SIMD Parallel Sparse Matrix-Vector and Transposed-Matrix-Vector Multiplication in DD Precision

Toshiaki Hishinuma[1], Hidehiko Hasegawa[12], and Teruo Tanaka[2]

[1] University of Tsukuba, Tsukuba, Japan
[2] Kogakuin University, Tokyo, Japan
hishinuma@slis.tsukuba.ac.jp

Abstract. We accelerate a double-precision sparse matrix and DD vector multiplication (DD-SpMV) and its transposition and DD vector multiplication (DD-TSpMV) using SIMD AVX2. AVX2 requires changing the memory access pattern to allow four consecutive 64-bit elements to be read at once. In our previous research, DD-SpMV in CRS using AVX2 needed non-continuous memory load, processing for the remainder, and the summation of four elements in the AVX2 register. These factors degrade the performance of DD-SpMV. In this paper, we compare the storage formats of DD-SpMV and DD-TSpMV for AVX2 to eliminate the performance degradation factors in CRS. Our result indicates that BCRS4x1, whose block size fits the AVX2 register's length, is effective for DD-SpMV and DD-TSpMV.

Keywords: Matrix storage format, SpMV, Transposed SpMV, Double-double precision arithmetic, AVX2

1 Introduction

High-precision arithmetic operations reduce rounding errors and improve the convergence of Krylov subspace methods [1]; however, they are very costly. Double-double-precision (DD) arithmetic, which is one type of high-precision arithmetic, is constructed by combining double-precision operations, but it requires more than 10 double-precision operations for one DD operation [2]. However, it can greatly speed up performance using SIMD because it has a smaller memory access rate than double-precision arithmetic [4].

A sparse matrix and vector multiplication take much time in Krylov subspace methods. We accelerated the double-precision sparse matrix and the DD vector multiplication (DD-SpMV) and its transposition and the DD vector multiplication (DD-TSpMV) using advanced vector extensions 2 (AVX2) [3] [4]. The AVX2 instruction set, which is a 256-bit single instruction multiple data streaming (SIMD) instruction set, provides fused multiply and add instruction (FMA). AVX2 simultaneously computes four double-precision FMA instructions.

AVX2 required changing the memory access pattern to allow four consecutive 64-bit elements to be read at once. In DD-SpMV and DD-TSpMV for a compressed row storage format (CRS) [5], a non-continuous memory load and store are needed for using AVX2. In addition, since it must simultaneously compute four double-precision data. Furthermore, processing for the remainder in each row is needed, because AVX2 must simultaneously compute four double-precision data. Consequently, the performance might be degraded. We call these CRS problems, collectively, performance degradation factors.

To avoid them, we use the BCRS format [5], which divides matrix A into $r \times c$ small dense submatrices (called blocks), which might include some zero-elements. BCRS4x1, 2x2, and 1x4 ($r \times c = 4$) can simultaneously compute four elements.

BCRS4x1 ($r = 4$, $c = 1$) is suitable for DD-SpMV using AVX2 because the block size fits the SIMD register's length [6]. However, since BCRS4x1 requires up to four times the amount of operations and data as CRS. In DD-TSpMV, BCRS4x1 fails to eliminate performance degradation factors. Consequently, we must compare BCRS1x4 and BCRS4x1.

In this paper, we show that the effective implementation of DD-SpMV and DD-TSpMV improves the AVX2 performance and analyze the optimal storage format to eliminate the performance degradation factors in CRS.

2 Related work

XBLAS [7] is a well-known extended precision BLAS whose input and output are double-precision that internally uses the DD operations. However, it does not accelerate them using SIMD.

Lis [8], which is an iterative solver library, internally uses DD operations, which are accelerated by SIMD SSE2. SSE2 has 128-bit SIMD registers.

On the other hand, Karakasis [9] and Im [10] accelerated a double-precision SpMV. Blocking, which fits the SIMD register's length, is effective for a double-precision SpMV. In AVX2, BCRS4x1 is effective. Xing [11] implemented a double-precision SpMV in ELLPACK and an ELLPACK sparse block format on MIC (Intel many integrated core architecture).

However, since these studies, which failed to evaluate TSpMV, are only double precision, we must compare DD-SpMV in BCRS1x4 and BCRS4x1.

3 Implementation of DD-SpMV and DD-TSpMV using AVX2

3.1 DD arithmetic

DD arithmetic, which is based on error-free, floating-point arithmetic algorithms by Dekker [12] and Knuth [13], only consists of combinations of double-precision values and uses two double-precision variables to implement one quadruple precision variable [2].

An IEEE 754 quadruple precision variable consists of a 1-bit sign part, a 15-bit exponent part, and a 112-bit significand part. A DD-precision variable consists of a 1-bit sign part, an 11-bit exponent part, and a 104-bit (52 × 2) significand part. The exponent part of a DD-precision variable is 4 bits shorter and the significand part is 8 bits shorter than the exponent and significand parts of an IEEE 754 quadruple precision variable, respectively.

The simplest way to use IEEE 754 quadruple precision is with Fortran REAL*16. We compared Fortran REAL*16 using an Intel Fortran compiler 13.0.1 (ifort) and DD arithmetic without any SIMD instructions. The compiler option in ifort was -O3. Fortran REAL*16 in ifort was only implemented by integer operations. We computed $\boldsymbol{y} = \alpha \times \boldsymbol{x} + \boldsymbol{y}$, where $\boldsymbol{x}$ and $\boldsymbol{y}$ are the quadruple precision vectors and α is quadruple precision variable. Two 10^5 vectors, $\boldsymbol{x}$ and $\boldsymbol{y}$, can be stored in the cache. The elapsed time of Fortran REAL*16 was 2.7 ms and that of the DD arithmetic was 0.64 ms in 1 thread, which means that the DD arithmetic was 4.2 times faster than Fortran REAL*16.

DD addition consists of 11 double-precision additions, and DD multiplication consists of 10 double-precision operations: three double-precision additions, three double-precision multiplications, and two double-precision FMA instructions (3 + 3 + 2 × 2 = 10 flops).

In DD multiplication, two sign inversions are needed. However, since AVX2 lacks sign inversion instruction, we use two double-precision multiplications for two sign inversion. This flop count consists of these multiplications.

We implemented DD vector $\boldsymbol{x}$ using two double-precision arrays ($\boldsymbol{x}.hi$ and $\boldsymbol{x}.lo$) for the SIMD acceleration.

The bytes/flops of the DD operations are lower than those of the double-precision operations. For example, in the DD-SpMV kernel stored in CRS, the memory requirement is 28 bytes: 8 bytes for matrix A, 16-byte vector $\boldsymbol{x}$, and 4 bytes for the vector column index. We postulate that loading vector $\boldsymbol{x}$ has a cache miss.

The bytes/flops of double-precision SpMV is 20 (bytes)/2 (flops) = 10, those of the DD matrix and the DD vector product is 36 (bytes)/23 (flops) = 1.56, and those of DD-SpMV is 28 (bytes)/21 (flops) = 1.33. The byte/flop value of DD-SpMV is 13% of double-precision SpMV and 85% of the DD matrix and the DD vector product.

DD-SpMV is expected to greatly speed up the SIMD acceleration because of the amount of data required for the memory. In many cases, for an iterative solver library, input matrix A, which is given in double precision, is iteratively used. To reduce the memory access of the sparse matrix and the vector product, we use double-precision sparse matrix A and DD-precision vector $\boldsymbol{x}$ product.

3.2 Intel SIMD AVX2

In this section, we introduce Intel SIMD AVX2. AVX2 must simultaneously compute, load, and store four double-precision variables. DD-SpMV and DD-TSpMV use three types of load AVX2 instructions (_mm256_load_pd (load), _mm256_broadcast_sd (broadcast), and _mm256_set_pd (set)) and one store instruction (_mm256_store_pd (store)).

These instructions have the following descriptions:

- The "load" instruction loads four continuous double-precision elements that begin with the same source memory address.
- The "broadcast" loads one double-precision element from one source memory address to all the elements of the SIMD register.
- The "set" loads four double-precision elements from four different source memory addresses.
- The "store" stores four continuous double-precision elements from the register to the memory beginning with the same source address.

We easily implemented the following three macro-functions (i.e. "SCATTER", "REDUCTION", "FRACTION_PROCESSING") to SIMD-ize DD-SpMV and DD-TSpMV.

To perform random store operation "scatter," we implemented a "SCATTER '' macro-function using "store" and ordinary instructions. We also implemented "SCATTER" to store the double-precision temporary array using the "store" instruction and stored valuables using ordinary double-precision store instructions.

To store the summation of the elements in the SIMD register storage into one source address, we implemented a "REDUCTION" macro-function that computes a summation of four DD variables in two SIMD registers (high and low). We implemented "REDUCTION" using the "shuffle" instruction, which rearranges the double-precision elements. For example, the AVX2 register has $\{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}\}$ elements. First, the "shuffle" instruction makes $\{\boldsymbol{b}, \boldsymbol{a}, \boldsymbol{d}, \boldsymbol{c}\}$ elements of AVX2 register from $\{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}\}$. Second, we operate the DD addition using the AVX2 of these registers, and then $\{\boldsymbol{a+b}, \boldsymbol{a+b}, \boldsymbol{c+d}, \boldsymbol{c+d}\}$ AVX2 register is made. Finally, we operate the DD addition using ordinary instructions for the second and third elements. 'REDUCTION" consists of "shuffle" instructions and 11 (DD addition using AVX2) + 11 (DD addition using ordinary instruction) = 22 flops.

To judge the processing for the AVX2 remainder, which is one, two, or three elements for each row in the case of CRS, we implemented a "FRACTION_PROCESSING" macro-function, which assigns zero to the "set" operand at the execution and three conditional branchings.

"FRACTION_PROCESSING" consists of the following C code:

```
av = load(A[value[j]]);
yv = set_zero();
case (r == 3)
   xv2 = set(x[index[j]], x[index[j+1]], x[index[j+2]], 0);
   yv2 = DD_MULT_ADD(yv2, av, xv2); break;
case (r == 2)
   set(x[index[j]], x[index[j+1]], 0, 0);
   yv2 = DD_MULT_ADD(yv2, av, xv2); break;
case (r == 1)}
   set(x[index[j]], 0, 0, 0);
   yv2 = DD_MULT_ADD(yv2, av, xv2);
```

The ***av*** is the 256-bit AVX2 register type variables. ***Xv2*** and ***yv2*** are the AVX2 register type variables for DD variables. "DD_MULT_ADD" computes DD multiplication and addition using AVX2: $\boldsymbol{yv2} + \boldsymbol{av} \times \boldsymbol{xv2}$. "Set_zero()" means an AVX2 register type variable initialization with zero. "FRACTION_PROCESSING" needs a maximum of three times as many branches on the conditions.

The "set" is costly compared to "load" and "broadcast [3]." "SCATTER," "REDUCTION," and "'FRACTION_PROCESSING" are costly because "SCATTER" occurs in the random memory store. "REDUCTION" requires more computations because it needs a "shuffle" instruction and 22 double-precision additions. "FRACTION_PROCESSING" occurs in the conditional branching.

4 Performance degradation factors of DD-SpMV and DD-TSpMV in CRS using AVX2

4.1 DD-SpMV

The CRS format is expressed by the following three arrays: ***ind***, ***ptr***, and ***val***. The double-precision ***val*** array stores the values of the non-zero elements of matrix A since they are traversed row-wise. The ***ind*** array is the column indices that correspond to the values, and ***ptr*** is the list of value indexes where each row starts. DD-SpMV in CRS using AVX2 consists of the following C code:

```
#pragma omp parallel for private (j, av, xv2, yv2)
for(i=0; i<N; i++){
   yv2 = set_zero();
   for(j=A->ptr[i]; j<A->ptr[i+1]-3; j+=4){
      xv2 = set(x[A->ind[j+0]],..,x[A->ind[j+3]]);
      av = load(&A->val[j]);
      yv2 = DD_MULT_ADD(yv2, av, xv2); break;
   }
   yv2 = FRACTION_PROCESSING();
   y[i] = REDUCTION(yv2);
}
```

X and ***y*** are a double-precision array, and ***A*** is the CRS format. DD-SpMV in CRS using AVX2 needs a "set" of ***x***, the "REDUCTION" of ***y***, and "FRACTION_PROCESSING."

4.2 DD-TSpMV in CRS using AVX2

DD-TSpMV in CRS using AVX2 consists of the following C code:

```
num_threads = omp_num_threads();
work = malloc(num_threads * N);
#pragma omp parallel private (i, j, k, av, xv2, yv2){
   k = omp_get_thread_num();
```

```
#pragma omp for
   for(i=0; i<N; i++){
      xv2 = broadcast(&x[i]);
      for(j=A->ptr[i]; j<A->ptr[i+1]-3; j+=4){
         jj = j + k + N;
         yv2 = set(y[A->ind[j+0]],..,y[A->ind[j+3]]);
         av = load(&A->val[j]);
         yv2 = DD_MULT_ADD(yv2, av, xv2); break;
         SCATTER(yv2, work[A->ind[jj+0],..,work[A->ind[jj+3]]);
      }
      av = load(&A->val[j]);
      yv2 = FRACTION_PROCESSING(A,x);
      SCATTER(yv2, work[A->ind[jj+0],..,work[A->ind[jj+3]]);
   }}
for(i=0;i<N,i++)
   for(j=0;j<num_threads,i++)
      y[i] = DD_ADD(y[i], work[A->ind[i+j*N]);
```

DD-TSpMV in CRS needs a “set” of ***y***, a “SCATTER” of ***y***, and “FRACTION_PROCESSING.”

In multi-threading, DD-TSpMV in CRS needs the number of thread work vectors and their array-reduction after computation.

5 Implementation and Evaluation of DD-SpMV and DD-TSpMV in other storage formats

CRS has some performance degradation factors, and AVX2 must change the memory access pattern to allow four consecutive 64-bit elements to be read at once. In this section, we compare the features of some storage formats using performance degradation factors.

5.1 DD-SpMV

BCRS r × c is expressed by the following three arrays: ***bind***, ***bptr***, and ***bval***. The length of the double-precision array ***bval*** is the number of blocks (blk) × r × c store values of the non-zero blocks since they are traversed row-wise. The ***bind*** array is the column indices that correspond to the blocks, and ***bptr*** is the list of block indexes where each block row starts.

Table 1 shows the features of CRS, BCRS1x4, BCRS4x1, and ELL [5].

BCRS4x1 does not need “set,” “REDUCTION,” or “FRACTION_PROCESSING.” It needs “REDUCTION,” and ELL needs a “set” of ***x***. In DD-SpMV, BCRS4x1 is the best estimation because it eliminates the performance degradation factors in CRS. However, it needs more operations and data.

DD-SpMV in BCRS4x1 using AVX2 consists of the following C code:

Table 1. Features of DD-SpMV in each storage format

	CRS	BCRS1x4	BCRS4x1	ELL
Loading $\boldsymbol{x}$	set	load	broadcast	set
Loading $\boldsymbol{y}$	set_zero	set_zero	set_zero	set_zero
Storing $\boldsymbol{y}$	REDUCTION	REDUCTION	store	store
FRACTION_PROCESSING	each row	none	none	each col.
Computation ratio (max)	1	4	4	the num. of row

```
#pragma omp parallel for private (jb, av, xv2, yv2)
for(i=0; i<N-3; i+=4){ // block_row is about N/4.
   yv2 = set_zero();
   for(j=A->bptr[i]; j<A->bptr[i+1]; j++){
      xv2 = broadcast(x[A->bind[j]]);
      av = load(&A->bval[j * 4]);
      yv2 = DD_MULT_ADD(yv2, av, xv2); break;
   }
   y[i] = store(yv2);
}
```

In the inner-loop (j-loop), DD-SpMV in CRS needs four double-precision elements of A and four non-contiguous and indirect DD elements of $\boldsymbol{x}$. BCRS4x1 only needs four double-precision elements of A and one indirect DD element of $\boldsymbol{x}$. The amount of bytes/flops of BCRS4x1 is smaller than that of CRS. The memory requirement of BCRS4x1 in the inner-loop is smaller than that of CRS.

5.2 DD-TSpMV

The performance degradation factors of DD-TSpMV in CRS are non-continuous load/store, "FRACTION_PROCESSING," and the initialization and summation of the work vectors in multi-threading. In DD-SpMV, the BCRS4x1 feature is the best. However, in DD-TSpMV, BCRS4x1 fails to eliminate the work vectors.

We improved DD-TSpMV in BCRS4x1 for high performance in DD-SpMV and DD-TSpMV on only one storage format. Its BCRS4x1 applied column-wise multi-threading; the others applied row-wise multi-threading. The DD-TSpMV performance in BCRS4x1 applied additional column-wise multi-threading, which is improved here because BCRS4x1 only computes one column in j-loop; i.e., it can easily be thread-partitioned. DD-TSpMV in BCRS4x1 using the AVX2 of column-wise multi-threading consists of the following C code:

```
num_threads = omp_num_threads();
work = malloc(4* N);  // The length of SIMD.
#pragma omp parallel private(work, jb, av, xv2, yv2){
k = omp_get_thread_num();
```

Table 2. DD-TSpMV features in each storage format

	CRS	BCRS1x4	BCRS4x1	ELL
Loading $\boldsymbol{x}$	broadcast	broadcast	load	broadcast
Loading $\boldsymbol{y}$	set	load	broadcast	set
Storing $\boldsymbol{y}$	SCATTER	store	store	REDUCTION
Fraction_processing	each row	none	none	each col.
Computation ratio (max)	1	4	4	number of rows

```
alpha = N / num_threads * k;
beta = N / num_threads * (k+1);
for (i = 0; i < N-3; i+=4){
   xv2 = load(x[i]);
   #pragma omp for
   for (j = bptr[i]; j < bptr[i+1]; j++){
      if ( alpha < bind[jb] <= beta){ //thread-partitioning
      av = load(A->bval[j]);
      yv2 = broadcast(work[bind[j]);
      yv2 = DD_MULT_ADD(yv2, av, xv2); break;
      y[i] = store(yv2);
}}}}
```

BCRS4x1 can eliminate "REDUCTION" and continuously store work vectors in j-loop. Since it needs only four work vectors, it is expected to speed up performance on more multi-core systems.

Table 2 shows the TSpMV features in each storage format. BCRS1x4 and BCRS4x1 do not need "set" "scatter," or "REDUCTION." ELL needs "set" and "REDUCTION." In addition, BCRS4x1 only needs four work vectors and continuous storage for them. In DD-TSpMV, BCRS1x4 or BCRS4x1 is the best.

6 Experimental results

We performed our tests on a machine with a 4-core 8-thread Intel Core i7 4770 3.4 GHz CPU, an 8-MB L3 cache, and 16-GB memory. We used Fedora 20 OS and Intel C/C++ compiler 15.0.0 as well as compiler options -O3, -xCORE-AVX2, -openmp, and -fp-model precise. Our code was written in C and used AVX2 intrinsic instructions. We also used an openMP guided scheduling option and 4-thread multi-threading.

6.1 DD-SpMV and DD-TSpMV overheads

We evaluated the performance in each storage format with 23 matrices, which were taken from The University of Florida Sparse Matrix Collection (Florida Collection) [14].

Fig. 1 and Fig. 2 shows the overhead of DD-SpMV and DD-TSpMV. We measured the elapsed time of the non-continuous load to change the load instruction from the set instruction. calculation kernel means elapsed time without overheads.

From Fig. 1, we compared the DD-SpMV in CRS overheads in the following results, where the calculation kernel is the baseline:

- Non-continuous load (set instruction) overheads are 74-630%.
- "FRACTION_PROCESSING" overheads are 4-89%.
- "REDUCTION" overheads are 27-380%.

The effect of the non-continuous load and "REDUCTION" is very large. When the nnz/row is small, the overhead effects are large because "FRACTION_PROCESSING" and "REDUCTION ,, occurred in each row.

The total time of BCRS1x4 is 1.6-7.3 times slower than the calculation kernel in CRS. The elapsed time of BCRS1x4 is more than four times slower because it needs "REDUCTION".

In all cases, BCRS1x4 is slower than BCRS4x1 because of the "REDUCTION" overhead. The total time of BCRS4x1 is 1.2-3.2 times slower than the calculation kernel in CRS. The computation ratios of BCRS4x1 are 1.1-3.9 times. The elapsed time and the computation ratio are proportional.

From Fig. 2, we compared the DD-TSpMV in CRS overheads in the following results, where the calculation kernel is the baseline:

- Non-continuous load/store (set instruction and "SCATTER") overheads are 140-640%.
- "FRACTION_PROCESSING" overheads are 3-79%.
- "Initialization and summation of work vectors" overheads are 12-150%.

The overheads of non-continuous load/store, Initialization, and the summation of work vectors are large.

The total time of BCRS1x4 is 1.4-5.1 times slower than the calculation kernel in CRS. The elapsed time of BCRS1x4 is more than four times slower.

The total time of BCRS4x1 is 1.2-3.8 times slower than the calculation kernel in CRS. The computation ratios of BCRS4x1 are 1.1-3.9 times. The elapsed time and the computation ratio are proportional.

6.2 Convert costs of BCRS4x1 from CRS

Next we evaluated the convert costs of BCRS4x1 from CRS using 100 matrices taken from the Florida Collection. The convert BCRS4x1 from CRS consists of the following C code:

```
for(bi=0;bi<nr;bi++){
   i = bi*r;
   ii = 0;
   kk = Aout.bptr[bi];
```

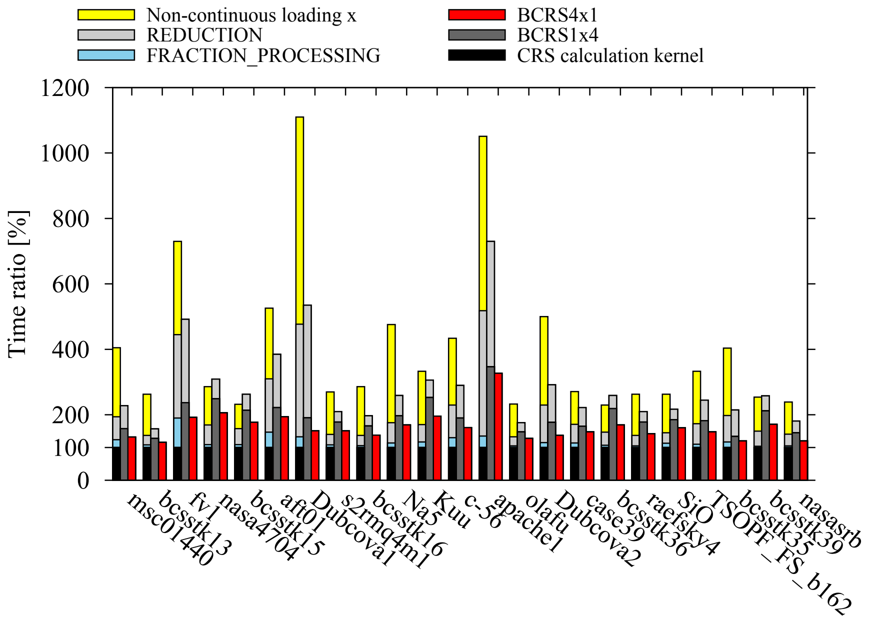

Fig. 1. DD-SpMV overhead

```
   while(i+ii<n && ii<=r-1){
      for(k=Ain.ptr[i+ii];k<Ain.ptr[i+ii+1];k++){
         ......
            Aout.bindex[kk] = Ain.index[k]/c;
         Aout.value[ij] = Ain.value[k];
         kk = kk+1;
         ......
         }
      ii = ii+1;
      }
   ......
}
```

We measured the convert times and compared them to the computation times of DD-SpMV in BCRS4x1. The average convert time was 4.2, the minimum convert time was 2.6, and the maximum convert time was 5.1 times slower than the elapsed time of DD-SpMV in CRS. The convert time is small.

6.3 BCRS4x1 effect

Fig 3 compares the time ratio of BCRS4x1 to CRS for 100 sparse matrices taken from the Florida Collection.

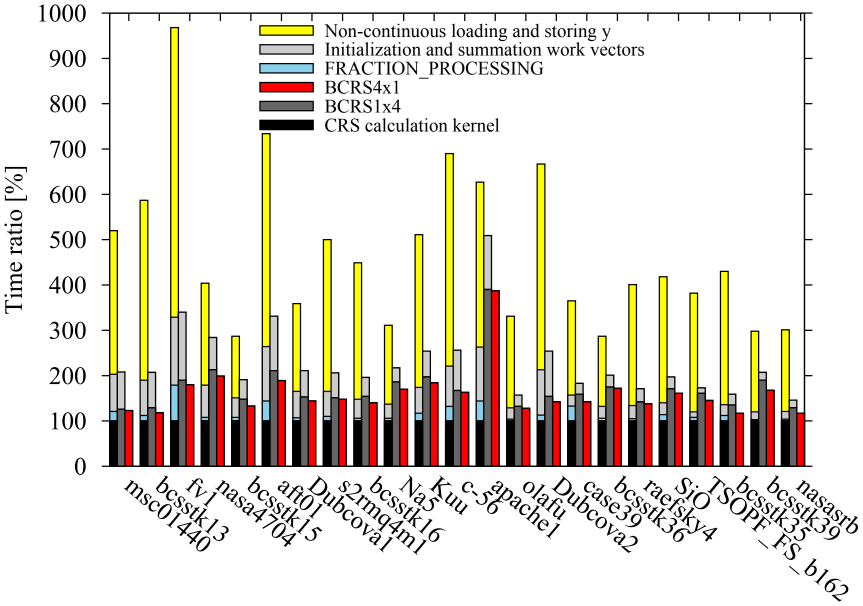

Fig. 2. DD-TSpMV overhead

In many cases, BCRS4x1 is faster than CRS using AVX2. The time ratios of DD-SpMV in CRS using AVX2 are 0.06 - 1.34 with an average of 0.38 compared with CRS without SIMD. The time ratios of DD-SpMV in BCRS4x1 are 0.09 - 1.96 times faster (average 0.43) than the case of CRS using AVX2. In DD-SpMV, the 96/100 matrix performance outperforms Scalar CRS.

The time ratios of DD-TSpMV in CRS using AVX2 are 0.06 - 1.14 with an average of 0.34 compared with CRS without SIMD. The times of DD-TSpMV in BCRS4x1 using AVX2 are 0.08 - 1.07 with an average of 0.38 compared with CRS using AVX2. In DD-TSpMV, the 100/100 matrix is better than Scalar CRS.

For example, the time ratio of "cell2" in BCRS4x1 using AVX2 is 1.96 times slower than that in CRS using AVX2. It has different placement of the non-zero elements in each row, nnz/row is 5, and the computation ratio is 1.9. When nnz/row is less than 8, BCRS4x1 is bad because it cannot improve the memory access.

7 Discussion

In DD-SpMV in CRS, the overheads by performance degradation factors are 130-1010% compared to the calculation kernel. The average overhead is about

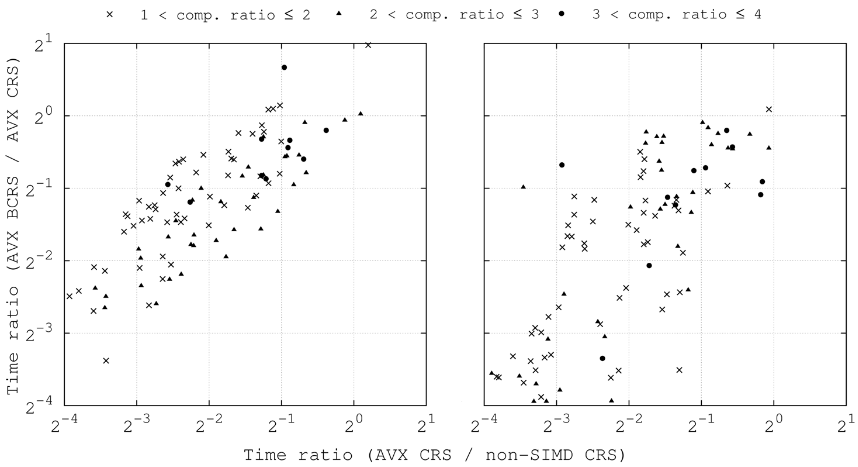

Fig. 3. Time ratio of BCRS4x1 compared with CRS [ms] (left: DD-SpMV, right: DD-TSpMV). The comp. ratio compares the amount of operations in BCRS4x1 to CRS.

300%. The overheads of the non-continuous load (set instruction) are 74-630%, and those of "REDUCTION" are 27-380%. These effects are very large.

BCRS4x1 may require at most four times the elapsed time of the calculation kernel. However, solving this trade-off problem is easy because CRS has 300% overheads.

As a result of DD-SpMV, BCRS4x1 is faster in 97/100 matrices than CRS. BCRS4x1 is effective for most cases because it eliminates the performance degradation factors. If the case of overhead is small and the computation ratio of BCRS4x1 is four, BCRS4x1 may be less than 2 times slower than CRS because it has a minimum 100% overhead in addition to the calculation kernel alone.

In the DD-TSpMV in CRS, the overheads by performance degradation factors are 180-890% compared to the calculation kernel. The overhead average is about 370%. The overheads of the non-continuous load/store (set instruction and "SCATTER") are 140-640%, and those of the Initialization and summation of work vectors are 12-150%. These effects are huge.

The maximum DD-TSpMV overheads are smaller than those of DD-SpMV. However, since the average is large, any large-sized matrix is affected.

As a result of DD-TSpMV, BCRS4x1 is faster in 99/100 matrices than CRS. If the case of overhead is small and the computation ratio of BCRS4x1 is four, BCRS4x1 may be less than 1.4 times slower than CRS because it has a minimum 180% overhead compared to the calculation kernel.

In this paper, We used row-wise access storage formats. On the other hand, there are column-wise access storage formats, for example, compressed column storage (CCS) and Block CCS (BCCS).

In BCCS, since DD-TSpMV needs work vectors for multi-threading, the column-wise access storage formats are better in DD-TSpMV than DD-SpMV. In many algorithms, the frequency use of DD-SpMV exceeds DD-TSpMV. Row-wise access storage formats have higher versatility than column-wise access storage formats.

We conclude that the effects of eliminating the performance degradation factors are large. In sparse matrix operation with SIMD, we must eliminate non-continuous memory access and horizontal vector summation.

8 Conclusion

We evaluated the performance degradation factors of CRS using AVX2 and a storage format that eliminated the performance degradation factors of CRS for DD-SpMV and DD-TSpMV. We compared DD-SpMV and DD-TSpMV in CRS, BCRS1x4, and BCRS4x1 formats.

AVX2 required the memory access pattern to be changed to allow four consecutive 64-bit elements to be read at once. Four consecutive 64-bit elements must be allowed with blocking.

In DD-SpMV in CRS using AVX2, three performance degradation factors occur: non-continuous memory load from $\boldsymbol{x}$, "FRACTION_PROCESSING," and the summation of the four DD variables in two SIMD registers. The overheads by the performance degradation factors are 130-1010% compared to the calculation kernel.

In DD-TSpMV in CRS using AVX2, three performance degradation factors occur: non-continuous load/store for $\boldsymbol{y}$, the summation of each variable of SIMD register ("REDUCTION"), and "FRACTION_PROCESSING." However, DD-TSpMV in BCRS4x1 in multi-threading needs the number of thread work vectors and their summation.

One of our improvements is column-wise multi-threading, but such thread-partitioning is difficult for row-wise access storage format. Column-wise multi-threading of BCRS4x1, which can be easily implemented, can factor out the "REDUCTION" in the storage and summation of four work vectors.

In DD-TSpMV in CRS, the overheads by performance degradation factors are 180-890% compared to the calculation kernel.

BCRS4x1 may require at most four times the elapsed time of the calculation kernel. However, solving this trade-off problem is easy because the CRS overheads are large. If the overhead case is small and the computation ratio of BCRS4x1 is four, DD-SpMV in BCRS4x1 may be less than 2.0 times slower than CRS, and DD-TSpMV in BCRS4x1 may be less than 1.4 times slower than CRS. The convert cost of BCRS4x1 is about five times more than the computation time of DD-SpMV in BCRS4x1. This convert time is small.

BCRS4x1 is suitable for AVX2 because the block size fits the SIMD register's length and BCRS4x1 eliminates the performance degradation factors in CRS. However, BCRS4x1 requires at most four times the amount of operations and data as CRS. Changing the memory access pattern and thread-partitioning for the multi-threading are good implementation for DD-SpMV and DD-TSpMV.

In the future, we will apply our technique to other SIMD lengths and multi-core systems. Column-wise multi-threading in the BCRS format only needs the length of the SIMD's register work vectors because they are expected to speed up performance on multi-core systems.

9 Acknowledgments

This work was supported by JSPS KAKENHI Grant Number 25330144. The authors thank the reviewers for their helpful comments.

References

1. Kouya, Tomonori: A Highly Efficient Implementation of Multiple Precision Sparse Matrix-Vector Multiplication and Its Application to Product-type Krylov Subspace Methods, International Journal of Numerical Methods and Applications, Vol. 7, Issue 2, pp. 107-119 (2012).
2. Bailey, D, H.: High-Precision Floating-Point Arithmetic in Scientific Computation, computing in Science and Engineering, pp. 54-61 (2005).
3. Intel, `http://software.intel.com/en-us/articles/intel-intrinsics-guide`
4. Hishinuma, T., Fujii, A., Tanaka, T., and Hasegawa, H.: AVX acceleration of DD arithmetic between a sparse matrix and vector, Parallel Processing and Applied Mathematics, LNCS 8384, Part 1, pp. 622-631 (2013).
5. Barrett, R., et al.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, SIAM pp. 57-65 (1994).
6. Hishinuma, T., Fujii, A., Tanaka, T., and Hasegawa, H.: AVX2 Acceleration of Double Precision Sparse Matrix in BCRS Format and DD Vector Product, IPSJ Transactions on Advanced Computing Systems, Vol. 7, No. 4, pp. 25-33 (2014) (in a Japanese).
7. X., Li, et al.: Design, implementation and testing of extended and mixed precision BLAS, ACS Transactions on Mathematical Software, Vol.28, Issue 2, pp. 152-205 (2002).
8. Lis: Library of Iterative Solvers for Linear Systems, `http://www.ssisc.org/lis/`
9. V., Karakasis, G., Goumas, and N., Koziris, Exploring the Effect of Block Shapes on the Performance of Sparse Kernels, 2009 IEEE International Symposium on Parallel & Distributed Processing, pp.1-8 (2009).
10. E., Im, K., Yelick, and R., Vuduc.: SPARSITY: Optimization Framework for Sparse Matrix Kernels, International Journal of High Performance Computing Applications, Vol.18, Issue 1, pp. 135-158 (2004).
11. Liu, Xing, Smelyanskiy, Mikhail, Chow, Edmond, and Dubey, Pradeep, Efficient Sparse Matrix-vector Multiplication on x86-based Many-core Processors, 27th International Conference on Supercomputing, pp.273-282 (2013).
12. Dekker, T.: A floating-point technique for extending the available precision, Numerische Mathematik, Vol. 18, pp. 224-242 (1971).
13. Knuth, D, E.: The Art of Computer Programming: Seminumerical Algorithms, Vol. 2, Addison-Wesley (1969).
14. The University of Florida Sparse Matrix Collection,
`http://www.cise.uhl.edu/research/sparse/matrices/`