Acceleration of Interactive Multiple Precision Arithmetic Toolbox MuPAT using FMA, SIMD, and OpenMP

Hotaka YAGI [a,1], Emiko ISHIWATA [a], and Hidehiko HASEGAWA [b]
[a] *Tokyo University of Science, Japan*
[b] *University of Tsukuba, Japan*

Abstract. MuPAT, an interactive multiple precision arithmetic toolbox for use on MATLAB and Scilab, enables users to handle quadruple- and octuple-precision arithmetic operations. MuPAT uses the DD and QD algorithms, which require from 10 to 600 double-precision floating-point operations for each DD or QD operation, which entails corresponding execution time costs. In order to reduce the execution time of vector and matrix operations, we apply FMA, AVX2, and OpenMP to MuPAT by using the MATLAB executable file. Unit stride access is required for high performance and it makes vectorization with AVX2 easier. Larger blocks are suitable for parallelization with OpenMP. That is, AVX2 is suitable for the inner-most loop and OpenMP is suitable for the outer loop. One result of adopting the described configuration is that matrix multiplication is nearly 13 times faster in a four-core environment. By using parallel processing in this way, the execution time of some DD vector operations is almost twice that of the original double-precision floating-point operations without parallel processing.

Keywords. DD, Double-Double, MATLAB, AVX2, Multicore

1. Introduction

In floating-point arithmetic, rounding error is unavoidable. The accumulation of rounding errors leads to unreliable and inaccurate results. One of the ways to reduce rounding errors is to use a high-precision arithmetic. For example, the high-precision arithmetic is used for improving the convergence of Krylov subspace methods [1] and is used in semidefinite programming problems [2]. Most high-precision arithmetics are implemented through software emulation such as the QD library [3].

Our team developed *MuPAT*, an open-source interactive *Multiple Precision Arithmetic Toolbox* [4,5] for use with the MATLAB and Scilab computing environments. MuPAT uses the DD (Double-Double) [6,7] and QD (Quad-Double) [3,7] algorithms, which are based on a combination of double-precision arithmetic operations. The high execution time cost is due to the large number of operations. We accelerate the DD arithmetics using FMA [8], AVX2 [8], and OpenMP [9].

[1]Corresponding Author: Department of Applied Mathematics, Graduate School of Science, 1-3 Kagurazaka, Shinjuku-ku, Tokyo 162-8601, Japan; E-mail: 1419521@ed.tus.ac.jp.

FMA (Fused Multiply-Add) can perform a double-precision floating-point multiply-add operation in one step with a single rounding, AVX2 (Advanced Vector Extensions 2) instructions can process four double-precision data at once, and OpenMP enables thread-level parallelism in a shared memory.

2. DD Arithmetic

DD (Double-Double) arithmetic [6,7] is based on an algorithm that enables quasi quadruple-precision arithmetic. A DD number a is represented by a combination of two double-precision numbers a_{hi} and a_{lo} such as $a = a_{hi} + a_{lo}$. According to the DD algorithm, each arithmetic operation of DD requires 10 to 30 double-precision floating-point operations and the order of computation must be maintained.

	$c = DD\ addition\ (a, b)$	$c = DD\ multiplication\ (a, b)$
1.	$s = a_{hi} \oplus b_{hi}$	$p = a_{hi} \otimes b_{hi}$
2.	$v = s \ominus a_{hi}$	$e = fl(a_{hi} \times b_{hi} - p)$
3.	$eh = a_{hi} \ominus (s \ominus v)$	$e = fl(a_{hi} \times b_{lo} + e)$
4.	$eh = eh \oplus (b_{hi} \ominus v)$	$e = fl(a_{lo} \times b_{hi} + e)$
5.	$eh = eh \oplus (a_{lo} \oplus b_{lo})$	$c_{hi} = p \oplus e$
6.	$c_{hi} = s \oplus eh$	$c_{lo} = e \ominus (c_{hi} \ominus p)$
7.	$c_{lo} = eh \ominus (c_{hi} \ominus s)$	

Figure 1. DD addition and multiplication. a, b, and c are DD numbers. The symbols $\oplus$, $\ominus$, and $\otimes$ denote the double-precision floating-point operators and the symbols $+$, $-$, and $\times$ denote mathematical operators. $fl(a \times b + c)$ means FMA.

The algorithms for DD addition and multiplication are shown in Figure 1. The number of double-precision floating-point operations for DD addition is 11. The DD multiplication algorithm utilizes FMA (Fused Multiply-Add). FMA can execute a double-precision multiply-add operation in one instruction with a single rounding. By using FMA instructions, the rounding error is reduced. The number of double-precision floating-point operations for DD multiplication is 7 with FMA and 24 without FMA. Thus, the number of double-precision floating-point operations for the DD multiply-add is 18 (=11+7).

3. Environment of Parallelization

Since the order of computation in DD arithmetic cannot be changed, we consider processing multiple data simultaneously by using data-level parallelism for acceleration. The unit of time of each operation is not changed, but if multiple results can be obtained in one unit of time, then the total execution time is reduced. We applied data-level parallelism to vector and matrix operations.

AVX2 (Advanced Vector Extensions 2) instructions [8] can process four double-precision data in one unit of time. The same arithmetic operations are applied to these four data. To do this, four double-precision data must be prepared on a SIMD register. An AVX2 load instruction can load four double-precision data from a continuous memory location in one unit of time. However, for a discontinuous memory location, four scalar load instructions are needed. AVX2 instructions cannot sum up the SIMD register elements. The performance may increase four-fold.

OpenMP [9] allows thread-level parallelism on shared memory for a multicore environment. Each thread is a separate process with its own instructions and data. By processing threads with the different cores simultaneously, the performance may be increased by the number of cores. A loop is parallelized by putting a pragma directive above the loop. There are two scheduling methods: block and cyclic scheduling.

We assume the memory references should be in column order, since MATLAB stores data column-wise.

Performance [Gflops/sec] is defined as the number of double-precision floating-point operations [flops] divided by the execution time [sec]. The upper bound of performance is defined as min(computational performance, memory performance$\times$operational intensity). The computational performance [Gflops/sec] is defined as the product of clock frequency for the CPU [Hz] and the number of flops which can be computed in one unit of time [flops/sec]. Performance is increased four-fold using AVX2 and by the number of cores using OpenMP. Memory performance [Gbytes/sec] is defined as 8 bytes/cycle times the product of clock frequency for memory [GHz] and the number of channels. Operational intensity (O. I.) [flops/bytes] is defined as the number of double-precision floating-point operations [flops] divided by the number of memory references [bytes].

We used an Intel Core i7 7820HQ, 2.90 GHz CPU, with LPDDR3-2133 memory and Intel compiler 18.0.3 with options -O2, -fma, -mavx, -fopenmp, and -fp-model precise. The peak computational performance of a single core including FMA is 5.80 Gflops/sec, and that of AVX2 or that of four cores is 23.20 Gflops/sec. Performance is 92.80 Gflops/sec using AVX2 with four cores. The peak memory performance is 34.13 Gbytes/sec because there are two channels.

Performance is bounded by computational performance or memory performance [10]. Performance is bounded by memory performance when operational intensity is 0.17 (= 5.80/34.13) or lower without parallelization, 0.68 (= 23.20/34.13) or lower when using AVX2 or OpenMP, and 2.72 (= 92.80/34.13) or lower when using both AVX2 and OpenMP. When operational intensity is higher than those values, performance is bounded by computational performance.

4. Experiment in DD Arithmetic for Matrix and Vector Operations

4.1. DD Vector Operations

In vector operations, the four elements of a vector are processed simultaneously using AVX2. When using OpenMP, different parts of the vector are processed by each thread. When we compute the inner product with AVX2, we must sum up the four SIMD register elements, with requires three scalar additions. In the case of using OpenMP, since we must sum up the p thread elements, $p-1$ scalar additions are needed. When using both AVX2 and OpenMP, each thread computes a partial sum with a vector of length N/p using AVX2. Then, these partial sums are converted to a global sum.

Table 1 shows the operational intensity (O. I.) and the experimental results of vector operations when the dimension is 4,096,000. In many vector operations, the upper bound is calculated by the product of memory performance and its operational intensity. According to Table 1, the performances of four operations ($\boldsymbol{y} = \alpha\boldsymbol{x}$, $\boldsymbol{z} = \boldsymbol{x}+\boldsymbol{y}$, $\boldsymbol{z} = \boldsymbol{x}+\boldsymbol{x}$, and $\boldsymbol{z} = \alpha\boldsymbol{x}+\boldsymbol{y}$) with a new vector variable on the left is nearly 20% of the upper bound of performance when using both AVX2 and OpenMP. However, the performances of six operations ($\boldsymbol{x} = \alpha\boldsymbol{x}$, $\boldsymbol{y} = \boldsymbol{x}+\boldsymbol{y}$, $\boldsymbol{x} = \boldsymbol{x}+\boldsymbol{x}$, $\boldsymbol{y} = \alpha\boldsymbol{x}+\boldsymbol{y}$, $\alpha = \boldsymbol{x}^T\boldsymbol{y}$, and $\beta = \boldsymbol{x}^T\boldsymbol{x}$) with no new vector variables on the left are nearly 70% of the upper bound of performance when using both AVX2 and OpenMP. Since the four operations $\boldsymbol{y} = \alpha\boldsymbol{x}$, $\boldsymbol{z} = \boldsymbol{x}+\boldsymbol{y}$, $\boldsymbol{z} = \boldsymbol{x}+\boldsymbol{x}$, and $\boldsymbol{z} = \alpha\boldsymbol{x}+\boldsymbol{y}$ require memory allocation, it is difficult to achieve high performance by parallelization. The other six operations do not have the overhead of allocating memory. Figure 2 shows the performances of $\alpha\boldsymbol{x}$, $\boldsymbol{x}+\boldsymbol{y}$, and $\alpha\boldsymbol{x}+\boldsymbol{y}$ with and without the overhead of allocating memory. As N becomes larger, the differences in performance increase between with and without overhead.

Table 1. Number of double-precision floating-point operations [flops], number of memory references [bytes], operational intensity [flops/bytes], memory requirement, and performances [Gflops/sec] for DD vector operations for N = 4,096,000.

	Flops	Memory references	O. I.	Memory requirement	Serial	AVX2	OpenMP	AVX2& OpenMP
$\boldsymbol{y} = \alpha\boldsymbol{x}$	$7N$	$2N\times16$	0.22	$2N$	1.19	1.51	1.30	1.25
$\boldsymbol{x} = \alpha\boldsymbol{x}$	$7N$	$2N\times16$	0.22	N	2.05	4.78	5.03	5.21
$\boldsymbol{z} = \boldsymbol{x}+\boldsymbol{y}$	$11N$	$3N\times16$	0.23	$3N$	1.96	2.05	2.15	1.67
$\boldsymbol{y} = \boldsymbol{x}+\boldsymbol{y}$	$11N$	$3N\times16$	0.23	$2N$	4.10	5.18	5.56	5.43
$\boldsymbol{z} = \boldsymbol{x}+\boldsymbol{x}$	$11N$	$2N\times16$	0.34	$2N$	2.25	2.25	2.15	2.25
$\boldsymbol{x} = \boldsymbol{x}+\boldsymbol{x}$	$11N$	$2N\times16$	0.34	N	4.10	8.19	7.51	8.34
$\boldsymbol{z} = \alpha\boldsymbol{x}+\boldsymbol{y}$	$18N$	$3N\times16$	0.38	$3N$	2.23	3.21	3.35	2.73
$\boldsymbol{y} = \alpha\boldsymbol{x}+\boldsymbol{y}$	$18N$	$3N\times16$	0.38	$2N$	2.84	7.37	8.57	8.78
$\alpha = \boldsymbol{x}^T\boldsymbol{y}$	$18N$	$2N\times16$	0.56	$2N$	2.30	7.76	8.19	14.18
$\beta = \boldsymbol{x}^T\boldsymbol{x}$	$18N$	$N\times16$	1.13	N	2.30	8.67	8.38	26.33

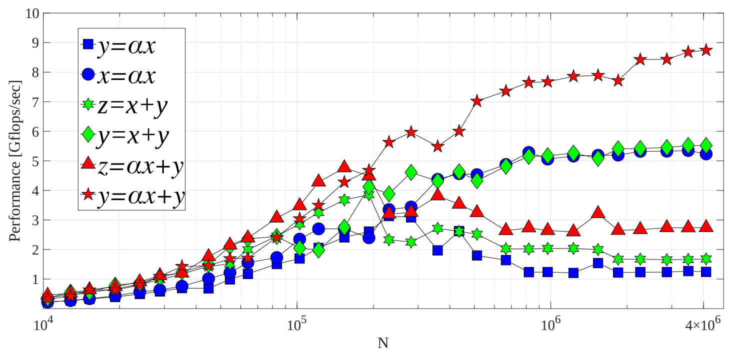

Figure 2. Performances [Gflops/sec] using AVX2 and OpenMP for $\alpha\boldsymbol{x}$, $\boldsymbol{x}+\boldsymbol{y}$, and $\alpha\boldsymbol{x}+\boldsymbol{y}$.

For all ten operations, a higher operational intensity results in a higher performance. For the same operational intensity, performance is higher for a smaller memory requirement. It is also important for high performance to reduce the number of memory references and the memory requirements.

In summary, for vector operations, when memory allocation overhead is not required, the performances are almost 70% of the upper bound by using AVX2 and OpenMP. Otherwise, the performances are degraded to 20%.

4.2. DD Matrix-Vector Multiplication

Matrix-vector multiplication $\boldsymbol{y} = A\boldsymbol{x}$ is written as $y_i = \sum a_{ij}x_j$. The operational intensity of matrix-vector multiplication is 1.13, because the number of double-precision floating-point operations is $7N^2 + 11N(N-1)$ and the number of memory references is $(N^2 + 2N) \times 16$, and this operation is limited by memory performance when using both AVX2 and OpenMP. Matrix-vector multiplication has two algorithms, PB and PDOT. The memory references for the matrix A are column order in PB, and those for matrix A are row order in PDOT. Since MATLAB stores data column-wise, the memory references for PB are continuous. Unit stride access can be used for PB.

4.2.1. AVX2

There are four algorithms using AVX2 shown in Figure 3. The order to load elements of matrix A is by column in PDOT$_{\text{PB}}$ and PB$_{\text{PB}}$, and by row in PDOT$_{\text{PDOT}}$ and PB$_{\text{PDOT}}$. Here, a prefix v indicates a vector and the variables are DD numbers. The variables va, vx, and vy hold the four DD numbers. The *vload* instruction loads four continuous data as $a(i,j)$ to $a(i+3,j)$ or $x(j)$ to $x(j+3)$. The $vmuladd(vy,va,vx)$ instruction performs the multiply-add operation $vy = vy + va * vx$ to compute four elements simultaneously and costs 18 double-precision floating-point operations. The $vmul(va,vx)$ instruction is the multiplication operation $vy = va * vx$ and costs 7 double-precision floating-point operations. The $sum(vy)$ instruction sums the data in the SIMD register and costs 11×3 double-precision floating-point operations.

PDOT$_{\text{PDOT}}$
```
for (i = 0; i < n; i++)
        vy = {0,0,0,0}     // setzero
    for(j = 0; j < n; j += 4)
                              // load 4 discontinuous data
        va = set(a(i,j), ..., a(i,j + 3))
        vx = vload(x(j))    // load 4 continuous data
        vy = vmuladd(vy, va, vx) //  vy = vy + va * vx
    y(i) = sum(vy) // sum 4 elements in SIMD register
```

PB$_{\text{PDOT}}$
```
y = setzeros(n, 1)  // set n×1 vector as zero
for (j = 0; j < n; j += 4)
    vx = vload(x(j))
    for(i = 0; i < n; i++)
        va = set(a(i,j), ..., a(i,j + 3))
        vy = vmul(va, vx)   // vy = va * vx
        y(i) = y(i) + sum(vy)
```

PDOT$_{\text{PB}}$
```
for (i = 0; i < n; i += 4)
    vy = {0,0,0,0}
    for(j = 0; j < n; j++)
        va = vload(a(i,j))
                    // load 1 element and fill SIMD register
        vx = broadcast(x(j))
        vy = vmuladd(vy, va, vx)
        vstore(y(i), vy)        // store 4 continuous data
```

PB$_{\text{PB}}$
```
y = setzeros(n, 1)
for (j = 0; j < n; j++)
    vx = broadcast(x(j))
    for(i = 0; i < n; i += 4)
        va = vload(a(i,j))
        vy = vload(y(i))
        vy = vmuladd(vy, va, vx)
        vstore(y(i), vy)
```

Figure 3. Algorithms using AVX2 for $\boldsymbol{y} = A\boldsymbol{x}$.

Table 2. Instructions and performances [Gflops/sec] for $\boldsymbol{y} = A\boldsymbol{x}$ for N=2,500.

	Computation	Load	Store	Serial	AVX2	OpenMP	AVX2& OpenMP
PDOT	N^2 *muladd*	$2N^2$ *load*	N *store*	1.25	-	4.96	-
PB	N^2 *muladd*	N *load* $2N^2$ *load*	N^2 *store*	3.55	-	12.77	-
PDOT$_{\text{PDOT}}$	$N^2/4$ *vmuladd* N *sum*	N *setzero* N^2 *load* $N^2/4$ *vload*	N *store*	-	1.64	-	6.11
PDOT$_{\text{PB}}$	$N^2/4$ *vmuladd*	$N/4$ *setzero* $N^2/4$ *vload* $N^2/4$ *broadcast*	$N/4$ *vstore*	-	4.96	-	10.58
PB$_{\text{PDOT}}$	$N^2/4$ *vmul* $N^2/4$ *sum* $N^2/4$ *add*	$N/4$ *vload* $5N^2/4$ *load*	$N^2/4$ *store*	-	3.89	-	14.72
PB$_{\text{PB}}$	$N^2/4$ *vmuladd*	N *broadcast* $N^2/2$ *vload*	$N^2/4$ *vstore*	-	11.97	-	25.63

It is clear from comparing PDOT with PB in Table 2 that unit stride access is required to achieve high performance. Since the performance for PB$_{\text{PB}}$ (11.97) is increased almost four-fold from that shown for PB (3.35), and that for PDOT$_{\text{PB}}$ (4.96) is also increased four-fold from the PDOT value (1.25) shown in Table 2, it is clearly important to use the *vmuladd* instruction instead of *muladd* in order to increase performance. To improve performance with the *vmuladd* instruction, the *vload* and *vstore* instructions must also be used.

4.2.2. AVX2 and OpenMP

The performance of PB$_{\text{PB}}$ using AVX2 in the innermost loop was the highest as shown in Section 4.2.1. Since we assume that the innermost loop i is parallelized by AVX2, we parallelize the outer loop by OpenMP.

As in Table 2, when using AVX2 and OpenMP, the performances of PB$_{\text{PB}}$ ($11.97 \rightarrow 25.63$) and PDOT$_{\text{PB}}$ ($4.96 \rightarrow 10.58$), which have improved performance with AVX2, are only twice as high as the case of just AVX2. By using AVX2 and OpenMP, the performances of PDOT$_{\text{PDOT}}$ ($1.64 \rightarrow 6.11$) and PB$_{\text{PDOT}}$ ($3.89 \rightarrow 14.72$), which did not improve much with AVX2, is nearly four times higher than the case of just AVX2. Since the OpenMP can parallelize and accelerate regardless of the order of memory references, the performances of PDOT$_{\text{PDOT}}$ and PB$_{\text{PDOT}}$ can be increased by using OpenMP. The performance of PB$_{\text{PB}}$ is the highest for using AVX2 and OpenMP, 25.63 Gflops/sec, and it is almost 7 times faster than before parallelization.

As shown in Table 2, even using both AVX2 and OpenMP, the execution time cannot be reduced 16-fold (AVX2 $\times$ four cores) compared to without parallelization. Since the operational intensity of matrix-vector multiplication is 1.13, which is lower than 2.72, matrix-vector multiplication is limited by memory performance when using AVX2 and OpenMP. Since the upper bound of performance is 38.53 Gflops/sec, which is calculated using operational intensity $\times$ memory performance, the performance 25.63 Gflops/sec is 67% of the upper bound of performance.

As for DD matrix-vector multiplication, when using the AVX2 *vmuladd*, *vload* and *vstore* instructions with OpenMP applied to the outer loop, the performance can reach 67% of the upper bound.

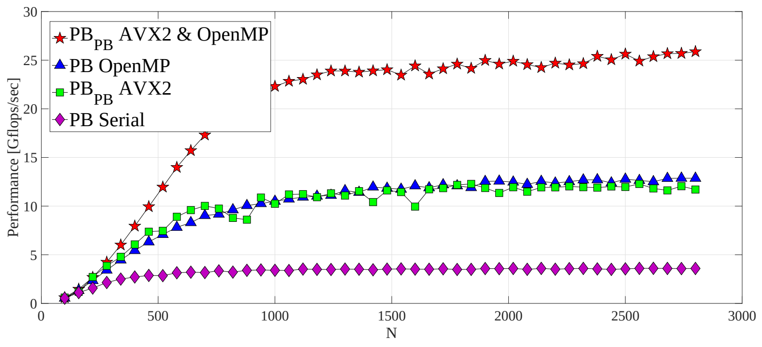

Figure 4. Performances [Gflops/sec] for Serial, AVX2 or OpenMP, and AVX2 and OpenMP for $\boldsymbol{y} = A\boldsymbol{x}$.

4.3. *DD Matrix-Matrix Multiplication*

Matrix multiplication $C = AB$ is written as $c_{ij} = \sum a_{ik}b_{kj}$ with three nested loops i, j, and k. Since operational intensity for matrix multiplication is quite high, at $O(N) = 18N^3/(3N^2 \times 16)$, this operation is limited by computational performance. As we have seen in Section 4.2, unit stride access is essential to achieve high performance. If the innermost processed loop is the k or j loop, then unit stride access cannot be performed, because MATLAB stores data in column-wise order. Since the index of the innermost loop for matrix multiplication must be i, there are two implementation algorithms: MP and PDOT. MP uses j-k-i order and PDOT uses k-j-i order for the loops.

AVX2 is easily applied to the loops in both algorithms, MP and PDOT. In order to parallelize using *vload* and *vstore* instructions, the loop of index i should be processed as a vector, in which case, its performance increases almost four-fold, as shown in Table 3. One of the remaining loops, with index j or k, will be parallelized by OpenMP. Figure 5 shows the four algorithms according to which loops are parallelized.

Table 3. Instructions and performances [Gflops/sec] for $C = AB$ for N = 2,500.

	Additional Instructions	Serial	AVX2	OpenMP block	OpenMP cyclic	AVX2& OpenMP block	AVX2& OpenMP cyclic
MP		3.60	13.73	13.04	12.88	46.71	45.58
MP$_{\text{PB}}$	N^2 *load/store* N^2 *sum*	-	-	13.04	13.08	29.46	29.36
PDOT		3.67	12.25	12.79	12.62	15.06	15.02
PDOT$_{\text{PDOT}}$	N *load/store* N *sum*	-	-	7.55	7.76	8.46	8.54

MP

```
#pragma omp for
$for\ (j = 0; j < n; j++)$
  $for\ (k = 0; k < n; k++)$
    $t = b(k,j)$
    $for(i = 0; i < n; i++)$
      $c(i,j) = c(i,j) + a(i,k) * t$
```

PDOT

```
$for\ (k = 0; k < n; k++)$
  #pragma omp for
  $for\ (j = 0; j < n; j++)$
    $t = b(k,j)$
    $for(i = 0; i < n; i++)$
      $c(i,j) = c(i,j) + a(i,k) * t$
```

MP$_{\text{PB}}$

```
$for\ (j = 0; j < n; j++)$
  $vtl = setzeros(n, 1)$   // set n×1 vector as zero
  #pragma omp for
  $for\ (k = 0; k < n; k++)$
    $t = b(k,j)$
    $for(i = 0; i < n; i++)$
      $vtl(i) = vtl(i) + a(i,k) * t$
  #pragma omp critical
  $c(:,j) = c(:,j) + sum(vtl)$
```

PDOT$_{\text{PDOT}}$

```
#pragma omp for
$for\ (k = 0; k < n; k++)$
  $for\ (j = 0; j < n; j++)$
    $vtl = setzeros(n, 1)$
    $t = b(k,j)$
    $for(i = 0; i < n; i++)$
      $vtl(i) = vtl(i) + a(i,k) * t$
  // vector addition $p$ times and store it to row of c
    #pragma omp critical
    $c(:,j) = c(:,j) + sum(vtl)$
```

Figure 5. Algorithms using OpenMP for $C = AB$.

MP and PDOT are easily parallelized with putting the "#*pragma omp for*" directive above an intended *for* statement, as shown in Figure 5. All the threads in MP$_{\text{PB}}$ and PDOT$_{\text{PDOT}}$ can potentially process and update the same data location in parallel. To avoid this problem, in each thread, we defined thread-local vector *vtl* for holding a partial sum as a private variable. Thread-local vector *vtl* requires memory equal to the product of $N \times 16$ and the number of threads. Accumulation to a global sum from each partial sum is processed serially by inserting a "#*pragma omp critical*" directive. Each $sum(vtl)$ in Figure 5 costs $11pN$ double-precision floating-point operations, where p denotes the number of threads. In Figure 5, $c(:, j)$ denotes the j-th column of array c. MP and PDOT can be processed in serial without OpenMP. However, executing MP$_{\text{PB}}$ and PDOT$_{\text{PDOT}}$ requires using OpenMP.

In the case of OpenMP, as shown in Table 3, the performances for MP and PDOT are almost 13 Gflops/sec, or about four times higher than without parallelization, but that for PDOT$_{\text{PDOT}}$ is about 8 Gflops/sec, which is lower than other cases. There is almost no difference between block and cyclic scheduling. PDOT$_{\text{PDOT}}$ needs one $sum(vtl)$ for each innermost loop, total cost of $sum(vtl)$ becomes $11pN^3$ double-precision floating-point operations. MP$_{\text{PB}}$ also needs $sum(vtl)$, but its total cost is $11pN^2$ double-precision floating-point operations because of once for each nested loop. Since DD Matrix multiplication requires $18N^3$ double-precision floating-point operations, the overhead for PDOT$_{\text{PDOT}}$ is extremely large and greater than original computations. Since the additional overhead for parallelization in MP is much less than that for MP$_{\text{PB}}$, as shown in Figure 5, the performance for MP was higher than that of MP$_{\text{PB}}$, as shown in Table 3. It is clear that less additional overhead for parallelization is required for high performance.

When using both AVX2 and OpenMP, there is a difference in performance between PDOT and MP for large N, as shown in Figure 6. Since the all elements of c_{ij} are updated N times in PDOT, but a column of c_{ij} are updated N times in MP. The data locality of MP is higher than that of PDOT. Thus, the performance of MP is higher than that of PDOT.

Since operational intensity for matrix multiplication is larger than 2.72, the upper bound of performance for matrix multiplication is 92.80 Gflops/sec. The DD matrix multiplication has a 16-fold increase, because this operation is limited by computational per-

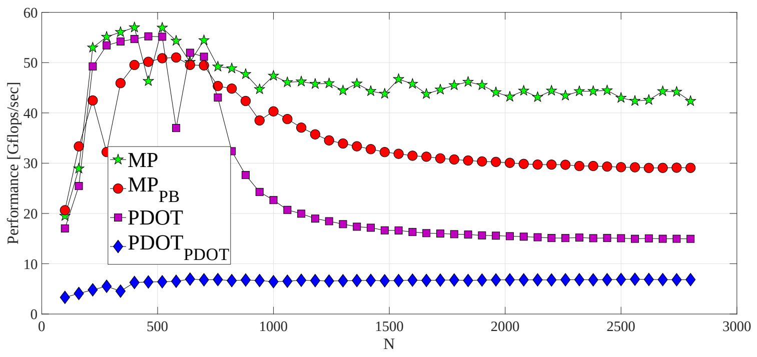

Figure 6. Performances [Gflops/sec] using AVX2 and OpenMP in block scheduling for $C = AB$.

formance. Although MP demonstrates the best performance 46.71 Gflops/sec, 50% of the upper bound of performance and 13-fold higher than without parallelization (46.71/3.60).

When operational intensity is high and the outer loop is parallelized by using OpenMP with less additional overhead, the operation is much accelerated. Thus, in order to use both AVX2 and OpenMP, it is important to vectorize the innermost loop by using AVX2 and parallelize outer loops by using OpenMP while avoiding the same memory location being updated by different threads.

Table 4. Execution time [sec] in double-precision and DD precision. N=4,096,000 for vector operations, and N=2,500 for matrix-vector multiplication.

	$\boldsymbol{x} = \alpha \boldsymbol{x}$	$\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{y}$	$\boldsymbol{y} = \alpha \boldsymbol{x} + \boldsymbol{y}$	$\alpha = \boldsymbol{x}^T \boldsymbol{y}$	$\boldsymbol{y} = A\boldsymbol{x}$
Double	0.0028	0.0042	0.0042	0.0027	0.0022
DD (AVX2 & OpenMP)	0.0055	0.0083	0.0084	0.0052	0.0044
DD / Double	1.96	1.98	2.00	1.93	2.00

5. Conclusion

In response to demands for ways to facilitate high-precision arithmetic with an interactive computing environment, we developed MuPAT on Scilab/MATLAB. MuPAT uses DD and QD arithmetics that require large numbers of double-precision floating-point operations. Executing DD arithmetic operations takes 10 to 30 times the execution time of double-precision floating-point operations, due to the heavy computation load and the need to maintain computation order.

We utilized computation offloading to call an outer C function with the MATLAB executable file, and parallelized the computation by using AVX2 and OpenMP. By using a C executable with MATLAB, the code becomes platform dependent, but we intended to achieve fast computation using data-level parallelism such as with AVX2 and OpenMP instead of platform independence. We used an FMA (Fused Multiply-Add) based algorithm to reduce rounding errors.

AVX2 (Advanced Vector Extensions 2) executes operations on four double-precision numbers simultaneously. To achieve high performance, it requires that vectorized fused multiply-add operations and load/store instructions be used. Unit stride access is essential for using vector load/store instructions. OpenMP enables the same operations to process different data within threads on different cores. To avoid the same data location from being accessed by different threads, we should apply OpenMP to as large a block as possible.

Thus, in order to use both AVX2 and OpenMP, it is important to vectorize the innermost loop by using AVX2 and parallelize outer loops by using OpenMP while avoiding the same memory location being updated by different threads. By utilizing both AVX2 and OpenMP, the performance of the matrix-vector multiplications became 25.63 Gflops/sec (67% of the upper bound), and the performance of matrix multiplication became 46.71 Gflops/sec (50% of the upper bound). Each DD arithmetic operation requires 10 to 30 double-precision floating-point operations, however the execution time of these DD operations for vector operations and matrix-vector multiplication in parallel processing became only about twice that of the original double-precision floating-point operations without parallel processing.

The performance of these DD operations is bounded by memory performance. It is possible to compute many more operations in the same time if no additional data are required. The execution times of $\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{y}$ and $\boldsymbol{y} = \alpha \boldsymbol{x} + \boldsymbol{y}$ are the same. These facts mean that parallel processing provides us more accurate results and/or processes a much larger workload for during the same time without an extra cost.

Acknowledgment

The authors would like to thank the reviewers for their valuable comments. This research was supported by a grant from the Japan Society for the Promotion of Science (JSPS : JP17K00164).

References

[1] T. Saito, E. Ishiwata, and H. Hasegawa, Analysis of the GCR method with mixed precision arithmetic using QuPAT, *Journal of Computational Science* **3** (2012), 87-91.

[2] H. Waki, M. Nakata, and M. Muramatsu, Strange behaviors of interior point methods for solving semidefinite programming problems in polynomial optimization, *Computational Optimization and Applications* **53** (2012), 823-844.

[3] Y. Hida, X. S. Li, and D. H. Baily, QD arithmetic: algorithms, implementation, and application, Technical Report LBNL-46996, Lawrence Berkeley National Laboratory, 2000.

[4] S. Kikkawa, T. Saito, E. Ishiwata, and H. Hasegawa, Development and acceleration of multiple precision arithmetic toolbox MuPAT for Scilab, *JSIAM Letters* **5** (2013), 9-12.

[5] MuPAT on MATLAB. `https://www.ed.tus.ac.jp/1419521/index.html`

[6] T. J. Dekker, A floating-point technique for extending the available precision, *Numerische Mathematik* **18** (1971), 224-242.

[7] J.- M. Muller, et al, Handbook of Floating-Point Arithmetic, *Birkhäuser*, 2010.

[8] Intel Intrinsics Guide. `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`

[9] P. S. Pacheco, An Introduction to Parallel Programming, *Morgan Kaufmann*, 2011.

[10] S. Williams, A. Waterman, and D. Patterson, Roofline: An insightful visual performance model for multicore architectures, *Communications of the ACM* **52** (2009), 65-76.