More Accurate Computation for Double-Double Arithmetic without Additional Execution Time by Parallel Processing

URL of Abstract

Hotaka Yagi* Emiko Ishiwata* Hidehiko Hasegawa†

*Tokyo University of Science, Japan †University of Tsukuba, Japan

1. Introduction

- **To reduce rounding errors** in floating-point arithmetic, the use of **high-precision arithmetic is effective.**
- **Our team developed *MuPAT***, an open-source interactive ***Multiple Precision Arithmetic Toolbox*** [1] for MATLAB and Scilab.
- ***MuPAT* uses the DD (Double-Double) algorithm** [2], which is based on a combination of double-precision arithmetic operations and **enables quasi quadruple-precision arithmetic.**
- **We accelerate DD vector and matrix operations by using AVX2 and OpenMP,** and achieve higher performance for heavier DD operations.
- **We found that some DD operations can be computed more accurately without additional execution time** in parallel processing environment.

2. DD Arithmetic

A DD number a is represented by a combination of two double-precision numbers a_{hi} and a_{lo},

$$a = a_{hi} + a_{lo} \qquad |a_{lo}| \leq \frac{1}{2} ulp(a_{hi}).$$

DD	s	$e_1 \dots e_{11}$	$m_1 \dots m_{52}$	s	$e_1 \dots e_{11}$	$m_1 \dots m_{52}$	
IEEE 754 Quadruple	s	$e_1 \dots e_{15}$	$m_1 \dots m_{112}$				

There are two implementations of DD addition, called **Cray-style** and **IEEE-style** [2].

	Cray-style	**IEEE-style**
# double-precision operations	cheap 11	heavy 20
Error bound	normal $DDadd\,(a,b) = (1+\delta_1)a + (1+\delta_2)b$ with $\lvert\delta_1\rvert, \lvert\delta_2\rvert \leq \epsilon_{dd}$	accurate $DDadd\,(a,b) = (1+\delta)(a+b)$ with $\lvert\delta\rvert \leq 2\epsilon_{dd}$, $\epsilon_{dd} = 2^{-105}$
Algorithm	1 $s = a_{hi} \oplus b_{hi}$ 2 $v = s \ominus a_{hi}$ 3 $eh = a_{hi} \ominus (s \ominus v)$ 4 $eh = eh \oplus (b_{hi} \ominus v)$ 5 $eh = eh \oplus (a_{lo} \oplus b_{lo})$ 6 $c_{hi} = s \oplus eh$ 7 $c_{lo} = eh \ominus (c_{hi} \ominus s)$	1 $s = a_{hi} \oplus b_{hi}$ 2 $v = s \ominus a_{hi}$ 3 $eh = a_{hi} \ominus (s \ominus v)$ 4 $eh = eh \oplus (b_{hi} \ominus v)$ 5 $t = a_{lo} \oplus b_{lo}$ 6 $v = t \ominus a_{lo}$ 7 $el = a_{lo} \ominus (t \ominus v)$ 8 $el = el \oplus (b_{lo} \ominus v)$ 9 $eh = eh \oplus t$ 10 $t = s \oplus eh$ 11 $eh = eh \ominus (t \ominus s)$ 12 $el = el \oplus eh$ 13 $c_{hi} = t \oplus el$ 14 $c_{lo} = el \ominus (c_{hi} \ominus t)$

more accurate, but not widely used, due to computation cost [3].

computational order cannot change !

3. Parallelization by AVX2 and OpenMP

- AVX2 [4] instructions can process four double-precision data in one unit of time.
- OpenMP [5] allows thread-level parallelism on shared memory for a multicore environment.

Algorithm of $y = Ax$

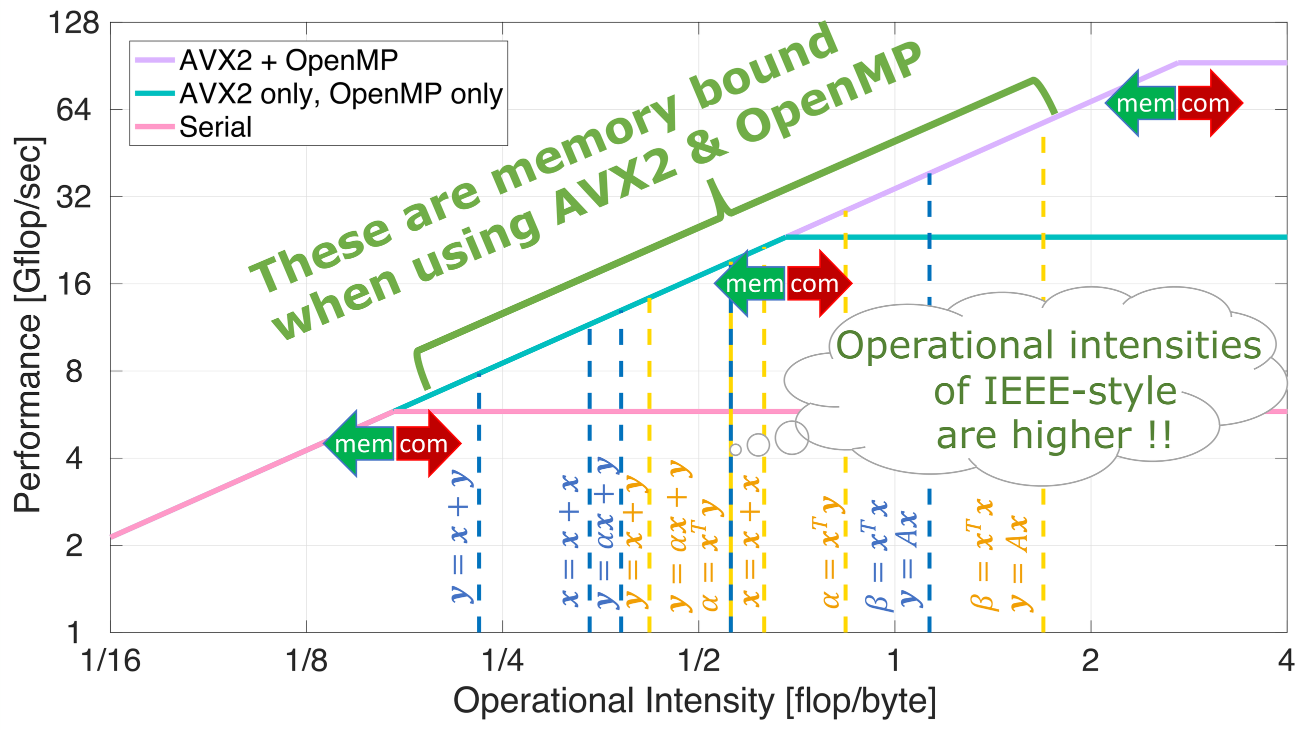
1. #pragma omp for
2. $for\ (j = 0; j < n; j++)$
3. $\quad for(i = 0; i < n; i += 4)$
4. $\quad\quad y(i) = DDadd(y(i), DDmul(a(i,j), x(j)))$

Since we use MATLAB, **column major order is unit stride access.** The order of loop should be ***j-i***.

- Unit stride access is key to use AVX2 load/store instructions. (The overhead is required for non unit stride access.)
  - We apply AVX2 for inner loop as in line 3.
- Parallelizing outer loop by OpenMP can offer much larger workload for each thread.
  - We apply OpenMP for outer loop as in line 1, 2.
- We implement two kinds of DD addition: Cray-style and IEEE-style in line 4.

4. Roofline Model Analysis [6]

Roofline is a **visual performance model** that sets **upper bound of performance** depending on **operational intensity** and hardware.



- **Operational intensity hits** the **diagonal line**: the operation is **memory bound** mem
- **Operational intensity hits** the **horizonal line**: the operation is **compute bound** com

Environment

CPU: Intel Core i7 7820HQ, 2.9 GHz processor
Memory: LPDDR-2133

Operational intensity [flops/bytes]	# double precision floating-point operations [flops] / # memory references [bytes]
Upper bound of performance [Gflops/sec]	min(computational performance, memory performance × operational intensity)

Performance [flops/sec]	# double precision floating-point operations [flops] / execution time[sec]
Computational performance [flops/sec]	clock frequency of CPU [Hz] × # flops can be computed in one unit of time [flops/cycles]
Memory performance [bytes/sec]	clock frequency of memory [Hz] × # channels × 8 [bytes/cycles]

5. Comparing Two Implementations between Cray-style and IEEE-style

Before acceleration (compute bound)

- Exec. times for all operations depend on **floating-point operations.**
- Exec. time for **IEEE-style** takes **1.5 times** than that for **Cray-style**.
- Performances do not depend on operational intensity.

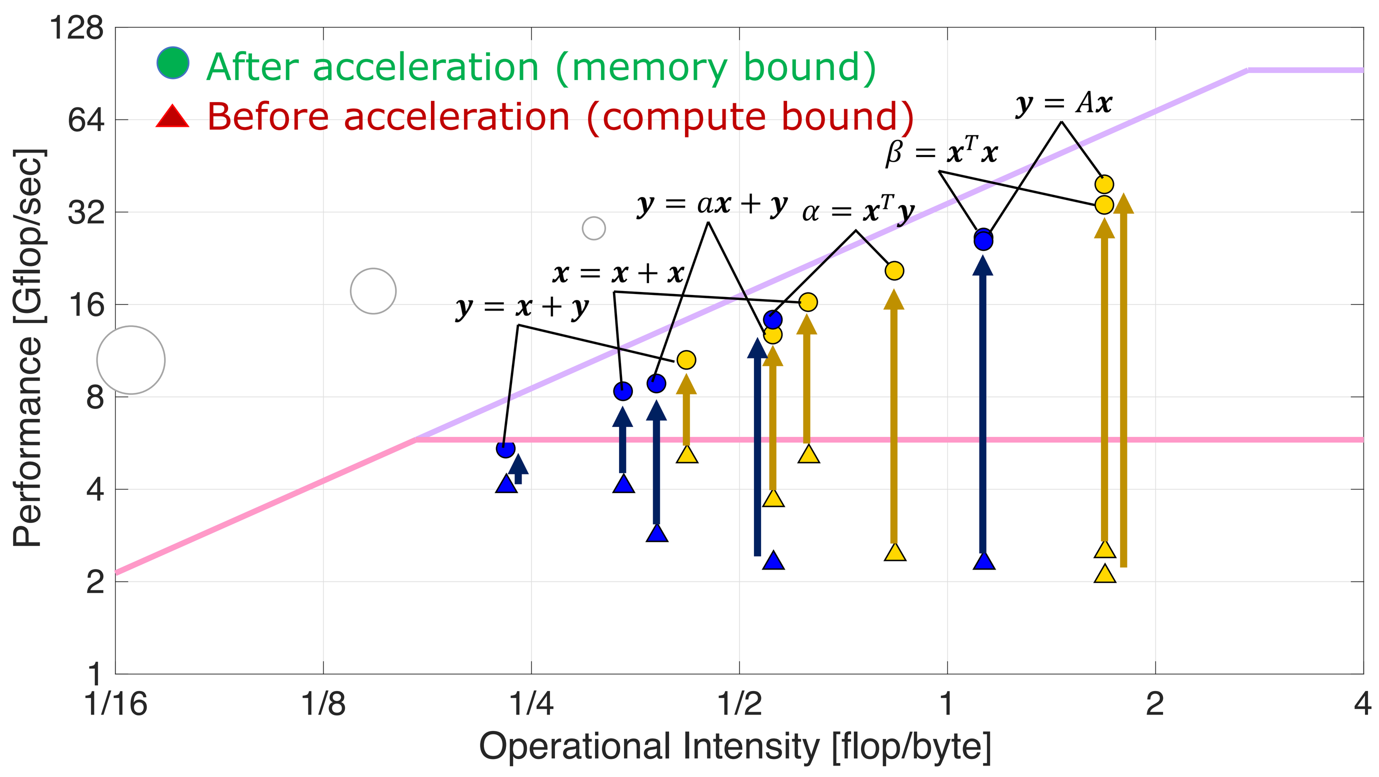
N = 4,092,000 for vector operations, N = 2,500 for $y = Ax$.

Cray-style / IEEE-style	Floating-Point Operations [flops]	Memory References [bytes]	Operational Intensity [flops/bytes]	Execution Time Serial / Accelerated [msec]	Speed-up	Performance Measured / Upper Bound [Gflops/sec]	Ratio of Performance to Upper Bound [%]
$y = x + y$	$11N$	$3N{\times}16$	0.23	11 / 8.3	1.3	5.4 / 7.8	69.5
$x = x + x$	$11N$	$2N{\times}16$	0.34	11 / 5.4	2.0	8.3 / 11.7	71.2
$y = \alpha x + y$	$18N$	$3N{\times}16$	0.38	26 / 8.4	3.1	8.8 / 12.8	68.6
$\alpha = x^T y$	$18N$	$2N{\times}16$	0.56	32 / 5.2	6.2	14.2 / 19.1	74.3
$\beta = x^T x$	$18N$	$N{\times}16$	1.13	32 / 2.8	11.4	26.3 / 38.5	68.3
$y = Ax$	$18N^2$	$(N^2{+}2N)\times16$	1.13	32 / 4.4	7.3	25.6 / 38.5	66.4
$y = x + y$	$20N$	$3N{\times}16$	0.42	16 / 7.8	2.1	10.5 / 14.2	73.9
$x = x + x$	$20N$	$2N{\times}16$	0.63	16 / 5.1	3.1	16.2 / 21.3	75.9
$y = \alpha x + y$	$27N$	$3N{\times}16$	0.56	30 / 8.7	3.4	12.7 / 19.2	66.3
$\alpha = x^T y$	$27N$	$2N{\times}16$	0.84	45 / 5.0	9.0	20.5 / 28.8	71.2
$\beta = x^T x$	$27N$	$N{\times}16$	1.69	44 / 3.3	13.3	33.6 / 57.5	58.4
$y = Ax$	$27N^2$	$(N^2{+}2N)\times16$	1.69	53 / 4.3	12.3	39.2 / 57.5	68.2

heavy same half same long half same long heavy same

After acceleration (memory bound)

- Exec. times for all operations depend on **memory references.**
- Exec. time for **IEEE-style** is almost the **same** with that for **Cray-style**.
- Performances are depending on operational intensity.



IEEE-style is much accelerated depending on **operational intensity.**

You can use DD addition in Cray-style and IEEE-style with parallelization in ***MuPAT*** on MATLAB.

Accelerated DD operations can be use in multi-core environment.

The detail for ***MuPAT*** is written in our web page !

URL of MuPAT

References

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

[2] Y. Hida, X. S. Li, and D. H. Bailey. 2000. Quad-Double Arithmetic: Algorithms, Implementation, and Application. Technical Report LBNL-46996.

[3] X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, A. Kapur, M. C. Martin, B. J. Thompson, T. Tung, and D. J. Yoo. 2002. Design, implementation and testing of extended and mixed precision BLAS. ACM Trans. Math. Softw. 28, 2 (June 2002), 152–205.

[4] Intel. 2019. *Intel Intrinsics Guides*. Retrieved November 11, 2019 from https://software.intel.com/sites/landingpage/IntrinsicsGuide/

[5] L. Dagum and R. Menon. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. IEEE Comput. Sci. Eng. 5, 1 (January 1998), 46-55.

[6] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. Commun. ACM 52, 4 (April 2009), 65–76.