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

Toshiaki Hishinuma[1], Hidehiko Hasegawa[1,2], 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) by using SIMD AVX2 for Krylov subspace methods. We compare some storage formats of DD-SpMV and DD-TSpMV for AVX2 to eliminate performance degradation factors in CRS. Our experience indicates that BCRS4x1, with fitting block size to the SIMD register's length, is effective.

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

1 Introduction

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

A sparse matrix and vector multiplication takes a lot of time of the Krylov subspace methods. We have accelerated a double precision sparse matrix and DD vector multiplication (DD-SpMV), and its transposition and DD vector multiplication (DD-TSpMV) using AVX2 (advanced vector extensions 2) [3] for an iterative solver library [4][5]. The AVX2 instruction set is a 256-bit single instruction multiple data streaming (SIMD) instruction set, and it provides a fused multiply and add instruction (FMA).

AVX2 must compute, load, and store four double precision variables at the same time. In DD-SpMV and DD-TSpMV for a compressed row storage format (CRS) [6], a non-continuous memory load and storage are needed for using AVX2. However, they degrade performance.

To avoid them, we use the BCRS format [6], which divides matrix A into $\mathrm{r} \times \mathrm{c}$ small dense submatrices (called blocks), which may include some zero-elements.

BCRS4x1 ($r = 4$, $c = 1$) would be suitable for DD-SpMV and DD-TSpMV using AVX2 because block size fits the SIMD register's length. However, BCRS4x1 requires at most four times the amounts of operations and data as CRS.

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

2 The implementation of DD-SpMV and DD-TSpMV using AVX2

2.1 DD arithmetic

DD arithmetic consists of combinations of double precision values only and uses two double precision variables to implement one quadruple precision variable [2]. A DD addition consists of 11 double precision additions, and a DD multiplication consists of two double precision additions, four double precision multiplications, and two double precision FMA instructions. We implemented DD vector $\boldsymbol{x}$ using two double precision arrays ($\boldsymbol{x}.hi$ and $\boldsymbol{x}.lo$) for SIMD acceleration.

In many cases, for an iterative solver library, input matrix A is given in double precision and iteratively used. To reduce the memory access of the sparse matrix and vector product, we used the double precision sparse matrix A and DD precision vector $\boldsymbol{x}$ product.

The byte / flop of DD operations are lower than those of double precision operations. For example, in the kernel of DD-SpMV stored in CRS, the memory requirement is 28 bytes, consisting of 8 bytes for matrix A, 16 bytes vector $\boldsymbol{x}$, and 4 bytes for vector column index. We postulate that loading vector $\boldsymbol{x}$ have cache miss. The byte / flop of double precision SpMV is 20 (bytes) / 2 (flops) = 10, that of DD matrix and DD vector product is 36 (bytes) / 21 (flops) = 1.71, and that of DD-SpMV is 28 (bytes) / 19 (flops) = 1.47. The byte / flop value of DD-SpMV is 15% that of double precision SpMV and 86% that of DD matrix and DD vector product, so DD-SpMV can be expected to greatly speed up the SIMD acceleration because of the amount of data required for the memory.

2.2 Intel SIMD AVX2

In this section, we describe SIMD AVX2. AVX2 must compute, load, and store four double precision variables at the same time. We used 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)). The "store" is storing four continuous double precision elements from register to memory that begin with the same source address. The "load" instruction is loading four continuous double precision elements that begin with the same source memory address. The "broadcast" is loading one double precision element from one source memory address to all elements of the SIMD register. The "set" is loading four double precision elements from four different source memory addresses.

We implemented three macro-functions to SIMD-ize DD-SpMV and DD-TSpMV easily. To perform random store operation "scatter", we implemented a "SCATTER" macro function by using "store" and ordinary instructions.

To store the summation of elements in the SIMD register storage to one source address, we implemented a "REDUCTION" macro function that computes a summation of four DD variables in two SIMD registers (hi and low). It consists of 11 (DD addition) $\times$ 3 = 33 double precision addition by ordinary instruction.

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

"Set", "SCATTER", "REDUCTION", and "'FRACTION_PROCESSING" are very costly. "Set" and "SCATTER" occur in random memory load and storage. The "REDUCTION" needs more computations because it cannot be SIMD-ized. The "FRACTION_PROCESSING" occurs in conditional branching.

2.3 DD-SpMV accelerated by AVX2

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

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

Variables; `av`, `xv`, and `yv` are 256 bit SIMD register variables, `x`, `y` are double precision array, and `A` is CRS format. The "set_zero" initializes SIMD register.

It needs "set" of $\boldsymbol{x}$, "REDUCTION" of $\boldsymbol{y}$, and "FRACTION_PROCESSING". They adversely affect the performance.

The BCRS r $\times$ c is expressed by the following three arrays: bind, bptr, and bval. The length of double precision array bval is the number of blocks (blk) $\times$ r $\times$ c stores values of non-zero blocks, as they are traversed row-wise. The bind array is the column indices corresponding to the blocks, and bptr is the list of block indexes where each block row starts. DD-SpMV in BCRS4x1 using AVX2 is 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, xv, yv)
for(ib=0;ib<block_row;ib++){ // block_row is about N/4.
    yv = set_zero();
    for(jb=A->bptr[ib];jb<A->bptr[ib+1];jb++){
        xv = broadcast(x[A->bind[jb]]);
        av = load(&A->bval[jb * 4]);
        yv += av * xv;
    }
    y[i*4] = store(yv);
}
```

Table 1 shows feature of CRS, BCRS1x4, BCRS4x1, and ELL[6]. The BCRS4x1 does not need "set", "REDUCTION", and "FRACTION_PROCESSING". The BCRS1x4 needs "REDUCTION", and the ELL needs "set" of $\boldsymbol{x}$. In DD-SpMV, BCRS4x1 would be the best because it eliminates performance degradation factors in CRS. However, it needs more operations and data. Block size must fit the SIMD register's length. In inner-loop (j-loop), DD-SpMV CRS needs four double precision elements of A and four non-contiguous and indirect DD elements of $\boldsymbol{x}$. Meanwhile, BCRS4x1 only needs four double precision element of A and one indirect DD element of $\boldsymbol{x}$. The amount of byte / flop of BCRS3x1 is smaller than that of CRS. The memory requirement of BCRS4x1 in the inner-loop is smaller than that of CRS.

2.4 DD-TSpMV accelerated by AVX2

In this section, we suggest fast computation methods of DD-TSpMV in making only A and changing the memory access pattern. DD-TSpMV in CRS using AVX2 is the following C code:

```
num_threads = omp_num_threads()
work = malloc(num_threads * N)
#pragma omp parallel private (i, j, k, av, xv, yv){
    k = omp_get_thread_num();
    #pragma omp for
    for(i=0;i<N;i++){
        xv = broadcast(&x[i]);
```

```
        for(j=A->ptr[i];j<A->ptr[i+1]-3;j+=4){
            jj = j + k + N //integer type
            yv = set(y[A->ind[j+0]],..,y[A->ind[j+3]]);
            av = load(&A->val[j]);
            yv += av * xv;
            yv = SCATTER(work[A->ind[jj+0],..,work[A->ind[jj+3]]);
        }
        work = FRACTION_PROCESSING(A,x);
}}
for(i=0;i<N,i++)
    y[i] = work[A->ind[j+0+k*N]+,...,+work[A->ind[jj+3]];
```

DD-TSpMV in CRS needs "set" of $\boldsymbol{y}$, "SCATTER" of $\boldsymbol{y}$, and "FRACTION_PROCESSING".

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

The performance of DD-TSpMV in BCRS4x1 applied additional column-wise multi-threading will be improved, because BCRS4x1 only computes one column in j-loop, i.e., it can be thread-partitioned easily. DD-TSpMV in BCRS4x1 using AVX2 of column-wise multi-threading is the following C code:

```
num_threads = omp_num_threads()
work = malloc(4* N)  // The length of SIMD.
#pragma omp parallel private(work, jb, av, xv, yv){
    alpha = N / num_threads * k
    beta = N / num_threads * (k+1)
    for (ib = 0 ; ib < brock_row ; ib++){
        xv = load(x[ib]):
        #pragma omp for
        for (jb = bptr[ib] ; jb < bptr[ib+1] -3 ; jb++){
            if ( alpha < bind[jb] <= beta){ //thread-partitioning
            av = load(A->bval[jb]);
            yv = broadcast(work[bind[jb]);
            yv += av * xv;
            work[num_threads][ib] =  store(yv);
}}}}
summation_of_work_vectors();
```

The `summation_of_work_vectors` consists of $3 \times$ N times "REDUCTION". BCRS4x1 can change the "REDUCTION" to "store" and the summation of four work vectors, which are the SIMD register's length. It can continuously store work vectors in j-loop. It needs only four work vectors, i.e., it can be expected to speed up the performance on the more multi-core systems.

Table 2 shows features of TSpMV in each storage format. Its BCRS4x1 applied column-wise multi-threading, and the others applied row-wise multi-threading.

BCRS1x4 and BCRS4x1 do not need "set" "scatter", or "reduction". ELL needs "set" and "REDUCTION". In addition, BCRS4x1 needs only four work

Table 2. Features of DD-TSpMV 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	the num. of row

Table 3. Elapsed times of DD-SpMV and DD-TSpMV in 4 threads [ms].

	SpMV	TSpMV	
		row-wise multi-threading	column-wise multi-threading
CRS	2.14	3.97	4.31
BCRS1x4	2.02	2.94	4.91
BCRS4x1	1.74	13.31	2.41

vectors and continuous storage for work vectors. In DD-TSpMV, BCRS1x4 or BCRS4x1 would be the best.

3 Experimental results

We test on a machine that have 4-core 8-thread Intel Core i7 4770 3.4 GHz CPU, 8 MB L3 cache, and 16 GB memory. Fedora 20 OS and Intel C/C++ compiler 15.0.0 are used. Compiler options -O3, -xCORE-AVX2, -openmp, and -fp-model precise are used. Our code is written in C and used AVX2 intrinsic instructions. OpenMP guided scheduling option and 4-thread multi-threading are used.

We used a band matrix with band width of 32 and a dimension N equal to 10^5. This matrix cannot be stored in L3 cache and does not need random access of loading for $\boldsymbol{x}$. Table 3 shows elapsed time of DD-SpMV and DD-TSpMV.

For DD-SpMV, BCRS4x1 performs the best and is 1.2 times faster than CRS. "set" and "REDUCTION" largely affect the performance.

For DD-TSpMV, BCRS4x1 with column-wise multi-threading performs the best and is 1.6 times faster than CRS with row-wise multi-threading. The elapsed times of DD-TSpMV in BCRS1x4 with row-wise and column-wise multi-threading are about 1.5 and 1.4 times slower than those of DD-SpMV, respectively. The best computation method of DD-TSpMV is BCRS4x1 with column-wise multi-threading. Column-wise multi-threading is only effective for DD-TSpMV in BCRS4x1 because of fraction processing by thread partitioning.

Figure 1 shows the time ratio of BCRS4x1 compared to CRS of 100 sparse matrices, which were taken from the University of Florida Sparse Matrix Collection (`http://www.cise.uhl.edu/research/sparse/matrices/`).

In many cases, BCRS4x1 is faster than CRS using AVX2. The time ratio of DD-SpMV in CRS using AVX2 are 0.06 - 1.34 with an average of 0.38 compared

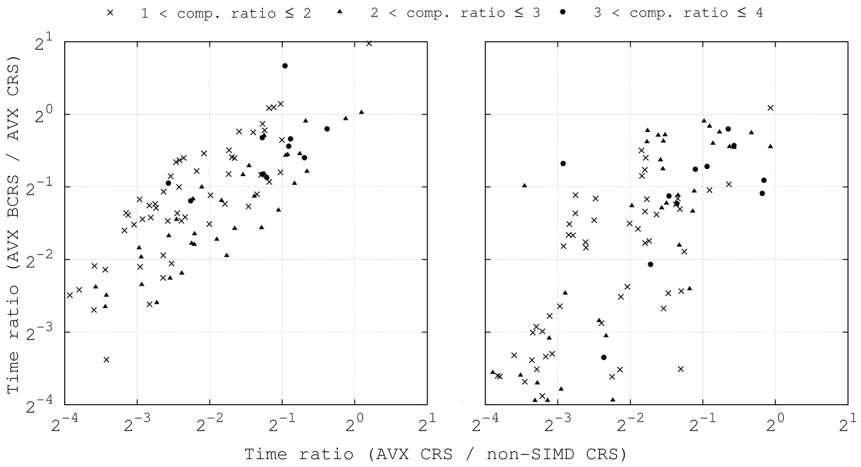

Fig. 1. Time ratio of BCRS4x1 compared with CRS [ms] (left:DD-SpMV, right:DD-TSpMV). The comp. ratio means the amount of operations in BCRS4x1 compared to CRS.

with CRS without SIMD. The time ratio of DD-SpMV in BCRS4x1 are 0.09 - 1.96 with an average of 0.43 compared with CRS using AVX2.

The time ratio of DD-TSpMV in CRS using AVX2 are 0.06 - 1.14 with an average of 0.34 compared with CRS without SIMD. The time of DD-TSpMV in BCRS4x1 using AVX2 are 0.08 - 1.07 with an average of 0.38 compared with CRS using AVX2.

There is the correlation between the effect of AVX2 in CRS and that in BCRS4x1. The correletion coefficient of DD-SpMV is 0.81, and that of DD-TSpMV is 0.65. When SIMD acceleration is not effective, BCRS4x1 may become worse. When the effect of SIMD-ization is low, nnz / row is low, or the computation ratio is more than 2.7.

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.

4 Conclusion

In this paper, we compared some storage formats of DD-SpMV and transposed DD-SpMV (DD-TSpMV) for AVX2. We analyzed the optimal storage formant to eliminate performance degradation factors in CRS.

In CRS, three performance degradation factors occur: non-continuous calculation, non-continuous memory access, and summation of four DD variables in two SIMD registers. BCRS4x1 is suitable for AVX2, because block size fits the SIMD register's length and eliminates degradation factors in CRS. However,

BCRS4x1 requires at most four times the amount of operations and data as CRS.

In DD-TSpMV using AVX2, CRS has non-continuous access of $\boldsymbol{y}$, the summation of each variable of SIMD register (REDUCTION), and "FRACTION_PROCESSING". BCRS1x4 can eliminate these problems. However, DD-TSpMV in multi-threading needs the number of thread work vectors and their summation.

One of the improvements is column-wise multi-threading, but thread-partitioning is difficult for row-wise access storage format. It is easy to implement column-wise multi-threading of BCRS4x1. It can factor out the "REDUCTION" to the storage and summation of four work vectors.

We concluded that there are two good conditions of implementation for AVX2. The first is fitting block size for the SIMD register's length. The second is making column-wise access blocking, which can access memory smoothly and is suitable for column-wise multi-threading for DD-TSpMV. A row-wise access storage format is not suitable for column-wise multi-threading because of thread-partitioning.

From the experimental results, the effect of BCRS4x1 is generally good. There is the correlation between the effect of AVX2 in CRS and that in BCRS4x1. When the effect of SIMD-ization is low, nnz / row is small, or the computation ratio is more than 2.7.

In the future, we will apply our technique for other SIMD lengths and multi-core systems. The column-wise multi-threading in the BCRS format needs only the length of SIMD's register work vectors, i.e., it can be expected to speed up the performance on the more multi-core systems.

5 Acknowledgment

This work was supported by JSPS KAKENHI Grant Number 25330144. The authors would like to 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, IJNMA, 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: Intrinsics Guide, `http://software.intel.com/en-us/articles/intel-intrinsics-guide`
4. Hishinuma, T., Fujii, A., Tanaka, T., Hasegawa, H.: AVX acceleration of DD arithmetic between a sparse matrix and vector, LNCS 8384, pp. 622-631, Springer, PPAM 2013, Part 1, Warsaw, Poland (2013).
5. Hishinuma, T., Fujii, A., Tanaka, T., Hasegawa, H.: AVX2 Acceleration of Double Precision Sparse Matrix in BCRS Format and DD Vector Product, ACS, Vol.7, No.4, pp.25-33 (2014) (in a Japanese journal).
6. Barrett, R., et al.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, SIAM pp. 57-65 (1994).