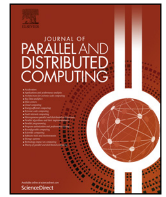
Journal of Parallel and Distributed Computing 158 (2021) 126–137

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc



Performance analysis and optimization for SpMV based on aligned storage formats on an ARM processor ☆

Yufeng Zhang [a,b], Wangdong Yang [a,b,*], Kenli Li [a,b], Dahai Tang [a,b], Keqin Li [a,b,c]

[a] College of Information Science and Engineering, Hunan University, Changsha, Hunan 410082, China
[b] The National Supercomputing Center in Changsha, Changsha, Hunan 410082, China
[c] Department of Computer Science, State University of New York, New Paltz, NY 12561, USA

ARTICLE INFO

Article history:
Received 24 December 2020
Received in revised form 10 July 2021
Accepted 5 August 2021
Available online 18 August 2021

Keywords:
ARM
NEON
SIMD
SpMV
Storage formats

ABSTRACT

Sparse matrix-vector multiplication (SpMV) has always been a hot topic of research for scientific computing and big data processing, but the sparsity and discontinuity of the nonzero elements in a sparse matrix lead to the memory bottleneck of SpMV. In this paper, we propose aligned CSR (ACSR) and aligned ELL (AELL) formats and a parallel SpMV algorithm to utilize NEON SIMD registers on ARM processors. We analyze the impact of SIMD instruction latency, cache access, and cache misses on SpMV with different formats. In the experiments, our SpMV algorithm based on ACSR achieves 1.18x and 1.56x speedup over SpMV based on CSR and SpMV in PETSc, respectively, and AELL achieves 1.21x speedup over ELL. The deviations between the theoretical results and experimental results in the instruction latency and cache access are 10.26% and 10.51% in ACSR and 5.68% and 2.91% in AELL, respectively.

© 2021 Elsevier Inc. All rights reserved.

1. Introduction

1.1. Motivation

Sparse matrix-vector multiplication (SpMV) is one of the core subroutines in numerical computation. The solution of large-scale linear equations is one of the major applications, and an exact solution is usually accessed by an iterative method. SpMV, a key step in solving systems of linear equations, may be performed thousands of times in the solution process. However, the complexity of the associated hardware and the load imbalance caused by the sparsity of sparse matrices can lead to memory bottlenecks, making it challenging to optimize the SpMV performance.

SpMV computes $\vec{y}$ as given by

$$\vec{y} = A\vec{x} + \vec{b}, \quad (1)$$

where A is a fixed sparse matrix in the iterative method. To improve the utilization of processors when calculating SpMV, the zero elements of A should be ignored in the calculation. Therefore, many sparse matrix storage formats are delivered, which reflect the distribution characteristics of nonzero elements in a sparse matrix. Some of these formats are suitable for the structural characteristics of the computing platform. Although researchers have performed many studies on SpMV, most of them are aimed at the x86 multicore platform or other accelerators, and little work has addressed ARM processors for SpMV.

ARMv8-A is an architecture for high-performance computing introduced by ARM. An increasing number of researchers have been drawn to the ARMv8-A architecture, as it supports 64-bit instruction sets, improving the double-precision floating-point arithmetic capability, and supporting single instruction multiple data (SIMD) operations via NEON, which is a SIMD instruction extension architecture of ARM. In addition, with the advent of the intelligent era, ARM processors have been widely used in mobile devices with their small size, low energy consumption, low cost, and good performance. Moreover, in the latest global supercomputer list re-

☆ The research was partially funded by the National Key R&D Program of China (grant nos. 2018YFB0204302), Scientific Challenges Special Subject (grant nos. TZZT2019-B2.1), the Key Program of National Natural Science Foundation of China (grant no. 92055213), and the National Natural Science Foundation of China (grant nos. 61872127 and 61751204).

* Corresponding author at: College of Information Science and Engineering, Hunan University, Changsha, Hunan 410082, China.
E-mail address: yangwangdong@163.com (W. Yang).

https://doi.org/10.1016/j.jpdc.2021.08.002
0743-7315/© 2021 Elsevier Inc. All rights reserved.

leased in November 2020, Fugaku with a 48-core A64FX SOC based on ARMv8-A retains the title, which is the first system supported by ARM processors to top the list.

1.2. Our contributions

In this paper, we

- propose the aligned storage formats ACSR and AELL, which are suitable for NEON in double-precision calculations on ARM processors.
- put forward a parallel SpMV algorithm based on ACSR and AELL formats.
- evaluate the performance of our ACSR and AELL formats and compare them with the CSR and ELL formats, and PETSc on Kunpeng 920 processors.

Our aligned compressed storage formats can improve the performance of SpMV with NEON on the ARMv8-A platform, and the acceleration of SpMV based on ACSR and AELL compared with general CSR and ELL is obtained from theoretical analysis and experimental verification.

First, we present the aligned storage formats ACSR and AELL based on the CSR and ELL formats, which align the SIMD registers of ARM processors. Second, we demonstrate four parallel algorithms of SpMV with NEON acceleration. Third, we conduct a performance analysis for the impact of two aligned formats, two general formats, and the difference between them on the execution latency of SIMD instructions, the cache access, and the cache misses. Fourth, we select 20 sparse matrices from the SuiteSparse Matrix Collection for experiments, which come from a wide range of practical application scenarios. Moreover, we examine the calculation performance of SpMV based on CSR and ELL with NEON and compare the SpMV based on ACSR and AELL with NEON accelerated SpMV in CSR, ELL, and SpMV in PETSc. Finally, we compare and analyze the experimental results with our performance analysis results. For the 20 matrices, the average performance improvements of NEON accelerated SpMV reach 19.91% in CSR and 19.15% in ELL. For the use of NEON, compared to ELL, AELL achieves an average performance improvement of 20.54%, 34.10%, and 28.85% in estimating time, cache access, and cache misses in double precision, respectively, while those of ACSR are 18.17%, 18.59%, and 18.00% compared to those of CSR. In addition, the average acceleration is 56.46% higher for SpMV based on ACSR than for SpMV in PETSc.

The experiment shows that our ACSR and AELL storage formats are well suited for NEON SIMD operations on ARM processors.

2. Related work

2.1. Compressed storage formats for sparse matrices

In general, to enhance the performance, the operations of zero elements in a sparse matrix must be reduced. Therefore, a sparse matrix can be compressed to store the nonzero elements, such as the original coordinate (COO) format and the compressed sparse column/row (CSC/CSR) [19]. Although these formats are widely used, they may not be adaptable enough for some special sparse matrices. Therefore, some storage formats for the specific sparse matrices have been introduced. For example, the diagonal (DIA) [14] format stores only the nonzero elements of a sparse matrix diagonally, and the ellpack (ELL) [5] format is suitable for the matrix with relatively uniform rows of nonzero elements. Designing a storage format aimed at the hardware features of a specific processor can maximize the performance [1]. Examples include the hybrid ELL/COO (HYB) [3], the sliced ellpack (SELL-C-σ) [13], compressed sparse row 5 (CSR5) [10], the blocked stored format mixed CSR and ELL (BCE) [21] and so on. All of the above formats are in full use as massive parallel computing features of the GPU or the SIMD device of the x86 CPU.

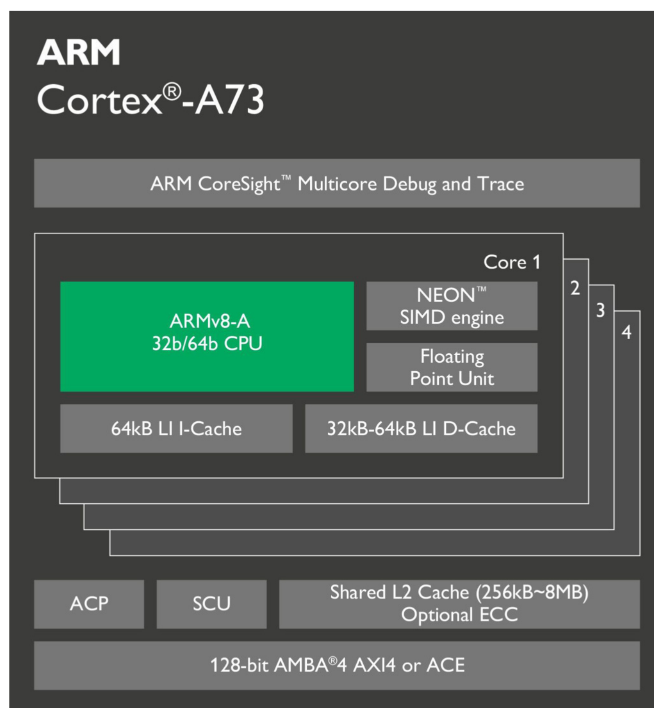


Fig. 1. ARMv8-A architecture.

Nevertheless, the scale of the sparse matrix obtained from practical applications is getting increasingly large, which may preclude obtaining the results immediately. As a consequence, matrix partitioning has become a valuable approach that ensures all elements in a block can be efficiently calculated immediately [12]. Karakasis et al. [7] compared the performance of several blocked storage formats. The blocked compressed sparse row (BCSR) [5] was used to exploit the computational performance of dense subblocks in a sparse matrix. Subsequently, Vuduc et al. [18] optimized the BCSR format to the unaligned block compressed sparse row (UBCSR) format and developed a better performance of dense subblocks of different sizes. In addition, to explore the substructures in a sparse matrix, the compressed sparse extended (CSX) [8] storage format was proposed to compress the index arrays.

2.2. ARMv8-A architecture

For far too long, mobile devices have been equipped with ARM cores because of the low price, high performance, and low power consumption [4]. With the debut of the ARMv8-A architecture, ARM has become a key research object for solving energy consumption in high-performance computing. Moreover, compared to the previous ARM architecture, 64-bit ARMv8-A benefits from increased support of double-precision and SIMD operations [9], making it more compelling [2,6,11,17].

Many companies worldwide have purchased an ARMv8-A license and have developed their cores based on ARM. Examples include the United States companies Cavium and Qualcomm, the Japanese company Fujistu, the Chinese companies Huawei and Phytium, etc. All of these companies have already released several server-level chips based on the ARMv8-A architecture shown in Fig. 1. Furthermore, there are four supercomputers equipped with

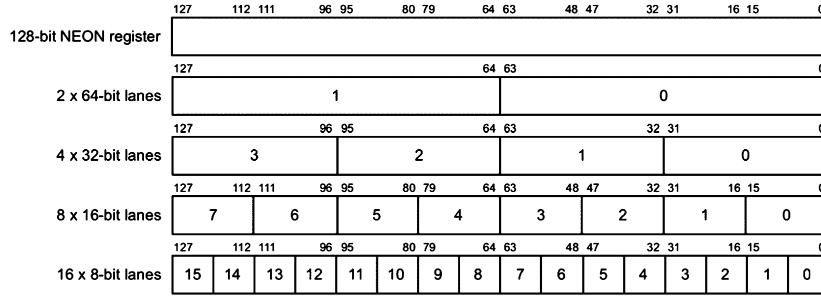


Fig. 2. NEON SIMD registers.

ARM processors in the latest TOP500. Fugaku with 48-core A64FX SOC produced by Fujistu [22] tops the list among them.

3. Sparse matrix compression and NEON technology

In this section, we introduce two general formats, CSR and ELL, which our aligned formats are based on. Because our algorithms are designed to deliver the performance of NEON on ARM processors, we provide a brief illustration of NEON in Section 3.2.

3.1. General storage formats for sparse matrices

To show the general storage formats visually, we assume that there is a sparse matrix

$$A = \begin{pmatrix} a & b & 0 & 0 & 0 \\ 0 & 0 & c & 0 & 0 \\ d & 0 & 0 & 0 & e \\ 0 & f & g & 0 & 0 \end{pmatrix}. \tag{2}$$

Then, we define three parameters, namely, M is the number of rows, N is the number of columns, and NNZ is the number of nonzero elements in the sparse matrix A.

3.1.1. Compressed sparse rows (CSR)

CSR, the most commonly used storage format, is not sensitive to the characteristics of a matrix. This format stores the nonzero elements in rows and stores the row information of the sparse matrix by indexing the first nonzero element of each row. Therefore, the CSR format employs two arrays of length "NNZ" to store all of the nonzero elements and the corresponding column index of nonzero elements, and another array of length "$M+1$" is used to store the row index. Therefore, the sparse matrix A can be represented by CSR as

$$values = \begin{pmatrix} a & b & c & d & e & f & g \end{pmatrix},$$
$$columns = \begin{pmatrix} 0 & 1 & 2 & 0 & 4 & 1 & 2 \end{pmatrix},$$
$$row_ptr = \begin{pmatrix} 0 & 2 & 3 & 5 & 7 \end{pmatrix}.$$

In addition, the memory space for the CSR format in double-precision S_{CSR} is given by

$$\begin{aligned} S_{CSR} &= NNZ \times (4+8) + (M+1) \times 4 \\ &= 12NNZ + 4(M+1). \end{aligned} \tag{3}$$

3.1.2. Ellpack (ELL)

In ELL format, a sparse matrix is compressed into two smaller dense matrices that contain the filled nonzero elements and the corresponding column indices. Assume that the maximum number of nonzero elements in a row in the sparse matrix is "k", so the size of two dense matrices is "$N \times k$". Moreover, the vacant places should be filled with zeros in the dense matrices. Thus, the sparse matrix A can be shown in ELL as

$$ELL_values = \begin{pmatrix} a & b \\ c & 0 \\ d & e \\ f & g \end{pmatrix}, ELL_columns = \begin{pmatrix} 0 & 1 \\ 2 & 0 \\ 0 & 4 \\ 1 & 2 \end{pmatrix}.$$

The size of the ELL in the memory in double precision is

$$\begin{aligned} S_{ELL} &= M \times K \times (4+8) \\ &= 12MK. \end{aligned} \tag{4}$$

3.2. NEON technology

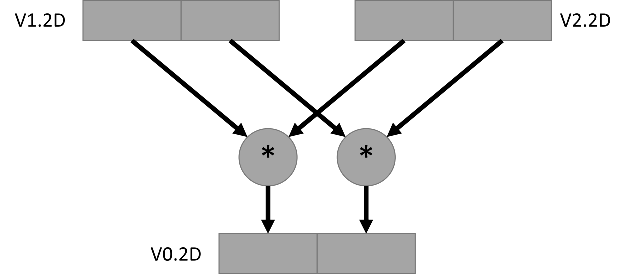


Fig. 3. Double-precision floating-point SIMD operation.

As an extended architecture of ARM, NEON offers 128-bit SIMD operations. For ARMv8-A, there are 32 128-bit NEON SIMD vector registers that support multiple data types and up to 64-bit double-precision floating-point precision as shown in Fig. 2. Moreover, using the NEON SIMD registers allows users to process data efficiently and minimize memory access.

When using NEON to accelerate SpMV, the vector x needs to be loaded into NEON SIMD registers one after another because of the discontinuity of column indices in CSR and ELL formats. In contrast, while applying the ACSR and AELL formats in SpMV, an SIMD register can be filled immediately both in sparse matrix A and in dense vector x.

For example, V0.2D, V1.2D, and V2.2D in Fig. 3 represent three SIMD registers, where "2D" means that a register stores a 2×64-bit vector. Suppose **V1** stores the value of a sparse matrix A and **V2** stores the value of the vector x in general SpMV. Therefore, the value of the vector should be assigned to the components of **V2** sequentially in scalar form. Subsequently, the SIMD operator simultaneously calculates two multiplication steps ($V1.2D[0] \times V2.2D[0]$, $V1.2D[1] \times V2.2D[1]$) and stores the result in register V0.2D.

4. Aligned compressed storage formats

When using the SIMD units for vector calculation, several consecutive elements are usually immediately loaded into the vector

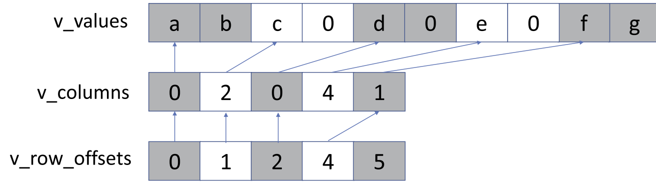


Fig. 4. ACSR format.

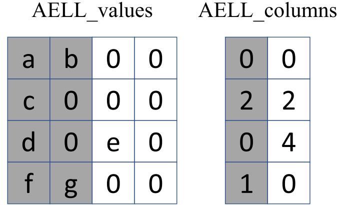


Fig. 5. AELL format.

registers in the form of vectors. However, in SpMV, since the distribution of nonzero elements is discontinuous as a whole, the constituent of the corresponding vector x can only be loaded to the vector register sequentially [20]. The increase in load steps for the vector registers leads to the efficiency of the SIMD unit degradation. Therefore, to improve the utilization of SIMD units, we design the aligned compressed storage formats ACSR and AELL for sparse matrices to fit the ARM advanced SIMD architecture. After the compression of a sparse matrix, if the column indices of two adjacent nonzero elements are also adjacent, the corresponding two elements in the vector multiplied by the two nonzero elements are adjacent when calculating the SpMV. If the column indices of the two nonzero elements are not adjacent, the corresponding two elements in the vector are also not adjacent. If the distance between the two nonadjacent elements exceeds the width of the cache line, it is impossible to load two elements into a vector register at one time. If the deviation between the column indices of the two is less than 8, we consider that they lie on the same cache line and insert zeros before the current nonzero element. If not, zeros are inserted after the current nonzero element, which guarantees that the filling operation does not cause redundant cross-cache lines as much as access permits.

The matrix A in Section 3 is an example to be used for ACSR and AELL storage formats as follows.

4.1. Aligned CSR (ACSR)

As shown in Fig. 4, there are three arrays in ACSR.

The v_values stores the vector values (including the filled zero elements) by rows, $v_columns$ stores the first column index of each vector, and v_rowptr stores the first index of $v_columns$ of each row. Assuming that the zero filling rate is "FR" ($0 \leq FR \leq 1$), there are $NNZ \times (1 + FR)$ elements in ACSR, and the storage space size of the sparse matrix compressed by the ACSR format is

$$\begin{aligned} S_{ACSR} &= NNZ \times (1 + FR) \times (8 + 4/2) + (M + 1) \times 4 \\ &= 10NNZ\,(1 + FR) + 4\,(M + 1)\,. \end{aligned} \tag{5}$$

According to the calculation formulas (3) (5) of the occupation space, we can conclude that the ACSR format needs more space than CSR while $FR > 0.2$.

4.2. Aligned ELL (AELL)

Similar to ELL, AELL uses two matrices to store the vectors in Fig. 5. The values of vectors are stored in $AELL_values$, and $AELL_columns$ stores the index of each vector. We set the number of vectors of each row as V_K in the two matrices of AELL so that the space occupied by AELL is

$$\begin{aligned} S_{AELL} &= V_K \times M \times (2 \times 8 + 4) \\ &= 20MV_K\,. \end{aligned} \tag{6}$$

Equally, we can find that the AELL formats occupy more space, while $V_K \geq 0.6K$ by Eqs. (4) and (6).

Table 1
NEON SIMD operations for 64-bit double-precision floating-point operation.

Function	Meaning
float64 × 2_t vector	Declare a vector of size 64 × 2
vdupq_n_f64(0.0)	Initialize the vector
vld1q_lane_f64(x+j,0)	Take $x[j]$ to the 0th lane of vector register
vld1q_f64(x+j)	Take two elements from $x[j]$ to vector register
vmlaq_f64(temp,v_A,v_x)	Execute vector operation: $temp+=v_A \times v_x$
vget_lane_f64(temp,0)	Get the 0th component of the vector $temp$

5. Parallel algorithms of SpMV

In the experiments in this paper, we use the NEON intrinsics provided by ARM to achieve NEON acceleration. The functions are shown in Table 1, where the letter "q" in the operation instruction indicates the use of 128-bit vector registers.

Algorithms 1 and 2 are the SpMV algorithms based on CSR and ELL formats with NEON acceleration, respectively, as we can only put x into the vector registers individually.

Algorithm 1 The kernel function of CSR SpMV with NEON.

Require: The vector x, row number of sparse matrix row_{num} and the arrays in CSR format $(values, columns, rowptr)$
Ensure: The result vector y
1: $float64 \times 2_t\ matrix$;
2: $float64 \times 2_t\ vector$;
3: $float64 \times 2_t\ temp_y$;
4: $temp_y = vdupq_n_f64(0.0)$;
5: **for** $i = 0$ to $row_{num} - 1$ **do**
6: **for** $j = rowptr[i];\ j < rowptr[i+1];\ j+=2$ **do**
7: $matrix = vld1q_f64(values + j)$;
8: $vector = vld1q_lane_f64(x + columns[j])$;
9: $vector = vld1q_lane_f64(x + columns[j+1])$;
10: $temp_y = vmlaq_f64(temp_y, matrix, vector)$;
11: **end for**
12: $y[i] += vget_lane_f64(temp_y, 0)$
13: $+\ vget_lane_f64(temp_y, 1)$;
14: **end for**

Algorithm 2 The kernel function of ELL SpMV with NEON.

Require: The vector x, row number of sparse matrix row_{num} and data in ELL format $(K, ELL_columns, ELL_values)$
Ensure: The result vector y
1: $float64 \times 2_t\ matrix$;
2: $float64 \times 2_t\ vector$;
3: $float64 \times 2_t\ temp_y$;
4: $temp_y = vdupq_n_f64(0.0)$;
5: **for** $i = 0;\ row_{num} - 1$ **do**
6: **for** $j = i \times K;\ j < (i+1) \times K;\ j+=2$ **do**
7: $matrix = vld1q_f64(ELL_values + j)$;
8: $vector = vld1q_lane_f64(x + ELL_columns[j])$;
9: $vector = vld1q_lane_f64(x + ELL_columns[j+1])$;
10: $temp_y = vmlaq_f64(temp_y, matrix, vector)$;
11: **end for**
12: $y[i] += vget_lane_f64(temp_y, 0)$
13: $+\ vget_lane_f64(temp_y, 1)$;
14: **end for**

As shown in the Algorithms 3 and 4, x can be continuously stored in the vector registers when performing SpMV in ACSR and

AELL formats. In addition, note that the length of x is $columns + 1$, as we need to add a zero element at the end of x, which may cause access overflow in error.

Algorithm 3 Kernel function of ACSR SpMV with NEON.

Require: The vector x, row number of sparse matrix row_{num} and arrays in ACSR format $(v_values, v_columns, v_rowptr)$
Ensure: The result vector y
1: $float64 \times 2_t\ matrix$;
2: $float64 \times 2_t\ vector$;
3: $float64 \times 2_t\ temp_y$;
4: $temp_y = vdupq_n_f64(0.0)$;
5: **for** $i = 0$ to $row_{num} - 1$ **do**
6: **for** $j = rowptr[i]$ to $v_rowptr[i+1] - 1$ **do**
7: $matrix = vld1q_f64(v_values + j \times 2)$;
8: $vector = vld1q_f64(x + v_columns[j])$;
9: $temp_y = vmlaq_f64(temp_y, matrix, vector)$;
10: **end for**
11: $y[i] += vget_lane_f64(temp_y, 0)$
12: $+\ vget_lane_f64(temp_y, 1)$;
13: **end for**

Algorithm 4 The kernel function of AELL SpMV with NEON.

Require: The vector x, row number of sparse matrix row_{num} and data in AELL format $(V_K, AELL_values, AELL_columns)$
Ensure: The result vector y
1: $float64 \times 2_t\ matrix$;
2: $float64 \times 2_t\ vector$;
3: $float64 \times 2_t\ temp_y$;
4: $temp_y = vdupq_n_f64(0.0)$;
5: **for** $i = 0$ to $row_{num} - 1$ **do**
6: **for** $j = i \times V_K$ to $(i+1) \times V_K$ **do**
7: $matrix = vld1q_f64(AELL_values + j \times 2)$;
8: $vector = vld1q_f64(x + AELL_columns[j])$;
9: $temp_y = vmlaq_f64(temp_y, matrix, vector)$;
10: **end for**
11: $y[i] += vget_lane_f64(temp_y, 0)$
12: $+\ vget_lane_f64(temp_y, 1)$;
13: **end for**

6. Performance analysis

Although our ACSR and AELL formats are filled with zeros based on the CSR and ELL formats that increase the amount of calculation, this can improve the efficiency of memory access by loading a vector to the vector register in one step. Therefore, with different filling rates, the performance of SpMV may have varying degrees of fluctuation. In this part, we analyze the execution latency of NEON SIMD instruction, the impact of cache access, and the cache miss in SpMV based on different compressed storage formats.

6.1. SpMV performance analysis for ARMv8 architecture

To perform performance analysis, some variables are shown in Table 2.

Table 2
Variables for analyzing.

Variable	Meaning
T	The computation time of SpMV
L	The execution time of instructions
A	The data access time
Li	The register fetch instruction latency
Ci	The calculation instruction latency
AC_{num}	The number of cache access
AC_t	The latency of cache access
$miss_{num}$	The times of cache misses
AM_t	The latency of memory access

Table 3
Latency of main instructions in SpMV.

Function	Instruction	Latency
	LDR	4
vld1q_lane_f64($x + j$,0)	LD1	5
vld1q_f64($x + j$)	LD1	5
vmlaq_f64($temp$,v_A,v_x)	FMLA	7(3)

For a computing program, the computing scale depends mainly on the number of instructions and the amount of data to be accessed. For an SpMV computation, the number of instructions is determined by the number of nonzero elements in the sparse matrix, and the amount of data accessed is related to the storage format of the sparse matrix and the access method of a right vector. However, different storage formats may have different proportions of zero elements filling and additional auxiliary data storage, resulting in different data access. Therefore, the actual running performance of a parallel program is related not only to the calculation scale of the program itself but also to the processor's instruction execution cycle, parallel execution of instructions, utilization efficiency of cache data, number of reads and writes of memory data, bandwidth utilization, and utilization rate of all computing cores of the processor, etc. To reduce the execution cycle of instructions, the instructions with a short execution cycle can be chosen. In addition, we can distribute the instructions to different computing cores for parallel execution. This can also improve the data amount of instructions by pipeline and SIMD. The performance of SpMV serial operation on the ARMv8 processor can be described by

$$T = f(L, A), \tag{7}$$

where L and A are the instruction execution time and data access time, respectively. The performance T of a program is positively correlated with the number of instructions and the number of data accesses. Thus, the function f is an increasing function. L is determined by the total number of instructions and the instruction latency of the processor. For a program, the execution latency mainly includes the latency of register loading and calculator operation. For the ARMv8 processor, the register loading includes data loading of a general register and vector loading of a vector register. Calculation instructions include general calculation instructions and vector calculation instructions. Thus, the execution latency time L can be calculated by

$$L = Li_n + Li_s + Ci_n + Ci_s, \tag{8}$$

where Li_n, Li_s, Ci_n, and Ci_s are the latency of general register loading, vector register loading, general calculation instructions, and vector calculation instructions, respectively. There are mainly addition and multiplication steps for SpMV, and the number of operating instructions is related to the nonzero elements of the sparse matrix. The number of multiplication and addition steps are NNZ and $NNZ - M$, respectively. Therefore, the total number of instructions for SpMV operation is determined according to the sparse matrix. However, the calculation time can be reduced by choosing instructions with short execution delay and using more instruction units simultaneously. Table 3 shows the clock cycles of different computing instructions on the ARMv8 processor. Instruction pipelining and SIMD technology can improve the concurrent operation of multiple instructions, which leads to the improvement of the throughput of instruction execution. For example, if the length of the vector arithmetic unit is M bytes and the floating-point number is 4 bytes, $M/4$ floating-point numbers can be calculated in a vector operation. In this way, it is equivalent to the

SpMV requirement to calculate NNZ multiplication steps to perform $NNZ/(M/4)$ vector multiplication steps. If the data access is limited to the cache data read by the instruction execution and the memory access caused by cache miss, A can be calculated by

$$A = AC_{num} \times AC_t + Miss_{num} \times AM_t, \tag{9}$$

where AC_{num}, AC_t, $Miss_{num}$, and AM_t are the number of accesses to the cache, the delay of cache access, the number of cache misses, and the delay of memory access, respectively. For the processor, the latency of accessing the cache and memory is fixed. Therefore, to improve the performance of data access, the program should reduce the number of cache accesses and the cache miss rate.

6.2. The execution latency of instructions

According to the execution latency described in the ARM official document, the minimum latency can be obtained by analyzing an operation dependent on an instruction in the described group. We analyze the difference of ASIMD instructions latency between the SpMV based on ACSR and AELL formats as well as CSR and ELL formats.

The main latency of instructions is shown in Table 3. The column indices of nonzero elements are loaded into the normal register by "LDR", and a 64-bit number or a 128-bit vector is loaded into the vector register by "LD1", which uses function vld1q_lane_f64($x+j$,0) or vld1q_f64($x+j$). Two vectors are multiplied and added to the third vector to be evaluated by "FMLA", the function vmlaq_f64($temp$,v_A,v_x) that evaluates

$$temp \leftarrow temp + v_A \times v_x.$$

In addition, for "FMLA", the number 3 in brackets in Table 3 indicates that the calculation result can be applied after only three cycles when the calculation is complete.

For SpMV using double precision, the column index of each nonzero element must be loaded into the normal register to perform an addressing instruction to obtain the corresponding element in the vector x. Therefore, it is necessary to execute the "LDR" instruction NNZ times. Moreover, in CSR formats, the nonzero elements can be loaded in vector, and there are $\frac{NNZ}{2}$ "LD1" instructions because the 128-bit vector register can load two nonzero elements at a time. However, the elements in vector x can be loaded only sequentially because the two nonzero elements loaded into the 128-bit vector register may not be continuous, which causes the discontinuity of the corresponding two elements in the vector x. Therefore, there are NNZ "LD1" instructions for loading x into the vector register sequentially. Two nonzero elements in a row of the sparse matrix can multiply by two corresponding elements in vector x, and the results of two multiplication steps may be added to the corresponding elements of the result vector by the "FMLA" instruction at the same time. Therefore, there are $\frac{NNZ}{4}$ "FMLA" instructions for SpMV because there are two ASIMD units on a Cortex-A72 processor, which is used by the tested computer in the paper. According to Eq. (8), we have

$$\begin{aligned} L_{CSR} &= NNZ \times 4 + \frac{NNZ}{2} \times 5 + NNZ \times 5 + \frac{NNZ}{4} \times (7+3) \\ &= 14NNZ. \end{aligned} \tag{10}$$

There are $\frac{NNZ'}{2}$ elements to be stored in ACSR format, which include zero elements filled by the alignment operation. Therefore, it is necessary to execute the "LDR" instruction NNZ' times. However, in ACSR format, the two nonzero elements loaded into the vector register are continuous, so the corresponding two elements of x can also be loaded into a vector register because the corresponding two elements in the vector x are continuous. Therefore, there are $\frac{NNZ'}{2} \times 2$ "LD1" instructions for ACSR format. Because the ARMv8 pipeline has two ASIMD units, two vectors can be calculated in one calculation instruction, so there are $\frac{NNZ'}{4}$ "FMLA" instructions for SpMV, and the latency of instructions in ACSR can be given by

$$\begin{aligned} L_{ACSR} &= \frac{NNZ'}{2} \times 4 + \frac{NNZ'}{2} \times 5 \times 2 + \frac{NNZ'}{4} \times (7+3) \\ &= \frac{19}{2} NNZ\,(1+FR)\,. \end{aligned} \tag{11}$$

A sparse matrix is stored in two $M \times K$ dense matrices in ELL format, where one of the dense matrices stores the values, and the other stores the column indices of the nonzero elements. Therefore, there are $M \times K$ "LDR" instructions to load column indices to the general register, and $\frac{M \times K}{2}$ "LD1" instructions to load the values matrices to the vector register. However, the elements in vector x can only be loaded sequentially because the two nonzero elements loaded into the 128-bit vector register may not continuously cause the discontinuity of the corresponding two elements in the vector x. Therefore, there are $M \times K$ "LD1" instructions for loading x into the vector register sequentially. Finally, there are $\frac{M \times K}{2}$ "FMLA" instructions for SpMV as with the CSR format. Because the ARMv8 pipeline has two ASIMD units, the execution latency of the "FMLA" is $\frac{\frac{M \times K}{2} \times (7+3)}{2}$; then, we have

$$\begin{aligned} L_{ELL} &= M \times K \times 4 + \frac{M \times K}{2} \times 5 + M \times K \times 5 + \frac{\frac{M \times K}{2} \times (7+3)}{2} \\ &= 14MK. \end{aligned} \tag{12}$$

For the AELL format, there are $M \times V_K$ "LDR" instructions to load the column indices to the general register. Due to the alignment operations, every two nonzero elements loaded into the vector register are continuous. Therefore, a row of data is divided V_K times and loaded into the vector register, and there are $M \times V_K$ "LD1" instructions to load the value matrix. The corresponding two elements of x can also be loaded into a vector register because the corresponding two elements in the vector x are continuous. Therefore, there are $M \times V_K \times 2$ "LD1" instructions for AELL format. Finally, there are $\frac{M \times V_K}{2}$ "FMLA" instructions for SpMV as with ELL format, and we have

$$\begin{aligned} L_{AELL} &= M \times V_K \times 4 + M \times V_K \times 2 \times 5 + \frac{M \times V_K}{2} \times (7+3) \\ &= 19MV_K. \end{aligned} \tag{13}$$

As a result, we can obtain the latency of ASIMD instructions of SpMV in different formats.

Using the above formulas to analyze the performance of SpMV operation based on different storage formats, we can obtain the following propositions.

Proposition 1. *When $FR < \frac{9}{19}$ in ACSR, the number of cycles spent in SpMV based on the ACSR format is less than that in the CSR format.*

Proof. With the increase in zero paddings, the amount of redundant data that needs to be read into the vector register increases, increasing the number of "LD1" and "FMLA" instructions. FR is the fill ratio of zero elements. Eqs. (10) and (11) represent the clock

cycles of all operation instructions of SpMV using CSR and ACSR formats, respectively. The upper bound of FR can be obtained by

$$LR_{ACSR} = \frac{L_{ACSR} - L_{CSR}}{L_{CSR}} = \frac{19NNZ}{2}\left(FR - \frac{9}{19}\right), \tag{14}$$

which calculates the difference between Eqs. (10) and (11).

To make the performance of SpMV based on the ACSR format not lower than that of SpMV based on the CSR format, LR_{ACSR} should be less than or equal to 0, so we can determine the range of FR as

$$\begin{aligned} LR_{ACSR} &\leq 0; \\ \frac{19NNZ}{2}\left(FR - \frac{9}{19}\right) &\leq 0; \\ FR &\leq \frac{9}{19}. \quad \square \end{aligned}$$

Proposition 2. *When $V_K < \frac{14K}{19}$ in AELL, the number of cycles spent in SpMV based on the AELL format is less than that of the ELL format.*

Proof. The storage space of the sparse matrix with ELL format depends on the row with the most nonzero elements. With the increase in zero padding, the number of redundant data points that need to be read into the vector register increases, increasing the number of “LD1” and “FMLA” instructions. V_K is the width of the row with zero paddings. Eqs. (12) and (13) represent the clock cycles of all operation instructions of SpMV using ELL and AELL formats, respectively. The upper bound of V_K is obtained by

$$LR_{AELL} = \frac{L_{AELL} - L_{ELL}}{L_{ELL}} = 19N\left(V_K - \frac{14}{19}K\right), \tag{15}$$

which calculates the difference between Eqs. (12) and (13).

To make the performance of SpMV based on AELL format not lower than that of SpMV based on ELL format, LR_{AELL} should be less than or equal to 0, so we can determine the range of V_K as

$$\begin{aligned} LR_{AELL} &\leq 0; \\ 19N\left(V_K - \frac{14}{19}K\right) &\leq 0; \\ V_K &\leq \frac{14}{19}K. \quad \square \end{aligned}$$

Through these formulas, we can obtain a general trend of the cycles changing with the zero element filling rate during the SpMV calculation based on ACSR and AELL.

6.3. Cache access times

Taking CSR and ACSR as an example, SpMV based on these two formats is required to access the row offsets $2M$ times to determine the nonzero elements corresponding to each row. Then, in the calculation process, the nonzero elements are accessed in vector form for $\frac{NNZ}{2}$ and $\frac{NNZ'}{2}$ times in CSR and ACSR, respectively. The column indices are all ordinary accesses, which are NNZ' and $\frac{NNZ'}{2}$ times. However, the access to the vector x is fundamentally different in the load mode with the vector register; this access needs NNZ times in CSR, but only $\frac{NNZ'}{2}$ times in ACSR because it accesses x consecutively after aligning the data to the vector register. Finally, the results must be stored in a vector y M times. Thus, we can obtain the number of cache accesses A_{CSR} and A_{ACSR} as

$$A_{CSR} = 2M + \frac{NNZ}{2} + NNZ + NNZ + M = 3M + \frac{5}{2}NNZ, \tag{16}$$

and

$$A_{ACSR} = 2M + \frac{NNZ'}{2} + \frac{NNZ'}{2} + \frac{NNZ'}{2} + M = 3M + \frac{3}{2}NNZ\left(1 + FR\right). \tag{17}$$

For SpMV using ELL format, the value matrix can be accessed in vector form, but the column index matrix, the vector x, and the result vector y can only be accessed sequentially. Therefore, the access cache times of SpMV using ELL format A_{ELL} are calculated by

$$A_{ELL} = \frac{M \times K}{2} + M \times K + M \times K + M = \frac{5}{2}MK + M. \tag{18}$$

However, for SpMV using AELL format, the value matrix, the column index matrix, and x can be vectors. Therefore, the access cache times of SpMV using AELL format are A_{AELL}, given by

$$A_{AELL} = 3MV_K + M. \tag{19}$$

Then, we can obtain the ratio of the reduction in cache access in our ACSR and AELL formats compared with the CSR and ELL formats, respectively:

$$AR_{ACSR} = \frac{A_{ACSR} - A_{CSR}}{A_{CSR}} = \frac{3NNZ}{2}\left(FR - \frac{2}{3}\right), \tag{20}$$

$$AR_{AELL} = \frac{A_{AELL} - A_{ELL}}{A_{ELL}} = 3M\left(V_K - \frac{5}{6}K\right). \tag{21}$$

Thus, we can use the above formulas to analyze the times of cache access of SpMV operation in different storage formats and obtain the following propositions.

Proposition 3. *When $FR < \frac{2}{3}$ of ACSR, the number of cache access in SpMV based on ACSR format is less than in CSR format.*

Proof. With the increase in zero paddings, although more time is needed to access the nonzero elements, every two elements in x can be read in a vector continuously with ACSR format, which is fewer than in CSR format. FR is the fill ratio of zeros. Equation (20) represents the reduction in cache access in ACSR compared with CSR.

To make the performance of SpMV based on the ACSR format not lower than that of SpMV based on the CSR format, AR_{ACSR} should be less than or equal to 0, so the upper bound of FR can be obtained by

$$\begin{aligned} AR_{ACSR} &\leq 0; \\ \frac{3NNZ}{2}\left(FR - \frac{2}{3}\right) &\leq 0; \\ FR &\leq \frac{2}{3}. \quad \square \end{aligned}$$

Proposition 4. *When $V_K < \frac{5}{6}K$ in AELL, the number of cache access in SpMV based on the AELL format is less than that based on the ELL format.*

Proof. The times of cache access depend on the data size and the access mode. AELL has a larger matrix to store the values than ELL, which leads to more cache accesses. However, in the column index matrix, this number may decrease, and the load of x is now continuous. V_K is the width of the row with zero paddings.

To make the times of cache access of SpMV based on AELL format not lower than that of SpMV based on ELL format, according to Equation (21), AR_{AELL} should be less than or equal to 0, so the upper bound of V_K is given by

$$AR_{AELL} \leq 0;$$
$$3M\left(V_K - \frac{5}{6}K\right) \leq 0;$$
$$V_K \leq \frac{5}{6}K. \quad \square$$

Thus, according to these formulas, we can find a general trend of the times of cache access with the zero element filling rate during SpMV calculation based on ACSR and AELL.

6.4. Cache miss times

As described in the previous part of this paper, SpMV using NEON acceleration can operate two vectors in a calculation instruction. It is necessary to access the cache ten times for nonzero elements in SpMV based on CSR and ELL formats and to include the nonzero elements in vector form only twice, the column index four times, and the vector x four times as well. However, we need only six times using ACSR and AELL formats in which values and x are accessed in vector form only twice, and the column index needs only the starting column index of two vectors.

In every calculation, in the worst case, four x values loaded in CSR and ELL may be distributed in four cache lines, while there are only two cache lines in ACSR and AELL. Our experiment is carried out on Kunpeng 920 with 4 cache lines of size 64 B at one set in the L1 cache. Therefore, during the calculation process, the CSR and ELL formats may have cache line conflicts and may be replaced for access to the cache 6 times in the worst case, while only 4 times for the ACSR and AELL formats. For this cause, we can infer that compared with SpMV based on CSR and ELL, the ACSR and AELL formats may reduce the probability of cache conflict and cause a decrease in cache misses [16,23].

7. Experimentals

7.1. Experiment platform

Our experiment is based on Kunpeng 920 produced by Huawei. Kunpeng 920 supports multiple parallel instructions based on ARMv8.2 architecture equipped with two floating-point units (FPUs). As shown in Table 4, the peak performance of Kunpeng 920 in double-precision floating-point computation achieves 665.6 Gflops. Moreover, Kunpeng 920 supports eight channels of memory, but only four channels are used in our experiment.

We have examined the effect on the performance of nonuniform memory access (NUMA) architecture on Kunpeng 920. Failure to consider NUMA may cause crossnode access to memory due to OS process switching, which will cause greater delays. Moreover, in this case, the experimental results are inaccurate. Therefore, to obtain a stable and reliable result in the experiment, we bind the threads and data through the tool for NUMA called "numactl", which can guarantee a thread to access the memory on a fixed node. Moreover, we use the perf tool on Linux to obtain the cycles, cache references, and cache misses in the calculation process.

Table 4
Experiment environment.

Processor	Kunpeng 920
Architecture	ARMv8.2
Frequency	2.6 GHz
Cores	64
L1 cache	64 kB L1I and 64 kB L1d
L2 cache	512 kB private per core
L3 cache	64 MB shared for all
Memory	4 × 2666 DDR4
OS	Centos 7.8
Linux kernel	4.18.0
Compiler	gcc 4.8.5
NUMA	nodes with 32 cores

Although the Algorithms 1 - 4 show the SpMV algorithms with NEON SIMD acceleration in single-threaded, they are also suitable for multi-threaded parallelism. All experimental procedures are implemented in C language with openMP and "-O2" complier option.

Since the non-zero elements of each row in the ACSR format are not uniform, and the operating system may schedule threads to ensure load balancing when using openMP automatic parallelism, which impact is not easy to predict in performance evaluation. In the experiments, all experiments choose the number of threads when the performance is optimal. And for the comparison experiment of each sparse matrix, the final number of threads executed is the same, which significantly reduces the impact of multi-threading on performance evaluation.

7.2. Experiment data

In this experiment, we selected 20 sparse matrices from the SuiteSparse Matrix Collection. They are all collected from a wide range of applications and are listed in Table 5, where "*sparse matrix*" is the name of the sparse matrices; "*rows*", "*columns*", "*NNZ*", and "*K*" indicate the rows, columns, number of nonzero elements of the sparse matrices, and threshold "*K*" in ELL, respectively.

8. Experimental results

The experimental results are divided into three parts:

- The acceleration of SpMV in CSR and ELL formats with NEON.
- The analysis of experimental data.
- Performance comparison of ACSR and AELL with CSR, ELL, and PETSc.

8.1. NEON acceleration for CSR and ELL

Fig. 6 shows the performance improvements of SpMV in original CSR and ELL formats after using NEON. The ordinate represents the relative performance improvements of NEON.

Through this experiment, we find that the performance of SpMV using NEON can achieve 19.91% and 19.15% improvements in CSR and ELL, respectively.

However, for some sparse matrices, we may find that the effect of NEON acceleration is not obvious. Combining the instruction delay in Table 3 and the number of nonzero elements in sparse matrices, we can infer that the access to the vector x is still not continuous. Because the number of accesses has not decreased but the instruction delay has increased, and because there is still

Table 5
Sparse matrices in the experiment.

sparse matrix	*rows*	*columns*	*NNZ*	*K*	FR_{ACSR}	V_K	FR_{AELL}
bbmat	38744	38744	1771722	126	24.16%	80	26.98%
car4	16384	33052	63724	111	56.32%	62	11.71%
cavity08	1182	1182	32747	62	2.20%	31	0.00%
Chebyshev2	2053	2053	18447	2053	55.50%	1027	0.05%
Chevron1	37365	37365	330633	9	33.34%	6	33.33%
EPA	4772	4772	8965	175	54.20%	95	8.57%
FEM_3D_thermal	17880	17880	430740	27	26.62%	18	33.33%
hangGlider_3	10260	10260	49201	4558	53.45%	2280	0.04%
hvdc1	24842	24842	159981	181	4.45%	100	10.50%
iprob	3001	3001	9000	3000	66.64%	1500	0.00%
lhr71	70304	70304	1528092	63	14.14%	32	1.59%
model4	1337	4962	45753	493	13.33%	248	0.61%
mycielskian10	767	767	22196	383	56.85%	192	0.26%
OPF_10000	43887	43887	255799	64	13.65%	32	0.00%
reorientation_1	677	677	3861	394	64.83%	197	0.00%
rosen10	2056	6152	64192	3886	6.51%	1996	2.73%
TSOPF_RS_b2383	38120	38120	16171169	983	0.90%	493	0.31%
viscorocks	37762	37762	1162244	42	0.00%	21	0.00%
water_tank	60740	60740	2035281	63	30.99%	36	14.29%
xenon1	48600	48600	1181120	27	27.37%	18	33.33%

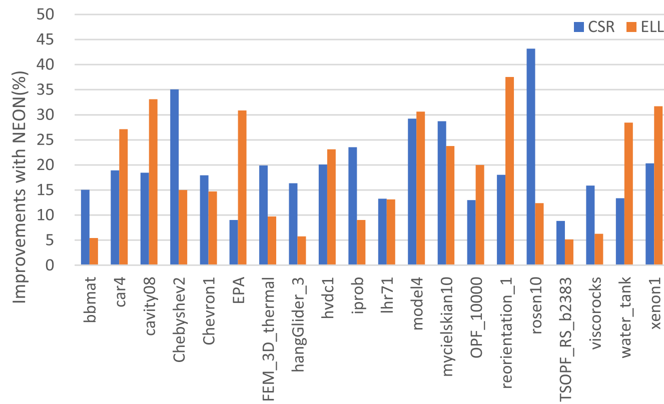


Fig. 6. Performance improvements of NEON in CSR and ELL formats. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

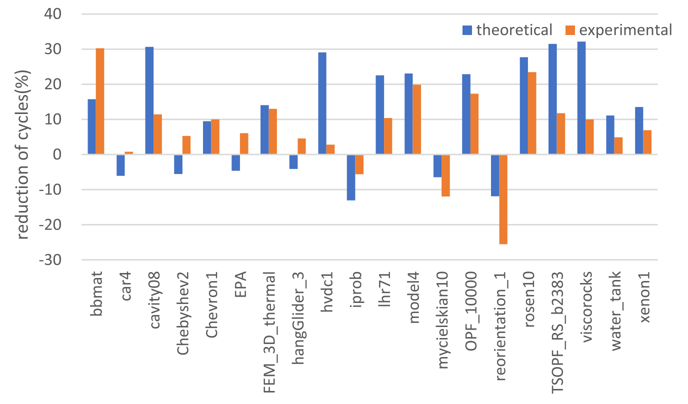


Fig. 7. Reduction in cycles of ACSR.

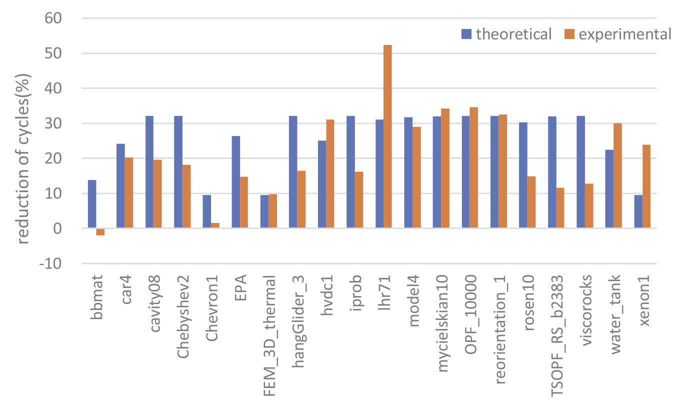


Fig. 8. Reduction in cycles of AELL.

waiting time for the NEON operation instruction itself, an increasing amount of data may lead to excessively high memory access instruction delays that reduce the performance of SpMV. For example, “bbmat” and “FEM_3D_thermal” in ELL and “TSOPF_RS_b2383” in both CSR and ELL. Moreover, the SpMV based on CSR and ELL with NEON may cause more cache misses because it needs extra two times of cache access.

In further experiments, the SpMV based on CSR and ELL are all accelerated by NEON.

8.2. Analysis of experimental data

In this part of the experiment, we obtain the cycles, cache references, and cache misses of SpMV for different formats through the perf performance profiling tool.

The ratios of the number of zero elements filled in ACSR and AELL based on the CSR and ELL are shown in Table 5.

8.2.1. Reduction in the execution latency of SIMD instructions

According to the formulas in Section 6 and the filling rates in Table 5, we can obtain the theoretical analysis results. In Figs. 7 and 8, we can find the reported improvement of the instruction latency and the actual improvement cycles.

The experimental results are consistent with our theoretical analysis. According to the chart, it can reduce cycles by 7.30% on average in ACSR compared with CSR, and the AELL value is 21.08% less than that of ELL.

However, there is a 10.26% average difference for ACSR and 10.51% for AELL between the theoretical and experimental re-

sults because we analyze only the main instructions. In the actual implementation process, we directly use the encapsulated NEON functions to use SIMD technology, but there are still some unknown instructions. Therefore, if the filling rate of ACSR theoretically reaches approximately 47%, an adverse effect arises, and there are some matrices whose filling rate reaches more than 56%, such as “iprob”, “mycielskian10”, and “reorientation_1” in the experiment too. Moreover, due to load unbalancing, the OS scheduling of threads may cause additional overhead in parallel SpMV calculations based on ACSR format.

For example, the distribution of nonzero elements in “TSOPF_RS_b2383” is too scattered, which may increase the number of memory accesses for ACSR and AELL. In the parallel SpMV calculation of some matrices in CSR format, the main performance depends on the long rows with the maximum non-zero elements. Although the instruction clock cycle of the serial calculation is theoretically increased compared to the CSR format, the opposite of the theory may occur in the parallel calculation for the zero-element filling rate of long rows in ACSR format may be zero. Such as “car4”, “Chebyshev2”, “hangGlider_3”, and “EPA”. The average length of their non-zero rows is hundreds or thousands of times different from their long rows, while the zero filling rate of long rows is almost zero. Therefore, the experimental results of these sparse matrices disagreed with the theoretical analysis in Fig. 7.

8.2.2. Cache improvements

We analyze the impact of cache access on the performance of SpMV for ACSR and AELL formats through the results given in Figs. 9, 10, and 11.

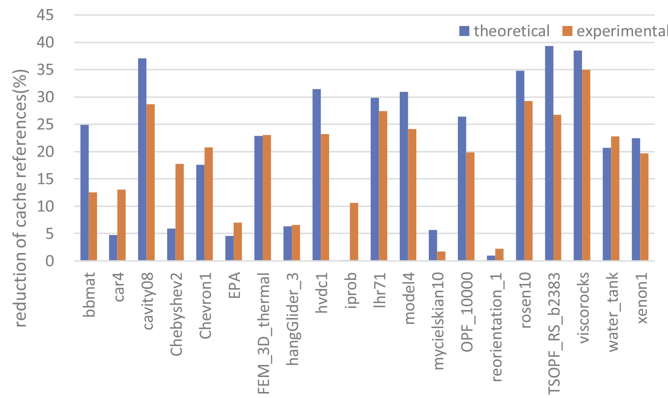


Fig. 9. Reduction in the cache references of ACSR.

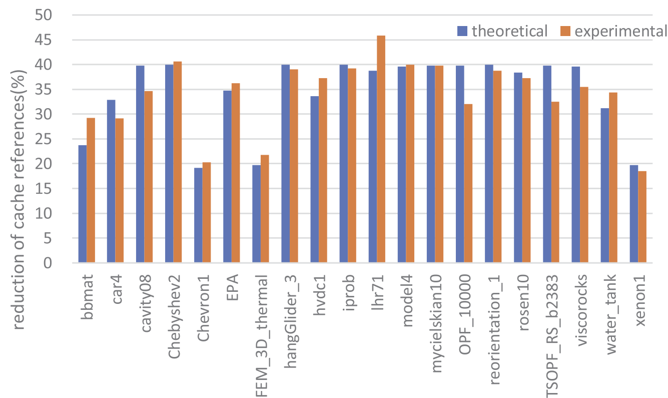


Fig. 10. Reduction in the cache references of AELL.

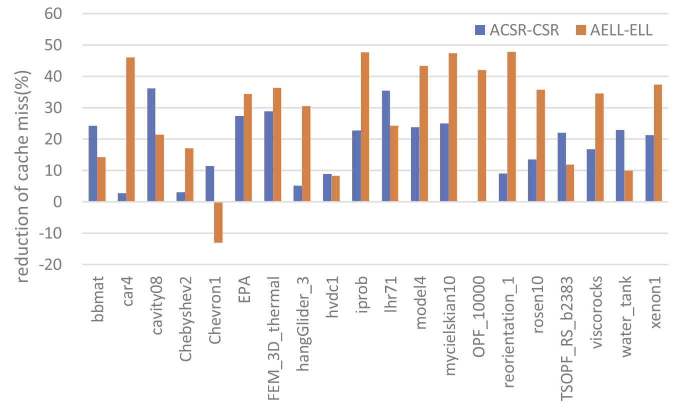


Fig. 11. Reduction in the cache misses of ACSR and AELL.

Figs. 9 and 10 present the number of cache access events reduced by ACSR and AELL formats, which is respectively 18.59% and 34.10% compared with those of CSR and ELL, and the deviation between the theoretical value and the actual value is 5.68% for ACSR and 2.91% for AELL.

Although the impact of cache access for ACSR and AELL formats is consistent with the theoretical analysis, there are still large deviations in the experimental results of a few matrices in ACSR. By analyzing the distribution of the nonzero elements in these matrices and comparing it with AELL, the reason for the large deviation may be the unbalanced load during the calculation, such as “bbmat” and “TSOPF_RS_b2383”.

In addition, we find that the ACSR and AELL formats can reduce the cache hit conflicts in a single calculation. As shown in Fig. 11, ACSR and AELL can reduce the number of cache misses of SpMV by 28.85% and 18.00%, respectively.

The ACSR and AELL formats can reduce the probability of cache conflict in each calculation process because the added zero elements and the previous nonzero elements usually remain on the same cache line. However, there are vectors across cache lines due to cache conflict. Therefore, when a sparse matrix has a relatively concentrated distribution of nonzero elements, the filling of zero in AELL format may cause the extra cache misses [15]. For example, “Chevron1” is a diagonal sparse matrix, and all nonzero elements are gathered on the diagonal line. Therefore, there are fewer nonzero elements across the cache line, such that the extra filling of zeros leads to more cache misses.

8.3. Performance of ACSR and AELL

In this part of the experiment, we compare the calculation time of SpMV using ACSR and AELL formats using CSR and ELL accelerated by NEON. We find that the ACSR format is the best in most cases. Fig. 12 records the performance improvements of SpMV based on ACSR compared with SpMV in CSR format and PETSc and makes a comparison between AELL and ELL on Kunpeng920 processor. At the same time, Fig. 13 shows the experimental results on FT2000+ processor to verify the performance improvements of our aligned storage formats.

The final experimental results are consistent with the analysis in Section 6. In this experiment, we find that SpMV in ACSR and AELL formats can achieve better performance than in the CSR and ELL formats. Compared with CSR and PETSc, ACSR can achieve an average performance improvement of 18.17% and 56.46%, and AELL achieves an average performance improvement of 20.54% over ELL on Kunpeng920 processor, while the average performance improvements are 24.83%, 61.14%, and 22.55% on FT2000+ processor.

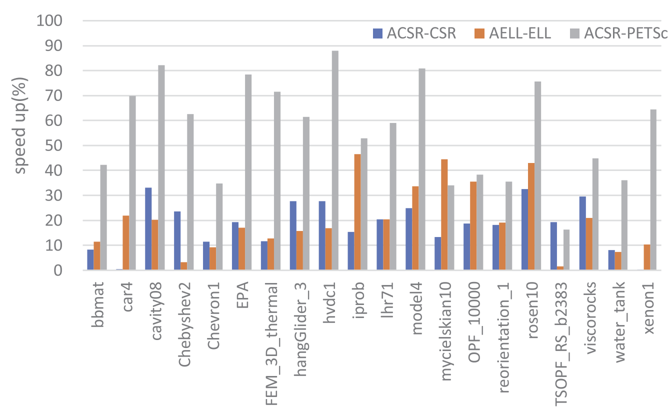


Fig. 12. Performance improvements of ACSR and AELL on Kunpeng920 processor.

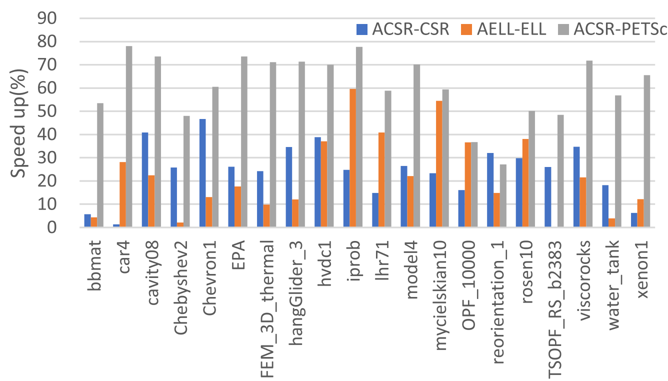


Fig. 13. Performance improvements of ACSR and AELL on FT2000+ processor.

However, the sparse matrix “car4” is a diagonal matrix with local continuity and the filling rate in ACSR reached 56.32%. The calculations of SpMV based on ACSR show almost no improvement in instruction latency, cache access and cache misses compared with SpMV using CSR.

9. Conclusions

In this paper, we propose two aligned storage formats, ACSR and AELL, which focus on the 128-bit SIMD operator to parallel optimize SpMV in double precision on ARM processors. Then, we analyze the improvement of ACSR and AELL formats in terms of the instruction delay, cache access, and cache miss that the deviations from the experimental results are 10.26%, 10.51% and 5.68%, 2.91% in execution latency of instructions and cache references, respectively. Moreover, as the experimental results show, SpMV based on ACSR can achieve an average improvement in the executive latency of instructions, cache references, cache misses, and calculation time of 7.3%, 18.59%, 28.85%, and 18.17%, respectively, compared with SpMV in CSR formats; the corresponding AELL parameters are 21.08%, 34.10%, 18.00%, and 20.54% higher than those of ELL. In addition, we choose PETSc for comparison and find that SpMV based on ACSR exhibits a 56.46% performance improvement compared to PETSc. In the future, we plan to consider more instruction types, and prefect the cache performance in performance analytical models of SpMV on ARM processors to predict more accurately and achieve better performance optimization.

CRediT authorship contribution statement

Yufeng Zhang: Conceptualization, Investigation, Methodology, Software, Validation, Writing – original draft. **Wangdong Yang:** Conceptualization, Methodology, Project administration, Resources, Writing – original draft. **Kenli Li:** Supervision. **Dahai Tang:** Software, Validation. **Keqin Li:** Writing – review & editing.

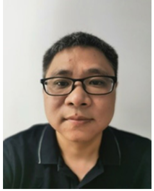
Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] N. Abubaker, K. Akbudak, C. Aykanat, Spatiotemporal graph and hypergraph partitioning models for sparse matrix-vector multiplication on many-core architectures, IEEE Trans. Parallel Distrib. Syst. 30 (2) (2018) 445–458.
[2] R.V. Aroca, L.M.G. Gonçalves, Towards green data centers: a comparison of x86 and ARM architectures power efficiency, J. Parallel Distrib. Comput. 72 (12) (2012) 1770–1780.
[3] N. Bell, M. Garland, Implementing sparse matrix-vector multiplication on throughput-oriented processors, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, 2009, pp. 1–11.
[4] E. Blem, J. Menon, T. Vijayaraghavan, K. Sankaralingam, ISA wars: understanding the relevance of ISA being RISC or CISC to performance, power, and energy on modern architectures, ACM Trans. Comput. Syst. 33 (1) (2015) 1–34.
[5] E.-J. Im, K. Yelick, R. Vuduc, Sparsity: optimization framework for sparse matrix kernels, Int. J. High Perform. Comput. Appl. 18 (1) (2004) 135–158.
[6] M. Jarus, S. Varrette, A. Oleksiak, P. Bouvry, Performance evaluation and energy efficiency of high-density HPC platforms based on Intel, AMD and ARM processors, in: European Conference on Energy Efficiency in Large Scale Distributed Systems, Springer, 2013, pp. 182–200.
[7] V. Karakasis, G. Goumas, N. Koziris, A comparative study of blocking storage methods for sparse matrices on multicore architectures, in: 2009 International Conference on Computational Science and Engineering, vol. 1, IEEE, 2009, pp. 247–256.
[8] K. Kourtis, V. Karakasis, G. Goumas, N. Koziris, CSX: an extended compression format for SpMV on shared memory systems, ACM SIGPLAN Not. 46 (8) (2011) 247–256.
[9] M.A. Laurenzano, A. Tiwari, A. Cauble-Chantrenne, A. Jundt, W.A. Ward, R. Campbell, L. Carrington, Characterization and bottleneck analysis of a 64-bit ARMv8 platform, in: 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), IEEE, 2016, pp. 36–45.
[10] W. Liu, B. Vinter, CSR5: an efficient storage format for cross-platform sparse matrix-vector multiplication, in: Proceedings of the 29th ACM on International Conference on Supercomputing, 2015, pp. 339–350.
[11] N. Liu, B. Zang, H. Chen, No barrier in the road: a comprehensive study and optimization of ARM barriers, in: Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2020, pp. 348–361.
[12] D. Merrill, M. Garland, Merge-based sparse matrix-vector multiplication (SpMV) using the CSR storage format, ACM SIGPLAN Not. 51 (8) (2016) 1–2.
[13] A. Monakov, A. Lokhmotov, A. Avetisyan, Automatically tuning sparse matrix-vector multiplication for GPU architectures, in: International Conference on High-Performance Embedded Architectures and Compilers, Springer, 2010, pp. 111–125.
[14] Y. Saad, SPARSKIT: a basic tool kit for sparse matrix computations.
[15] J.M. Sabarimuthu, T. Venkatesh, Analytical miss rate calculation of L2 cache from the RD profile of L1 cache, IEEE Trans. Comput. 67 (1) (2017) 9–15.
[16] D. Shen, M. Chabbi, X. Liu, An evaluation of vectorization and cache reuse tradeoffs on modern CPUs, in: Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores, 2018, pp. 21–30.
[17] S. Streit, F. De Santis, Post-quantum key exchange on ARMv8-A: a new hope for neon made simple, IEEE Trans. Comput. 67 (11) (2017) 1651–1662.
[18] R.W. Vuduc, H.-J. Moon, Fast sparse matrix-vector multiplication by exploiting variable block structure, in: International Conference on High Performance Computing and Communications, Springer, 2005, pp. 807–816.
[19] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, J. Demmel, Optimization of sparse matrix-vector multiplication on emerging multicore platforms, in: SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, IEEE, 2007, pp. 1–12.
[20] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, L. Zhang, CVR: efficient vectorization of SpMV on x86 processors, in: Proceedings of the 2018 International Symposium on Code Generation and Optimization, 2018, pp. 149–162.
[21] W. Yang, K. Li, K. Li, A parallel computing method using blocked format with optimal partitioning for SpMV on GPU, J. Comput. Syst. Sci. 92 (2018) 152–170.
[22] T. Yoshida, Fujitsu high performance CPU for the post-K computer, in: Hot Chips, vol. 30, 2018.
[23] Z. Zhang, H. Wang, S. Han, W.J. Dally, SpArch: efficient architecture for sparse matrix multiplication, in: 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2020, pp. 261–274.

Yufeng Zhang received M.S. degree in computer science and technology from Hunan University, China, in 2021, and the B.S. degree in information and computing science from Hunan University, China, in 2018. His research interests include parallel algorithms, high-performance computing, and computer architectures.



Wangdong Yang received the Ph.D. degree in computer science from Hunan University, China, and the M.S. degree in computer science from Central South University, China. He is a professor of computer science and technology at Hunan University, China. His research interests include modeling and programming for heterogeneous computing systems, parallel and distributed computing, and numerical computation. He has published more than 60 papers in International conferences and journals. He is currently served on the editorial boards of IEEE Internet of Things Journal.

Kenli Li received the Ph.D. degree in computer science from Huazhong University of Science and Technology, China, in 2003, and the M.S. degree in mathematics from Central South University, China, in 2000. He was a visiting scholar at University of Illinois at Urbana-Champaign from 2004 to 2005. He is a full professor of computer science and technology at Hunan University. The main research fields are parallel and distributed processing, supercomputing and cloud computing, high-performance computing for big data and artificial intelligence, etc. He has published more than 300 papers in international conferences and journals. He is currently served on the editorial boards of IEEE Transactions on Computers. He is an outstanding member of CCF and a member of the IEEE.

Dahai Tang was born in 1996. He is a second-year doctoral student at Hunan University, China. His research interests include parallel computing and operating system.

Keqin Li is a SUNY Distinguished Professor of computer science with the State University of New York. He is also a National Distinguished Professor with Hunan University, China. His current research interests include cloud computing, fog computing and mobile edge computing, energy-efficient computing and communication, embedded systems and cyber-physical systems, heterogeneous computing systems, big data computing, high-performance computing, CPU-GPU hybrid and cooperative computing, computer architectures and systems, computer networking, machine learning, intelligent and soft computing. He has authored or coauthored more than 780 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He holds over 60 patents announced or authorized by the Chinese National Intellectual Property Administration. He is among the world's top 10 most influential scientists in distributed computing based on a composite indicator of Scopus citation database. He has chaired many international conferences. He is currently an associate editor of the ACM Computing Surveys and the CCF Transactions on High Performance Computing. He has served on the editorial boards of the IEEE Transactions on Parallel and Distributed Systems, the IEEE Transactions on Computers, the IEEE Transactions on Cloud Computing, the IEEE Transactions on Services Computing, and the IEEE Transactions on Sustainable Computing. He is an IEEE Fellow.