pzqd: PEZY-SC2 acceleration of double-double precision arithmetic library for high-precision BLAS

Toshiaki Hishinuma[1] and Maho Nakata[2]

[1] PEZY Computing, 5F Chiyoda Ogawamachi Crosta, 1-11, Tokyo 101-0052, Japan
hishinuma@pezy.co.jp
[2] RIKEN, Wako, Saitama 351-0198, Japan maho@riken.jp

Abstract. We implemented pzqd, a high precision arithmetic library for the PEZY-SC2 that is based on Hida *et al.*'s QD library. PEZY-SC2 is an MIMD (multiple instruction stream, multiple data stream) -type many-core processor. We optimized matrix-matrix multiplication (Rgemm) in double-double precision (DD) on the PEZY-SC2. Porting the CPU code to PEZY-SC2 code is relatively easy because PEZY-SC2 is a MIMD-type processor; it runs all the threads independently. As a proof of concept, we ported pzqd with minimal modifications to the original QD library; pzqd can treat a DD type variable in a unified way on the host CPU and the PEZY-SC2. The performance of our implementation of Rgemm in DD (DD-Rgemm) on the PEZY-SC2 attained 75% of the peak performance of DD, which is 20 times faster than an Intel Xeon E5-2618L v3, even including the communication time between the host CPU and the PEZY-SC2. The most important technique for optimizing the DD-Rgemm on the PEZY-SC2 is to make use of the high-speed scratch-pad memory (local memory) installed in each core. We stored the 2x2 DD block matrices and other temporary variables in local memory by reducing the number of threads to increase the local memory size per thread as they occupy local memory even for this block size.

Keywords: PEZY-SC2, double-double precision, MIMD, many-core, matrix-matrix multiplication

1 Introduction

The binary64 [1] (so-called double-precision floating-point numbers) of IEEE Std 754-2008, is a kind of floating point number commonly used for numerical simulations in computational science. It has finite precision and the calculation time and error may increase due to rounding errors when using it to solve problems requiring higher precision than 16 decimal digits.

Various high-precision calculation methods are used to reduce the influence of errors on floating point operations in problems in physics, chemistry, mathematics, etc. [2]. High-precision arithmetic has become relatively more straightforward

to perform because of the dramatic increase in numbers of floating point operations per second of computers as a result of Moore's law. Therefore, we expect an increase in demand for high-precision arithmetic to solve numerically difficult problems that are ill-conditioned or so huge that they require enormous numbers of floating-point operations. Demand will also grow for using high-precision arithmetic for numerical verification [3] and examining numerical reproducibility [4].

High-precision calculations using software are very costly, because they need a lot of computations and memory, and they are not supported in general programming languages. In any case, we should reduce the computational and implementation costs.

Double-double precision (DD) arithmetic is frequently used because of its relatively high speed and low implementation cost. DD arithmetic does not need any special hardware and runs on general-purpose processors, which only use double-precision operations for DD operations. DD floating point numbers are cheap versions of the binary128 (quadruple precision floating point numbers) of IEEE Std 754-2008 and are defined as two non-overlapping double precision floating numbers [5].

Hida *et al.*'s QD library [5] significantly reduces the implementation cost of DD, as it implements DD floating point numbers as a class of C++. This essentially overcomes the first obstacle. To remove the second obstacle, we can use accelerators such as GPUs or SIMD (single instruction multiple data) processors [6–8].

One of the problems that currently limits the performance of computers is power consumption. In particular, it is important to find ways how to increase power efficiency and how to dissipate the heat generated by a computer efficiently.

To solve these problems, PEZY Computing and ExaScaler have been developed from the PEZY-SC2 many-core processor [9] and ZettaScaler 2.2 series of PEZY-SC2-based supercomputers that use a liquid immersion cooling system [10]. In the Green 500 supercomputer energy conservation ranking [11], One of our supercomputers "Shoubu system B" [9] has been certified for three consecutive terms from 2017 to 2018. Presently, we are working on new hardware and are conducting numerical simulations on it [12, 13].

In this paper, we describe the pzqd library, which is a port of the high-accuracy arithmetic library QD [5] on the PEZY-SC2 processor for high-precision BLAS. We implemented and evaluated the DD matrix-matrix product (DD-Rgemm) on the PEZY-SC2 using the pzqd library.

We obtained about 74% of the peak performance of DD for the DD-Rgemm. The key to attaining high execution performance in this case was to launch four threads per processor element (the processor element is the minimal computation unit of the PEZY-SC2) so that we could effectively use the local memory space. The peak performance was 59 GFlops in DD, or equivalently, 1297 GFlops in double precision.

The rest of the paper is organized as follows. Section 2 describes related work, and section 3 shows the architecture and programming model of the PEZY-SC2 processor. An overview of DD arithmetic is given in section 4. We describe the pzqd library, how we optimized DD-Rgemm using it, and how we measured its performance in section 5. Section 6 summarizes the paper.

2 Related work

Some hardware implements binary128[14, 15]; however, most hardware can use only double precision.

Recently, the GNU Compiler Collection and Intel C/C++ and Fortran Compiler implemented "__float128" or "_Quad" binary128 in software.

Double-double precision, sometimes referred to as double-word arithmetic, is widely used. It is used by the IBM XL FORTRAN compiler, IBM XL C compiler, and gcc for the Power series (RS6000), including the MacOSX until 10.6 implemented "long double" and "REAL*16" as double-double precision [16–19]. (Note: double-double precision and binary128 are not compatible.)

Hida *et al.*'s QD library is available for other systems [5]. By using the QD library, users can use dd_real and qd_real (octuple precision) like float or double in C++ or Fortran 90.

A most critical application of DD would be T. Aoyama *et al.*'s numerical evaluation of the electron anomalous magnetic moment from quantum electron dynamics [20]. They used DD and even quad-double precision for calculating certain Feynman diagrams.

There is a lot of research on accelerating DD operations. Because DD operations can only be done using double operations, we can use traditional optimization techniques for DD, i.e., thread parallelization, distributed parallelization, SIMD, and so forth.

Mukunoki *et al.*[7] reported an acceleration on an NVIDIA GPU and accelerated Krylov subspace methods on GPUs [21] for matrix-matrix multiplication in double-double precision. Mukunoki *et al.*[22] also reported an implementation and optimization of double-double and triple-double precision GEMM, GEMV, and AXPY on GPUs.

Nakata *et al.*implemented a similar acceleration for double-double precision to replace the CPU implementation of MPACK (multiple precision arithmetic BLAS and LAPACK) [23] and applied it to semidefinite programming [24, 6].

In CAMPARY, M. Joldes *et al.*implemented double-double, triple-double and quad-double precision [25]. They also accelerated matrix-matrix multiplication and other routines on GPU to apply semidefinite programming solver for these precisions [26].

For solving sparse linear equations, Kotakemori *et al.* implemented and accelerated quadruple precision for the Lis library [27]. Hishinuma *et al.* accelerated sparse matrix-vector multiplication for CPUs accelerated by SIMD AVX / AVX2 [8, 28, 29] for the DD iterative solver library.

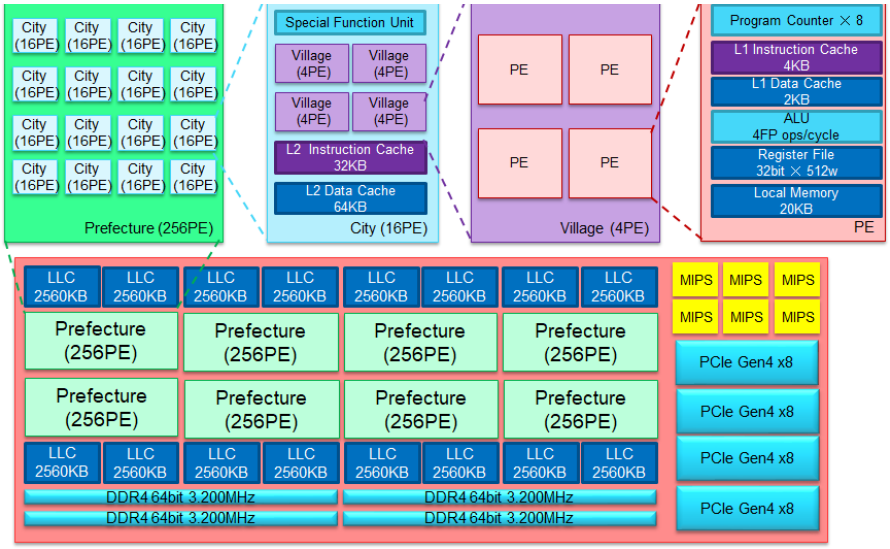

Fig. 1. Block diagram of the PEZY-SC2 processor.

3 Architecture of the PEZY-SC2 processor

3.1 Overview

Figure 1 shows the block diagram of the PEZY-SC2 processor, and table 1 shows the specifications of the PEZY-SC2 processor. The PEZY-SC2 processor is a MIMD (multiple instruction, multiple data) type many-core processor, and the calculation core of the PEZY-SC2 is called the Processing Element (PE).

The PEZY-SC2 processor has in total 2048 PEs in a three-layer hierarchical structure called "Prefecture," "City," and "Village." Each Village has four PEs, each City has four Villages, and each Prefecture has sixteen Cities. The PEZY-SC2 has eight Prefectures.

The inside of each PE has eight register files and eight program counters for running eight threads independently.

Figure 2 shows the thread control mechanism, and figure 3 shows the latency hiding mechanism. The PEZY-SC2 can launch eight threads ($= 4 \times 2$) per PE. There are four active threads, called "front threads"; the remaining four inactive threads are called "back threads."

These front and back threads can be switched sequentially in each cycle (i.e., by using fine-grained multi-threading [30]). When the original front threads stall (e.g., when loading data from memory), we can hide some of the latency by switching back threads to the front and front threads to the back.

Each PE has arithmetic units (an adder and a multiplier). To process an instruction, we need four cycles to read from memory, perform operations, and

Table 1. Specifications of the PEZY-SC2 processor.

Process	16 nm
Clock freq.	1 GHz
L1 cache	4 MB (D), 8 MB (I)
L2 cache	8 MB (D), 4 MB (I)
LLC	40 MB (X-bar connection)
Local memory	40 MB (20KB / PE)
PCIe I/F	PCIe Gen4 8 Lane 4 port (64 GB/s)
DDR I/F	DDR4 64 bit 3200 MHz 4 port (100 GB/s)
# of PEs (cores)	2048 MIMD cores
SIMD vector length	64 bit
Peak performance (DP)	4.1 TFlops
Peak performance (SP)	8.2 TFlops (x2 SIMD)
Peak performance (HP)	16.4 TFlops (x4 SIMD)
Power consumption	200 W (Peak)

write to registers or memory. Consequently, the processor needs at least four threads to fully occupy the arithmetic unit in the pipeline of PE.

Next, let us focus on the instructions and computing unit of the PEZY-SC2. The PEZY-SC2 can use the MAD (Multiply-Add; $d = a + b \times c$) instruction; MAD is addition and multiplication. Unlike the FMA (Fused-Multiply-Add) instruction, it rounds the result of the multiplication. The PE performs the MAD by running the adder and multiplier at the same time. If we run eight MAD instructions on eight threads on one PE, each MAD instruction is processed in one cycle on average.

The PEZY-SC2 processor supports 64-bit SIMD instructions. It can compute one double-precision operation, two single-precision operations, or four half-precision operations simultaneously.

One special function unit (SFU) is installed in each City to calculate division, modulo, square root, and inverse of the square root. Thus, the PEZY-SC2 processor has 128 SFUs (= 16 (Cities) $\times$ 8 (Prefecture)) in total.

Finally, let us focus on the cache memory. Each PE has a 2 KB L1 data cache, and each City has a 64 KB L2 data cache. Each Prefecture has a 2.5 MB LLC (Last Level Cache), and the LCCs of each Prefecture are connected by an X-bar.

PEZY-SC2 has 20 KB worth of small and fast scratchpad memory called "local memory." Figure 4 shows the address layout of the local memory. The PEZY-SC2 processor allocates local memory as a stack, and a user can use the remainder. The local memory can load or store data in one cycle.

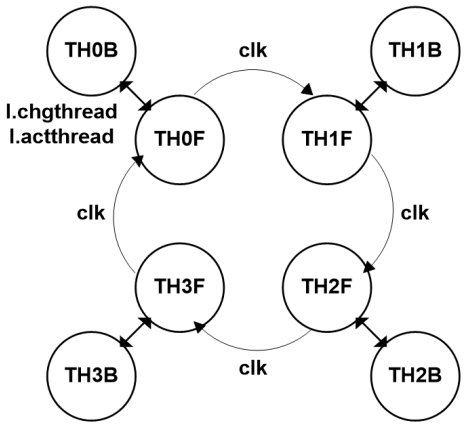

Fig. 2. Thread control mechanism of the PEZY-SC2 processor. THxF means "front threads," THxB means "back threads." The front and back threads can be switched by using I.chgthread or I.actthread instructions. Threads are controlled by fine-grained multi-threading.

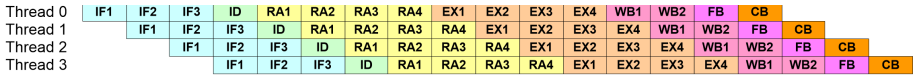

Fig. 3. Pipeline architecture of the PEZY-SC2 processor.

3.2 PEZY-SC2 programming model

The PEZY-SC2 processor supports PZCL, an OpenCL-like programming interface. To run a program on the PEZY-SC2 processor, it needs two types of program: a kernel program and a host program.

A host program runs on the host CPU. A host program is written in C/C++, and the PZCL API compiles it by using an ordinary C compiler (e.g., GNU C compiler). The PZCL API allocates the PEZY-SC2 memory, transfers data between the CPU and PEZY-SC2, launches the kernel program, and so forth. In addition, the SDK (Software Development Kit) for the kernel programs provides mathematical function libraries and atomic operations libraries.

A kernel program runs on the PEZY-SC2 device. It is written in "PZCL C" and compiled with LLVM, which is almost the same as OpenCL C. PZCL C has built-in functions for the PEZY-SC2 architecture, such as thread control using the thread ID (tid) and process ID (pid), synchronization, flushing data, switching between front and back threads, and so forth.

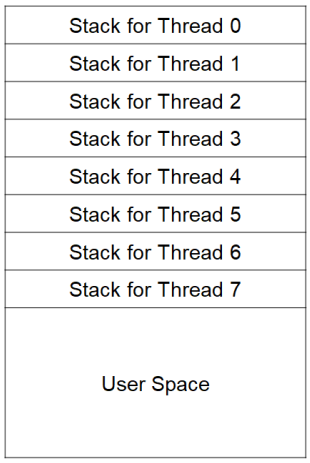

Fig. 4. PEZY-SC2 address layout of the local memory. The size of the local memory is 20 KB. This area is shared by the stack and the userspace.

4 Implementation of pzqd library

4.1 Double-double precision arithmetic

Table 2. Flop count of double-double precision arithmetic.

Algorithm	Add	Mult.	Sum
Two-Sum	6	0	6
Split	3	1	4
Two-Prod	10	7	17
QuadAdd-IEEE	20	0	20
QuadMul	15	9	24

A DD number is represented by two double-precision numbers as $a = (a_{hi}, a_{lo})$ (see figure 4.1). It consists of a sign part of 1 bit, an exponent part of 11 bits, and a significant part of 104 (52×2) bits. For comparison, the binary128 of IEEE Std 754-2008 is composed of a 1-bit sign part, 15-bit exponent part and 112-bit significant part; the DD number has four fewer bits in its exponent part and eight fewer bits in its significant part.

Next, we show how we realize addition and multiplication of two DD numbers. First, we use the fact that we can add, subtract, and multiply floating point numbers with numerical round offs. Still, these round off errors can correctly be evaluated [5, 31, 32].

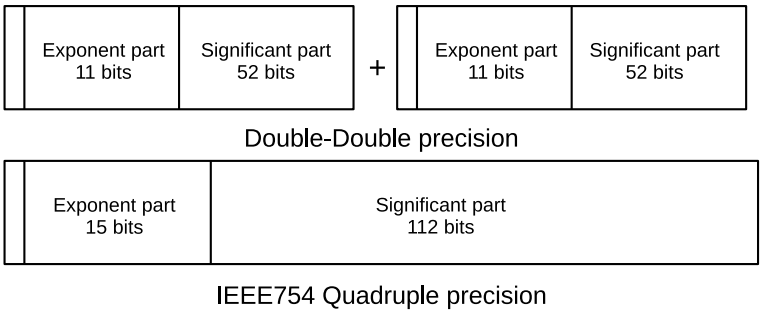

Fig. 5. Schematic view of a double-double precision number in comparison with IEEE 754 quadruple precision. DD has a 1-bit sign part, 11-bit exponent part, and 104-bit significant part.

Here, let us calculate *exactly* the addition $s = a \oplus b$ and its error $e = a + b - (a \oplus b)$ for a floating point number a, and b, where $\oplus$ denotes addition as a floating point operation.

When $|a| \geq |b|$, the addition $s = a \oplus b$ and its error $e = a + b - (a \oplus b)$ can be evaluated with the following Quick-Two-Sum (a, b) algorithm:

Quick-Two-Sum (a, b):

1. $s \leftarrow a \oplus b$
2. $e \leftarrow b \ominus (s \ominus a)$
3. **return**(s, e).

When the relation of a and b is not known, we can use the Two-sum (a, b), although it requires more floating point number operations:

Two-Sum (a, b) :

1. $s \leftarrow a \oplus b$
2. $v \leftarrow s \ominus a$
3. $e \leftarrow (a \ominus (s \ominus v)) \oplus (b \ominus v)$
4. **return**(s, e),

where $\ominus$ denotes subtraction as a floating point operation. The Two-Sum algorithm requires six double-precision operations compared with the three of QuickSum.

Next, let us evaluate the multiplication $p = a \otimes b$ and its error $e = a \times b - (a \otimes b)$ *exactly*; multiplication as a floating point operation is denoted by $\otimes$.

Using the sub-function Split (a), the double-precision number a is divided into two double-precision numbers a_{hi}, a_{lo} and $a = a_{hi} + a_{lo}$ as follows:

Split (a):

1. $t \leftarrow (2^{27} + 1) \otimes a$
2. $a_{hi} \leftarrow t \ominus (t \ominus a)$
3. $a_{lo} \leftarrow a \ominus a_{hi}$

4. **return**(a_{hi}, a_{lo})

Moreover, we use Two-Prod (a, b) to calculate p and e above, as follows:

Two-prod (a, b):

1. $p \leftarrow a \otimes b$
2. $(a_{hi}, a_{lo}) \leftarrow \text{Split}(a)$
3. $(b_{hi}, b_{lo}) \leftarrow \text{Split}(b)$
4. $e \leftarrow ((a_{hi} \otimes b_{hi} \ominus p) \oplus a_{hi} \otimes b_{lo} \oplus a_{lo} \otimes b_{hi}) \oplus a_{lo} \otimes b_{lo}$
5. **return**(p, e)

Finally, let us define addition (QuadAdd-IEEE) and multiplication (QuadMul) for two arbitrary DD numbers a and b:

QuadAdd-IEEE (a, b):

1. $(s_{hi}, e_{hi}) = \text{Two-Sum}(a_{hi}, b_{hi})$
2. $(s_{lo}, e_{lo}) = \text{Two-Sum}(a_{lo}, b_{lo})$
3. $e_{hi} = e_{hi} \oplus s_{lo}$
4. $(s_{lo}, e_{lo}) = \text{Quick-Two-Sum}(s_{hi}, e_{hi})$
5. $e_{hi} = e_{hi} \oplus s_{lo}$
6. $(s_{hi}, e_{lo}) = \text{Quick-Two-Sum}(s_{hi}, e_{hi})$
7. **return**(c)

and

QuadMul (a, b):

1. $(p_{hi}, p_{lo}) = \text{Two-Prod}(a_{hi}, b_{hi})$
2. $p_{lo} = p_{lo} \oplus (a_{hi} \otimes b_{lo} \oplus a_{lo} \otimes b_{hi})$
3. $(c_{hi}, c_{lo}) = \text{Quick-Two-Sum}(p_{hi}, p_{lo})$
4. **return**(c)

. Table 2 shows the number of flops (FLOating-Point operationS) of the double-precision arithmetic operations that constitute the DD arithmetic operation. Note that it is also possible to reduce the number of calculations by lowering the operation precision and using the FMA instruction [5, 6].

We ran a simple benchmark of calculating the dot product of a vector of length 10^5. We compared the elapsed time of FORTRAN REAL*16 in the Intel FORTRAN compiler 13.0.1 (this is software-implemented binary128) and the DD on Intel Xeon-E5-2618L v3.

By quadruple precision calculation (REAL*16) took 3.5 [ms], while the DD calculation took about 0.45 [ms]. The DD operations were approximately 7.7 times faster than the FORTRAN REAL*16 quadruple-precision operations.

Finally, let us point out another feature of the DD arithmetic operation; the data request amount in bytes (Byte / Flop) for the one floating point operation is smaller than in the double-precision operation. For example, the MAD ($d = a \times b + c$) operation for DD numbers requires 44 floating operations using double-precision numbers, whereas the memory requirement is $48 = 16 \times 3$ Bytes. Thus, Byte / Flop ratio is about 0.91. The double-precision operation requires 24 bytes to be read from memory and performs two floating operations; its Byte / Flop ratio is therefore 12.

4.2 Implementation details of the pzqd library

We developed a definition of the DD type, DD mathematical functions, and DD arithmetic operations for the kernel program by using the syntax of PZCL C++ (under development).

All of the operations in the QD library are single-threaded programs. As mentioned above, the kernel program supports C syntax, and the program is parallelized with tid (0 - 7) and pid (0 - 1983) as indices; each thread can be operated independently. Therefore, we could port most of the functions without modification. An exception was that the pzqd library does not support I/O functions.

We implemented operator overloading of DD arithmetic operations by using PZCL C++. The kernel program can handle a DD-type variable that is transferred from the CPU in the same way as the host program using the QD library.

We confirmed that the pzqd library and the QD library give bit-wise the same result for sample codes of the QD library. We deleted the I/O parts on the PEZY-SC2 and run only one thread.

Figures 6 and 7 show examples of a host program using the QD library and kernel program using the pzqd library.

The code of figure 6 runs the following flow:

1. It declares dd_real type array $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$,
2. transfers $\boldsymbol{a}$ and $\boldsymbol{b}$ to the PEZY-SC2 from the CPU,
3. calls the "pzc_Add_dd" function written in the kernel program, and
4. receives the answer array $\boldsymbol{c}$ from the PEZY-SC2.

The code of figure 7 runs the following flow:

1. It gets its own thread ID (tid) and PE ID (pid),
2. computes its own global thread ID by using tid and pid,
3. computes the locations of the arrays $\boldsymbol{a}$ and $\boldsymbol{b}$ corresponding to the global thread ID, and
4. computes $\boldsymbol{c}$[index] += $\boldsymbol{a}$[index] $\times$ $\boldsymbol{b}$[index].

As shown above, pzqd treats the dd_real type in the PEZY-SC2 kernel program in the same way as the CPU code using the QD library.

5 Experimental results

We conducted two experiments. The first evaluated DD elementary functions in pzqd; the second evaluated an implementation of matrix-matrix multiplication in DD using pzqd.

We used a PEZY-SC2 system that operated 1984 PEs at 700 MHz and used an Intel Xeon D-1571 as the host CPU (with DDR4 at 2400 MHz, 64GB of memory and a memory bandwidth of 76 GB/s). Initially, four Cities were disabled to improve the yield. The peak performance of the PEZY-SC2 in double

```
#include "qd.h"
...
int main(){
dd_real *a, *b, *c;
...
size = sizeof(dd_real) * N;
cl_mem mem_a = clCreateBuffer(..., N);
cl_mem mem_b = clCreateBuffer(..., N);
cl_mem mem_c = clCreateBuffer(..., N);
...
enqueueWriteBuffer(mem_a, true, 0, size, a);
enqueueWriteBuffer(mem_b, true, 0, size, b);
...
Add_dd.setArg(0, mem_a);
Add_dd.setArg(1, mem_b);
Add_dd.setArg(2, mem_c);
Add_dd.setArg(3, N);
...
enqueueNDRangeKernel(Add_dd, ThreadsNum, ...);
enqueueReadBuffer(mem_C, true, 0, size, C);
```

Fig. 6. Example of host program using QD library written in C++ with PZCL API

precision was 2777 GFlops ($\approx$ 0.7 [GHz] $\times$ 1984 [cores] $\times$ 2 [MAD operations]). The operating system was CentOS 7.2, the host program compiler was gcc 4.8.5, and the kernel program compiler was pzSDK 4.1 + LLVM 3.6.2. We verified that the LLVM compiler issued the MAD instructions for the kernel program at the assembly level.

For comparison, we used an Intel Xeon E5-2618L v3@2.3 GHz with eight cores and 64 GB of memory. In this experiment, we did not explicitly use the FMA or the SIMD AVX2 instructions, i.e., the SIMD extension instruction set of the Xeon CPU. Without the SIMD AVX2 instructions, the theoretical peak performance in double precision is 73.6 GFlop for Intel Xeon E5-2618L v3@2.3 GHz eight cores. Note that when we used the SIMD AVX2 instructions, it reached 294.4 GFlops. The Turbo Boost feature was disabled. While we did not write SIMD or FMA instructions, we did not suppress FMA instructions generated by the compiler.

5.1 Performance of elementary operations in one thread on the PEZY-SC2

We roughly estimated the ratio of the speeds of the CPU and the PEZY-SC2 in one thread. For the PEZY-SC2 processor, the peak performance dropped to 1 / 4 when only one thread was running, because the PEZY-SC2 operates one thread every four clocks.

```
#include "pzqd_real.h"
void pzc_Add_dd(dd_real* a, dd_real* b, dd_real* c, int N)
{
int tid = get_tid();
int pid = get_pid();
int index = pid * get_maxpid() + tid;
int maxid = get_maxpid * get_maxtid();

for (   ; index < N ; index += maxid)
{
  a[index] = "3.14159265358979323846264338327950288";
  b[index] = "2.249775724709369995957";
  c[index] += a * b;
}
flush();
}
```

Fig. 7. Example of kernel program using pzqd library written in PZCL C

Usually, we fill up the pipeline of the PEZY-SC2 processor by executing every four threads sequentially in the PE to hide latency. Therefore, the performance ratio of the PEZY-SC2 processor between Intel Xeon E5-2618 v3@2.3 GHz at peak performance in one thread is 13.1 = (2.3 [GHz] / (0.7 [GHz] / 4)). Thus, we estimate that the PEZY-SC2 is least 13.1 times slower than Xeon E5-2618 when we operate both with only one thread.

The implementations of the elementary functions of pzqd are similar to those of the QD library; we used a Taylor expansion to obtain them. We verified that our implementation gave the same bit-wise results as the QD library by inputting random DD values. Consequently, we found that we can run the same math functions of the QD library on the host machine and on the PEZY-SC2.

Table 3 shows the results of benchmarking the elementary functions by using one thread of the Xeon E5-2618Lv3 CPU and one thread of the PEZY-SC2 10^6 times. In these cases, we fixed the input to 0.5 in order to obtain more systematic results because some functions vary in performance depending on their input.

The elapsed time of the PEZY-SC2 was about 15 to 25 times longer than that of the CPU. Addition and multiplication were 14.7 and 14.5 times slower, respectively; these values are very reasonable. However, the results for other functions (sin, cos, acos, and asin) were much slower. We suspect this was due to the differences in the special function unit and compiler between the CPU and PEZY-SC2. Nevertheless, the performance losses were not too serious.

Table 3. Elapsed times of executing elementary functions 10^6 times in one thread [second] (ratio) on E5-2618L v3 and the PEZY-SC2.

	Xeon E5-2618L v3	PEZY-SC2
add	0.007 (1.00)	0.11 (14.7)
mult	0.010 (1.00)	0.15 (14.5)
div	0.422 (1.00)	6.96 (16.4)
sin	0.394 (1.00)	8.44 (21.3)
cos	0.411 (1.00)	8.51 (20.7)
pow(x,2)	0.038 (1.00)	0.73 (18.7)
sqr	0.022 (1.00)	0.37 (16.6)
sqrt	0.005 (1.00)	0.09 (15.3)
asin	0.588 (1.00)	14.3 (24.2)
acos	0.583 (1.00)	14.5 (24.8)
sinh	0.444 (1.00)	7.45 (16.8)
cosh	0.466 (1.00)	7.32 (15.7)
log	0.425 (1.00)	5.91 (13.9)
exp	0.433 (1.00)	6.98 (16.1)

5.2 DD-Rgemm in PEZY-SC2

Here, we explain the details of the matrix-matrix multiplication in DD (DD-Rgemm). The DD-Rgemm routine calculates

$$C = \alpha AB + \beta C,$$

where A, B, and C are DD square dense matrices of size $N \times N$, and α and β are scalar values in DD. This routine is a straightforward extension of DD to the BLAS Level 3 gemm.

The core operation of the DD-Rgemm is the DD multiply-add operation. This operation consists of 35 double-precision additions and nine multiplications. We defined the peak performance of DD as 1745 GFlops = 2777 / 35 × 22, where 2777 is the peak performance of the PEZY-SC2.

We defined the peak performance of DD because the numbers of additions and multiplications are not uniform for double-precision arithmetic; the DD multiply-add operation cannot be processed in (35 + 9) / 2 = 22 cycles on the MAD unit. It takes 35 cycles even if we use the MAD instruction for all *double* operations; however, the numbers of additions and multiplications of DD multiply-add are not equal. Therefore, the peak performance of DD-Rgemm on the PEZY-SC2 should be 1745 GFlops.

The peak performance of the Intel Xeon E5-2618L v3 is 73.6 GFlops, when we issue the FMA instructions. Therefore, the peak performance of the DD calculation is 73.6 / 35 × 22 = 46 GFlops, if we consider the unequal numbers of additions and multiplications in DD arithmetic.

To speed up DD-Rgemm, we made 2 × 2 blockings [33]. The block size can be small, and we set the block size to 2 × 2 since the amount of data requested

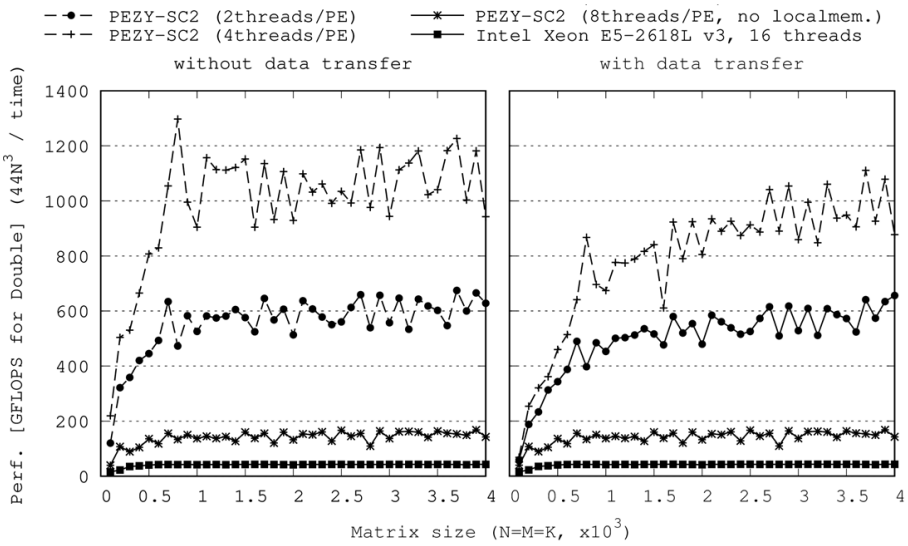

Fig. 8. Performance of DD-Rgemm on the PEZY-SC2 (right: includes the communication overhead between the CPU and the PEZY-SC2, left: pure kernel execution time; i.e., the communication overhead is not included). The horizontal axis is the matrix size N, and the vertical axis is performance, which is equal to the floating point operations per second divided by $44 \times N^3$.

to be sent to memory per operation is small in DD arithmetic. In this way, we can store all the blocking matrix and temporary variables of DD arithmetic in the local memory. Also, the DD addition and multiplication operations are inline expanded to eliminate the overhead of function calls.

As the matrix size increases, the data size increases and exceeds the capacity of the local memory space. We reduced the number of threads and in turn the usage of the stack area, so that all the data fit in the local memory.

5.3 Performance of DD-Rgemm in the PEZY-SC2

Figure 8 shows the results of DD-Rgemm of several implementations, with various matrix sizes and with / without communications between the CPU and the PEZY-SC2. The matrices used in the experiment were dense square matrices, and we filled them with random numbers. The horizontal axis is the matrix size N, and the vertical axis is performance, which is equal to the floating point operations per second divided by $44 \times N^3$. We performed the calculation six times for each size N. We discarded the first result and averaged the remaining five. We verified the resultant matrices against the QD library; the pzqd library was equal bit-wise to the QD library.

The results for the cases including communications included (i) the time to allocate memory to the matrices A, B, and C on the PEZY-SC2, (ii) the time to transfer the data of each of the matrixes from the CPU to the PEZY-SC2, (iii) the time to compute DD-Rgemm, and (iv) the time to transfer the resultant matrix C from the PEZY-SC2.

We tested the following implementations:

PEZY-SC2 (2 threads / PE) Launch total 3968 threads (= 1984 ×2), two threads per PE. 2 × 2 blocking, and temporary variables stored in local memory.

PEZY-SC2 (4 threads / PE) Launch total 7936 threads (= 1984 ×4), four threads per PE. 2 × 2 blocking, and temporary variables stored in local memory.

PEZY-SC2 (8 threads / PE, no localmem.) Launch total 15872 threads (= 1984 ×8), eight threads per PE. 2 × 2 blocking, all variables stored in global memory because the stack overflowed local memory.

Intel Xeon E5-2618L v3 16 threads in total launched using OpenMP on the Intel Xeon. 2 × 2 blocking, the same program except for the PEZY-SC2-specific stuff.

We could not perform 4 × 4 blocking because the stack size overflowed the local memory for any number of threads, and we could not obtain correct results.

Using four threads per PE, the performance reached 1111 GFlops counting communications and 1297 GFlops not counting communications when all data were stored in local memory. These values represent 40 % and 47 % of peak performance in double precision (2777 GFLOPS) and 64 % and 74 % of peak performance in DD (1745 GFLOPS).

Using two threads per PE, performance reached 656 GFlops counting communications and 674 GFlops not counting communications when all data were stored in local memory. The performance of four threads without counting communications was 1.97 times faster than that of two threads, i.e., almost two times faster. Since the PEZY-SC2 fills up the pipeline with four threads per PE (as shown in figure 3), when we use only two threads per PE, the performance is halved, and there are no operations for the remaining two clocks.

The memory bandwidth is not a bottleneck because the performance of four threads is twice that of two threads. The amount of data required for the calculation is small enough for the DD arithmetic to fill up the instruction pipeline.

Using eight threads per PE, the performance reached 167 GFlops counting communications and 168 GFlops not counting communications when all data were stored in global memory. These values are about 10% of peak performance in DD. It seems that overflow of the local memory caused a performance degradation. The blocking matrices and temporary data of DD arithmetic were too large.

From the above results, it turns out that the most effective implementation is one with four threads per PE. Even including communications, its performance was the highest.

We obtained 43 GFlops at maximum on the CPU, which was about 93% of peak performance in DD. The PEZY-SC2 runs were 4.3 (13.8) times faster than the CPU with (without) communication for a matrix size of 100, and 19.5 (22.5) times faster for a matrix size of 2000. When the matrix is small, the kernel startup overhead time and data communication overhead are large on the PEZY-SC2; in this case, it was only four times faster than the CPU.

Next, let us focus on the overhead of the kernel startup time and evaluate the performance when the matrix is small. We measured the time for kernel launch. It was 30 [ms] on average and 50 [ms] at maximum.

The elapsed time when for a matrix size of 100 was 220 [ms] without communications and 740 [ms] with communications. The communication time, kernel startup time, and calculation time were 520, 50, and 170 [ms], respectively. As the kernel startup time amounted to about 30% of the calculation time, including communications caused the PEZY-SC2 to be only about four times faster than the CPU.

On the other hand, when the matrix size was 200, the total calculation took 650 [ms] without communications and 1350 [ms] with communications. The communication time, kernel activation time, and calculation time were 700, 50, and 600 [ms] in this case, so the speedup when including communications was 11 times relative to the times of the CPU.

Even if the kernel startup time is assumed to be the maximum, it occupies 10% or less of the total elapsed time. Thus, the kernel startup time is not a big problem; we can expect a performance speedup even when the matrix is small.

Looking at the case of even smaller matrixes, the performance for a matrix size of 60 was 50.9 GFlops without communications and 13.6 GFlops with communications, whereas the CPU ran at 14.2 GFLOPS. Consequently, the PEZY-SC2 performs DD-Rgemm faster than the CPU when the matrix size is larger than 60.

From these results, the performance reached 168 GFlops when we did not use the local memory, and it reached a ceiling of 1297 GFlops, or 74% of peak performance in DD, when we used the local memory. This value is equivalent to 59 GFlops by DD. All of the results were faster than those of the CPU for a matrix size of 60 or more.

5.4 Efficiency of thread parallelization of DD-Rgemm on the PEZY-SC2

We analyzed the causes of the performance degradation of DD-Rgemm on the PEZY-SC2. To analyze the parallelization efficiency, we increased the number of Cities and evaluated PEZY-SC 2's three-layer hierarchical cache structure.

Table 4 shows the efficiency of thread parallelization for a matrix size of 2000 in four threads per PE and increasing the number of PEs to be activated. The total size of the DD matrices A, B, and C comes to 192 MB, too big to fit in the LLC (40 MB).

We obtained the highest performance for 64 threads (128 PE, 1 City), i.e., 95% of peak performance. The efficiency of thread parallelization was more than

Table 4. Efficiency of thread parallelization (matrix size $N = M = K = 2,000$, no communication, four threads per PE). Performance means $44 \times N^3$ / time.

# of threads (PE)	time [sec.]	perf. (ratio)	peak of DD ratio
64 (16, 1 City)	25.4	13.4 (1.0)	95%
128 (32, 2 Cities)	13.0	26.7 (2.0)	95%
256 (64, 4 Cities)	6.8	52.7 (3.9)	94%
512 (128, 8 Cities)	3.5	102.2 (7.6)	91%
1024 (256, 1 Prefecture)	1.9	194.3 (14.5)	86%
2048 (512, 2 Prefectures)	1.0	356.1 (26.6)	79%
4096 (1024, 4 Prefectures)	0.5	712.9 (53.2)	79%
7936 (1984, 8 Prefectures)	0.3	1234.0 (92.1)	71%

90% of the theoretical value, even when we increased the number of threads to 1024 (256 PE, 1 Prefecture), and performance increased linearly as we increased the number of threads.

For a matrix size of 2000, the CPU performed at about 41.3 GFlops. The PEZY-SC2 exceeded the performance of the CPU when we used 64 PEs and 256 or more threads. The peak performance ratio of the CPU and the PEZY-SC2 was about 24 times. The PEZY-SC2 needed to use 83 or more PEs (=1984/24) to beat the CPU. However, 95% of peak performance was high enough to exceed the performance of the CPU with 64 PEs or more.

Although the efficiency of thread parallelization was less than 90% when the number of threads was 2048 or more (512 PE or more, 2 Prefecture or more), the efficiency of thread parallelization was still very high (75 %) even with 7986 threads (i.e., when we used all of the PEs); this amounts to 71% of peak performance.

The performance loss when we used a large number of threads (especially when we used more than 1024 threads) may be due to LLC misses. One Prefecture is up to 1024 threads, and it completely occupies the LLC. If we had used more than one Prefecture, thrashing of the LLC may have occurred.

When we look at the performance counter, the cache hit rate of the LLC at 1024 threads is about 95%, whereas for 7936 threads it drops to about 88%. For 1024 threads, one Prefecture exclusively occupies the LLC, and when the number of threads is 2048 or more, accesses across the LLCs start to occur; thus, LLC thrashing may occur. We expect that the LLC thrashing would resulted in a substantial loss in performance.

6 Summary

We developed the pzqd library; a double-double precision arithmetic library based on Hida *et al.*'s QD library for the PEZY-SC2 processor. The features of PEZY-SC2 are: (i) 2048 processor elements (PEs) in total, (ii) a three-layer hierarchical cache structure consisting of Village, City, and Prefecture, and (iii)

fast local memory (20 KB per PE). It can load or store in one cycle; it has (iv) an efficient threading mechanism with eight threads using fine-grained multithreading, and (v) a MIMD-type processor, making it easy to port conventionally threaded CPU codes. We also implemented DD matrix-matrix multiplication (DD-Rgemm) using pzqd and evaluated its performance.

To make use of these features, we reduced the number of threads from eight to four to increase the remaining area of local memory. This allowed us to store the intermediate variables for DD arithmetic and the small blocking matrix in local memory, so that it became possible to calculate matrix-matrix multiplications of any size.

Our optimized Rgemm routine attained 74% of peak performance at maximum, not counting the communication time between the CPU and the PEZY-SC2. This level of performance is equivalent to 59G Flops in DD operations, or 1297 GFlops in double-precision operations. Moreover, it is faster by 23 times than the Intel Xeon CPU. Even when we included the communication time between the CPU and the PEZY-SC2, the PEZY-SC2 outperformed the CPU when the matrix size was 60 or more. Thus, we also demonstrated the usefulness of the PEZY-SC2 even in comparatively small problems.

The execution efficiency of our implementation for the PEZY-SC2 was 91% of peak performance when running in 256 PEs, i.e., one Prefecture. This value is quite good; it is 14.5 times faster than that of running 16 PEs. Using all 1984 PEs was 92 times faster than using 16 PEs.

Our future tasks will be (i) to improve the parallelization efficiency by optimizing the data management in the cache memory and (ii) to implement quad-double precision and BLAS functions other than matrix-matrix multiplication.

7 Acknowledgments

We used the Shoubu System B installed at RIKEN in collaboration with RIKEN, PEZY Computing, and ExaScaler Inc. A subsidy supporting this research for advanced use of high-performance general-purpose computers was provided by the Ministry of Education, Culture, Sports, Science, and Technology. A Grant-in-Aid for Scientific Research (B) (KAKENHI: 18H03206) also supported this research.

References

1. IEEE: IEEE standard for floating-point arithmetic, IEEE Std 754-2008, 1–70 (2008). DOI:10.1109/IEEESTD.2008.4610935
2. Bailey, D.H.: High-Precision Floating-Point Arithmetic in Scientific Computation. Computing in Science and Engineering. 7, 54–61 (2005). DOI:10.1109/MCSE.2005.52
3. Rump, S.M.: Verification methods: Rigorous results using floating-point arithmetic, Acta Numerica, 19 287–449 (2010). DOI:10.1017/S096249291000005X

4. Demmel, J., Nguyen, H.D.: Numerical Reproducibility and Accuracy at ExaScale, In: 2013 IEEE 21st Symposium on Computer Arithmetic, Austin, TX, 2013, pp. 235–237. DOI:10.1109/ARITH.2013.43
5. Hida, Y., Li, X.S., Baily, D.H.: Library for Double-Double and Quad-Double Arithmetic. http://crd-legacy.lbl.gov/~dhbailey/mpdist/ and reference therein.
6. Nakata, M.: Numerical evaluation of highly accurate multiple-precision arithmetic version of semidefinite programming solver: SDPA-GMP, -QD and -DD, In: 2010 IEEE International Symposium on Computer-Aided Control System Design, IEEE Press, New York (2010). DOI:10.1109/CACSD.2010.5612693
7. Mukunoki, D., Takahashi, D.: Implementation and Evaluation of Quadruple Precision BLAS Functions on GPUs. In: PARA 2010: Applied Parallel and Scientific Computing, LNCS, vol. 7133, pp. 249–259. Springer, Heidelberg (2012). DOI:10.1007/978-3-642-28151-8_25
8. Hishinuma, T., Tanaka, T., Hasegawa, H.: SIMD Parallel Sparse Matrix-Vector and Transposed-Matrix-Vector Multiplication in DD Precision. In: VECPAR2016: 12th International Meeting on High Performance Computing for Computational Science LNCS, vol. 10150, pp. 21–34, Springer, Heidelberg (2016). DOI:10.1007/978-3-319-61982-8_4
9. PEZY Computing: PEZY-SC2 module & processor. https://www.pezy.co.jp/products/pezy-sc2module-processor/ (In Japanese).
10. Torii, S., Ishikawa, H., Kimura, Y., Saitoh, M.: Technologies and Future Prospects of Green Supercomputer ZettaScaler. IEICE Transactions C, J100-C, 537–544 (2017), (In Japanese).
11. Green500. https://www.top500.org/green500/
12. Tanaka, H., Ishihara, Y., Sakamoto, R., Nakamura, T., Kimura Y., Nitadori, K., Tsubouchi, M., Makino, J.: Automatic Generation of High-Order Finite-Difference Code with Temporal Blocking for Extreme-Scale Many-Core Systems, In: ESPM2 2018: Fourth International Workshop on Extreme Scale Programming Models and Middleware, pp. 1–8, Dallas (2018).
13. Hishinuma, T., Kurosawa, N.: Development and Evaluation of OpenFOAM for PEZY-SC series toward to PEZY-SC3, OpenCAE Symposium 2018, No. A25, pp. 1–6, Tokyo (in Japanese).
14. Lichtenau, C., Carlough, S., Mueller, S.M.: Quad Precision Floating Point on the IBM z13, In: 2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH), IEEE Press, New York (2016). DOI:10.1109/ARITH.2016.26
15. David Patterson, Andrew Waterman: The RISC-V Reader: An Open Architecture Atlas. Strawberry Canyon, 1 edition, pp. 1–200 (2017). ISBN:0999249118
16. 128-bit long double floating-point data type. https://www.ibm.com/support/knowledgecenter/en/ssw_aix_71/com.ibm.aix.genprogc/128bit_long_double_floating-point_datatype.htm
17. IBM XL Fortran for AIX, V16.1.0, Language Reference. https://www.ibm.com/support/knowledgecenter/SSGH4D_16.1.0/com.ibm.compilers.aix.doc/langref.pdf?view=kc
18. IBM XL Fortran for Linux, V16.1.1, Language Reference. https://www.ibm.com/support/knowledgecenter/SSAT4T_16.1.1/com.ibm.compilers.linux.doc/langref.pdf?view=kc
19. Libm source. https://opensource.apple.com/source/Libm/Libm-315/Source/PowerPC/
20. Aoyama, T., Hayakawa, M., Kinoshita, T., Nio, M.: Tenth-order electron anomalous magnetic moment: Contribution of diagrams without closed lepton loops, Phys. Rev. D 91, 033006 (2015). DOI:10.1103/PhysRevD.91.033006

21. Mukunoki, D., Takahashi, D.: Using Quadruple Precision Arithmetic to Accelerate Krylov Subspace Methods on GPUs, In: PPAM2013: Proc. 10th International Conference on Parallel Processing and Applied Mathematics. LNCS, Vol. 8384, pp. 632–642, Springer, Heidelberg (2014). DOI:10.1007/978-3-642-55224-3_59
22. Mukunoki, D., Takahashi, D.: Implementation and Evaluation of Triple and Quadruple Precision Floating-point Operations on GPUs IPSJ Transactions on Advanced Computing System (ACS) 6, 66–77 (2013) (in Japanese).
23. Nakata, M.: The MPACK; Multiple precision arithmetic BLAS (MBLAS) and LAPACK (MLAPACK). http://mplapack.sourceforge.net/
24. Nakata, M., Takao, Y., Noda, S., Himeno, R.: A fast implementation of matrix-matrix product in double-double precision on NVIDIA C2050 and its application to semidefinite programming, In: 2012 Third International Conference on Networking and Computing, pp. 68-75. IEEE Press, New York (2012). DOI:10.1109/ICNC.2012.19
25. Joldes, M., Muller, JM., Popescu, V., Tucker, W.: CAMPARY: Cuda Multiple Precision Arithmetic Library and Applications, In: ICMS 2016: Mathematical Software, LNCS, Vol 9725, pp. 232–240, Springer, Charm (2016). DOI:10.1007/978-3-319-42432-3_29
26. Joldes, M., Muller, JM., Popescu, V.: Implementation and Performance Evaluation of an Extended Precision Floating-Point Arithmetic Library for High-Accuracy Semidefinite Programming, In: IEEE 24th Symposium on Computer Arithmetic (ARITH), London, 2017, pp. 27–34. DOI:10.1109/ARITH.2017.18
27. Kotakemori, H., Fujii, A., Hasegawa, H., Nishida, A.: Implementation of Fast Quad Precision Operation and Acceleration with SSE2 for Iterative Solver Library, IPSJ Transactions on Advanced Computing Systems (ACS) 1, 73–84. (2008) (in Japanese).
28. Hishinuma, T., Fujii, A., Tanaka, T., Hasegawa, H.: AVX acceleration of DD arithmetic between a sparse matrix and vector, In: PPAM 2013: the Tenth International Conference on Parallel Processing and Applied Mathematics LNCS, vol. 8384, pp. 622–631, Springer, Heidelberg (2014). DOI:10.1007/978-3-642-55224-3_58
29. DD-AVX Library. https://sourceforge.net/projects/dd-avx/
30. Kongetira, P., Aingaran, K., Olukotun, K.: Niagara: A 32-way multithreaded sparc processor, IEEE Micro, 25, 21–29 (2005). DOI:10.1109/MM.2005.35
31. Knuth, D.E.: The Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd Edition), Addison-Wesley Longman Publishing, Boston (1997).
32. Dekker, T.J.: A floating-point technique for extending the available precision, Numerische Mathematik, 18, 224–242 (1971).
33. Kågström, B., Ling, P., van Loan, C.: GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark, ACM Trans. Math. Softw. 24, 268–302 (1998). DOI:10.1145/292395.292412