AVX acceleration of DD arithmetic between a sparse matrix and vector

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

[1] Major of Informatics, Kogakuin University, Tokyo, Japan
[2] Faculty of Lib., Info. & Media Sci., University of Tsukuba, Tsukuba, Japan
em13015@ns.kogakuin.ac.jp

Abstract. High precision arithmetic can improve the convergence of Krylov subspace methods; however, it is very costly. One system of high precision arithmetic is double-double (DD) arithmetic, which uses more than 20 double precision operations for one DD operation. We accelerated DD arithmetic using AVX SIMD instructions. The performances of vector operations in 4 threads are 51-59% of peak performance in a cache and bounded by the memory access speed out of the cache. For SpMV, we used a double precision sparse matrix A and DD vector $\boldsymbol{x}$ to reduce memory access and achieved performances of 19-46% of peak performance using padding in execution. We also achieved performances that were 9-33% of peak performance for a transposed SpMV. For these cases, the performances were not bounded by memory access.

Keywords: Double-double arithmetic, AVX, SpMV, high precision

1 Introduction

In many cases, the kernel of a numerical simulation is the solution of a large and sparse system of linear equations. Well-known algorithms for this solution are the Krylov subspace methods, but these methods diverge, stagnate, and increase iterations because of rounding errors. High precision arithmetic may be able to improve the convergence of these methods[1];

However, it is very costly[2]. One system of high precision arithmetic is double-double (DD) arithmetic[3], which does not need any special hardware and runs on general-purpose processors but uses more than 20 double precision operations for one DD operation.

In this study, we accelerate DD arithmetic using AVX (Intel Advanced Vector Extensions)[4]. The targeted operations of Krylov subspace methods are the vector operation, double precision sparse matrix and DD vector product (SpMV), and transposed double precision sparse matrix and DD vector product.

2 SIMD instruction and DD arithmetic

2.1 Test bed

The CPU is a 4-core 8-thread Intel Core i7 2600 K @ 3.4 GHz (Sandy Bridge), which can use AVX. It has an 8 MB L3 cache and 16 $\times$ 256 bit SIMD registers.

ALU of Sandy Bridge operates an FP adder and multiplier in parallel. AVX calculates four double precision variables at once. The peak performance of this CPU is 108.8 GFLOPS (3.4 × 4 (cores) × 2 (adder and multiplier) × 4 (AVX)).

Memory is a 16 GB DDR3-1333 dual channel and memory bandwidth is 21.2 GB/s (10.6 GB/s × 2 (dual channel)).

OS is Fedora 16 and the compiler is an Intel C/C++ compiler 12.0.3. Compiler options -O3, -xAVX, -openmp, and -fp-model precise are used for C-code optimization, enabling AVX instructions, OpenMP based multithreading, and enabling value-safe optimization.

2.2 DD arithmetic

DD arithmetic consists of combinations of double precision values only and uses two double precision variables to implement one quadruple precision variable[3]. It is based on the error-free floating-point arithmetic algorithms by Dekker[5] and Knuth[6]. A DD add consists of 11 double precision addition instructions, and a DD multiplication consists of 15 double precision addition instructions and nine double precision multiplication instructions.

An IEEE 754 quadruple precision variable consists of a 1 bit sign part, 15 bit exponent part, and 112 bit significant part. A DD precision variable consists of a 1 bit sign part, 11 bit exponent part, and 104 (52 × 2) bit significant part. The exponent part of a DD precision variable is 4 bits shorter and the significant part is 8 bits shorter than the exponent and significant 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 12.0.3 (ifort) and DD arithmetic without any SIMD instructions. The compiler option in ifort was -O3. Fortran REAL*16 in ifort was implemented only by integer operations. We computed $\boldsymbol{y} = \mathrm{a} \times \boldsymbol{x} + \boldsymbol{y}$, where $\boldsymbol{x}$ and $\boldsymbol{y}$ are quadruple precision vectors and a is a quadruple precision variable. Two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ whose sizes are 10^5 can be stored in the cache. The elapsed time of Fortran REAL*16 was 3 ms and that of DD arithmetic was 0.76 ms in 1 thread, which means that DD arithmetic was 3.9 times faster than Fortran REAL*16.

The exponent and significant parts of DD variables were shorter than those of quadruple variables but DD arithmetic was faster than quadruple precision arithmetic in Fortran.

3 Double-double vector operations

3.1 Vector operations

Table 1 lists DD vector operations. "Load" means the number of elements of DD vector referred in a kernel, "store" means the number of elements of DD vector moved to memory in a kernel and "complexity" means the number of double precision operations in a kernel. We computed "GFLOPS for double precision" using (Complexity × N)/elapsed time.

Table 1. Double-double vector operations

	Operation	Load	Store	Complexity (add + sub:mult)
axpy	$\boldsymbol{y} = \alpha\boldsymbol{x}+\boldsymbol{y}$	2	1	35 (26:9)
axpyz	$\boldsymbol{z} = \alpha\boldsymbol{x}+\boldsymbol{y}$	2	1	35 (26:9)
xpay	$\boldsymbol{y} = \boldsymbol{x}+\alpha\boldsymbol{y}$	2	1	35 (26:9)
dot	val $= \boldsymbol{x}\cdot\boldsymbol{y}$	2	0	35 (26:9)
nrm2	val $= \|\boldsymbol{x}\|$	1	0	31 (24:7)
scale	$\boldsymbol{x} = \alpha\boldsymbol{x}$	1	1	24 (15:9)

NOTE:α and val are DD variables, $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{z}$ are DD vectors.

We used an average at least 70 experiments and used the static scheduling in OpenMP. Figure 1 shows the performances of DD vector operations in 1 thread and 4 threads on AVX when N is 10^5. In this case, all variables were stored in the cache.

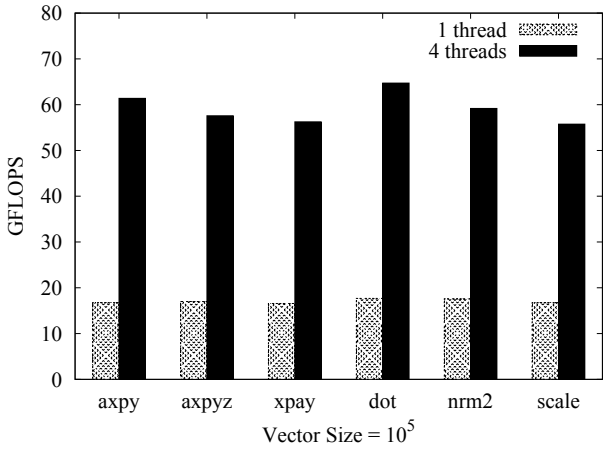

Fig. 1. Performances of vector operations on AVX

The performances of vector operations in 1 thread are from 16.6 to 17.7 GFLOPS and 61-65% of peak performance of one core, while those in 4 threads are from 55.7 to 64.7 GFLOPS and 51-59% of peak performance. The performances of vector operations in 4 threads are 3.4-3.7 times higher than those of 1 thread. Multithreading worked well for vector operations in the cache on AVX.

The performance of dot is the highest, i.e., 64.7 GFLOPS in 4 threads and 59% of peak performance. All vector operations were expressed by the same instructions and only in the case of dot, store was eliminated from inside the loop by the compiler. The performance of scale is the lowest, i.e., 55.7 GFLOPS

and 51% of peak performance. Scale consisted of one load and one store. This ratio was considered to be low performance.

The peak performance on the Intel core i7 2600 K is on the premise that FP adder and multiplier are performed in parallel. However, DD arithmetic has a different number of double precision addition and multiplication instructions. For example, axpy consists of 26 double precision addition and nine double precision multiplication instructions in a kernel. Theoretically, ALU calculates 26 double precision addition and 26 double precision multiplication instructions, i.e., a total of 52 flops. However, axpy can calculate a maximum of 35 flops. Therefore, the peak performance of axpy diminishes 67% (35/52) of the peak performance of hardware. We defined diminished the peak performance as corrected peak performance. The corrected peak performance of axpy, axpyz, xpay dot was 73.2, that of nrm2 was 70.4, and that of scale was 87.2 GFLOPS. The performances of vector operations in 1 thread were 91-97% of corrected peak performances of one core, and those in 4 threads were 65-98% of corrected peak performances.

3.2 Bound of memory access

The 8 MB L3 cache can store two vectors when N is less than 2.6×10^5. Figure 2 shows the performances of axpy in 1 thread and 4 threads on AVX when the vector size changes from 10^3 to 8.0×10^5.

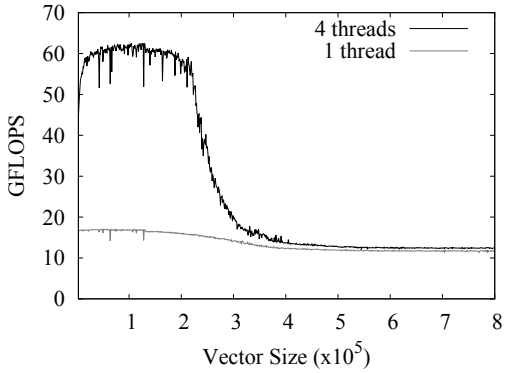

Fig. 2. Performances of axpy on AVX

In the cache, when N is 10^5, the performance of axpy in 1 thread is 17 GFLOPS, which is 62% of peak performance of one core and 93% of corrected peak performance of one core. The performance of axpy in 4 threads is 61.4 GFLOPS, which is 56% of peak performance and 84% of corrected peak performance, meaning that the performance of axpy in 4 threads is 3.6 times higher than that in 1 thread. Out of the cache, when N is 8.0×10^5, the performance of axpy diminishes to 12 GFLOPS in 1 thread and 4 threads. The speed of data

moved from/to memory was ($8.0 \times 10^8 \times 16$ (bytes) $\times$ 3)/elapsed time (2.2 ms) = 17.5 GB/s. It is 83% of the peak memory bandwidth of 21.2 GB/s. In the DDR3-1333 dual channel, the maximum theoretical performance of axpy is 15.4 GFLOPS. Out of the cache, we considered performance to be bounded by the memory access speed and multi-threading was not effective.

4 Sparse matrix vector product in DD arithmetic

4.1 Product of the double precision matrix and DD vector

Out of the cache, DD arithmetic for a vector is bounded by the memory access speed; therefore, we need to reduce memory access to accelerate computation. In many cases, for an iterative solver library, input matrix A is given by double precision and iteratively used. To reduce memory access and accelerate the sparse matrix and vector product, we used the double precision sparse matrix A and DD precision vector $\boldsymbol{x}$ product (SpMV). This allowed the size of value in the sparse matrix to half, compared to storage in DD values.

The complexity of the DD matrix and DD vector product is 35 flops. SpMV consists of 25 double precision addition and eight double precision multiplication instructions and its complexity is 33 flops. We computed GFLOPS for SpMV using (33 $\times$ the number of non-zero elements (nnz))/elapsed time. The corrected peak performance of SpMV is 72 GFLOPS in 4 threads.

To store the double precision sparse matrix A, Compressed Row Storage (CRS)[7] is used. The CRS format is expressed by the following three arrays:col_ind, row_ptr, and val; one for matrix value (val), and the other two for integers (col_ind and row_ptr). The val array stores the values of the nnz of matrix A, as they are traversed row-wise. The col_ind array stores the column indexes of the elements in the val array, i.e., if val[k] = a_{ij} then col_ind[k] = j. The row_ptr array stores the locations in the val array that start a row, i.e., if val[k] = a_{ij} then row_ptr[i], k $<$ row_ptr (i + 1).

The memory requirements of SpMV in its kernel is 50 bytes, consisting of 8 bytes for matrix A, 16 bytes for vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, and 4 bytes for vector col_ind. However, the DD matrix and DD vector product needs 58 bytes in a kernel. The value of byte per flops is 1.5 for SpMV and 1.7 for the DD matrix and DD vector product. SpMV can reduce the required memory to 88% of that of the DD matrix and DD vector product.

4.2 Fraction processing on AVX

AVX must calculate four double precision instructions at once. Processing for the remainder, which has one, two, or three elements, occurs at most once for a vector. However in SpMV, the remainder are occurs for each row. We call the processing of the remainder as fraction processing.

Four methods exists for fraction processing. Table 2 lists these methods and the performance of fraction processing in a band matrix in 4 threads, where

N is 10^4 and the bandwidth is 63 and 1023. There were three elements in the remainder in each row. This was the worst case. In SpMV, we used an average of 500 experiments and guided scheduling in OpenMP.

Table 2. Performance of fraction processing in ms (GFLOPS) (N = 10^4, 4 threads)

Bandwidth	63	1023
Padding in execution	49 (42.2)	71 (47.4)
Padding in creation CRS	47 (44.3)	71 (47.4)
Using SSE2 and normal instruction (without padding)	53 (39.0)	81 (41.1)
Using normal instruction (without padding)	48 (41.1)	71 (47.1)

"Padding" means that unnecessary elements are added to the remainder and the remainder is eliminated. "Padding in execution" assigns zero to the operand of AVX instructions at the execution. "Padding in creation" assigns zero to val and col_ind of CRS when creating CRS, but it enlarges the size of vector val and col_ind. "Using SSE2 (Streaming SIMD Extensions 2) and normal instruction" means the following code, where r represents the number of elements in the remainder:

```
if (r ≧ 2)
    process two elements with SSE2 instruction; r = r - 2;
if (r == 1)
    process one element with normal instruction;
```

"Using normal instruction" means the following code:

```
for ( ; r < 0 ; r = r -1)
    process one element with normal instruction;
```

The performance of using SSE2 and normal instruction was the least (from 39.0 to 41.1 GFLOPS), because the ymm register is the same hardware as the xmm register. When switching from AVX to SSE2, there is some processing to save the register's values; therefore, frequently switching AVX and SSE2 instructions is not recommended.

The performance of padding in creation of CRS was the best case (from 44.3 to 47.4 GFLOPS). However, the difference between padding in execution and using normal instructions which was from 1% to 7%. We chose the padding in execution and using normal instructions, which did not need an extra cost for the creation of the CRS matrix or any extra storage for a matrix.

We used a set of 15 sparse matrices that were taken from The University of Florida Sparse Matrix Collection[8]. Figure 3 shows the performances of SpMV. It arranges by nnz/row in 1 thread. The performances of SpMV using padding in execution are from 5.1 to 12.4 GFLOPS, 19-46% of peak performance of one core, and 28-69% of corrected peak performance of one core. The Performances

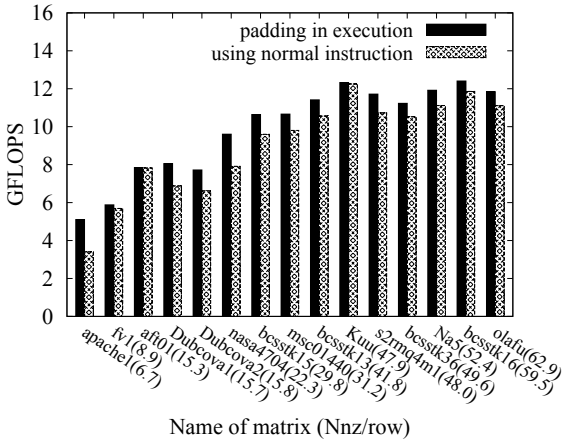

Fig. 3. Performances of sparse matrix and vector product on AVX (1 thread)

of SpMV using normal instruction are from 3.4 to 12.2 GFLOPS, 12-45% of peak performance of one core, and 19-68% of corrected peak performance of one core. The performances of padding in execution are 1.0-1.2 times higher than those of using normal instruction, except for apache1.

In this result, padding in execution is the best overall condition because it can calculate any remaining numbers at once. However, padding in normal instruction needs a loop of processing fraction, which is the number of the remainder.

4.3 Multithreading and memory access

Figure 4 shows the performances of SpMV in 1 thread and 4 threads using padding in execution. The performances of SpMV in 1 thread are from 5.1 to 12.4 GFLOPS, 19-46% of peak performance of one core, and 28-69% of corrected peak performance of one core. The performances of SpMV in 4 threads are from 18.3 to 44.5 GFLOPS, 17-41% of peak performance, and 26-62% of corrected peak performance, meaning that the performances of SpMV in 4 threads are 3.3-3.6 times higher than those in 1 thread. The performances of SpMV show that matrices with more nnz/row show higher performances.

For the evaluation of memory access, we changed the size of the band matrix from 10^3 to 4.0×10^5 when matrix bandwidth was 32. The 8 MB L3 cache can store one double precision band matrix and two DD vectors when the matrix size is less than 1.9×10^4, In the cache, when N is 10^4, performance of SpMV in 1 thread is 12.7 GFLOPS, while that of SpMV in 4 threads is 41.2 GFLOPS, meaning that the performance of SpMV in 4 threads is 3.2 times higher than that in 1 thread. Out of the cache, when N is 4.0×10^5, the performance of SpMV in 1 thread is 11.8 GFLOPS, while that of SpMV in 4 threads is 38.1 GFLOPS, meaning that the performance of SpMV in 4 threads is 3.2 times higher than that in 1 thread. The performance of SpMV in 4 threads in the cache is 1.1

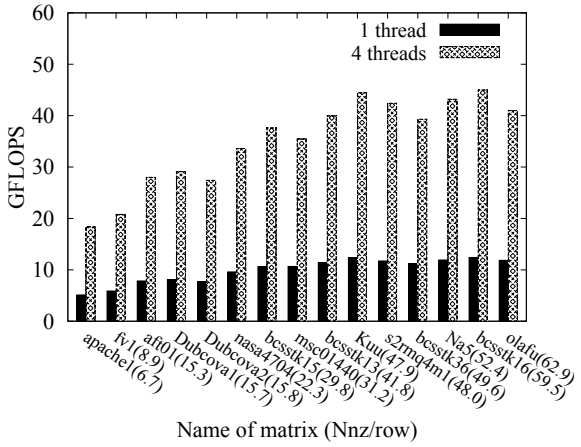

Fig. 4. Performances of the sparse matrix vector product on AVX (padding in execution)

times higher than out of the cache. As the difference in performance between in and out of the cache is small. We concluded that SpMV is not bounded by the memory access speed.

4.4 Product of the transposed sparse matrix and vector

A transposed SpMV is also necessary for a Krylov subspace method. However, the performance and memory access patterns of the transposed SpMV are different from those of the original SpMV with the same storage format. An effective storage format for a transposed SpMV is available, but storage space needs twice because of the transposed matrix. We evaluated a double precision transposed sparse matrix A and DD vector product $\boldsymbol{x}$ ($\boldsymbol{y} = A^T\boldsymbol{x}$).

The difference between $A^T\boldsymbol{x}$ and A$\boldsymbol{x}$ is the cache hit ratio of DD vector $\boldsymbol{x}$ and $\boldsymbol{y}$. When a sparse matrix has a complicated structure, SpMV has a diminishing cache hit ratio of loading $\boldsymbol{x}$, while a transposed SpMV has a diminishing cache hit ratio of loading and storing $\boldsymbol{y}$.

Figure 5 shows the performances of $A^T\boldsymbol{x}$ and A$\boldsymbol{x}$ in 1 thread and 4 threads using padding in execution. The performances of the transposed SpMV in 1 thread are from 6.8 to 11.3 GFLOPS and those in 4 threads are from 9.3 to 36.3 GFLOPS. The performances of the transposed SpMV in 4 threads are 1.3-3.3 times higher than those in 1 thread. The performances of SpMV are 1.3-1.7 times higher than those of the transposed SpMV in 4 threads, except for apache1 and aft01 that have a few nnz/row. The difference in performances between SpMV and transposed SpMV were small.

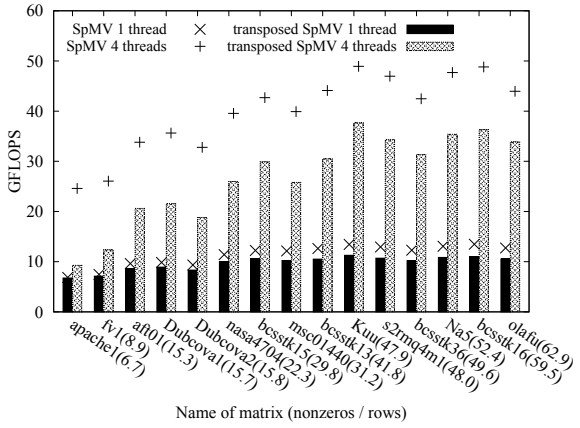

Fig. 5. Performances of the transposed SpMV on AVX (padding in execution)

5 Conclusion

We accelerated DD arithmetic using AVX SIMD instructions. The peak performance is on the premise that FP adder and multiplier were performed in parallel. However, DD arithmetic has a different number of double precision addition and multiplication instructions. It does not reach peak performance of hardware. We defined diminished peak performance as corrected peak performance.

In the cache, the performances of DD vector operations were 51-59% of peak performance and 65-98% of corrected peak performances. The performances of DD vector operations in 4 threads were 3.4-3.7 times higher than those in 1 thread. Multithreading worked well, but performances were bounded by memory access speed out of the cache. In the theoretical memory access speed, the maximum performance of axpy was 15.4 GFLOPS. We concluded that performance was bounded by the memory access speed and multithreading was not effective.

For SpMV, we used the double precision sparse matrix A and DD precision vector $\boldsymbol{x}$ product to reduce the required memory access to 88% of that of the DD matrix and DD vector product. To use the AVX instructions, processing of the remainder was necessary for each rows. We chose "padding in execution" and "using normal instructions", which did not need any extra cost for the creation of the CRS matrix or any extra storage for a matrix. Padding in execution was 1.0-1.2 times faster than using normal instruction. Padding in execution was the best overall condition.

In the cache, when the size of the bandmatrix was 10^4 and bandwidth was 32 using padding in execution, the performance of SpMV in 1 thread was 47% of peak performance of one core and 70% of the corrected peak performance of one core. The performance of SpMV in 4 threads was 38% of peak performance and 57% of corrected peak performance, meaning that the performances of SpMV

in 4 threads were 3.2 times higher than those in 1 thread. Out of the cache, when the size of the bandmatrix was 4.0×10^5, the performance of SpMV in 1 thread was 44% of peak performance of one core and 66% of corrected peak performance of one core. The performance of SpMV in 4 threads was 35% of peak performance and 53% of corrected peak performance, meaning that the performance of SpMV 4 threads was 3.2 times higher than that in 1 thread. For these cases, the performance was not bounded by memory access. The performances of SpMV was 1.3-1.7 times higher than that of the transposed SpMV in 4 threads. The difference in performance between SpMV and the transposed SpMV were small, except in some matrices which had a few nnz/row.

The performances of corrected peak performance were good except for vector operations out of the cache. ALU and multithreading worked well. AVX acceleration of DD arithmetic was effective. The problem of acceleration was in the different number of addition and multiplication instructions of DD arithmetic. In the future, we will improve the number of addition and multiplication instructions.

References

1. Hasegawa, H.: Utilizing the Quadruple-Precision floating-Point Arithmetic Operation for the Krylov Subspace Methods, The 8th SIAM Conference on Applied Linear Algebra (2003).
2. Bailey, D ,H.: High-Precision Floating-Point Arithmetic in Scientific Computation, computing in Science and Engineering, pp. 54-61 (2005).
3. Bailey, D, H.: QD (C++ / Fortran-90 double-double and quad-double package), `http://crd-legacy.lbl.gov/~dhbailey/mpdist/`
4. Intel: Intrinsics Guide, `http://software.intel.com/en-us/articles/intel-intrinsics-guide`
5. Knuth, D, E. : The Art of Computer Programming: eminumerical Algorithms,Vol. 2, Addison-Wesley (1969).
6. Dekker, T.: A floating-point technique for extending the available precision, Numerische Mathematik, Vol. 18, pp. 224-242 (1971).
7. Barrett, R., et al.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, SIAM pp. 57-65 (1994)
8. The University of Florida Sparse Matrix Collection, `http://www.cise.ufl.edu/research/sparse/matrices/`