Fast Quadruple Precision Arithmetic Library on Parallel Computer SR11000/J2

Takahiro Nagai¹, Hitoshi Yoshida¹, Hisayasu Kuroda^{1,2},
and Yasumasa Kanada^{1,2}

¹ Dept. of Frontier Informatics, The University of Tokyo,
2-11-16 Yayoi Bunkyo-ku Tokyo, Japan

{[takahiro.nagai](mailto:takahiro.nagai@klab.cc.u-tokyo.ac.jp),[hitoshi.yoshida](mailto:hitoshi.yoshida@klab.cc.u-tokyo.ac.jp)}@klab.cc.u-tokyo.ac.jp

² The Information Technology Center, The University of Tokyo,
2-11-16 Yayoi Bunkyo-ku Tokyo, Japan

{[kuroda](mailto:kuroda@pi.cc.u-tokyo.ac.jp),[kanada](mailto:kanada@pi.cc.u-tokyo.ac.jp)}@pi.cc.u-tokyo.ac.jp

Abstract. In this paper, the fast quadruple precision arithmetic of four kinds of basic operations and multiply-add operations are introduced. The proposed methods provide a maximum speed-up factor of 5 times to gcc 4.1.1 with POWER 5+ processor used on parallel computer SR11000/J2. We also developed the fast quadruple precision vector library optimized on POWER 5 architecture. Quadruple precision numbers, which is 128 bit long double data type, are emulated with a pair of 64 bit double data type on POWER 5+ processor used on SR11000/J2 with Hitachi Optimizing Compiler and gcc 4.1.1. To avoid rounding errors in computing quadruple precision arithmetic operations, emulation needs high computational cost. The proposed methods focus on optimizing the number of registers and instruction latency.

1 Introduction

Some numerical methods require much more computation complexity due to rounding errors as increasing the scale of a problem. For example, CG method, one of the solutions for linear equation $A\mathbf{x}=\mathbf{b}$ and using Krylov subspace, is affected by computation errors on a large scale problem. Floating point arithmetic operations generate rounding errors because a real number is approximated with the finite number of significant figures. To reduce errors in floating point arithmetic, quadruple precision arithmetic, e.g. higher precision arithmetic is required.

Quadruple precision number, which is a 128 bit long double data type, can be emulated with a pair of 64 bit double precision numbers on POWER 5 architecture by the run-time routine. The cost of the quadruple precision operations takes much more than the double precision operations. In this paper, we present the fast quadruple precision arithmetic of four basic arithmetic operations, i.e. $\{+, -, \times, \div\}$ and multiply-add operation, and vector library for POWER 5 architecture based machine such as parallel computer SR11000/J2. We implemented

Table 1. IEEE 754 data type of 64 bit double and 128 bit long double on SR11000/J2

Data type	Total bit length	Exponent bit length	Exponent range	Significand bit length	number of significant figures in decimal
IEEE 754	64	11	$-1022 \sim 1023$	52	about 16.0
SR11000/J2	128	11×2	$-1022 \sim 1023$	52×2	about 31.9

fast quadruple precision arithmetics and made up a quadruple precision vector library including four basic arithmetics and multiply-add operations. We achieved high performance that maximum speed up factor of 5 times to gcc 4.1.1.

2 128-Bit Long Double Floating Point Data Type

POWER5+ processors, which are CPUs of SR11000/J2, have 64 bit floating point registers. In 64 bit architecture, to store floating point data with quadruple precision, a pair of 64 bit registers is used by the software. Quadruple precision can handle up to about 31 decimal digits precision number, compared to $(1 + 52) \times \log_{10} 2 \simeq 16$ handled by double precision. The point to notice is that the exponent range is the same as that of double precision. Although the precision is greater, the magnitude of representable numbers is the same as 64 bit double precision numbers. That is, while 128 bit data type can store numbers with more precision than 64 bit data type, it does not store numbers of greater magnitude.

The details are as follows. Each pair of 64 bit numbers has two 64 bit floating point data type with sign, exponent and significand. We show the data format with IEEE 754 standard explained in Table 1[5]. Typically, the low-order part has a magnitude that is less than 0.5 units in the last place of the high-order part, so the value of two parts never overlap and the entire significand of the low-order number adds precision beyond the high-order number.

3 Quadruple Precision Arithmetic

All of algorithms as follows are double or quadruple precision data type based on the round-to-nearest rounding mode. We express the floating operations using $\{\oplus, \ominus, \otimes, \oslash\}$ for $\{+, -, \times, \div\}$ respectively. For example, floating point addition $a + b = \text{fl}(a + b) + \text{err}(a + b)$ exactly, then we use $a \oplus b = \text{fl}(a + b)$ to denote the result of addition, and $\text{err}()$ is the error caused by the operation.

Now, we explain quadruple precision arithmetic operations consisted of two kinds of basic algorithms, Quick-TwoSum() and TwoSum(). These 64 bit double precision algorithms are already used and implemented on gcc 4.1.1 for 128 bit long double data type[3] and they have explained in papers[1,2,4,8].

This type of 128 bit long double data type does not support IEEE special numbers, NaN and INF. The quadruple precision algorithms introduced in this paper are not satisfying the IEEE compliance.

3.1 About Precision

There are two kinds of quadruple precision algorithms for addition operation for the accuracy.

- Accuracy of 106 bit significand
- Accuracy of about 106 bit significand permitting a few bits rounding errors in the last part

Compared with these algorithms, the latter method is realized with the half number of instructions of the former one by permitting a few bits rounding error in the last part. This reason is extra instructions in error compensation. In this paper, we select the latter algorithm by focusing on speeding up of quadruple precision arithmetic.

We have already quantitatively analyzed the quadruple precision arithmetic of addition and multiplication[11]. Here we introduce the quadruple precision algorithms optimized and implemented as vector library.

3.2 Addition

Quadruple precision addition algorithm of Quad-TwoSum(a, b), which is consisted of floating point addition TwoSum(), computes $(s_H, s_L) = \text{fl}(a + b)$. Here, (s_H, s_L) is a part of s , $s = s_H + s_L$. Each of s_H and s_L is 64 bit data type and indicates high-order and low-order part respectively. Then, a, b, s are 128 bit data type. We do not need to consider separating the quadruple precision numbers because each number is stored into memory as 64 bit data type automatically. TwoSum() algorithm[2] also computes $s = \text{fl}(c + d)$ and $e = \text{err}(c + d)$. We have

<pre> Quad-TwoSum(a, b) { (t, r) $\leftarrow$ TwoSum(a_H, b_H) $e \leftarrow r \oplus a_L \oplus b_L$ $s_H \leftarrow t \oplus e$ $s_L \leftarrow t \ominus s_H \oplus e$ return (s_H, s_L) } </pre>	<pre> TwoSum(c, d) { $s \leftarrow c \oplus d$ $v \leftarrow s \ominus c$ $e \leftarrow (c \ominus (s \ominus v)) \oplus (d \ominus v)$ return (s, e) } </pre>
--	---

to pay attention that both of c and d are not 128 bit data type but 64 bit double data type.

Quad-TwoSum() is addition routine for quadruple precision numbers with error considerations using TwoSum() algorithm.

Then the number of operation steps is 11 Flop (FLoating point OPeration), sum of Two-Sum 6 Flop and 5 Flop from addition and subtraction operations. We see that this quadruple precision algorithm requires 11 times more operations compared to 1 Flop of double precision. Flop indicates the number of floating point operations.

3.3 Multiplication

Quadruple precision multiplication algorithm Quad-TwoProd(a, b) computes $(p_H, p_L) = \text{fl}(a \times b)$. (p_H, p_L) is a part of p , $p = p_H + p_L$. Then, a, b, p are also 128 bit data type.

```

Quad-TwoProd( $a, b$ ) {
     $m_1 \leftarrow a_H \otimes b_L$ 
     $t \leftarrow a_L \otimes b_H \oplus m_1$ 
     $p_H \leftarrow a_H \otimes b_H \oplus t$ 
     $e \leftarrow a_H \otimes b_H \ominus p_H$ 
     $p_L \leftarrow e \oplus t$ 
    return  $(p_H, p_L)$ 
}

```

Some processors have FMA (Fused Multiply-Add) instruction set that can compute expressions such as $a \times b \pm c$ with a single rounding error. It is a merit of this instruction that there are not double rounding errors for addition following multiplication operation. FMA instruction is comparatively fast because it is implemented on hardware as well as addition or multiplication instruction. A series of POWER processor can handle FMA instruction, so we made up multiplication algorithm using FMA instruction. It costs 8 Flop in quadruple precision multiplication operation.

3.4 Division

Quadruple precision division Quad-TwoDiv(a, b) computes $(q_H, q_L) = \text{fl}(a \div b)$. (q_H, q_L) is a part of q , $q = q_H + q_L$. Then, a, b, q are also 128 bit data type.

```

Quad-TwoDiv( $a, b$ ) {
     $d_1 \leftarrow 1.0 \oslash b_H$ 
     $m_1 \leftarrow a_H \otimes d_1$ 
     $e_1 \leftarrow -(b_H \otimes m_1 \ominus a_H)$ 
     $m_1 \leftarrow d_1 \otimes e_1 \oplus m_1$ 
     $m_2 \leftarrow -(b_H \otimes m_1 \ominus a_H)$ 
     $m_2 \leftarrow a_L \oplus m_2$ 
     $m_2 \leftarrow -(b_L \otimes m_1 \ominus m_2)$ 
     $m_3 \leftarrow d_1 \otimes m_2$ 
     $m_2 \leftarrow -(b_H \otimes m_3 \ominus m_2)$ 
     $m_2 \leftarrow d_1 \otimes m_2 \oplus m_3$ 
     $q_H \leftarrow m_1 \oplus m_2$ 
     $e_2 \leftarrow m_1 \ominus q_H$ 
     $q_L \leftarrow m_2 \oplus e_2$ 
    return  $(q_H, q_L)$ 
}

```

This algorithm is based on the Newton-Raphson method. The number of operation steps is 18 Flop and 1 double division operation. The definition of Flop

does not include the double division operation because it is costly compared to the cost of double precision addition and multiplication operations.

This algorithm is applicable in the usual case that special numbers such as NaN, INF are not generated by first operation of double division ($1.0 \oslash b$).

4 Speeding-Up the Quadruple Precision Arithmetic

We quantitatively evaluate each algorithms of addition and multiplication on parallel computer SR11000/J2 at Information Technology Center, the University of Tokyo. In terms of the number of operations, addition takes 11 Flop and multiplication takes 8 Flop. Division takes 18 Flop and 1 double precision data division. From the analysis in term of the number of addition and multiplication operations, it is possible to speed up by reducing data dependency between instructions under condition that the each instruction latency such as fadd, fmul, fnadd of processors is the same clocks as others.

And we have considered the multiply-add operation in quadruple precision with the combination of multiplication and addition.

5 Optimizing Quadruple Precision Arithmetic

First, the theoretical peak performance is 9.2 GFlops in one processor on SR11000/J2. Quadruple precision arithmetic operations are rarely affected by the delay of data transfer from main memory to register because computation time of one quadruple precision operation is large. To get high performance, it is most important to increase throughput and hide instruction latency by pipelining the operations for vector data. To realize pipeline processing, we focus on the loop unrolling.

We see that latency of floating point instructions on POWER 5+ such as fadd, fmul, fsub, fabs and fnadd is 6 clocks. Throughput is 2 clocks for fnadd and 1 clock for others. Fig.1 shows the pipeline processing in case of instruction latency is 6 clock.

Fig. 1. Pipelining for 6 clock instruction latency

5.1 Hiding Instruction Latency

Because of loop unrolling, we can optimize performance by way of hiding instruction latency. Data dependency of quadruple precision arithmetic operations is solved by loop-unrolling, which lines up same instructions as follows. An example of solution is shown below. Here, fr means a 64 bit floating point register in POWER architecture. In Problem(), there is data dependency in three instructions, $\{+, \times, \div\}$. It is possible to hide latency by loop unrolling like Solution(), whose unrolling size is 2.

<pre> Problem(){ fr1 ← fr2 + fr3 fr5 ← fr1 × fr4 fr7 ← fr5 ÷ fr6 } </pre>	<pre> Solution(){ fr1 ← fr2 + fr3 fr9 ← fr7 + fr8 fr5 ← fr1 × fr4 fr11 ← fr9 × fr10 fr7 ← fr5 ÷ fr6 fr13 ← fr11 ÷ fr12 } </pre>
---	---

5.2 Number of Registers

Loop unrolling prevents from stall of CPU resource among instructions. As POWER 5+ processor has 32 logical registers, we used the full logical registers. In fact, there are 120 physical registers and they are utilized by the register renaming function. If m is the number of registers needed for 1 quadruple operation,

$$\text{maximum unrolling size} = \lfloor 32 / m \rfloor \quad (1)$$

Quadruple precision addition needs 4 registers in 1 operation of $c_i = a_i + b_i$, that is, m is 4. We can realize maximum unrolling size of 8.

$$\text{maximum unrolling size} = 32 / 4 = 8 \quad (2)$$

In a similar way, quadruple precision multiplication of $c_i = a_i \times b_i$ also needs 4 registers, then m is 4. The maximum unrolling size is 8.

Quadruple precision division of $c_i = a_i / b_i$ needs 5 registers in 1 operation, then m is 5. Maximum unrolling size is $32/5 = 6$. To attain unrolling size 8 as well as addition or multiplication operation, we store 1 register data into memory and reload when it is needed. This method achieves unrolling size of $32/4 = 8$.

6 How to Use Quadruple Precision Arithmetic Operations Library

We have discussed algorithms and how to optimize quadruple precision arithmetic for vector data in sections 3, 4 and 5. The interfaces of each quadruple precision arithmetic operations are shown in this section. This library is especially effective for vector data and implemented in C with optimized assembler-code. Users specify the include file "quad_vec.h" in C and call each arithmetic function in library. We have to note here that it is easy for adaptation to FORTRAN.

Table 2. Compile option

	compile option
Optimizing C Compiler 01-03/C	cc -Os +Op -64
No paralleled	-noparallel
Quadruple precision (add, multiply, div)	-roughquad
gcc 4.1.1	gcc -maix64 -mlong-double-128 -O3

- Addition $c_i = a_i + b_i$
`void qadd_vec(long double a[], long double b[], long double c[], int n)`
- Subtraction $c_i = a_i - b_i$
`void qsub_vec(long double a[], long double b[], long double c[], int n)`
- Multiplication $c_i = a_i \times b_i$
`void qmul_vec(long double a[], long double b[], long double c[], int n)`
- Division $c_i = a_i / b_i$
`void qdiv_vec(long double a[], long double b[], long double c[], int n)`
- Multiply-Add $c_i = s \times b_i + c_i$ (s : constant)
`void qmuladd_vec(long double *s, long double b[], long double c[], int n)`

Here are the sample program routine computing matrix-multiplication in size N using `qmuladd_vec()` described above.

```
long double a[N][N], b[N][N], c[N][N];
...
for(i=0; i<N; i++)
    for(j=0; j<N; j++)
        qmuladd_vec(&a[i][j], &b[j][0], &c[i][0], N);
...
```

7 Numerical Experiment

We implemented and evaluated four kinds of arithmetic operations, addition, multiplication, division and multiply-add operation. Subtract is same operation as addition except for sign. Our proposed methods were optimized with assembler-code and compared with Hitachi Optimizing Compiler of SR11000/J2 [10] and gcc 4.1.1. OS is IBM AIX version 5.3 with large page setting[9]. Compile options are shown in Table 2.

The experimented data size is six patterns, i.e. size of L1 cache, half of L2, L2, half of L3, L3 and out of L3. We measured the MQFlops value (1 quadruple precision operation in 1 second is defined as 1 QFlops). Figures from Fig.2 to Fig.9 show the quadruple precision arithmetic operation performances. The effective clocks, which is the clocks in each loop unrolling size in one loop in our proposed method, is shown in Fig.2 and the computational performance is shown in Fig.3 in addition. Our proposed methods in quadruple precision arithmetic operations show high performance in all of data ranges. Performances of our proposal and Hitachi optimizing compiler in quadruple precision addition are

Fig. 2. Effective Clocks in our proposed addition

Fig. 3. MQFlops in addition

Fig. 4. Effective Clocks in our proposed multiplication

Fig. 5. MQFlops in multiplication

Fig. 6. Effective Clocks in our proposed division

Fig. 7. MQFlops in division

almost same. Operations in gcc 4.1.1 are much slow because its execution calls library in each steps and it takes much cost in function overhead.

From the result of quadruple precision addition, our proposed method attained $73.70/38.34 \simeq 1.9$ times speed up than that of Hitachi optimizing compiler and $73.70/13.96 \simeq 5.3$ times speed up than that of gcc 4.1.1 when data size is just on

Fig. 8. Effective Clocks in multiply-add

Fig. 9. MQFlops in multiply-add

Fig. 10. Matrix multiplication using multiply-add arithmetic operation

L1 cache. At the end, matrix-multiplication result using optimized multiply-add operation is shown in Fig.10.

8 Concluding Remarks

In this paper, fast quadruple precision arithmetic of four kinds of basic arithmetic operations and multiply-add operation are developed and evaluated. The proposed methods provide a maximum speed-up 5 times faster for vector data than gcc 4.1.1 with POWER 5+ processor on parallel computer SR11000/J2. Even though proposed method in quadruple precision addition operation is almost the same with Hitachi optimizing compiler in performance, other quadruple precision arithmetic operations' results show high performance in all of data ranges. We developed the fast quadruple precision library for vector date optimized on POWER 5 architecture. Quadruple precision arithmetic operations are costly, compared with double precision operations because compensating rounding errors. Then we applied the best optimization of hiding latency to fit the number of registers by loop unrolling to quadruple precision arithmetic operation.

As a future work, we have to develop quadruple precision library, which will be available in various architecture such as Intel and AMD. POWER architecture as well as PowerPC has FMA instructions which can operate in same clocks as add or multiply. Especially, in some environment where there is no FMA instruction, we have to develop fast algorithm in quadruple precision arithmetic operations.

References

1. Dekker, T.J.: A Floating-Point Technique for Extending the Available Precision. *Numer. Math.* 18, 224–242 (1971)
2. Knuth, D.E.: *The Art of Computer Programming*, 2nd edn. Addison-Wesley Series in Computer Science and Information. Addison-Wesley Longman Publishing Co., Inc, Amsterdam (1978)
3. The GNU Compiler Collection, <http://www.gnu.org/software/gcc/index.html>
4. A fortran-90 double-double library, <http://www.nersc.gov/~dhbailey/mpdist/mpdist.html>
5. ANSI/IEEE754-1985 Standard for Binary Floating-Point Arithmetic (1985)
6. Akkas, A., Schulte, M.J.: A Quadruple Precision and Dual Double Precision Floating-Point Multiplier. In: *DSD 2003: Proceedings of the Euromicro Symposium on Digital Systems Design*, pp. 76–81 (2003)
7. Hida, Y., Li, X.S., Bailey, D.H.: Algorithms for quad-double precision floating point arithmetic. In: *Proceedings of the 15th Symposium on Computer Arithmetic*, pp. 155–162 (2001)
8. Bailey, D.H.: High-Precision Floating-Point Arithmetic in Scientific Computation. In: *Computing in Science and Engineering*, vol. 07, pp. 54–61. IEEE Computer Society, Los Alamitos (2005)
9. *AIX 5L Rifferences Guide Version 5.3 (IBM Redbooks)*. IBM Press (2004)
10. *Optimizing C User’s Guide For SR11000*. Hitachi, Ltd. (2005)
11. Nagai, T., Yoshida, H., Kuroda, H., Kanada, Y.: Quadruple Precision Arithmetic for Multiply/Add Operations on SR11000/J2. In: *Proceedings of the 2007 International Conference on Scientific Computing CSC, Worldcomp 2007, Las Vegas*, pp. 151–157 (2007)