csrv_vals* array, because it is much smaller than *csr_val* and it reduces much memory-access overload. Meanwhile, the *csrv_vals* with a small size is easier to put into the cache, which means a better locality.

Algorithm 3 Scaler CSR&RV based SpMV

```
1: for i = 0 to numRows - 1 do
2:     sum ← 0
3:     for j=csrv_rows_ptr[i] to csrv_rows_ptr[i+1]-1 do
4:         sum ← sum + csrv_vals[csrv_vals_idx[i]] × x[csrv_cols[j]]
5:     end for
6:     y[i] ← sum
7: end for
```

C. SpMV Algorithm Used AVX512

Modern CPUs have vector operation units, which can use SIMD to accelerate the SpMV algorithm. In our implementation, we use the AVX512 (Advanced Vector Extensions) instructions in Intel C++ intrinsic API, which supporting 512 bits vector arithmetic both in integral and float numbers. By using these instructions, we can process eight *float64* numbers or 16 *float32* numbers or 16 *int32* numbers or 32 *int16* numbers or 64 *int8* numbers at the same time. Algorithm 4 shows the SpMV method with AVX512 instructions, which uses the *uint8_t* as the type of *val_idx* array for example. We vectorize the data by rows, which means the elements in the same rows will be packed together into several vectors. This kind of vectorial pattern may potentially harm the efficiency when the number of *nnz* per row is small. Because we need some control statements and some mask instructions to vectorize the data, which have some extra overhead. But in large general sparse matrixes, the *nnz* is always higher than the number of rows. Therefore, the situation of little elements in a row is infrequently. And we can still get high benefits through the vectorization even in some rows it may take adverse reactive.

Cause we vectorize the data by rows, we need to calculate the end positions of vectorial processing for each row (Algorithm 4, lines 8-10). The AVX512 instructions support eight *double* numbers (64 bits) to calculate at once, therefore the width of the vectors is eight. To match the width, other operations all use eight as operated width. There are five major steps to accomplish one-time floating multiplication and addition, which is a fundamental operation in SpMV (can be compared with Algorithm 3, line 4) :

1) Step1: Loading eight value indexes from *csrv_vals_idx* array to the *_val_idx_uint8* vector. We only need to load half of a 128 bits vector because the value indexes are stored by *uint8_t*. To save the memory bandwidths, we use the maskload instruction and set the first eight masks as zero (Algorithm 4, line 12).

2) Step2: Converting the indexes vector from *uint8_t* to *int32*, which can be used as offsets to gather the value vectors. Because the Intel C++ intrinsic only has the instruction to use 32 bits index to gather data, the convection is necessary. After convection, the length of *_val_idx* vector is 256 bits (line 13).

3) Step3: Loading the columns from the *csrv_cols* array, which can be used for gathering the *x* vector. The columns are stored in *int32*; thus, we can use a 256 bits vector to load the data directly and need not any other convection (line 14).

4) Step4: Gathering the vector operand *_x* from the *x* array using *_col_ptr* as the offset; and gathering the vector operand *_val* from *csrv_vals* array using *_val_ptr* as offset. The length of both vector operands are 512 bits (lines 15-16).

5) Step5: Finally, using the Fused Multiply and Add (FMA) instruction to execute once floating multiply and add operation in the FMA units (line 17).

Algorithm 4: SpMV using AVX512 instructions.

```
1:  __m128i _val_idx_uint8
2:  __m256i _col_ptr, _val_idx
3:  __m512d _x, _y, _val
4:  __mmask8 _mask
5:  for row = 0 to numRows - 1 do
6:      _mm_prefetch(y[row])
7:      _y ← _mm512_setzero_pd()
8:      start = csrv_rowDelimiters[row]
9:      non_vectoral _end ← h_rowDelimiters[row + 1]
10:     vectoral_end ← start + (padding_end – start) / 8 * 8
11:     for j = start to vectoral_end-1 do
12:         _val_idx_uint8 ← _mm_mask_loadu_epi8
                    (&csrv_vals_idx[i], 0x00FF)
13:         _val_idx ← _mm256_cvtepu8_epi32(_vals_ptr_uint8)
14:         _col_ptr ← _mm256_loadu_epi32(&csrv_cols[i])
15:         _val ← _mm512_i32gather_pd(_val_ptr, csrv_vals, 8)
16:         _x ← _mm512_i32gather_pd(_col_ptr, x, 8)
17:         _y ← _mm512_fmadd_pd(_x, _val, _y)
18:         j ← j + 8
19:     end for
20:     PRODUCE_TAIL(vectoral_end, non_vectoral_end - 1)
21:     y[j] ← _mm512_reduce_add_pd(_y)
22: end for
```

Looping above five steps, until the vectorial portion is finished. Next, we need to produce the non-vectorial portion (line 20). That procedure is similar to the inner loop of simple SpMV-based CSR&RV (Algorithm 3, lines 3-6), but started in *vectoral_end* and ended in *non_vectoral_end-1*. After that, the computation of one row is accomplished. We can use the *reduce_add* instruction to sum the *_y* vector horizontally and write back the result to the *y* array corresponding to the row rank (line 21). The write position in the *y* is prefetched before the computation (Algorithm 4, line 6). Repeating the above procession until all the rows are computed and stored to the *y* array. Fig. 4 shows the data flow in the SpMV method, which helps the reader to comprehend the algorithm further.

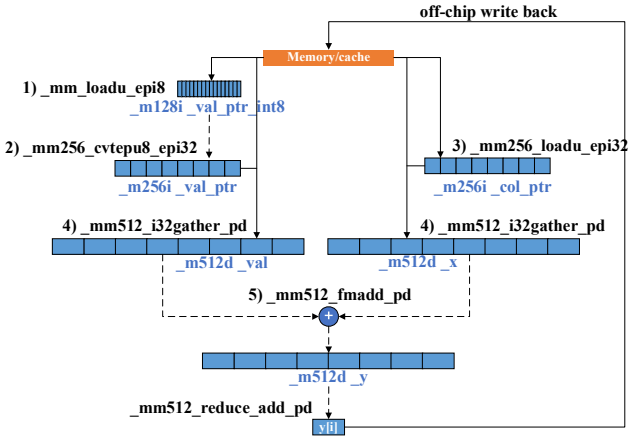


Fig. 4. The data flow in the SpMV algorithm.

V. Experimental Methodology

A. Experimental Setup

We conduct our experiments on an Intel CPU: Xeon (R) Platinum 8163 @ 2.50GHz, with 24 cores and 48 threads. This CPU has 32KB Data Cache and 32KB Instruction Cache in the L1 Cache, 1MB L2 Cache, and 32MB L3 Cache. The RAM is 192GB DDR4 running at 2666MHz. The operating system is CentOS Linux release 7.9.2009 (Core). We use Intel classic compiler (ICC 2021.5.0) to compiler all the experimental test source codes.

B. Experimental Matrixes

We use 30 real word sparse matrixes in different scenarios and different applications as the benchmark, the detail of the matrixes is shown in TABLE I. The *dimension* column shows the number of rows and cols in one matrix, and the *nnz* column shows the number of nonzero elements. Those two columns can reflect the scale of one matrix. According to TABLE I. , all the *nnz* are large than millions. The *nnz / row* column shows the average number of *nnz* per row, this column can reflect the beneficial extent through SIMD operation to certain degrees. The *numVals* column shows the number of different values; while the *nnz / numVals* column represents the average repeated times of one value, both columns can reflect the repeated extent in a matrix. In TABLE I. , the *numVals* in most matrixes is small while the *nnz / numVals* in most matrixes is high. The higher *nnz / numVals* signified a higher compressed rate, which can save more memory space and get a better locality. And the smaller *numVals* means we can use index compression more possibly, this method can enhance the compressed rate according to Section III. Fig. 5 shows the distribution of nonzero elements in some matrixes on our benchmark. All the matrixes can be downloaded from the official website of the University of Florida Sparse Matrix Collection [14].

C. Baseline Formats

We compare our works with state of art open-source sparse matrix storage formats and SpMV algorithms, all the formats implemented with the AVX512 instructions. The following 5 formats are compared: MKL-CSR, Intel math kernel library [23] that provides a state of art SpMV implementation based on CSR; MKL-OPT, which uses the *mkl_sparse_optimize* function to optimize the MKL-CSR; CSR5, a tailed CSR-based format, which conducts the segment sum and blocking algorithm; CVR, a vector-oriental format, it has low cache miss ratio and low pre-processing overhead; SPV8 [15], a new format which takes minimum branch-prediction miss as a design goal.

The CSR5, CVR, and SPV8 formats are downloaded from the GitHub website and running with the default setting, we only change the data-input function. The MKL library is downloaded from Intel OneAPI official website [18], the version is 2022-01. We run each matrix for 1000 iterations and use the single-iteration average time to evaluate. The SpMV is run in double precision, and the thread numbers are increased from 1 to 48, we record the SpMV run-time in different formats and different threads. When evaluating the multi-core performance, we use the best run-time among all run-times in different threads as a result. Normally, the maximal thread numbers (48) can get the best result.

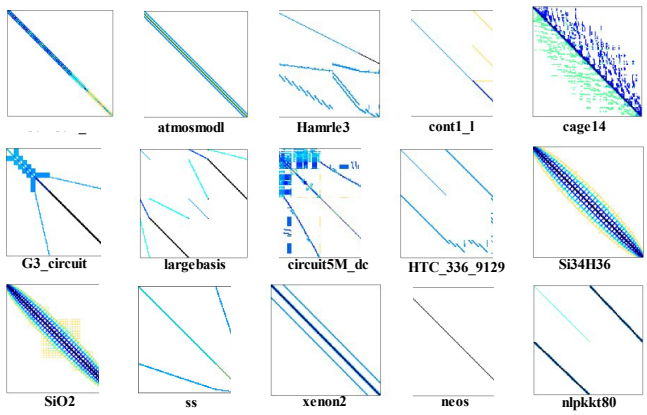


Fig. 5. The distribution of nonzero elements in some matrixes of our benchmark.

TABLE I. Sparse Matrixes Benchmark Detail

NO.	Matrix name	Dimension	*nnz*	*nnz* / row	numVals	*nnz* / numVals	problem
1	af_shell3	0.50M*0.50M	9.05M	17.92	0.28M	32.49	Subsequent Structural Problem
2	apache2	0.72M*0.72M	2.77M	5.48	40	69.16K	Structural Problem
3	atmosmodl	1.50M*1.50M	10.32M	103.88	4	2.58M	CFD Problem
4	boneS01	0.13M*0.13M	3.42M	10.63	40	85.52K	Model Reduction Problem

5	boneS10	0.91M*0.91M	28.19M	87.64	40	70.48K	Model Reduction Problem
6	cage14	1.51M*1.51M	27.13M	39.73	465	58.34K	Directed Weighted Graph
7	circuit5M_dc	3.52M*3.52M	19.19M	15.11	37.04K	51.83K	Circuit Simulation Problem
8	CO	0.22M*0.22M	3.94M	3.1	0.11M	34.48K	Quantum Chemistry Problem
9	cont1_l	0.22M*0.22M	3.94M	84.32	0.11M	34.48K	Linear Programming Problem
10	CurlCurl_4	2.38M*2.38M	14.45M	113.56	1.60K	85.09K	Model Reduction Problem
11	degme	0.18M*0.66M	8.13M	63.88	19	0.43M	Linear Programming Problem
12	ecology2	1.00M*1.00M	3.00M	3.28	18	0.17M	2D/3D Problem
13	G3_circuit	1.59M*1.59M	4.62M	9.91	241	19.18K	Circuit Simulation Problem
14	Goodwin_127	0.18M*0.18M	5.78M	44.37	4.58K	1.26K	CFD Problem
15	Hamrle3	1.45M*1.45M	5.51M	12.38	53	0.10M	Circuit Simulation Problem
16	largebasis	0.44M*0.44M	5.56M	138.26	317	17.54K	Optimization Problem
17	LargeRegFile	2.11M*0.80M	4.94M	122.94	19	0.26M	Circuit Simulation Problem
18	neos	0.48M*0.52M	1.52M	6.9	4	0.38M	Linear Programming Problem
19	nlpkkt80	1.06M*1.06M	14.88M	82.28	915	16.27K	Optimization Problem
20	rajat31	4.69M*4.69M	20.32M	10.59	3.40K	5.01K	Circuit Simulation Problem
21	s3dkq4m2	90.45K*90.45K	2.46M	10.84	74.28K	33.06	Structural Problem
22	Si34H36	97.57K*97.57K	2.62M	3.26	0.11M	23	Quantum Chemistry Problem
23	SiO2	0.15M*0.15M	5.72M	4.69	0.24M	24.31	Quantum Chemistry Problem
24	sls	1.75M*62.73K	6.80M	2.86	4	1.70M	Least Squares Problem
25	ss	1.65M*1.65M	34.75M	187.35	87.90K	395.39	Semiconductor Process Problem
26	stat96v3	33.84K*1.11M	3.32M	51.02	94	35.30K	Linear Programming Problem
27	stormG2_1000	52.81M*1.37M	3.46M	3.46	99	34.95K	Linear Programming Problem
28	tp-6	0.14M*1.01M	11.54M	11.54	19	0.61M	Linear Programming Problem
29	ts-palko	35.70K*35.70K	8.78M	528.59	0.25M	34.58	Linear Programming Problem
30	xenon2	0.16M*0.16M	3.87M	1.29	93.36K	41.42	Materials Problem

VI. EVALUATION

A. CSR&RV vs CSR

In this section, we give the test results for different implantations of CSR&RV format. We implement CSR&RV with different methods, as shown in TABLE II. The CSR&RV-simple is a basic version only using OpenMP for multi-threads; the CSR&RV-int8 is implemented with the index compression method, and the CSR&RV-AVX512 uses the Intel intrinsic API to rewrite with the AVX512 instructions; as for the CSR&RV-prefetch, we enable prefetch instructions on it.

TABLE II. DIFFERENT IMPLEMENTATIONS OF CSR&RV FORMATS

	CSR&RV-simple	CSR&RV-int8	CSR&RV-AVX512	CSR&RV-prefetch
Index compression		✓	✓	✓
AVX512 instructions			✓	✓
Prefetch instructions				✓

Fig. 6 shows the speedup of different implantations in CSR&RV format compared to the MKL-CSR format. The blue bars represent the speedup of CSR&RV-simple, we can get an average of 1.40× (maximum of 3.11×) speedup in multi-core SpMV. However, not all the matrixes can get benefits, which means the CSR&RV-simple is not suitable for all matrixes. Compared to the MKL-CSR, the CSR&RV-simple can reduce a part of memory overload and enhance the locality to certain degrees. When the value-repetition extend is not high, the improvement is unobvious.

The orange bars represent the speedup of CSR&RV-int8, we can get an average of 1.72× (maximum of 3.20×) speedup. The improvement by using the index compression method is evident, and the speedup is influenced by the number of values. When the number of values is very small, for example, in *atmosmodl*, *Goodwin_127* and *sls*, we can get very high speed up, and even beyond the theory. That is influenced by the cache and data locality, for example, when the *csrv_vals* array can be completed loaded into the L1 cache coincidentally. And the same situation for L2 and L3 cache.The gray bars represent the speedup of CSR&RV-AVX512, we can get an average of 1.87× (maximum of 3.63×) speedup. Most matrixes can get benefits by using the AVX512 instructions because the SIMD units have the capability for parallel computing. However, some matrixes cannot get good speedup when the *nnz / row* is low. Because we need many mask SIMD instructions which reduces the vectorial extent and leads to low throughput.

The yellow bars represent the speedup of CSR&RV-prefetch. By using the prefetch instructions, all the matrixes can get benefits. We can get an average of 3.67× (maximum of 10.52×) speedup. This result shows that prefetch can improve the SpMV largely because it is a typical memory-bound application.

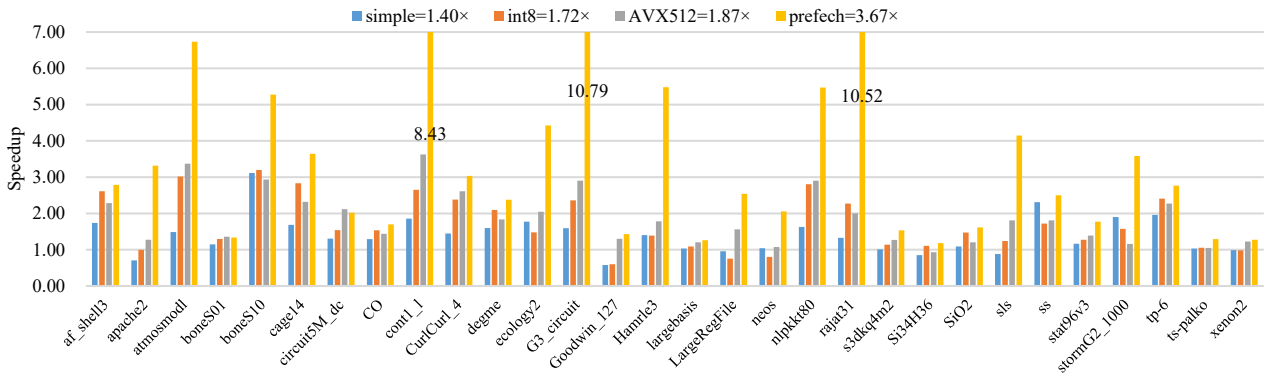


Fig. 6. The speedup of different implantations in CSR&RV format comparing to the MKL-CSR.

B. *Single-Core Performance*

Fig. 8 shows the single-core performance of different formats in our benchmark. The x-axis expresses different matrixes, and the y-axis expresses the GFlops. The green bars represent the GFlops of the CSR&RV format, which is the highest in most matrixes. For some matrixes, the speedup can be very high. For example, in *atmosmodl* which is described in Section II (each element repeats nearly 2.5 million times). TABLE III. shows the single-core throughput (GFlops) speedup of the CSR&RV format compared to other formats. Compared to the MKL-CSR, CSR&RV can get an average of 1.66× speedup and a maximum of 3.86× speedup; and compared to the best state-of-art formats, our format can get an average of 1.66× speedup and a maximum of 3.86× speedup. The performance of MKL_CSR, CSR5, and SPV8 are similar, and about 80% of matrixes in our benchmark show better performance in CSR&RV than those formats. As for MKL_OPT, our format can still have an average of 1.37× speedup, and 66% of matrixes in CSR&RV are better than it.

TABLE III. THE SINGLE-CORE SPEEDUP COMPARES TO DIFFERENT FORMATS

	MKL_CSR	MKL_OPT	CVR	CSR5	SPV8
Max speedup	3.86×	2.65×	4.72×	3.90×	4.11×
Ave. speedup	1.66×	1.37×	1.94×	1.60×	1.66×

C. *Multi-Core Performance*

Fig. 8 and TABLE IV. show the multi-core performance of different formats in benchmark, the setting is the same as Fig. 7 and TABLE III. The green bars represent the GFlops of the CSR&RV format, which has the best throughput in most matrixes. Compared to the MKL-CSR, the multi-core throughput (GFlops) of our format can get an average of 3.22× speedup and a maximum of 12.42× speedup; and compared to the best state-of-art formats, we can get an average of 7.92× speedup and a maximum of 1.85× speedup. The results in multi-core are better than single-sore. When compared to the single-core performance, the bandwidth ability for one thread is enough and the performance is mainly influenced by the optimization of arithmetic instructions. However, when considering multi-core, the bandwidth will be limited gradually with the increasing of thread numbers. This leads to memory-access ability to be the major influence. And the memory-access ability is reflected by the storage space of formats, the accessed pattern of algorithms, and the feature of matrixes. That is why the CSR&RV format is better than other formats and the muti-core results are better than single-core results.

TABLE IV. THE MULTI-CORE SPEEDUP COMPARES TO DIFFERENT FORMATS

	MKL_CSR	MKL_OPT	CVR	CSR5	SPV8
Max speedup	12.42×	52.82×	9.63×	9.51×	21.31×
Ave. speedup	3.22×	3.71×	2.77×	2.76×	6.38×

D. *Scalability*

SpMV is a typical memory access limited application, and the efficiency can be influenced remarkably by the storage space. In the last two sections, we have discussed the single-core and multi-core performance in the CSR&RV format, and we know that the performance in multi-core is better than in single-core. To study the scalability of CSR&RV further, we split our benchmark by the type of *csrv_vals_idx* into three parts: *int8*, *int16*, and *int32*. We measure the throughput by increasing the thread numbers from 1 to 48 in the above parts on different formats, the results shown in Fig. 9. The x-axis expresses the number of threads, we test 1, 2, 4, 8, 16, 24, and 48 threads respectively, and the 48 threads use the hyper-threading technology. The y-axis expresses the corresponding GFlops.

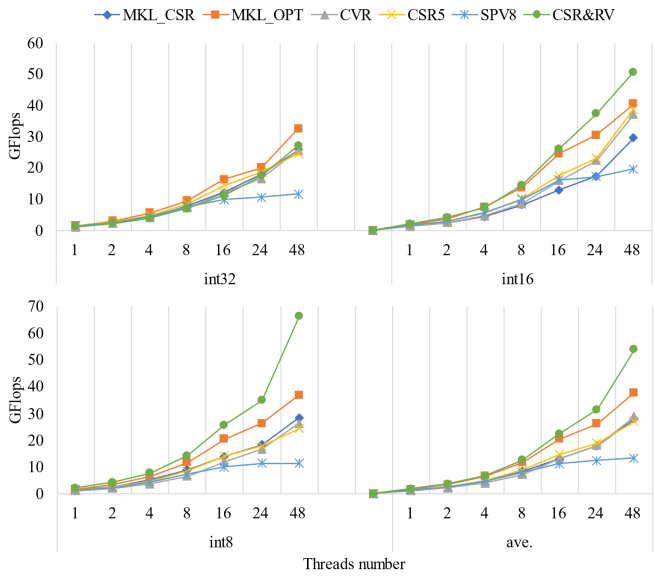


Fig. 7. The performance of different formats in different thread numbers.

According to Fig. 9, the average throughput in *int8* is the best, *int16* the second, and *int32* the last. In *int32*, the performance in MKL_OPT is litter better than CSR&RV; while in *int16* and *int8*, the performance in CSR&RV is

largely better than other formats. The test results explain the main speedup in CSR&CR is driven by the *int16* and *int8* matrixes, and it also illustrates the index compression method can improve the SpMV efficiency significantly. With increasing thread numbers, the gap between CSR&RV and other formats is larger except in *int32*. Taking *int8* as an example, when the thread numbers increased from 24 to 48, the throughput nearly double in CSR&RV; while other formats cannot get such increase through hype-threads. That means in the same bandwidth ability, the CSR&RV format has better throughput performance. Therefore, for the matrixes in benchmark, our format has preferable scalability.

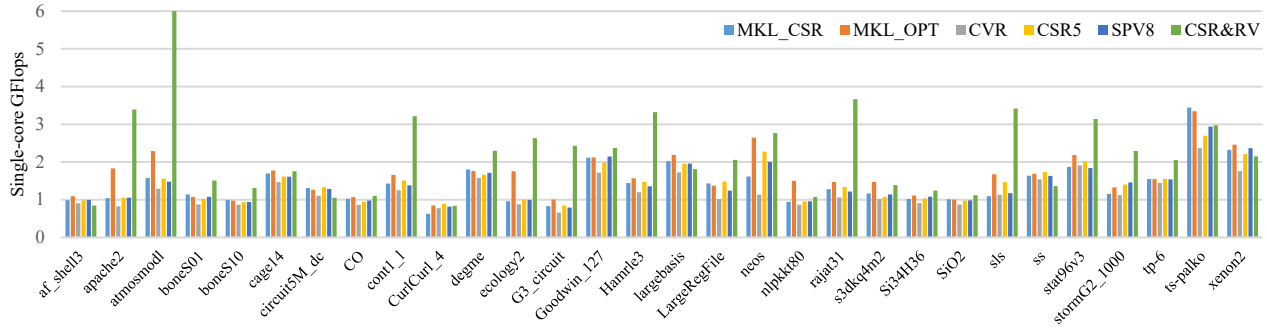


Fig. 8. The single-core performance of different formats in benchmark.

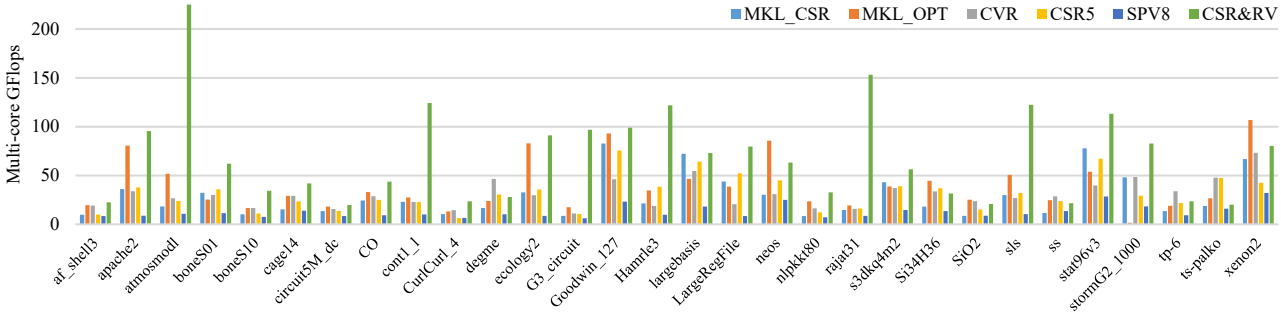


Fig. 9. The multi-core performance of different formats in benchmark.

E. Memory overhead

In this section, we talk about the memory overhead in different formats. Cause the memory space in our baseline formats is similar to the CSR format, we only compare our work with the CSR format. Fig. 10 shows the memory overhead tested results in the benchmark matrixes. The x-axis represents the different matrixes introduced in TABLE I. The main y-axis represents the memory space taken by one nnz for average, on the left. The memory space token by CSR&RV is smaller than CSR in all matrixes. For CSR format, most matrixes take place about 12 bytes for one nnz, which fits to (3). For CSR&RV format, when using *int32* as the type of *csrv_vals_idx* array, the bytes per nnz are near eight, e.g., *af_shell3*, *CO*, and *xenon2*; when using *int16* as the type, the bytes per nnz are close to six, e.g., *apache2*, *cage14*, *cont1_l*; when uses *int8*, the bytes per nnz are five approximately, e.g., *boneS10*, *degme* and *stat96v3*. That results in accord with (4). The subsidiary y-axis represents the memory-reduction rate, which is defined as the reduced storage size of CSR&RV divided by the original storage size of CSR. The experimental results are between 0.3 and 0.6, which conform to the theoretical computations in (6) and (7). Compared with CSR, the memory space in the CSR&RV format is reduced by 48.57% on average and 58.13% on maximum.

F. Pre-processing

Fig. 11 shows the pre-processing overload in different formats on our benchmark matrixes. The pre-processing overload is defined as T_{conv} / T_{comp}. The T_{conv} presents once single-core converting time and the T_{comp} presents once single-core SpMV time. According to Fig. 11, the pre-processing overload in CSR&RV is the second lowest (an average of about 3×), only CVR is faster than it (an average of about 2×). And the overload in MKL_OPT is the highest, which needs about an average of 10× SpMV time to complete once format convection. This result shows that the CSR&RV has the capability for actual use.

VII. Related Work

SpMV is an important kernel function in scientific computation, parallel computing, machine learning et.al. and it has been studied for decades. Scholars had designed many formats according to different architectures, applications, and the feature of matrixes. The classic format CSR [5] is one of the most widely used formats and many formats for overcoming the defects of CSR are presented. For example, the BCSR [16] uses a block algorithm to enhance the locality; the RCSR [11] uses the half-word index to reduce the memory space, and the CSX [12] format exploits the substruction in sparse matrixes to compress the storage space.

Bell et al. [17] proposed the hybrid format called HYB. It improves the SpMV efficiency in irregular distribution sparse matrixes on high throughput processors, which uses the ELL to save the regular part of sparse matrixes and uses the COO to save the irregular part. However, this format does not use vectorization. The VHCC [18] proposed by Tang et al. is a vectorized HYB, which uses the 2D jagged method and prefix sum method to improve the efficiency in scale-free matrixes on the Intel Xeon Phi.

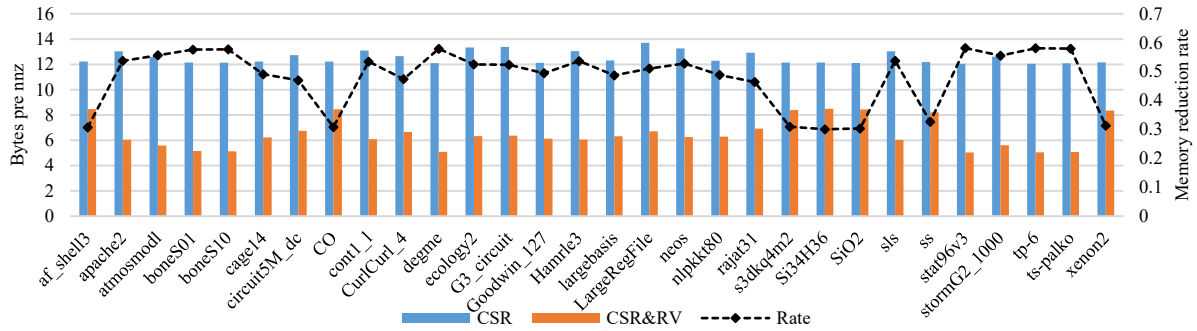


Fig. 10. Memory reduction in the CSR&RV format compares to CSR.

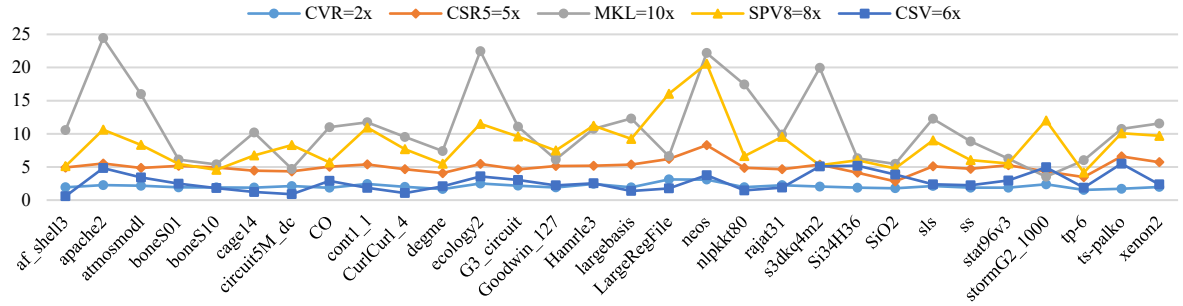


Fig. 11. The pre-processing overload in different formats.

There are many formats designed for utilizing the vectorization ability in modern processors. For example, Liu et al. present the CSR5 [8] implemented with SIMD instructions, which uses segmented sum for better loading balance and tiled mothed for better memory accessed efficiency. This format has a well cross-platform ability in CPU, GPU, and Intel Xeon Phi, but the representation of this format is complex. To streamline the CSR5 format, Bian et al. present an easier understood version called CSR2 and implemented it with the AVX2 instructions. The CSR5 format runs efficiently in the HPC matrixes but is not suited for scale-free matrixes. Targeting the scale-free matrixes, Xie et al. [10] proposed the CVR format. It is a vectorized format that uses a column-based storage pattern and uses trackers to convert all rows fitted to the SIMD lane. This format has a high vectorial rate and gets high efficiency both in scale-free and HPC matrices. Although the CVR has remarkably vectoral rate, the reuse of x is still not high. Thus, the CVB [7] uses the block algorithm to optimize the data locality in CVR format moreover. Noticing the above formats forget the problem of bad branch forecasting in the scale-free matrixes, Li et al. [15] propose the SPV8 format. This format sorts the rows by their length and split the matrix into multiple panels and one fragment stack. Cause the length of rows in the same panel are equal, the missing branch forecasting decreased.

There are also many formats using block algorithms as the main tool to enhance the data locality. For example, the ESB [9] (ELLPACK Sparse Block), Sliced-ELL [19] and SELL-C-σ [20] are blocking formats based on the ELLPACK (ELL), because ELL is quite suitable for vectorization [21]. The only difference in these formats is the blocking strategy. Sliced-ELL format divides the data from ELL into equal-size chunks and SELL-C-σ reduces the padding size in Sliced-ELL by sorting the rows. While ESB uses bitmaps instant of padding zeros, this blocking strategy makes it easier to vectorize. However, when the distribution of elements is very uneven, above formats cannot get high efficiency. Therefore, Chen et al. [24] proposed a one-dimensional storage format. This format padding every row to the same length which ignored the 2-d feature in sparse matrixes and is friendly to SIMD, but when the length of rows in successive rows is variable, the performance will be damaged. Li et al. [21] proposed the VBSF format which does not restrict the blocks to a fixed size, the variable blocking size can improve the data re-use of matrix A and vector x moreover, but it still needs pad many zeros. To overcome the padding used in block algorithms, Bramas et al. [22] propose the SPC5, a blocking format without padding, which uses bitmasks to replace the zero elements in padding blocks.

VIII. Conclusion

In this paper, we propose the CSR&RV, an efficient value compression format to solve the problem that existing compressed formats save too many repetitive values in the value array. The CSR&RV format is designed for compressing the value array in CSR format to reduce the memory space and bandwidth overload, and to improve the SpMV efficiency. We conduct a series of experiments on an Intel Xeon CPU and compare it with five state-of-art formats in 30 real-word sparse matrixes. The experimental results show that our format can get the best throughput both in single-core and multi-core. By using index compression, our format can reduce an average of about 50% (maximum of near 60%) of memory space. Moreover, the CSR&RV format is program friendly and the pre-processing overhead is low, which means our format has the potential to be employed in real-world applications.

Acknowledgment

This research work was supported in part by the National Key Research and Development Program of China (2021YFB0300101). The datasets used and/or analyzed during the current study are available from the corresponding author on reasonable request. The authors declare no conflict of interest. The code of CSR-RV is available at GitHub: https://github.com/junjun-yan/CSR-RV.

REFERENCES

[1] J. Shen, A. L. Varbanescu, P. Zou, Y. Lu, and H. Sips, "Improving Performance by Matching Imbalanced Workloads with Heterogeneous Platforms," in *ICS '14: Proceedings of the 28th ACM international conference on Supercomputing*, Munich Germany, 2014.

[2] Y. Saad, "Iterative methods for sparse linear systems second edition," in *SIAM*, 2003.

[3] X. Sun, Y. Zhang, T. Wang, G. Long, X. Zhang, and Y. Li, "CRSD: Application Specific Auto-tuning of SpMV for Diagonal Sparse Matrices," Bordeaux, FRANCE, 2011.

[4] K. A. B. R. Asanovic, "The Landscape of Parallel Computing Research: A View from Berkeley," 2006.

[5] B. R., "Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods," in *SIAM,* 1994.

[6] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, "Performance evaluation of the sparse matrix-vector multiplication on modern architectures," in *Journal of Supercomputing,* pp. 36-77, 2009.

[7] N. Namashivayam, S. Mehta and P. Yew, "Variable-sized blocks for locality-aware SpMV," in *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, Virtual Event, Republic of Korea, 2021, pp. 211‑221.

[8] Liu, W., Vinter, B, "CSR5: An Efficient Storage Format for Cross Platform Sparse Matrix-Vector Multiplication," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015.

[9] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *ICS '13:Proceedings of the 27th international ACM conference on International conference on supercomputing*, Eugene, Oregon, USA, 2013.

[10] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, and L. Zhang, "CVR: efficient vectorization of SpMV on x86 processors," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, Vienna, Austria, 2018, pp. 149‑162.

[11] M. Martone, S. Filippone, M. Paprzycki, and S. Tucci, "On the Usage of 16 Bit Indices in Recursively Stored Sparse Matrices," in *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2010)*, Timisoara, Romania, 2010.

[12] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris, "CSX: An Extended Compression Format for SpMV on Shared Memory Systems," in *ACM SIGPLAN NOTICES,* vol. 46, pp. 247-256, 2011-01-01 2011.

[13] S. S. L. G. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix‑vector multiplication on emerging multicore platforms," in *Parallel Computing,* pp. 178-194, 2009.

[14] T. A. A. E. Davis, "The University of Florida Sparse Matrix Collection(Article)," in *ACM Transactions on Mathematical Software,* p. 1(1-25), 2011.

[15] C. Li, T. Xia, W. Zhao, N. Zheng, and P. Ren, "SpV8: Pursuing Optimal Vectorization and Regular Computation Pattern in SpMV," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, San Francisco, CA, USA, 2021.

[16] R. W. Vuduc and H. J. Moon, "Fast sparse matrix-vector multiplication by exploiting variable block structure," in *1st International on High Performance Computing and Communications (HPCC 2005)*, 2005, pp. 807-816.

[17] N. Bell, M. Garland and IEEE, "Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors," in *Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[18] W. T. Tang, R. Zhao, M. Lu, Y. Liang, P. H. Huynh, X. Li, R. S. M. Goh, and IEEE, "Optimizing and Auto-Tuning Scale-Free Sparse Matrix-Vector Multiplication on Intel Xeon Phi," in *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015, pp. 136-145.

[19] A. Monakov, A. Lokhmotov and A. Avetisyan, "Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures," in *International Conference on High Performance Embedded Architectures and Compilers (HiPEAC 2010) 5th*, Pisa, Italy, 2010.

[20] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-vector Multiplication in Modern Processors with Wide SIMD Units," in *SIAM Journal on Scientific Computing,* vol. 36, pp. C401-C423, 2014-01-01 2014.

[21] Y. Li, P. Xie, X. Chen, J. Liu, B. Yang, S. Li, C. Gong, X. Gan, and H. Xu, "VBSF: a new storage format for SIMD sparse matrix-vector multiplication on modern processors," in *Journal of Supercomputing,* vol. 76, pp. 2063-2081, 2020-01-01 2020.

[22] B. Bramas and P. Kus, "Computing the sparse matrix vector product using block-based kernels without zero padding on processors with AVX-512 instructions," in *Peerj Computer Science,* 2018-01-01 2018.

[23] Wang, E., Zhang, Q., Shen, B., Zhang, G., Lu, X., Wu, Q., and Wang, Y, "Intel Math Kernel Library," in *Springer International Publishing*, 2014.

[24] X. Chen, P. Xie, L. Chi, J. Liu, and C. Gong, "An efficient SIMD compression format for sparse matrix-vector multiplication," in *Concurrency & Computation: Practice & Experience*, p. e4800, 2018.