Error-Free Transformation in Rounding Mode toward Zero

Stef Graillat[1], Jean-Luc Lamotte[1], and Diep Nguyen Hong[2]

[1] CNRS, UMR 7606, LIP6, University Pierre et Marie Curie, 4 place Jussieu, 75252 Paris cedex 05, France
Stef.Graillat@lip6.fr, Jean-Luc.Lamotte@lip6.fr
[2] Laboratoire LIP, CNRS-ENSL-INRIA-UCBL, 46 alle d'Italie, 69364 Lyon Cedex 07, France
hong.diep.nguyen@ens-lyon.fr

Abstract. In this paper, we provide new error-free transformations for the sum and the product of two floating-point numbers. These error-free transformations are well suited for the CELL processor. We prove that these transformations are error-free, and we perform numerical experiments on the CELL processor comparing these new error-free transformations with the classic ones.

1 Introduction

For numerical computing, traditional processors are now in competition with new processors using new architecture.

Over the last 5 years, the main evolution of traditional processors has been towards multi-core architecture, but there is no new approach to design the floating-point unit. The power of the new processor is directly dependent on the number of cores.

On the other hand, new architectures are currently being used for specific numerical codes. The most popular are GPU (see `http://www.gpgpu.org`) and the CELL processor. These new possibilities are the result of the convergence between the multimedia system (mainly graphics operations) and numerical computation. These solutions offer huge power for numerical computation. The peak performance of traditional processors is around 50 Gflops and should be compared with the 200 Gflops of the CELL processors and the 500 Gflops of best graphic cards. Unfortunately, very often, their high level of performance is obtained with specific implementations of floating-point numbers which do not respect the IEEE 754 standard [1]. For example, on the CELL processor, the most powerful unit has only a rounding mode toward zero (truncated mode) and there are no subnormal numbers and no representation for infinity. In both architectures, to obtain a high level of performance, it takes a lot of hard works to find the dependencies of the numerical instructions and to use them carefully to write instructions that use the instruction pipeline fully. The algorithm study must take into account this situation: an algorithm which needs two or three

A. Cuyt et al. (Eds.): Numerical Validation, LNCS 5492, pp. 217–229, 2009.

times more operations can be more efficient if these operations can be easily pipelined.

More and more scientific applications need more accurate computations, whether for specific algorithms (accurate summation, accurate dot product) or for an entire method by using extended precision. One of the main way to achieve this higher precision is to use **Error-Free Transformation** (EFT). An EFT is an algorithm which transforms any arithmetic operation $\circ$ of two values a and b into a sum of two values s and e, where s is an approximation of the result and e is an approximation of the error on the result. Such that $a \circ b = s + e$. A lot of publications have been written on EFT and their applications (see for example [2,3,4,5,6]) but most of the transformation algorithms use only the rounding mode to the nearest except for some papers by Priest [7]. With the new architectures, it is necessary to study the implementation of EFT on processors that perform computation with the rounding mode toward zero.

In this paper, the `TwoSum-toward-zero` proposed by Priest in [7] is studied from an implementation point of view. Its main limitation is found in the dependencies of its instructions. Another version is proposed which reduces the dependencies and allows a more efficient implementation. Concerning multiplication, we study a well-known algorithm that uses a rounding mode to the nearest and is based on FMA (Fused Multiply-Add). We prove that this algorithm is usable with a rounding mode toward zero.

The rest of the article is organised as follows. Section 2 gives a reminder of properties of floating-point numbers that will be used in the paper and results on EFT. In Section 3, we present the main characteristics of the CELL processor and motivates our work. Section 4 details the known EFT algorithms for the sum and the product with rounding mode toward zero. In Section 5, we provide a new EFT algorithm in rounding mode toward zero more suitable for the CELL processor; the proof is given in Section 6. Finally, Section 7 is devoted to performance measurements which show that our new algorithm is faster on the CELL processor.

2 Floating-Point Arithmetic and EFTs (Error-Free Transformations)

Let $\mathbb{F}$ denote the set of all floating-point numbers, and $x \in \mathbb{F}$ be a normalized floating-point number. It can be written as:

$$x = s \times \underbrace{x_0.x_1 \dots x_{p-1}}_{\text{mantissa}} \times B^e, \quad 0 \leq x_i \leq B-1, \quad x_0 \neq 0, \tag{1}$$

with $s = \pm 1$ the sign, B the base, p the precision, and e the exponent of x. We can say that x is a p-bit floating point number. The value $\texttt{eps} = B^{1-p}$ is the relative error of x.

The IEEE 754 standard [1] specifies the base ($B = 2$), $x_0 = 1$ and two main representations: the single precision ($s = 1, p = 24$ bits with the hidden bit, and

$e = 8$) and the double precision ($s = 1, p = 53$ bits with the hidden bit, and $e = 11$).

Floating-point numbers are approximations of real numbers. Let r be a real number. The approximation of r, denoted $\mathrm{fl}(r)$, in the floating-point set $\mathbb{F}$ is equal to r if $r \in \mathbb{F}$. In the other cases, there are two consecutive floating-point numbers $f^-, f^+ \in \mathcal{F}$ such that: $f^- < r < f^+$, and then

$$\mathrm{fl}(r) \in \{f^-, f^+\}.$$

.

The value $\mathrm{fl}(r)$ is chosen between those two values depending on the current rounding mode. There are four rounding modes.

1. to the nearest: $\mathrm{fl}(r)$ is equal to the nearest floating point value of r.
2. toward $+\infty$: $\mathrm{fl}(r) = f^+$.
3. toward $-\infty$: $\mathrm{fl}(r) = f^-$.
4. toward zero: if $r < 0$ then $\mathrm{fl}(r) = f^+$ else $\mathrm{fl}(r) = f^-$.

The **approximation error** of r is defined to be $\mathrm{err}(r) = r - \mathrm{fl}(r)$.

Another binary representation can be used to represent floating-point numbers. Let x be a floating-point number with a binary representation. x can be written as:

$$x = s \times 1.m \times 2^e,$$

where s, m, e are respectively the sign, the mantissa coded with $p - 1$ bits and the exponent. Another representation of x is

$$x = s \times 1m \times 2^{e-p+1},$$

where

- $1m$ is an integer such that $2^{p-1} \leq 1m < 2^p$,
- 2^{e-p+1} is usually named $\mathrm{ulp}(x)$ (unit in the last place).

A bound of the relative error is $\mathtt{eps} = 2^{1-p}$. Since,

$$\mathrm{ulp}(x) = 2^e\mathtt{eps} = \frac{|x|}{1.m}\mathtt{eps},$$

it follows that

$$\frac{\mathtt{eps}}{2}|x| < \mathrm{ulp}(x) \leq \mathtt{eps}|x|.$$

The following lemmas which can be found in [7] are used in this paper.

Lemma 1. *Let $a = m \times \mathrm{ulp}(b)$ a floating-point number of p-bits, and k an integer such as $|k| \leq |m|$, then $k \times \mathrm{ulp}(b)$ is representable by a floating-point number of p-bits.*

Lemma 2. *Let a and b be two floating-point numbers of p-bits such that $1/2 \leq a/b \leq 2$, the difference of a by b is representable by a floating-point number of p-bits i.e. $\mathrm{fl}(a - b) = a - b$.*

Lemma 3. *Let $\circ$ be a floating-point operation. The following inequality is always true,*

$$|\,\mathrm{err}(a \circ b)| < \mathrm{ulp}(\mathrm{fl}(a \circ b)) < \mathtt{eps}|a \circ b|.$$

3 The CELL Processor

The CELL processor [8] uses a new architecture optimized for multimedia applications. It can be used for scientific computation [9] as well. It implements two different cores. The main core is a PowerPC processor (named PPE) with some elements removed (for example, the reordering instruction mechanism) to free place for the 8 SPEs (Synergetic Processor Element) which provide the numerical computation power of the chip.

The PPE is a standard PowerPC processor. It manages the memory, the IO and runs the operating system. It is fully IEEE 754 compliant [1]. An SPE, on the contrary, is a small processor with a SIMD unit. It has only 256 KB of memory, for instructions and data, named the "local store" (LS) and 128 registers of 128 bits. All exchanges with the main memory are managed by the MFC (Memory Flow controller) through DMA access. The SIMD processor is based on a FMA (Fused Multiply-Add) and uses 128-bit registers. So, it performs 4 multiplications on single precision floating-point numbers in a single instruction. Another important characteristic is that it is fully pipelined. That means that it can provide 4 results of 4 FMA operations at each cycle. Its peak performance with a clock at 3.2 Ghz is around 25.6 Gflops. With the 8 SPE on a processor, the peak performance of the entire processor is around 200 Gflops. In double precision, the SIMD processor is not fully pipelined and the peak performance is only 1.8 Gflops/SPE.

The price we pay for the enhanced performance is the incompatibility with the IEEE 754 standard. For single precision, we should note that:

- There is no division.
- Only the 12 first bits of $\frac{1}{x}$ and $\frac{1}{\sqrt{x}}$ are exact.
- `Inf` and `NaN` are not recognized.
- Overflows saturate the largest representable values.
- There are no denormalized results.

Some SPE instructions will be explained in detail to facilitate the understanding of the algorithms. The variables correspond to a 128-bit registers which can contain 16 8-bit integers, or 8 16 bit integers, or 4 32-bit integers or 4 32-bit floating-point numbers or 2 64-bit floating point numbers. Let `u`,`v` and `w` be a 128-bits registers of integer or floating point variables and let `comp` be a field of 128 bits.

The instructions `comp=spu_cmpabsgt(u,v)` and `comp=spu_cmpeq(u,v)` compare the values of `u` and `v`. All the bits of `comp` are set to 1 if the corresponding elements of `u` and `v` are respectively greater or equal in absolute value. An example is provided in Table 1 on two vectors u, v of 4 elements.
Important instructions are:

- `c=spu_sel(u, v, comp)` selects the bits of `u` or `v` in relation to the bits of `comp`. Number of cycles: 2 (see table).
- `c=spu_add(u, v)` adds the four values of `u` with the four values of `v`. Number of cycles: 6

Table 1. Example of the `spu_cmpabsgt` function result

u	1.0	1.0	−1.0	−1.0
v	0.5	−2.0	−0.5	−2.0
comp	0xFFFFFFFF	0x00000000	0xFFFFFFFF	0x00000000

Table 2. Example of the `spu_sel` function result

u	1.0	1.0	−1.0	−1.0
v	0.5	−2.0	−0.5	−2.0
comp	0xFFFFFFFF	0x00000000	0xFFFFFFFF	0x00000000
v	0.5	1.0	−0.5	−1.0

- `c=spu_sub(u, v)` subtracts the four values of `u` from the four values of `v`. Number of cycles: 6
- `c=spu_madd(u, v, w)` multiplies the four values of `u` with the four corresponding values of `v` and then adds the four values of `w` . Number of cycles: 6

The code optimisation on an SPE is very tricky. The SIMD programming is based on the Altivec system. The interested reader can visit the following website `http://www.freescale.com/altivec`. Just a specific note: with the compiler used by the CELL SDK (Software Development Kit), it is possible to have a good estimation of instruction orders that will be run on the processor. This estimation takes into account notably the number of cycles used by the instructions and the capabilities of the two pipelines. The estimation is given by the `SPU_TIMING` option of the compiler which indicates for each instruction the start cycle modulo 10 and the instruction cycle number. In this paper, the result of this option is slightly improved for a better understanding. Figure 1 shows how the instructions are run. On this program, the instruction `inst1` is run first and its duration is 6 cycles. The instruction `inst2` starts at the cycle 2. Its duration is 6 cycles. The instructions `inst3` and `4` start at cycle 8 on two separate pipelines. The sign `-` shows that no instruction can be run during these cycles due to variable dependencies between `inst2` and `inst3`. The cycle lost is also known as a pipeline bubble.

```
inst1 123456
inst2  234567
inst3  ------890123
inst4  ------89
```

Fig. 1. The left column shows the instructions and the right column the cycle number of each instruction and its start cycles

4 The "Error-Free Transformations" (EFT)

An EFT is an algorithm which transforms an arithmetic operation $\circ \in \{+, -, \times, /\}$ on two values a and b into a sum of two floating-point values r and e such that $a \circ b = r + s$. We also require that $r \approx \mathrm{fl}(a \circ b)$ and $e \approx \mathrm{err}(a \circ b)$. EFT are very useful to implement extended precision number [10,2] and accurate operators [4,3,5,6].

Let a and b be two floating-point numbers and $\circ$ any operation in $(+, -, \times, /)$ then we have

$$a + b = \mathrm{fl}(a + b) + \mathrm{err}(a + b),$$

where $\mathrm{fl}(a \circ b)$ is a floating-point corresponding to the result and $\mathrm{err}(a \circ b)$ is the rounding error.

It is known that the error obtained during the operation $a \circ b$ in the rounding mode to the nearest is a floating-point number for $\circ \in (+, -, \times)$. In that case, the result of an EFT must be $r = \mathrm{fl}(a + b)$ and $e = \mathrm{err}(a + b)$.

But it turns out that with other rounding modes, in most cases the error is a floating-point number but there are exceptions. For example, as noticed by Priest, with rounding toward zero, if we subtract a very small positive number from a very large positive number then the rounding error is not a floating-point number.

4.1 The Sum Operation

A set of algorithms has been proposed for the sum of two numbers in the rounding mode toward the nearest and used in a lot of libraries. We can cite `TwoSum` algorithm of Knuth [11] and `FastTwoSum` algorithm of Dekker [10],

For rounding mode toward zero, Priest proposed in [7] the following algorithm to compute the sum of two floating-point number:

```
1  TwoSum-toward-zero (a,b)
2  if (|a| < |b|)
3      swap(a,b)
4  s = fl(a + b)
5  d = fl(s - a)
6  e = fl(b - d)
7  if (e + d != b)
8      s = a, e = b
9  return (s,e)
```

If $[s, e] = \texttt{TwoSum-toward-zero}(a, b)$ then $a + b = s + e$ with either $s = e = 0$ or $|e| < \mathrm{ulp}(c)$.

Figure 2 shows the implementation on the CELL processor of the `TwoSum-toward-zero` algorithm and how the code is run. The instruction running sequence has been given by the SPU_TIMING tool.

```
1  TwoSum-toward-zero(a,b)          | cycles
2    comp = spu_cmpabsgt(b,a)       |  12
3    a = spu_sel(a, b, comp)        |   -34
4    b = spu_sel(b, a, comp)        |     45
5    s = spu_add(a , b)             | 012345
6    d = spu_sub(s , a)             |      -678901
7    e = spu_sub(b , d)             |         ----234567
8    tmp = spu_add(e , d)           |              -----890123
9    comp = spu_cmpeq(d, tmp)       |                        45
10   s = spu_sel(s, a, comp)        |                         -67
11   e = spu_sel(e, b, comp)        |                           89
12   return s,e)                    |
```

Fig. 2. Implementation on the CELL processor of the `TwoSum-toward-zero` algorithm and its instruction running sequence. Cycle cost: 29.

This implementation is not efficient. It is obvious that there are important dependencies between the line 9 and lines 6, 7 and 8. The effect is clearly visible in the information generated by the SPU_TIMING tools. There are a lot of pipeline "bubbles" marked by the '-' character.

4.2 The Product Operation

For the product, there is an algorithm called `TwoProduct` in rounding mode to the nearest proposed by Veltkamp [10] using Dekker `Split` algorithm [10]. The Veltkamp algorithm is not efficient since it costs 17 floating point operations.

The `TwoProduct` algorithm can be re-written in a very simple way if a Fused-Multiply-and-Add (`FMA`) operator is available on the targeted architecture [12]. Some computers have a Fused-Multiply-and-Add (`FMA`) operation that enables a floating point multiplication followed by an addition to be performed as a single floating point operation. As a consequence, there is only one rounding error. The Intel IA-64 architecture, implemented in the Intel Itanium processor, has an `FMA` instruction as well as the IBM RS/6000 and the PowerPC before it and as the new Cell processor [13].

Thanks to the `FMA`, the `TwoProduct` algorithm can be re-written as follows, which costs only 2 operations.

```
1  TwoProductFMA (a,b)
2      p = fl(a * b)
3      e = FMA(a,b,-p)
4      return (p,e)
```

The `TwoProductFMA` function is very efficient with only 2 operations in a rounding mode to the nearest. From a pipeline point of view, the `TwoProductFMA` is not as efficient as it looks because the two operations cannot be pipelined.

In spite of this bad characteristic, on most processors this algorithm is much more efficient than the Veltkamp's algorithm.

5 A New Algorithm for the Sum

With rounding mode toward zero, Priest's `TwoSum-toward-zero` algorithm uses a comparison between $e+d$ and b. This comparison should wait for the end of all the previous instructions to be executed. We propose replacing this comparison by another one which uses only the variables b and d.

```
1  TwoSum-toward-zero2 (a,b)
2  if (|a| < |b|)
3      swap(a,b)
4  s = fl(a + b)
5  d = fl(s - a)
6  e = fl(b - d)
7  if (|2 * b| < |d|)
8      s = a, e = b
9  return (s,e)
```

There is not a lot of difference between our algorithm and those proposed by Priest except that the instruction of line 7 relaxes the dependencies which allows an increasing in performance. Figure 3 shows how the instructions are run on the CELL. The cycle number is equal to 20 and should be compared with the 29 of the Priest algorithm.

The proof of this algorithm correctness is presented in the next section.

```
1  TwoSum-toward-zero2(a,b)                cycles
2    comp = spu_cmpabsgt(b,a)               12
3    a = spu_sel(a, b, comp)                 -34
4    b = spu_sel(b, a, comp)                   45
5    s = spu_add(a , b)                    012345
6    d = spu_sub(s , a)                         -678901
7    e = spu_sub(b , d)                            ----234567
8    tmp = spu_mul(2 , b)                       789012
9    comp = spu_cmpabsgt(d, tmp)                      34
10   s = spu_sel(s, a, comp)                           -56
11   e = spu_sel(e, b, comp)                             --89
12   return s,e)
```

Fig. 3. Implementation on the CELL processor of the `TwoSum-toward-zero2` algorithm and its instructions running sequence. Cycle cost: 20.

6 Proof

This section explains the proof in the rounding mode toward zero of the `TwoSum-toward-zero2` and the `TwoProductFMA` algorithms. The lines in the proof refer to the algorithm and not to its implementation on the CELL processor.

6.1 The Correctness Proof for the `TwoSum-Toward-Zero2` Algorithm

Let a and b be two floating-point numbers. After the two instructions of line 2 and 3 of algorithm `TwoSum-toward-zero2`, we have $|a| \geq |b|$. The proof will take into account the case $a > 0$. For the case $a < 0$, the proof is very similar.

When $a > 0$, we will study three cases carefully: $b \geq 0$, $-a \leq b \leq -a/2$ and $-a/2 < b < 0$.

Case b ≥ 0

It is clear that $a + b > 0$. In rounding mode toward zero, $\mathrm{err}(a + b) \geq 0$ and $a + b = \mathrm{fl}(a + b) + \mathrm{err}(a + b)$ so we deduce that $b \leq a \leq \mathrm{fl}(a + b) \leq a + b$ and $0 \leq \mathrm{err}(a + b) \leq b$.

Let b be equal to $h \times \mathrm{ulp}(b)$ with h a positive integer. If $a > b > 0$ then $\mathrm{ulp}(a) = n \times \mathrm{ulp}(b)$ implies $a = k \times \mathrm{ulp}(b)$ with k a positive integer. From line 4 of the `TwoSum-toward-zero2` algorithm, $s = \mathrm{fl}(a + b) \geq b$ hence $s = \mathrm{fl}(a + b) = l \times \mathrm{ulp}(b)$, l being a positive integer. As a consequence

$$\begin{aligned}\mathrm{err}(a + b) &= a + b - \mathrm{fl}(a + b),\\ &= k \times \mathrm{ulp}(b) + h \times \mathrm{ulp}(b) - l \times \mathrm{ulp}(b),\\ &= (h + k - l) \times \mathrm{ulp}(b),\\ &= m \times \mathrm{ulp}(b).\end{aligned}$$

Moreover $0 \leq \mathrm{err}(a + b) \leq b$, hence $\mathrm{err}(a + b)$ is representable. This is a consequence of Lemma 1.

From line 5 of the `TwoSum-toward-zero` algorithm, it holds

$$\begin{aligned}d &= \mathrm{fl}(s - a),\\ &= \mathrm{fl}(a + b - \mathrm{err}(a + b) - a),\\ &= \mathrm{fl}(b - \mathrm{err}(a + b)),\\ &= \mathrm{fl}((h - m) \times \mathrm{ulp}(b)).\end{aligned}$$

As we have $0 \leq \mathrm{err}(a + b) \leq b$, it follows that $0 \leq b - \mathrm{err}(a + b) \leq b$ and therefore $b - \mathrm{err}(a+b) = (h-m) \times \mathrm{ulp}(b)$ is representable and $b = (h-m) \times \mathrm{ulp}(b)$ is the exact result.

To conclude

$$\begin{aligned}e &= \mathrm{fl}(b - d),\\ &= \mathrm{fl}(h \times \mathrm{ulp}(b) - (h - m) \times \mathrm{ulp}(b)),\\ &= \mathrm{fl}(m \times \mathrm{ulp}(b)),\\ &= \mathrm{err}(a + b),\end{aligned}$$

so e is the exact result. Moreover

$$\begin{aligned} |d| &= (h-m) \times \mathrm{ulp}(b), \\ &< h \times \mathrm{ulp}(b), \\ &< b, \\ &< |2b|, \end{aligned}$$

As a consequence, the comparison of line 7 of `TwoSum-toward-zero2` algorithm is not satisfied. Then the return result is: $(s = \mathrm{fl}(a+b), e = \mathrm{err}(a+b))$.

Case $-a \leq b \leq -a/2$

We then have $1/2 \leq -b/a \leq 1$. As consequence, $a + b = a - (-b)$ is representable (by Lemma 2). So $s = \mathrm{fl}(a+b) = a+b$, $d = \mathrm{fl}(s-a) = b$ and $e = \mathrm{fl}(b-d) = 0$. Then $d = b$ so the inequality of line 7 of the `TwoSum-toward-zero2` algorithm is not satisfied. The following result is returned: $(s = a+b, e = 0)$.

Case $-a/2 < b < 0$

Hence $a > a+b > a/2 > |b| > 0$ and so $\mathrm{err}(a+b) > 0$. We know that $a/2$ is a representable floating-point, so we have $\mathrm{fl}(a+b) \geq a/2$. It follows that $a > s \geq a/2$, $1/2 \leq s/a < 1$. Then $s - a$ is representable and so:

$$\begin{aligned} d &= \mathrm{fl}(s-a), \\ &= s-a, \\ &= a+b-\mathrm{err}(a+b)-a, \\ &= b-\mathrm{err}(a+b), \end{aligned}$$

and

$$\begin{aligned} e &= \mathrm{fl}(b-d), \\ &= \mathrm{fl}(b-(b-\mathrm{err}(a+b))), \\ &= \mathrm{fl}(\mathrm{err}(a+b)). \end{aligned}$$

As $a > a+b > |b|$, we can deduce $a > s = fl(a+b) \geq |b|$, $a = h \times \mathrm{ulp}(b)$, $s = k \times \mathrm{ulp}(b)$, $b = -l \times \mathrm{ulp}(b)$, h, k, l being positive integers with $h > k > l > 0$. It follows that

$$\begin{aligned} \mathrm{err}(a+b) &= a+b-\mathrm{fl}(a+b), \\ &= h \times \mathrm{ulp}(b) - l \times \mathrm{ulp}(b) - k \times \mathrm{ulp}(b), \\ &= (h-l-k) \times \mathrm{ulp}(b). \end{aligned}$$

As $b < 0$ and $\mathrm{err}(a+b) \geq 0$ the comparison of line 7 can be rewritten as follows:

$$\begin{aligned} |2b| &< |d|, \\ &< |b-\mathrm{err}(a+b)|, \end{aligned}$$

and so

$$2 * b > b - \mathrm{err}(a+b),$$
$$|b| < \mathrm{err}(a+b).$$

If the comparison of line 7 is satisfied, the returned result is $(s = a, e = b)$. As $0 < a + b < a$ we have

$$|e| = |b| < \mathrm{err}(a+b) < \mathrm{ulp}(\mathrm{fl}(a+b)) \leq \mathrm{ulp}(a) = \mathrm{ulp}(s).$$

It is in that case that the error is not representable and so $s \neq \mathrm{fl}(a+b)$.

If the comparison of line 7 is not satisfied, that means that $|b| \geq \mathrm{err}(a+b) \geq 0$. Moreover $\mathrm{err}(a+b) = (h-l-k) \times \mathrm{ulp}(b)$, by Lemma 1 $\mathrm{err}(a+b)$ is representable. Hence $e = fl(\mathrm{err}(a+b)) = \mathrm{err}(a+b)$. Therefore the returned result by this algorithm is $(s = \mathrm{fl}(a+b), e = \mathrm{err}(a+b))$.

In both cases, the equality $s + e = a + b$ and the inequality $e < \mathrm{ulp}(s)$ are always correct if $s \neq 0$. So, the couple (s, e) is the exact transformation of the sum of a and b.

6.2 The Correctness Proof for the `TwoProductFMA` Algorithm

Let a and b be two floating-point numbers of t-bits. They can be written as $a = s_1 \times 1m_1 \times 2^{e1-t}$, $b = s_2 \times 1m_2 \times 2^{e2-t}$ with $2^t \leq 1m_1, 1m_2 < 2^{t+1}$.

The product $a \times b$ is equal to $(s_1 \times s_2) \times (1m_1 \times 1m_2) \times 2^{e1+e2-2t}$ As $2^t \leq 1m_1, 1m_2 < 2^{t+1}$, then we have $2^{2t} \leq 1m_1 \times 1m_2 < 2^{2t+2}$.

The intermediate result of the product $a \times b$ is a floating-point of $(2t+1)$-bits without taking into account the first bit. In the rounding mode toward zero, the computed result of $a \times b$ is represented exactly by the $t+1$ first bits of the intermediate result. Then the subtraction of $\mathrm{fl}(a \times b)$ by $a \times b$ is exactly the $(t+1)$ last bits of the intermediate result. That means that $\mathrm{err}(a \times b)$ is representable by a floating-point of t-bits and that $a \times b - \mathrm{fl}(a \times b) = \mathrm{err}(a \times b)$. This function is usable with two rounding modes: to the nearest and toward zero.

7 Performance Measurements

The performances have been measured on the sum of two vectors of 64 floating-point numbers. To have a accurate estimate, a sum of two 64 elements vector have been done 10^7 times on 1 SPE, without memory exchange with the main memory.

Inside both code it is necessary to copy the data to registers. An empty program which contents only the load and store of the registers has been written. The cost of this part is around 10 cycles. In practice, the performance measurements show clearly that our algorithm is better than those proposed by Priest. If we remove the number of cycle due to the load and store of the registers, we find the theoretical performance. The performance of `TwoSum-toward-zero` and the `TwoSum-toward-zero2` algorithm are given in Table 3.

Table 3. Performance of the `TwoSum-toward-zero` and the `TwoSum-toward-zero2` algorithms on a CELL processor

Algorithm	computation time in second	performance (MFLOPS)	cycle/operation
`TwoSum-toward-zero`	7.93	80.7	39.65
`TwoSum-toward-zero2`	6.13	104.4	30.65
Only load-store registers	2.15	-	10.75

8 Conclusion

In this paper, we have proposed an improvement of `TwoSum-toward-zero` algorithm which reduces the variable dependencies. It allows a better implementation on processors which use pipeline instructions.

Future work: the next step consists in using this algorithm to implement algorithms which use EFT on processors which compute only in rounding mode toward zero.

Acknowledgements

The authors are very grateful to the CINES (Centre Informatique National de l'Enseignement Supérieur, Montpellier, France) for providing us access to their CELL blades.

References

1. IEEE Computer Society: IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985. Institute of Electrical and Electronics Engineers, New York (1985); Reprinted in SIGPLAN Notices 22(2), 9–25 (1987)
2. Li, X.S., Demmel, J.W., Bailey, D.H., Henry, G., Hida, Y., Iskandar, J., Kahan, W., Kang, S.Y., Kapur, A., Martin, M.C., Thompson, B.J., Tung, T., Yoo, D.J.: Design, implementation and testing of extended and mixed precision BLAS. ACM Trans. Math. Softw. 28, 152–205 (2002)
3. Ogita, T., Rump, S.M., Oishi, S.: Accurate sum and dot product. SIAM J. Sci. Comput. 26, 1955–1988 (2005)
4. Graillat, S., Louvet, N., Langlois, P.: Compensated Horner scheme. Research Report 04, Équipe de recherche DALI, Laboratoire LP2A, Université de Perpignan Via Domitia, France, 52 avenue Paul Alduy, 66860 Perpignan cedex, France (2005)
5. Rump, S.M., Ogita, T., Oishi, S.: Accurate floating-point summation part I: Faithful rounding. Technical Report 07.1, Faculty for Information and Communication Sciences, Hamburg University of Technology (2007)
6. Rump, S.M., Ogita, T., Oishi, S.: Accurate floating-point summation part II: Sign, K-fold faithful and rounding to nearest. Technical Report 07.2, Faculty for Information and Communication Sciences, Hamburg University of Technology (2007)

7. Priest, D.M.: On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations. Ph.D thesis, Mathematics Department, University of California, Berkeley, CA, USA (1992), ftp://ftp.icsi.berkeley.edu/pub/theory/priest-thesis.ps.Z
8. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the cell multiprocessor. IBM J. Res. Dev. 49(4/5), 589–604 (2005)
9. Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., Yelick, K.: The potential of the CELL processor for scientific computing. In: CF 2006: Proceedings of the 3rd conference on Computing frontiers, pp. 9–20. ACM Press, New York (2006)
10. Dekker, T.J.: A floating-point technique for extending the available precision. Numer. Math. 18, 224–242 (1971)
11. Knuth, D.E.: The Art of Computer Programming, 3rd edn. Seminumerical Algorithms, vol. 2. Addison-Wesley, Reading (1998)
12. Nievergelt, Y.: Scalar fused multiply-add instructions produce floating-point matrix arithmetic provably accurate to the penultimate digit. ACM Trans. Math. Software 29, 27–48 (2003)
13. Jacobi, C., Oh, H.J., Tran, K.D., Cottier, S.R., Michael, B.W., Nishikawa, H., Totsuka, Y., Namatame, T., Yano, N.: The vector floating-point unit in a synergistic processor element of a cell processor. In: ARITH 2005: Proceedings of the 17th IEEE Symposium on Computer Arithmetic, Washington, DC, USA, pp. 59–67. IEEE Computer Society, Los Alamitos (2005)