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 the users to treat quadruple and octuple precision arithmetics. MuPAT uses DD and QD algorithm that require from 10 to 600 double precision floating-point operations for each operation. That causes taking much time for computation. In order to reduce the execution time in basic matrix and vector operations, FMA, AVX2 and OpenMP are used. We applied these features to MuPAT using the function of MATLAB executable file, and analyzed the performance. Using AVX2 and OpenMP to unit stride memory references and to avoid synchronization can increase the performance of DD operations. We confirm that the effectiveness of AVX2 and OpenMP is depending on the operational intensity.

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

1. Introduction TODO. 高精度演算の意義を足す (査読 4)

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 the rounding errors is the use of high-precision arithmetic. Most of high-precision arithmetics are accomplished by software emulation such as QD library [1].

Our team developed *MuPAT*, an open-source interactive *Multiple Precision Arithmetic Toolbox* [2,3] for use with the MATLAB and Scilab computing environments. MuPAT uses DD (Double-Double) [4] and QD (Quad-Double) [1,4] algorithms which are based on the combination of double precision arithmetic, and excessively long computation times often result. DD and QD arithmetics are possible to accelerate by applying FMA [5], AVX2 [5], and OpenMP [6]. A floating-point multiply-add operation is performed in one step with a single rounding by FMA, four double precisions of data are processed at once with AVX2 instruction, and OpenMP enables thread-level parallelism in a shared memory. We implement the combination of these features, and precisely examine some basic matrix and vector operations for DD arithmetic.

---

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

2. DD Arithmetic TODO.DD 演算の説明足す (査読 1, 3), FMA の説明足す (査読 2)

DD (Double-Double) arithmetic [4] is based on the algorithm that enables quasi quadruple precision arithmetic. A DD number is defined by two double precision numbers. A DD vector is defined by two vectors with structure of arrays. DD algorithm requires 10 to 30 of double precision operations for each operation and has data dependency of flow, i.e. the order of computation must be kept. The accuracy of this algorithm consists of the order of the computation.

DD multiplication algorithm [1] can utilize FMA (Fused Multiply-Add). FMA is a floating-point units that can execute double precision multiply-and-add operation in one instruction. Since a double precision multiply-and-add operation is performed in one step via a single rounding FMA instructions, the rounding error is reduced. The algorithms for DD addition and multiplication are shown in Figure 1.

	DD addition (a, b)	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 using FMA. The symbols $\oplus, \ominus, \otimes$ mean the floating-point operators performed on computer, and the symbols $+, -, \times$ mean mathematical operators. $fl(a \times b + c)$ means FMA. TODO.input と output が何かを明確にする (査読 1)

The number of floating-point operations for DD multiplication is 7 with FMA and 24 without FMA. That for DD addition is 11. DD addition in Figure 1 is called Cray-style [7]. Highly accurate, more expensive (20 times of floating-point operations) algorithm is called IEEE-style [7]. Cray-style is mainly used, because it is faster [8]. TODO. IEEE-style の説明をするなら詳しくする (査読 1)

We measured the execution time of DD multiplication and multiply-and-add that are repeated 10^7 times using FMA. We used Intel Core i7 7820HQ, 2.9 GHz CPU and Intel compiler 18.0.3 with options -O2, -fma, -mavx, -fopenmp, and -fp-model precise.

The execution time of DD multiplication is 0.018 sec, and that of multiply-and-add is 0.048 sec. The ratio for execution time (0.048/0.018 = 2.67) and the ratio for the floating-point operations (18/7 = 2.57) is almost the same. The number of floating-point operations determine the execution time for scalar operation.

3. Performance Prediction TODO.AVX2 の説明足す (査読 2), 実験環境を強調?(査読 3), Roofline の図を入れる?(査読 3)

Since the order of computation in DD arithmetic can not be changed, we consider to process multiple data simultaneously for parallelization. The unit time of one operation is not changed, but if multiple results can be obtained in one unit time, then the total execution time is reduced. We examine to accelerate for basic matrix and vector operations. AVX2 instructions [5] can perform the single instruction for four double precision of

data. The computational performance increases four times, not the memory performance. OpenMP [6] can be used for a multicore environment. The memory performance and the computational performance per each core do not change, but it enables to process using multicore. For nested-loop operations, we should discuss which loops to parallelize.

We assume the memory references should be the column order, since MATLAB stores data in column wise. Unit stride memory reference can use the data read by cache line of length of 64 bytes; however, when reading data N-stride (N is longer than the length of cache line), each data is read from memory to the register [9].

The performance (= the number of floating-point operations / the execution time [Gflops/sec]) is determined by the memory performance [Gbytes/sec] and the computational performance [Gflops/sec]. For the experiment environment, in the case of single core, the peak computational performance is 5.8 Gflops/sec because there are two FPUs. It increases to 23.2 Gflops/sec by using AVX2. In the case of four cores with AVX2, it is 92.8 Gflops/sec. The peak memory performance is 34.1 Gbytes/sec, because LPDDR3-2133 can handle 8-bytes data with 2,133 MHz of memory clock frequency and two channels.

Table 1 shows the number of floating-point operations, memory references and operational intensity [10] which is floating-point operations per memory references. By comparing the ratio of the computational performance for the memory performance and operational intensity, the bottleneck of the application can be predicted [10]. When operational intensity is higher than 0.17 flops/bytes (5.8/34.1) for single core processing, the execution time is bounded on the time for computation. When lower than 0.17, the execution time is bounded on the time for data transfer. In processing of a single core without parallelization, the problem of DD arithmetic is not the number of memory references, but the number of floating-point operations, as shown in Table 1.

Table 1. The number of floating-point operations [flops], the number of memory references [bytes], and operational intensity [flops/bytes] for DD arithmetic. Let $\alpha \in \mathbb{R}$, $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{z} \in \mathbb{R}^N$, A, B, $C \in \mathbb{R}^{N \times N}$.

	floating-point operations	memory references	operational intensity
$\boldsymbol{y} = \alpha \boldsymbol{x}$	$7N$	$2N \times 2 \times 8$	$7/32 \simeq 0.22$
$\boldsymbol{z} = \boldsymbol{x} + \boldsymbol{y}$	$11N$	$3N \times 2 \times 8$	$11/48 \simeq 0.23$
$\alpha = \boldsymbol{x}^T \boldsymbol{y}$	$18N$	$2N \times 2 \times 8$	$18/32 \simeq 0.56$
$\boldsymbol{y} = A\boldsymbol{x}$	$18N^2$	$(N^2 + 2N) \times 2 \times 8$	$18/16 \simeq 1.13$
$C = AB$	$18N^3$	$3N^2 \times 2 \times 8$	$O(N)$

4. Experiment for Matrix and Vector Operations in DD arithmetic TODO. グラフ・表に対応した説明に直す

4.1. DD Vector Operations

Vector operations are single loop processing. When we compute the inner product with AVX2, we must sum up the four SIMD register elements after the loop, three scalar additions are needed. In the case of using OpenMP, since we must sum up the p thread elements after the loop, $p - 1$ scalar additions are needed. When using both AVX2 and OpenMP, each thread processes a vector of length N/p using AVX2.

For vector of order $N = 4,096,000$, the performance of $\alpha = \boldsymbol{x}^T \boldsymbol{y}$ is 2.27 Gflops/sec for serial computing, 7.73 Gflops/sec for AVX2, and 8.20 Gflops/sec for four threads OpenMP. When using AVX2 and OpenMP, the performance is 14.08 Gflops/sec. The performance of $\alpha = \boldsymbol{x}^T \boldsymbol{x}$ is 27 Gflops/sec, which having half of memory references. When using AVX2 and OpenMP, the data supply cannot catch up. Since both inner products are bounded on the computational performance, they have the same tendency of the performance as 2.32 Gflops/sec for serial computing, 8.68 Gflops/sec for AVX2, and 8.42 Gflops/sec for OpenMP in Figure 2. For memory bounded operations, the upper bound of performance can be predicted by the roofline model [10]. The upper bound is the product of memory performance and operational intensity in Table 1, i.e. the performance levels is 74% (14.08/(34.1*0.56)).
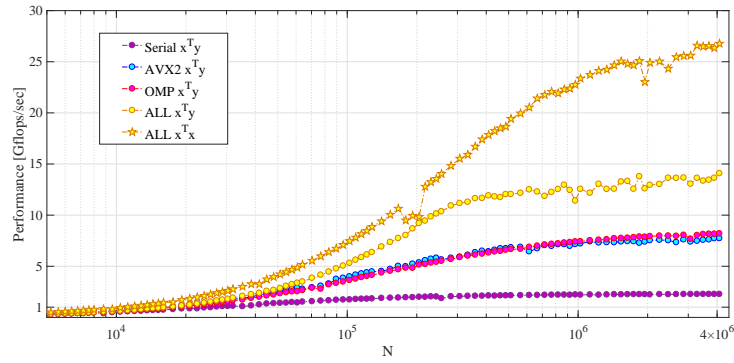
Figure 2. Performances [Gflops/sec] of inner products $\boldsymbol{x}^T \boldsymbol{y}$ and $\boldsymbol{x}^T \boldsymbol{x}$.

For scalar multiplication of vector, vector addition, and axpy, they require to allocate new memory address for output vector before calling outer C function. This process would become degrade the performance level because roofline model do not consider it.
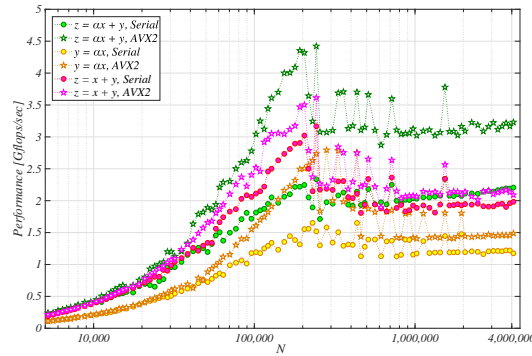
Figure 3. 図は資料 20190930_matome の p13 から p16 の図で AVX2+OpenMP のケースを追加したもの (y=x+y, z = x+y, y=ax+y, z=ax+y, (y = ax, x = ax も追加する) を比較) を載せる、軸も変える

Even OpenMP is used, the situation is the same. The operational intensity of axpy is 0.38 and this value is higher than that of scalar multiplication of vector and vector addition. When using AVX2, we can estimate that the upper bound of performance of axpy is higher. For the vector of order $N = 4{,}096{,}000$, the execution time of axpy is 0.033 sec when serial computing. This time is longer than 0.023 sec for that of the vector addition. However, when we use AVX2, the execution time of axpy is 0.023 sec and that of vector addition is 0.022 sec, as almost the same. Since the workload of memory reference is same for these two operations, the time for computation is reduced by AVX2.

4.2. DD Matrix-Vector Multiplication

$y_i = \sum_i \sum_j a_{ij} x_j$, and $y_i = \sum_j \sum_i a_{ij} x_j$ indicate the matrix-vector multiplication. Matrix-vector multiplication can be implemented in two types of algorithms depending on the order of index i and j. We call i-j type as PDOT, and j-i type as PB. These algorithms are differ in the memory access. PB is the unit stride access in MATLAB. We use letter P to represent the shape of Panel, and letter B to represent the shape of Block [11]. Hereafter, the loops in the program is called first, second, and third from the innermost.

4.2.1. AVX2

Since matrix-vector multiplication has nested loops, we should consider the way to use AVX2. The four ways of using AVX2 are shown in Figure 4. Here, the letter v means vector instruction and letter s means scalar instruction. Blue words are the instructions that can use vector instructions. Red words are the instructions that cannot use vector instructions or overheads. (青文字・赤文字の説明は入れた方が良い？冗長な気も。。) TODO.Figure4 を受けての説明を足す (AVX2 の load が使える、使えない)

```
for (i = 0; i < n; i += 4)          PDOT_PB
    vy = setzero( )
    for(j = 0; j < n; j ++)
        va = vload(a(i, j))
        vx = broadcast(x(j))
        vy = vmuladd(vy, va, vx)
    vstore(y(i), vy)
```

```
for (j = 0; j < n; j ++)            PB_PB
    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)
```

```
for (i = 0; i < n; i ++)            PDOT_PDOT
    vy = setzero( )
    for(j = 0; j < n; j += 4)
        va = set(sload(a(i, j)),
                 ..., sload(a(i, j + 3)))
        vx = vload(x(j))
        vy = vmuladd(vy, va, vx)
    sy = sum(vy)
    sstore(y(i), sy)
```

```
for (j = 0; j < n; j += 4)          PB_PDOT
    vx = vload(x(j))
    for(i = 0; i < n; i ++)
        va = set(sload(a(i, j)),
                 ..., sload(a(i, j + 3)))
        vy = vmul(va, vx)
        sy = sum(vy)
        sy = sadd(y(i), sy)
        sstore(y(i), sy)
```

Figure 4. Algorithm for using AVX2 to PDOT or PB for $\boldsymbol{y} = A\boldsymbol{x}$.

For using AVX2 for the column order (PDOT_PB, PB_PB), we can use AVX2 loading instruction to load four continuous data of matrix. For using AVX2 for row order (PDOT_PDOT, PB_PDOT), we should use four times of scalar loading instruction to load four
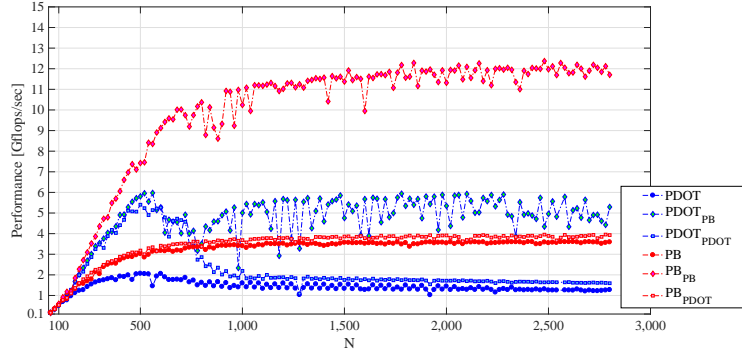
Figure 5. Performances [Gflops/sec] using AVX2 to PDOT and PB for $\mathbf{y} = A\mathbf{x}$.

discontinous data of matrix. Furthermore, $PDOT_{PDOT}$ and PB_{PDOT} require overhead for summing up, and they cannot use AVX2 storing instruction.

TODO. 冗長な説明を直す (パワポの p20 の説明に直す) $PDOT_{PB}$ and PB_{PB} that can use AVX2 loading and storing instructions are higher in the performance than others. The performance is increased almost four times by using AVX2 for $PDOT_{PB}$ and PB_{PB}. We then use OpenMP for these two algorithms.

4.2.2. *AVX2 + OpenMP TODO. Inner loop or First loop*

TODO. OpenMP のみのコメントを追加 (外側のループに OpenMP を使うこと) For using OpenMP, there are two ways depending on which loop (outer or inner) to parallelize. For PDOT with $N = 2{,}500$, the performance for outer loop parallelism is 10.76 Gflops/sec and that for inner loop parallelism is 4.55 Gflops/sec. For PB with $N = 2{,}500$, the performance for outer loop parallelism is 13.07 Gflops/sec and that for inner loop parallelism is 4.79 Gflops/sec. These result shows that parallelizing the outer loop is better to use OpenMP for this problem. The Figure 6 shows that the combinations of using AVX2 to load four data of matrix continuously and using OpenMP to parallelize the outer loop.

PB_{PB} with OpenMP is the best in performance, so we decided to use this algorithm for MuPAT. Since the operational intensity of matrix-vector multiplication is 1.13 and it is lower than 2.7 (=92.8/34.1), matrix-vector multiplication would become being bounded on memory when using AVX2 and OpenMP. Even using AVX2 and OpenMP, the execution time cannot be reduced 16 times (AVX2 x 4-core) from the case of serial due to memory references. The upper bound of performance can predict by the product of memory performance and operational intensity, i.e. the performance levels when $N = 2{,}500$ is 67% (25.63/(34.1*1.13).

4.3. *DD Matrix-Matrix Multiplication*

TODO. ループのインデックスの順番、どのループを OpenMP で並列化するか。について説明を追加 (パワポ p24) The matrix multiplication ($y_{ij} = \sum_{ijk} a_{jk}x_{kj}$) has six types of algorithms depending on the order of index *j-k-i*.
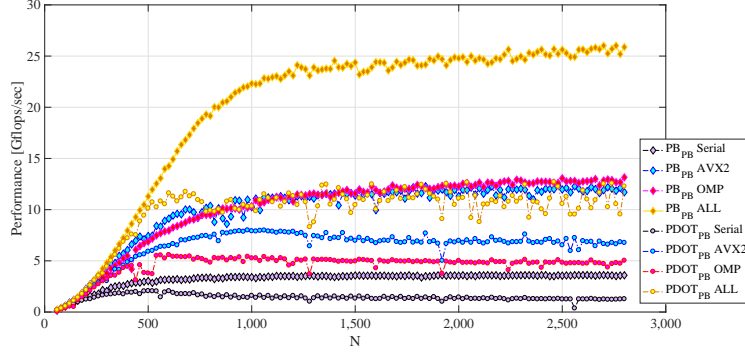
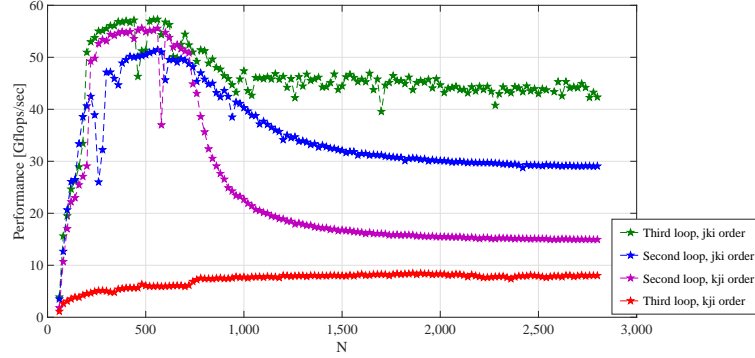**Figure 6.** Performances [Gflops/sec] using OpenMP, and using AVX2 to PDOT and PB for $y = Ax$.

Table 2. The performance of matrix multiplication when N is 2,500 depending on the order of index (Serial).

Index of Outer loop	i	i	j	j	k	k
Index of Inner loop	k	j	i	k	j	i
Index of Inner most loop	j	k	k	i	i	j
Performance [Gflops/sec]	0.80	0.81	0.89	3.62	3.52	0.83

When the index of inner most loop is i, the algorithm become unit stride access for MATLAB and result in higher performance, that is 3.6 Gflops/sec. When using AVX2 to load four data of matrix continuously, the performance for matrix multiplication of j-k-i order and that of k-j-i order are almost the same. When $N = 2,500$, the formar is 12.00 Gflops/sec and the latter is 12.27 Gflops/sec. For using OpenMP, the performance for matrix multiplication with $N = 2,500$ of j-k-i order with the outer loop parallelism is 12.26 Gflops/sec and that of the inner loop parallelism is 11.93 Gflops/sec. As for the k-j-i order, the performance of the outer loop parallelism is 6.87 Gflops/sec and the inner loop parallelsm is 11.78 Gflops/sec. The only case of using OpenMP to the outer loop for k-j-i order is low performance, and the others are not differ. However, when using AVX2 and OpenMP, the performance of j-k-i order with outer loop parallelism is the highest as in Figure 7.

TODO. 一番性能が高いケースの行列積の傾向を表を使って説明し直す (パワポ p29) For the case that matrix of order $N = 2,500$, the performance of PB$_{PB}$ is 42.98 Gflops/sec. For the order of matrix of order $N = 560$, the performance is peak, i.e. 57.31 Gflops/sec and the performance levels of 61.76%. For the serial computing, the performance is 3.56 Gflops/sec. The performance is from 40 Gflops/sec to 60 Gflops/sec when parallelizing third loop. However, when parallelizing second loop, the performance is from 30 Gflops/sec to 50 Gflops/sec.

Since, the operational intensity of matrix multiplication is too large as $O(N)$, the execution time is bounded on time for computation. For parallelizing third loop, the performance improve almost theoretically 16 times by utilizing both AVX2 and OpenMP. For other operations such as vector operations and matrix-vector operations, the performances are not increased theoretically, because their operational intensity is lower than 2.7 (=92.8/34.1).

Figure 7. Performances [Gflops/sec] using both AVX2 and OpenMP (parallelizing Third loop and Second loop) for $Y = AX$.

5. IEEE-style TODO. 主張をまとめ, 別紙で取捨の確認 (パワポ **p46, p47** あたり)

5.1. Summary TODO. 文章の修正

Before offloading to outer C function, it is required to allocate memory address in MAT-LAB. This overhead degrades some vector operations such as vector addition, scalar multiplication of vector, and axpy, which require output of vector. This overhead cannot consider in the roof line model, so the performance levels are low. Since the inner product is not require output of vector, its performance level is high when using the roof line model. However, the performances for vector operations are low , because their operational intensities are low, and being bounded on memory references.

For DD matrix-vector operations that have nested loops, the way to use AVX2 and OpenMP should be considered. In order to achieve high performance by AVX2, it is necessary to avoid synchronization and using AVX2 loading instructions, not the set of scalar loading instructions. Using AVX2 in this way, the number of floating-point operations can be reduced almost 1/4, and the performance can be increased. Since the operational intensity is high as twice as inner product, the performance is increased to almost the upper bound of 85% when using both AVX2 and OpenMP.

For DD matrix multiplication, the performance degraded after the peak, using both AVX2 and OpenMP are not enough to optimize. However, when using AVX2 in the effective way, the performance increases almost theoretically and the execution time is reduced, because its operational intensity is so high.

6. Conclusion TODO. 文書の修正

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 arithmetic that requires large number of floating-point operations. Executing DD arithmetic takes much time for computation. It is possible to offload to outer C function by MATLAB executable file and use FMA, AVX2, OpenMP supported by modern CPU.

8

Since FMA can reduce the number of floating-point operations, we assume to use FMA. For matrix-vector multiplication and matrix multiplication, there is some way to use AVX2 and OpenMP. For using AVX2, when loading, computing, and storing is executed with four pack of double precision numbers, the performance is increased. To execute AVX2 loading, computing, and storing instructions, we should avoid synchronization and avoid using four scalar loading instruction. For using OpenMP, we should avoid synchronization and avoid disturbing unit stride memory references. DD matrix-vector multiplcation and matrix multiplication can be implemented in these way.

The innermost structures of inner products, matrix-vector multiplication, and matrix multiplication are the one multiply-and-add. MuPAT on Scilab is implemented these three operations into one as matrix multiplication routine. Since the way of implementation to increase the performance for these three operations are different, MuPAT on MATLAB is implemented as three different routines. There is the trade-off between high performance program and general purpose program.

The effectiveness of AVX2 and OpenMP is depending on the operational intensity. Although the performance become almost the upper bound, it does not mean that the execution time is reduced theoretically. Reducing the number of memory references and increasing the number of floating-point operations let the operational intensity become high. For DD arithmetic, using large routines such as axpy and using IEEE-style can increase the operational intensity. As a result, the performance can be increased with almost the same execution time before increasing the operational intensity.

Even the DD arithmetic, matrix and vector operations are bounded on the memory performance except for matrix multiplication. Recent CPUs can hide the computational workload of these operations in DD arithmetic by using FMA, AVX2, and OpenMP. The next problem for some matrix and vector operations in DD arithmetic is the number of memory references.

Acknowledgment

References

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

[2] 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.

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

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

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

[6] L. Dagum, and R. Menon, OpenMP: An Industry-Standard API for Shared-Memory Programming, *IEEE Computational Science & Engineering* **5** (1998), 46-55.

[7] I. Yamazaki, S. Tomov, T. Dong, J. Dongarra, Mixed-precision orthogonalization scheme and adaptive step size for improving the stability and performance of CA-GMRES on GPUs, in: Proceedings of International Meeting on High Performance Computing for Computational Science (VECPAR), 2014, 17-30.

[8] X. S. Li et al., Design, Implementation and Testing of Extended and Mixed Precision BLAS, *ACM Transactions on Mathematical Software* **28** (2002), 152-205.

[9] K. Dowd and C. Severance, High Performance Computing, 2nd Edition, *O'Reilly*, 1998.

[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.

[11] K. Goto, and R. van de Geijn, Anatomy of High-Performance Matrix Multiplication, *ACM Transactions on Mathematical Software* **34** (2008), Article 12 (25 pages).