Performance Analysis of BigDecimal Arithmetic Operation in Java

Jos Timanta Tarigan*, Elviawaty M. Zamzami & Cindy Laurent Ginting

Faculty of Computer Science and Information Technology, Universitas Sumatera Utara,
Jalan Universitas No.9, Padang Bulan, Medan Baru, Kota Medan 20222,
Sumatera Utara, Indonesia
*E-mail: jostarigan@usu.ac.id

Abstract. The Java programming language provides binary floating-point primitive data types such as float and double to represent decimal numbers. However, these data types cannot represent decimal numbers with complete accuracy, which may cause precision errors while performing calculations. To achieve better precision, Java provides the BigDecimal class. Unlike float and double, which use approximation, this class is able to represent the exact value of a decimal number. However, it comes with a drawback: BigDecimal is treated as an object and requires additional CPU and memory usage to operate with. In this paper, statistical data are presented of performance impact on using BigDecimal compared to the double data type. As test cases, common mathematical processes were used, such as calculating mean value, sorting, and multiplying matrices.

Keywords: *BigDecimal arithmetic operation; floating-point arithmetic; numerical programming; optimization; programming language*.

1 Introduction

Due to the nature of computer processors, providing data as a sequence of binary digits to represent a 10-base decimal number requires a conversion process that follows a certain standard. Most programming languages use an approximation approach to represent decimal numbers. This method may produce precision problems, depending on the complexity of the calculation process and the length of the binary digits [1].

A commonly used approach for binary floating point representation is IEEE 754-1985 [2], which has been succeeded by IEEE 754-2008 [3]. This standard defines four different precisions for binary floating points: single, double, single extended, and double extended. The single and double precision use 32-bit and 64-bit binary digits to represent decimal numbers. The Java programming language uses this standard to represent decimal numbers. Java is one of the most widely used programming languages in computer science. Although it has consistently been the most popular programming language in recent years [4],

Received March 1st, 2018, Revised December 20th, 2018, Accepted for publication December 26th, 2018.
 ISSN: 2337-5787, DOI: 10.5614/itbj.ict.res.appl.2018.12.3.5

Java's popularity in numerical/scientific programming is not equal to that of its counterparts such as C++, Matlab, or Fortran. There are a few drawbacks that restrain Java as a scientific computing programming platform, as shown by Hale, *et al.* [5]. However, as Bull, *et al.* has pointed out that the gap between Java and C++/Fortran has become smaller over the years, making it a viable platform for scientific computing [6]. Moreover, *et al.* [6] have proposed an optimization that may increase numerical computation in Java, making it a compatible platform for technical computing. The research concluded that while Java has not reached the level of performance of C++ or Fortran, its features provide a huge advantage for programmers, which overshadows its deficiencies.

In scientific computation, numerical precision is important. However, detecting numerical errors such as unexpected rounding results during a lengthy computational process is not a trivial task. IEEE 754 binary floating point, which has a limited precision, is prone to this type of error. Fortunately, Java provides BigDecimal, a built-in class available on its standard platform. This class is dedicated to provide an exact representation of decimal numbers and gives full control to the user over how to define decimal digits (both integers and fractional parts) and how to perform rounding during calculation. These properties make BigDecimal a suiltable option to perform computations that require accurate precision, such as scientific and technical computation processes. However, the advantages of using BigDecimal come at a cost. Performing calculation using BigDecimal requires more CPU and memory usage. This issue has motivated us to investigate the performance difference between using primitive data type double and BigDecimal in common mathematical computation processes. The objective of this research was to measure the performance of Java's BigDecimal in common mathematical operations. We used three mathematical formula as our test cases: mean calculation, sort, and matrix multiplication. Two metrics were collected during the tests: running time and precision.

2 Related Works

Cowlishaw [7] has shown how decimal representation using binary floating point may cause correctness issues. One of the examples given is the problem of calculating a 5% sales tax on a $0.70 telephone call. Using manual calculation, we can easily figure out that total cost would be 0.735. However, a double precision binary floating point calculation would give us 0.7349999999999999 (which could be rounded to $0.74).

Researches on binary floating point mostly fall into two categories: correctness and performance. Kamble, *et al.* pinpointed a trend in floating point computer arithmetic researches [8]. Erle, *et al.* offers a hardware specification that may

increase the performance of floating point-based calculation while maintaining its precision [9]. They proposed hardware that supports decimal floating point calculation natively, which eliminates the need to convert decimals into binary format. Beauchamp, *et al.* proposed three architectural modifications of field programmable gate arrays (FPGA) that may increase the efficiency of floating point operations. Some researches focused on modifying FPGA to perform a specific floating point-based operation such as Gaussian filtering [10], matrix multiplication [11], and image-classifier neural network training [12].

Researches focusing on precision have also been done previously. Joldes, *et al.* introduced arithmetic algorithms using floating points expansion, yielding better precision. The work offers improvement in performing normalization, division, and square root [13]. Similar work has also been done by Muller, *et al.*, who focused their research on a multiplication operation algorithm using floating-point expansions [14]. Ruibo-Gonzalez, *et al.* proposed Precimonious [15], a program that is able to assist developers to tune the precision of floating points. It allows developers to define the desired precision and set floating point variables to satisfy constraints. It also allows developers to increase the performance of the program by allowing the use of less precise floating point variables. Rubio-Gonzalez proposed a method to automate precision analysis of a floating point operand [16] called Blame Analysis. This method, combined with Precimonious, allows programmers to have automated data precision and control in application code. Hull, *et al.* proposed an architecture design for a coprocessor that is able to perform variable precision floating point arithmetic. Ho, *et al.* created a similar program, which allows developers to perform floating point precision tuning efficiently [17]. A research from NASA by Goodloe, *et al.* focused on verifying the correctness of numerical programs [18]. Chiang, *et al.* developed a heuristic search algorithm called Binary Guided Random Testing [19]. The program evaluates floating point routines and finds input that maximizes the error. By using the evaluation results, a programmer can allocate resources to a certain routine that requires more precision.

3 Binary Based Floating Point

In Java, there are two primitive data types to represent decimal numbers: float and double. Both of these data types use IEEE Standard 754 Floating Point Numbers. This binary based floating point uses base and exponent digits. For example, the number 145.231 could be represented in its normalized form 1.45231 x 102. IEEE 754 Floating Point Numbers has three basic components: the sign, the exponent, and the mantissa. These fields are filled as follows:

1. The sign uses 1-bit binary. The value 0 represents a negative number and 1 represents positive numbers.

2. The exponent is a base-2 exponent with bias value to split the positive and negative bias value. In 32-bit floating point, the exponent is represented by 8-bit binary, which has 127 as bias value. In 64-bit floating point, the exponent is represented with an 11-bit exponent, which has 1023 as bias value.
3. The mantissa represents the fractional bits of the number in normalized form. A 32-bit floating point has a 23-digit mantissa while a 64-bit floating point has 52 digits. Since the normalized form does not allow 0 as an integer part of a number, the mantissa always has one hidden bit with value 1.

The structure of a binary floating point is visualized in Figure 1.

Figure 1 Structure of 32-bit and 64-bit floating point.

The conversion from decimal to binary floating point can be seen in this example: given a number of 5.5, the representation in 32-bit floating point is as follows:

1. The sign would be 1 since it is a positive number.
2. The exponent would be the largest number smaller than 5.5, which is 4 (or 22). This number gives us an exponent value of 129.
3. The mantissa would be 1.375, which is the result of 5.5/4. Since the integer part has already been defined in the rule, we only need to represent 0.375 in binary form, which is 011 0000 0000 0000 0000 0000.

Hence, the 32-bit floating point of 5.5 is 1 1000 0001 011 0000 0000 0000 0000 0000.

Another way to represent decimal numbers in Java is to use the BigDecimal class. This class is dedicated to represent signed decimal numbers with unlimited precision. Unlike float and double, which use approximation with a certain precision, BigDecimal uses the exact value of the number and has arbitrary precision. BigDecimal also gives users control over the rounding and precision used in the calculation. The data representation of BigDecimal consists of a BigInteger to represent the unscaled value and an integer to represent the scale of the value. The scale value will then define the radix point that separates the integer part of the number and its fractional part. Given a number 313.0123, the unscaled value is 3130123 and the scale value is 4, noting

that there are 4 numbers to represent the frictional part. BigInteger itself is a class of Java that represents a virtually unlimited integer value. This class uses an array of integers to create large integer values. Theoretically, since an integer value is represented by a 32-bit value and the maximum index of an integer indexed array is 232, the maximum value of BigInteger is $232^{\text{Integer.MAX\_VALUE}}$. However, as of Java 8 the value supported by BigInteger is in the range of $-2^{\text{Integer.MAX\_VALUE}}$ to $+2^{\text{Integer.MAX\_VALUE}}$. Hardware-wise, this range of values may take up 2 GB of memory space, depending on the number represented.

The current version of Java's BigDecimal has been substantially improved to support many functionalities including math operations, multiple rounding modes, and formatting. However, it is important to note that due to its nature as a class, BigDecimal requires an additional process of creating a new object when performing an operation that returns a new number, such as addition, multiplication or exponentiation. BigDecimal also does not support the use of mathematical operators. Instead, we have to perform a method call, such as add(BigDecimal val1, BigDecimal val2) or mult(BigDecimal val1, BigDecimal val2).

4 Implementation

Our objective is to observe the performance behavior of using BigDecimal compared to double data type. The observation was focused on two data: running time and precision. We collected running time data by calling Java's built-in method System.nanoTime() at the beginning and the end of each process. As for data precision, we used an absolute error formula: given a value p and its approximation p^*, the absolute error $e = |p^* - p|$.

To conduct the test, a routine was created that generates random numbers based on a certain length of digits. These numbers were created by generating sequences of characters (strings) consisting of numbers. These strings were assigned to our BigDecimal objects using a built-in constructor that converts strings into decimal numbers. To test whether the length of the number affected the performance, we used two kinds of numbers in the test: 5 integer and fractional digits (denoted by (5, 5)) and 10 integer and fractional digits (denoted by (10, 10)). An identical set of double numbers was created by converting BigDecimal objects to primitive type double. While it is important to note that this data initialization may take a long amount of time during the test, this process was excluded from the data since our observations focused on operation performance.

Our test consisted of three cases: mean (average) value calculation, sorting, and matrix multiplication. For each data, the test was performed 10 times and the

values were averaged to justify the result. Without trying to manipulate the result, outlier data that could distort the result were filtered out. It was assumed that such events are rare and irrelevant since they may be caused by OS service interruption, Java Garbage Collection, or simply a 'bad' set of random numbers generated during the initialization.

The hardware specification and software environment used for the test were as follows:

1. Retina Macbook Pro (late 2013) with macOS High Sierra 10.13.3,
2. Intel Core i7 2.3 GHz Quad Core, 256 KB L2 cache (per code), 6 MB L3 cache,
3. Memory 16 GB 1600 MHz DDR3,
4. Java SE Runtime Environment 1.8.0.

It is important to note that modern CPUs have caches to store recently used/accessed data other than the memory. Moreover, modern CPUs and compilers have various optimization methods that may also affect the performance. While these features may in some cases affect performance significantly, it was assumed that the massive amount of data overwhelmed these optimizations.

4.1 Mean Calculation

The purpose of this test was to observe the performance of using BigDecimal to do addition. Mean calculation consists of a series of addition operations followed by one division operation. The formula used to calculate the mean value was as follows: given a set of number $S = \{s_1, s_2, s_3, s_4, \ldots, s_n\}$, the mean value $\bar{s}$ of set S is:

$$\bar{s} = \frac{\sum_{i=0}^{n} s_i}{n} \quad (1)$$

Based on this equation, we can conclude that finding the average of a set with n amount of numbers takes n amount of additions and one division. The test result was in Table 1. In the first test, a series of tests was performed with the amount of numbers from 10 to 10^7 (using 10^8 numbers exceeded the physical memory). As expected, the test showed that using BigDecimal gave a significant performance hit compared to double. The performance ratio, however, varied based on the amount of data. Interestingly, the performance ratio decreased for the data from (5, 5) until $n = 10^6$. At $n = 10^6$, using BigDecimal was only 2-3 times slower than using primitive type double. However, increasing the amount to 10^7 increased the ratio 10.98 times on average. The data from (10, 10) showed a much higher average ratio, up to 76.79 times at 10^2. It dropped to 24.75 at 10^2 but increased from 10^3 upward.

Table 1 First test of mean value calculation.

N	(5,5)			(10,10)		
	BigDecimal (ms)	**double (ms)**	**Ratio**	**BigDecimal (ms)**	**double (ms)**	**Ratio**
10	$6.4x10^{-2}$	$7.85x10^{-4}$	81.52	$1.11x10^{-2}$	$5.97x10^{-4}$	18.58
10^2	$1.034x10^{-1}$	$1.8x10^{-3}$	55.40	$1.79x10^{-1}$	$2.33x10^{-3}$	76.79
10^3	$9.48x10^{-1}$	$2.04x10^{-2}$	46.47	1.45	$2.24x10^{-2}$	60.41
10^4	2.90	$1.58x10^{-1}$	18.35	4.53	0.18	24.75
10^5	4.82	$3.5x10^{-1}$	13.714	11.68	0.23	50.78
10^6	16.02	5.72	2.80	319.01	5.67	56.26
10^7	169.22	10.986	10.98	642.91	10.50	61.22

A larger amount of numbers was also tested to see the consistency of the ratio behavior. The result was in Table 2.

Table 2 First test of mean value calculation.

N ($x10^7$)	(5,5)			(10,10)		
	BigDecimal (ms)	**double (ms)**	**Ratio**	**BigDecimal (ms)**	**double (ms)**	**Ratio**
1	169.22	10.986	10.98	642.91	10.50	61.22
1.5	194.93	13.95	13.97	897.05	18.18	49.34
2	216.48	19.90	10.87	1,564.91	21.83	71.68
2.5	267.95	28.46	9.41	1,882.82	25.84	72.86
3	322.82	27.14	11.89	2220.82	31.676	70.11
3.5	353.74	44.195	8.00	2629.99	44.19	59.51

The table above shows that using BigDecimal (5, 5) requires 20 times more computation time than using double on average. Using a larger BigDecimal (10, 10) may require more than 70 times extra computation time.

The precision error from the test was also collected. The data show a higher error in the (10, 10) tests with an average value of $2.04x10^{-4}$ compared to $8.6x10^{-6}$ for the (5, 5) tests. However, the data precision fluctuated and highly depended on the random numbers generated during the initialization.

4.2 Sorting

In the sorting algorithm, most of the operations consist of comparing and swapping. This method is effective to observe how BigDecimal may affect memory access operation. Our sorting algorithm was based on this rule: given a set of random number $S = \{s_1, s_2, s_3, s_4, \ldots, s_n\}$ as the input, the algorithm will

sort until all members of the set follow the rule $s_k < s_{k+1}$ where $0 < k < n$. In this test, we used selection sort, which follows the following steps:

1. Split the list into two lists: sorted and unsorted. Initially set the size of the sorted list to 0.
2. Find the smallest number in the unsorted list and put this number in the sorted list (usually performed by swapping it with the first number of the unsorted list).
3. Increase the size of the sorted list by one and repeat step two until the size of the sorted list is the same as that of the unsorted list (hence, no more numbers are left in the unsorted list).

We also used 2 sets of data for this test: sorted and reversely sorted. These two cases are known as the best and worst cases for selection sort. Based on the steps previously explained, sorting a sorted list with n amount of numbers requires n^2 compare operations and no swap operations. In contrast, sorting a reversely sorted list requires n^2 swap operations and n swap operations.

It is also important to note that the BigDecimal compare operation starts by truncating both numbers to integer-representable numbers, starting from the most significant digit (most left digit). If both numbers are different, the numbers is compared and returned. If both numbers are equal, more digits to the right are truncated and the process is repeated. Based on this algorithm, we can conclude that the performance of the compare operation in BigDecimal depends on the number.

In our test, random numbers were used without supervision; hence, there was a possibility, although unlikely, that a set of numbers would require additional steps during the comparison operation. In the first test, the best case of selection sort was used, of which the result is shown in Table 3. The second sorting test consisted of sorting the reversely sorted numbers. The result is shown in Table 4.

Table 3 Test result for sorting the best case.

N	(5,5)			(10,10)		
	BigDecimal (ms)	**double (ms)**	**Ratio**	**BigDecimal (ms)**	**double (ms)**	**Ratio**
10	1.08×10^{-2}	2.51×10^{-3}	4.302	4.72×10^{-2}	2.25×10^{-3}	20.97
10^2	3.87×10^{-1}	8.52×10^{-2}	4.54	5.28×10^{-1}	1.16×10^{-1}	4.55
10^3	4.19	1.64	2.55	7.66	1.58	4.84
10^4	387.58	30.37	12.76	906.07	27.19	33.32
10^5	25,725.57	2,636.58	9.75	45,297.85	2,802.05	16.16

Table 4 Test result for sorting the worst case.

N	(5,5)			(10,10)		
	BigDecimal (ms)	**double (ms)**	**Ratio**	**BigDecimal (ms)**	**double (ms)**	**Ratio**
10	$6.64\text{x}10^{-3}$	$2.58\text{x}10^{-3}$	2.57	$4.75\text{x}10^{-2}$	$2.70\text{x}10^{-3}$	17.59
10^2	$4.29\text{x}10^{-1}$	$9.78\text{x}10^{-2}$	4.38	$7.26\text{x}10^{-1}$	$1.16\text{x}10^{-1}$	6.28
10^3	4.47	1.67	2.67	9.93	1.91	5.19
10^4	393.44	27.50	14.30	1,082.44	26.49	40.86
10^5	31,035.92	3,672.31	8.45	52,949.53	7,707.19	6.87

As expected, the result showed a higher value compared to the best case since there was an additional swap operation during the sort process. The increment, however, was not significant since the swapping operation only swaps the reference of the object instead of the value. The running time ratio behavior was also similar to that in the previous test.

4.3 Matrix Multiplication

Our third test was to perform matrix multiplication. The process of matrix multiplication consists of multiplication and addition. Given an $n \times m$ matrix A and $m \times p$ matrix B in Eq.(2):

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{bmatrix}, B = \begin{bmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{m1} & \cdots & b_{mp} \end{bmatrix} \tag{2}$$

and the matrix product $C = AB$ is an $n \times p$ matrix in Eq. (3):

$$C = \begin{bmatrix} c_{11} & \cdots & c_{1p} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{np} \end{bmatrix} \tag{3}$$

then each entry of the matrix is defined with the following Eq.(4):

$$C_{ij} = \sum_{k=1}^{m} A_{ik} * B_{kj} \tag{4}$$

The test was performed using various amounts of numbers, but the interesting part was between 100 to 1000 numbers. The result of our test was in Table 5. The highest ratio was at 100 numbers, where it reached 115.63 for (5, 5) numbers and 207.05 for (10, 10) numbers.

We also observed the precision error data during the test and the result was interesting. Since the multiplication operation has the possibility of doubling the digits, the primitive data type double encountered a massive error when

representing large numbers in the (10, 10) test. The table below shows the precision error comparison between the (5, 5) tests and the (10, 10) test.

Table 5 Test result for matrix multiplication.

N (x100)	(5,5)			(10,10)		
	BigDecimal (ms)	**double (ms)**	**Ratio**	**BigDecimal (ms)**	**double (ms)**	**Ratio**
1	123.73	1.07	115.63	242.25	1.17	207.05
2	837.34	25.69	32.59	2,169.00	30.93	54.32
3	3,625.13	93.44	38.79	7,889.80	78.55	100.44
4	8,569.21	190.76	44.92	18,412.64	191.57	96.11
5	16,810.54	382.35	43.96	35,704.13	372.11	95.95
6	28,187.18	581.75	48.45	60,252.33	757.72	79.51
7	44,076.20	3,062.68	14.39	95,303.30	2,877.64	33.11
8	68,344.10	5,673.34	12.04	142,352.66	5,933.57	23.99
9	66,905.45	5,826.40	11.48	207,514.55	9,580.51	21.66
10	135,797.72	14,087.24	9.63	283,057.84	13,834.28	20.46

Table 6 Test result for sorting the worst case.

N (x100)	(5, 5)	(10, 10)
1	5.394×10^{-5}	555,204.92
2	1.491×10^{-4}	1,595,412.88
3	3.075×10^{-4}	2,898,569.24
4	4.172×10^{-4}	4,540,951.57
5	6.572×10^{-4}	6,641,028.56
6	8.702×10^{-4}	8,207,129.30
7	1.033×10^{-3}	9,567,372.45
8	1.179×10^{-3}	12,828,427.27
9	1.486×10^{-3}	16,106,312.21
10	1.896×10^{-3}	18,805,729.36

The data show that the errors were marginally small in the (5, 5) numbers. However, the errors were significant in the (10, 10) numbers due to its inability to represent very large numbers. It is important to note that the error in the (10, 10) numbers reached the integer digit of the original number.

5 Conclusion and Future Work

In this paper, statistical data on how BigDecimal affects performance were presented. Tests were conducted to compare the performance between primitive data type double and BigDecimal objects. As expected, BigDecimal required more CPU and memory usage and in some cases the difference was significant.

However, depending on the purpose of the software, this issue may be overshadowed by the ability to have a precise value. It is safe to conclude that Java's BigDecimal library is a feasible option to perform numerical/scientific programming that either uses large digit numbers or requires exact precision.

Acknowledgements

The author would like to thank Universitas Sumatera Utara as the author's current institution. Moreover, the author would like to thank the head and staff of 'Lembaga Penelitian USU' (Research Center of Universitas Sumatera Utara), Prof. Dr. Erman Munir, MSc, and staff for all their support during this research. The author would also like to thank the Dean of the Faculty of Computer Science and Information Technology, University of Sumatera Utara, Prof. Dr. Opim Salim Sitompul, M.Sc for his extensive support of this work.

References

[1] Burden, R.L., Faires, J.D. & Burden, A.M., *Numerical Analysis*, 10th edition, Boston, MA, United States: Cengage Learning, 2016.

[2] IEEE Computer Society, Microprocessor Standards Committee, Institute of Electrical and Electronics Engineers, and IEEE-SA Standards Board, *754-1985 – IEEE Standard for Floating-point Arithmetic*, New York, NY: Institute of Electrical and Electronics Engineers, 1985.

[3] IEEE Computer Society, Microprocessor Standards Committee, Institute of Electrical and Electronics Engineers, and IEEE-SA Standards Board, *754-2008- IEEE Standard for Floating-point Arithmetic*, New York, NY: Institute of Electrical and Electronics Engineers, 2008.

[4] Cass, S., *The 2017 Top Programming Languages*, IEEE Spectrum, 18-Jul-2017.

[5] Bull, J.M., Smith, L.A., Pottage, L. & Freeman, R., *Benchmarking Java against C and Fortran for Scientific Applications*, Proceedings of the 2001 Joint ACM-ISCOPE Conference, Palo Alto, California, pp. 97-105, 2001.

[6] Moreira, J.E., Midkiff, S.P. & Gupta, M., *From Flop to Megaflops: Java for Technical Computing*, ACM Transactions on Programming Languages and Systems, **22**(2), pp. 265-295, Mar. 2000.

[7] Cowlishaw, M.F., *Decimal Floating-point Algorithm for Computers*, Proceedings of 16th Symposium on Computer Arithmetic, pp. 104-111, 2003.

[8] Kamble, L., Palsodkar, P. & Palsodkar, P. *Research Trends in Development of Floating Point Computer Arithmetic*, pp. 0329-0333, International Conference on Communication and Signal Processing (ICCSP), 2017.

[9] Erle, M.A. Schulte, M.J. & Linebarger, J.M., *Potential Speedup using Decimal Floating-point Hardware*, Conference Record of the 36th Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, California, **2**, pp. 1073-1077, 2002.

[10] Pham-Quoc, C. Tran-Thanh, B. & Thinh, T.N., *A Scalable FPGA-based Floating-Point Gaussian Filtering Architecture*, 2017 International Conference on Advanced Computing and Application (ACOMP), Ho Chi Minh City, Vietnam, pp. 111-116, 2017.

[11] Jia, X., Wu, G. & Xie, X., *A High-Performance Accelerator for Floating-Point Matrix Multiplication*, 2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC), Guangzhou, China, pp. 396-402, 2017.

[12] O'uchi, S.I., Hiroshi, F., Tsutomu, I., Wakana, N., Takashi, M., Tomohiro, K., Ryousei, T., *Image-Classifier Deep Convolutional Neural Network Training by 9-bit Dedicated Hardware to Realize Validation Accuracy and Energy Efficiency Superior to the Half Precision Floating Point Format*, 2018 IEEE International Symposium on Circuits and Systems (ISCAS), Florence, Italy, pp. 1-5, 2018.

[13] Joldes, M., Marty, O., Muller, J-M. & Popescu, V., *Arithmetic Algorithms for Extended Precision Using Floating-Point Expansions*, IEEE Transactions on Computers, **65**(4), pp. 1197-1210, Apr. 2016.

[14] Muller, J.-M., Popescu, V. & Tang, P.T.P., *A New Multiplication Algorithm for Extended Precision Using Floating-point Expansions*, presented at the IEEE 23rd Symposium on Computer Airthmetic (ARITH), Santa Clara, California, pp. 39-46, 2016.

[15] Rubio-González, C., Nguyen, C., Hong Diep, N., Demmel, J., William, K., Sen, K., Bailey, D.H., Iancu, C. & Hough, D., *Precimonious: Tuning Assistant for Floating-point Precision*, Proceedings of the International Conference on High Performance Computing, Networking, Storage, and Analysis, Denver, Colorado, pp. 1-12, 2013.

[16] Rubio-González, C., Hough, D., Nguyen, C., Sen, K., Demmel, J., William, K., Iancu, C., Lavrijsen, W. & Bailey, D.H., *Floating-point Precision Tuning using Blame Analysis*, Proceedings of the 38th International Conference on Software Enginering, Austin, Texas, pp. 1074-1085, 2016.

[17] Ho, N-M., Manogaran, E., Wong, W.-F. & Anoosheh, A., *Efficient Floating Point Precision Tuning for Approximate Computing*, Proceedings of 22nd Asia and South Pacific Design Automation Conference, Chiba, Japan, pp. 63-68, 2017.

[18] Goodloe, A. E., Muñoz, C., Kirchner, F. & Correnson, L. *Verification of Numerical Programs: From Real Numbers to Floating Point Numbers*, in

NASA Formal Methods, **7871**, G. Brat, N. Rungta, and A. Venet, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 441-446, 2013.

[19] Chiang, W-F., Gopalakrishnan, G., Rakamaric, Z. & Solovyev, A., *Efficient Search for Inputs Causing High Floating-point Errors*, ACM SIGPLAN Notices, **49**(8), pp. 43-52, Feb. 2014.