Extended precision with a rounding mode toward zero environment. Application to the Cell processor

Hong Diep Nguyen, Stef Graillat and Jean-Luc Lamotte*

CNRS, UMR 7606, LIP6,
Université Pierre et Marie Curie,
4 place Jussieu,
F-75252, Paris cedex 05, France
Email: hong.diep.nguyen@ens-lyon.fr
Email: stef.graillat@lip6.fr
Email: Jean-Luc.Lamotte@lip6.fr
*Corresponding author

Abstract: In the field of scientific computing, the exactness of the calculation is of prime importance. That leads to efforts made to increase the precision of the floating point algorithms. One of them is to increase the precision of the floating point number to double or quadruple the working precision. The building block of these efforts is the Error-Free Transformations (EFT). In this paper, we develop EFT operations in truncation rounding mode optimised for the Cell processor. They have been implemented and used in double precision library using only single precision numbers. We compare the performance of our library with the native one in double precision on vectors operations. In the best case, the performance of our library is very closed to the standard double precision implementation. The work could be easily extended to obtain quadruple precision.

Keywords: extended precision; rounding mode toward zero; Cell processor.

Reference to this paper should be made as follows: Nguyen, H.D., Graillat, S. and Lamotte, J-L. (2009) 'Extended precision with a rounding mode toward zero environment. Application to the Cell processor', *Int. J. Reliability and Safety*, Vol. 3, Nos. 1/2/3, pp.153–173.

Biographical notes: Hong Diep Nguyen is pursuing his PhD at the Ecole Normale Superieure de Lyon. He received his Master's degree in Computer Science from Université Pierre et Marie Curie in September 2007. His main research interests are reliable computing and computer arithmetic.

Stef Graillat received his PhD in Computer Science from Université de Perpignan, France, in 2005. He has been an Associate Professor of Computer Science at the Laboratory LIP6 of Université Pierre et Marie Curie since September 2006. His research interests are computer arithmetic and reliable computing.

Jean-Luc Lamotte received his PhD in Computer Science from Université de Caen, France, in 1992. He was appointed Associate Professor in 1995 in the Computer Science Laboratory (LIP6) of the Université Pierre et Marie Curie, and Professor in 2006. His research interests are reliable computing and new architectures for numerical computing.

Copyright © 2009 Inderscience Enterprises Ltd.

1 Introduction

The Cell processor, developed jointly by Sony, Toshiba, and IBM, provides a great power of calculation with a peak performance in single precision of 204.8 Gflop/s. This performance is obtained with a set of SIMD processors which use single precision floating point numbers with rounding mode toward zero.

Most scientific computation utilises floating point arithmetic. For a growing number of computations, much higher precision is needed, especially for applications which carry out very complicated and enormous tasks in scientific fields, for example[1]:

- Quantum field theory
- Supernova simulation
- Semiconductor physics
- Planetary orbit calculations
- Experimental and computational mathematics, etc.

Even if other techniques, methods or algorithms are employed to increase the accuracy of numerical results, some extended precision is still required to avoid severe numerical inaccuracies.

In this paper, we will study how to implement the double working precision library named single-single on the SPEs (Synergistic Processing Elements), which are the workhorse processors of the Cell. Our approach is similar to those used by Hida et al. (2001) for the quad-double precision arithmetic in the rounding mode to the nearest. The next Cell generation will provide powerful computing power in double precision with a rounding toward zero. Our library will be easily fit into double-double library which will emulate the quad precision.

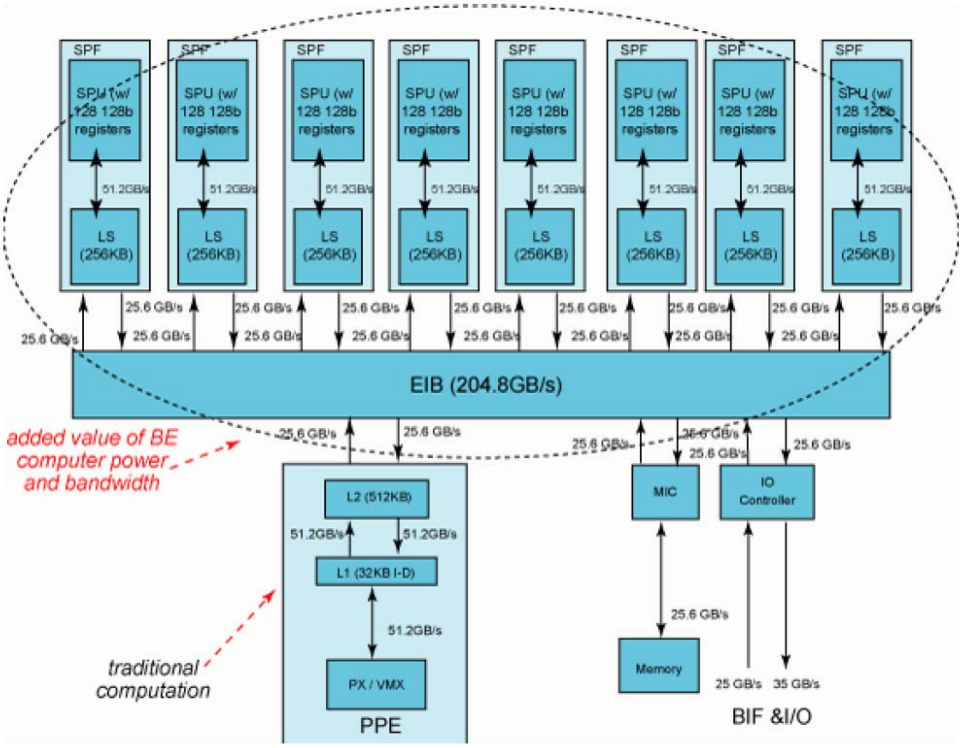
This paper begins with a brief introduction to the Cell processor. Then we propose algorithms for the operators (+, −, ×, /) of extended precision based on the error-free transformations for the rounding mode toward zero. The next section is devoted to the implementation of the single-single library on the SPE by taking into account the advantages of the SIMD characteristics, among which the most important are the fully pipelined single precision instructions set and the FMA (Fused Multiply-Add). Finally, the numerical experiments and the test results showing the library performance are presented. The code of the library is available at http://pequan.lip6.fr/~lamotte/software/extendedprecision/.

2 Introduction to the Cell processor

The Cell processor (Kahle et al., 2005; Williams et al., 2006) is composed of one 'Power Processor Element' (PPE) and eight 'Synergistic Processing Elements' (SPEs). The PPE and SPEs are linked together by an internal high-speed bus called 'Element Interconnect Bus' (EIB) (see Figure 1).

The PPE is based on the Power Architecture. Despite its important computing power, in practical use, it only serves as a controller for the eight SPEs, which perform most of the computational workload.

Figure 1 Cell processor architecture (taken from IBM website) (see online version for colours)



The SPE is composed of a 'Synergistic Processing Unit' (SPU) and a 'Memory Flow Controller' (MFC), which is devoted to memory transfer via the DMA access. The SPE contains an SIMD processor for single and double precision (Jacobi et al., 2005; Gschwind et al., 2006), which can perform four simultaneous operations in single precision or two operations in double precision. It supports all the four rounding modes for the double precision and only the rounding mode toward zero for the single precision.

The instruction set in single precision of the SPE is fully pipelined; one instruction can be issued for each clock cycle. It is based on the FMA function, which calculates the term $a*b+c$ in one operation and one rounding. With a frequency of 3.2 GHz, each SPE can achieve the performance of 2 × 4 × 3.2 = 25.6 GFLOPs on single precision numbers.

For the double precision, the instruction set is not fully pipelined. It is only possible to issue one instruction for each seven cycles, so the peak performance of each SPE for the double precision is 2 × 2 × 3.2/7 = 1.8 GFLOPs.

Each SPE has a 'Local Storage' (LS) of 256 KB for both data and code. In the opposite of the cache memory management, there is no mechanism to load data in the LS. It is up to the programmer to explicitly transfer data via DMA function calls. The SPE have a large set of registers (128 128-bit registers) which can be used directly by the program, avoiding the load-and-store time.

3 Floating-point arithmetic and extended precision

In this section, we briefly introduce the floating point arithmetic and the method to extend the precision. In this paper, due to the specific environment of the Cell processor, we work only with the rounding mode toward zero.

In a computer, the set of floating point numbers denoted F is used to represent real numbers. A binary floating point number is represented as

$$x = (\pm)\underbrace{1.x_1 \ldots x_{p-1}}_{mantissa} \times 2^e, x_i \in \{0,1\},$$

with p the precision and e the exponent of x. We use $\varepsilon = 2^{1-p}$ as the machine precision, and the value corresponding to the last bit of x is called *unit in the last place*, denoted $ulp(x)$, and $ulp(x) = 2^{e-p+1}$.

In the case of the Cell's single precision, $p = 24$ and $\varepsilon = 2^{-23}$. These values are similar to the IEEE 754 single precision format (IEEE, 1985). These values will be used throughout this paper.

Let x and y be two floating point numbers and let $\circ$ be a floating point operation $\left(\circ \in \{+,-,\times,/\}\right)$. It is clear that $(x \circ y)$ is a real number, but in most cases it is not representable by a floating point number. Let $fl(x \circ y)$ be the representative floating point number of $(x \circ y)$ obtained by a rounding. The difference $(x \circ y) - fl(x \circ y)$ corresponds to the rounding error denoted $err(x \circ y)$.

Given a specific machine precision, the precision of calculation can be increased by software. Instead of using a floating point number, multiple floating point numbers can be used to represent multiple parts of a real number. This is the idea of the extended precision. In our case, a single-single is defined as follows:

Definition 1: *A single-single is a non-evaluated sum of two single precision floating point numbers. The single-single represents the exact sum of these two floating point numbers:*

$$a = a_h + a_l.$$

There may be multiple couples of two floating point numbers whose sums are equal. To ensure a unique representation, a_h and a_l should have the same sign and should satisfy

$$|a_l| < ulp(a_h). \tag{1}$$

To implement the extended precision, we have to calculate the error produced by single precision operations using the error-free transformations presented below.

3.1 The error-free transformations (EFT)

Let x and y be two floating point numbers and $\circ$ be a floating point operation. The error-free transformations are intended to calculate the rounding error incurred by this operation. The EFTs transform $(x \circ y)$ into a couple of two floating point numbers (r, e) so that

$$r \approx x \circ y \text{ and } r + e = x \circ y.$$

3.1.1 Accurate sum

There are two main algorithms for the accurate sum of two floating point numbers. For example, for the rounding mode to nearest, there is the algorithm proposed by Knuth (1998), which uses six standard operations, or the algorithm proposed by Dekker (1971), which uses only three standard operations, but with the assumption on the order between the absolute values of two input numbers.

In this paper, we focus only on the rounding mode toward zero, so, it is necessary to adapt these algorithms. Priest (1992) has proposed an algorithm for an accurate sum using a rounding mode toward zero. To better use the pipelines, we proposed another algorithm.

Algorithm 2: *Error-free transformation for the sum with rounding toward zero.*

```
Two-Sum-toward-zero2 (a, b)
   if (|a| < |b|)
      swap (a , b)
   s = fl (a + b)
   d = fl(s - a)
   e = fl (b - d)
   if (|2 * b| < |d|)
      s = a, e = b
return (s, e)
```

The exactness of the proposed algorithm is provided in the following theorem:

Theorem 3: *Let a and b be two floating point numbers. The result (s, e) = Two-Sum-toward-zero2(a, b) satisfies:*

$$s + e = a + b,$$
$$|e| < ulp(s).$$

The proof of all the theorems of this paper can be found in Nguyen (2007) (in French).

3.1.2 Accurate product

The calculation of the error-free transformation for the product is much more complicated than the sum (Dekker, 1971), but if the processor has a FMA (Fused Multiply-Add) which calculates the term $a * b + c$ in one operation, the classic algorithm for the product can be used.

Algorithm 4: *The error-free transformation for the product of two floating point numbers.*

```
Two-Product-FMA (a, b)
p = fl (a * b)
e = fma (a, b,-p)
return (p, e)
```

This algorithm is applicable for all the four rounding modes. The basic operation on the SIMD unit of the SPE being a FMA, our library implements this algorithm.

4 Basic operations of single-single

4.1 Renormalisation

Using the rounding toward zero EFTs, we can implement the basic operations for the single-single. Most of the algorithms described hereafter often produce an intermediate result of two overlapping floating point numbers. To respect the definition of the normalisation (1), it is necessary to apply a renormalisation step to transform these two floating point numbers into a normalised single-single. The following function is proposed:

```
1  Renormalise2-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  return (s, e)
```

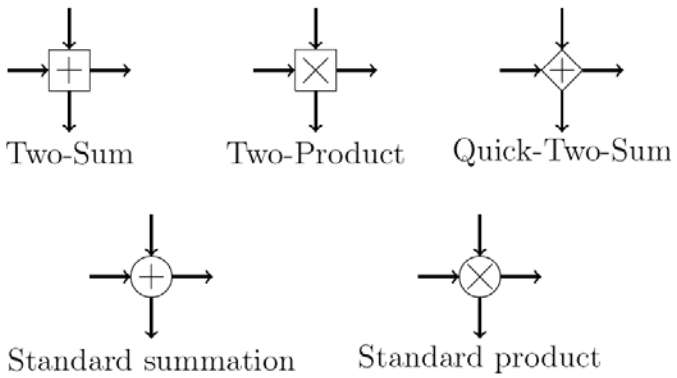
Renormalisation is the same for the rounding mode toward zero and to the nearest, but in the case of the rounding mode toward zero, it is not possible to give an exact result. The following theorem provides an error bound for this algorithm.

Theorem 5: *Let a and b be two floating point numbers. The result returned by* `Renormalise2-toward-zero` *is a pair of two floating point numbers (s, e) which satisfies*

- *s and e have the same sign, and* $|e| < ulp(s)$, *and*
- $a + b = s + e + \delta$, *where* δ *is error of normalisation, and* $|\delta| \leq \frac{1}{2} \times \varepsilon^2 \times |a+b|$.

As we will see later, this error is much smaller than the errors produced by the following algorithms. To describe them, we use the notations in Figure 2.

Figure 2 Notations

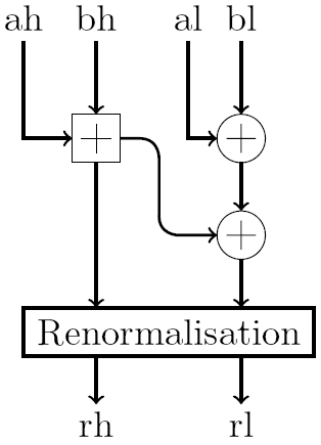


4.1.1 Addition

Figure 3 represents the algorithm for the addition of two single-singles *a* and *b*. The source code follows:

```
1  add_ds_ds (ah,  al,  bh,  bl)
2     (th,  tl) = Two-Sum-toward-zero  (ah,  bh)
3     tll = fl (al  +  bl)
4      tl = fl(tl  +  tll)
5    (rh, rl) = Renormalise2-toward-zero  (th,  tl)
6  return  (rh,  rl)
```

Figure 3 Algorithm for the addition of two single-singles



With two sums, a `Two-Sum-toward-zero` and a `Renormalise2-toward-zero`, the cost of the `add_ds_ds` algorithm is 11 FLOPs. The following theorem provides an error bound for this algorithm.

Theorem 6: *Let $a_h + a_l$ and $b_h + b_l$ be two input single-singles and $r_h + r_l$ be the result of* `add_ds_ds`*. The error δ produced by this algorithm satisfies*

$$r_h + r_l = (a_h + a_l) + (b_h + b_l) + \delta,$$
$$|\delta| < max(\varepsilon \times |a_l + b_l|,\ 6 \times \varepsilon^2 \times |a_h + a_l + b_h + b_l|.$$

4.1.2 The subtraction

The subtraction of two single-singles $a - b$ is implemented by a sum $a + (-b)$. To compute the opposite of a single-single, it is just necessary to get the opposite of the floating point components. Therefore, the algorithms for the addition and the subtraction are similar.

4.1.3 Product

The product of two single-singles a and b can be considered as the product of two sums $a_h + a_l$ and $b_h + b_l$ so the exact product has four components:

$$\begin{aligned} p &= (a_h + a_l) \times (b_h + b_l) \\ &= a_h \times b_h + a_l \times b_h + a_h \times b_l + a_l \times b_l. \end{aligned}$$

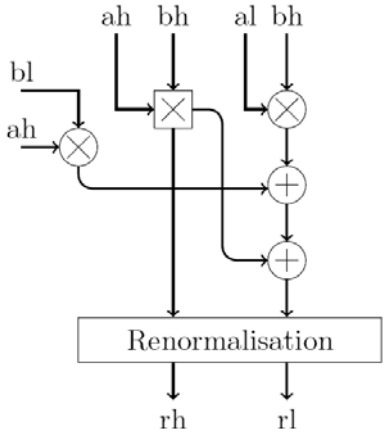
Considering $a_h \times b_h$ as a term of order $\mathcal{O}(1)$, this product consists of one term $\mathcal{O}(1)$, two terms $\mathcal{O}(2)$, and one term $\mathcal{O}(3)$. To decrease the complexity of the algorithm, the terms of order below $\mathcal{O}(2)$ will not be taken into account. Additionally, using the EFT for the product, $a_h \times b_h$ can be transformed exactly into two floating point numbers of orders $\mathcal{O}(1)$ and $\mathcal{O}(2)$, respectively. Hence, the product of two single-singles can be approximated by:

$$p \approx \underbrace{fl(a_h \times b_h)}_{\mathcal{O}(1)} + \underbrace{(err(a_h \times b_l) + a_l \times b_h + a_h \times b_l)}_{\mathcal{O}(2)}.$$

This approximation can be translated into the following algorithm described in Figure 4.

```
1  mul_ds_ds (ah,  al,  bh,  bl)
2     (th, tl) = Two-Product-FMA (ah, bh)
3     tll = fl (al * bh)
4     tll = fl (ah * bl + tll)
5      tl = fl (tl + tll)
6     (rh, rl) = Renormalise2-toward-zero  (th, tl)
7  return  (rh,  rl)
```

Figure 4 Algorithm for the product of two single-singles



The error bound of the algorithm `mul_ds_ds` is provided by the following theorem.

Theorem 7: *Let $a_h + a_l$ and $b_h + b_l$ be two single-singles. Let $r_h + r_l$ be the result returned by the algorithm* `mul_ds_ds` *applying to $a_h + a_l$ and $b_h + b_l$. The error δ of this algorithm satisfies*

$$|(r_h + r_l) - (a_h + a_l) \times (b_h + b_l)| < 8 \times \varepsilon^2 \times |(a_h + a_l) \times (b_h + b_l)|.$$

4.1.4 The division

The division of two single-singles is calculated by the classic division algorithm.

Let $a = (a_h, a_l)$ and $b = (b_h, b_l)$ be two single-singles. To calculate the division of a by b, we calculate the approximate quotient, $q_h = a_h/b_h$.

Then we calculate the residual $r = a - q_h \times b$, which allows us to calculate the correction term $q_l = r/b_h$.

```
1  div_ds_ds (a, b)
2       qh = fl(ah / bh)
3       tmp1 = fl (ah - qh * bh)
4       tmp2 = fl (al - qh * bl)
5       r = fl (tmp1 + tmp2)
6       ql = fl(r / bh)
7      (qh,ql) = Renormalise2-toward-zero (qh , ql)
8  return  (qh, ql)
```

The following theorem provides an error estimate for this algorithm.

Theorem 8: *Let $a = (a_h, a_l)$ and $b = (b_h, b_l)$ be two single-singles, ε the machine precision and ε_1 the error bound for the single precision division with $\mathcal{O}(\varepsilon_1) = \mathcal{O}(\varepsilon)$. The error of the algorithm* `div_ds_ds` *is bounded by:*

$$\left|div\_ds\_ds(a,b)) - a/b\right| < \left[\varepsilon^2 \times \left(6.5 + 7 \times \varepsilon_1/\varepsilon + 2 \times (\varepsilon_1/\varepsilon)^2\right) + \mathcal{O}\left(\varepsilon^3\right)\right] \times |a/b|.$$

In most of cases we have $\varepsilon_1 = \varepsilon$. In this case, the error bound of this algorithm is:

$$\left|q - a/b\right| < \left[15.5 \times \varepsilon^2 + \mathcal{O}\left(\varepsilon^3\right)\right] \times |a/b|.$$

This inequality means that our division algorithm of two single-singles is accurate to 42 bits on a maximum of 48 bits. The accuracy of this algorithm can be increased by calculating another correction term q_2, but doing so has a great impact on the performance. The execution time over-cost cost more than twice because of two factors:

- the calculation of $(a_h + a_l) - (b_h + b_l) \times (q_h + q_l)$, which is one single-single multiplication and one single-single subtraction, and
- the renormalisation of three floating point numbers, which requires two accurate sums and one renormalisation of two floating point numbers (Nguyen, 2007).

5 Implementation

The SPE (Synergistic Processor Element) of the Cell processor contains a 32-bit 4-way SIMD processor together with a large set of 128 128-bit registers. It can perform the operations on the vectors of `16 char/unsigned char, 4 int/unsigned int, 4 float, or 2 double`.

The operations on scalars are implemented by using the vector operations. In this case, only one operation is performed on the preferred slot instead of 4 on vectors. For this reason, we implement only the vector operations for the single-singles.

Table 1 lists some Cell-specific instructions used in the sequel to write algorithms. As we work with vectors, the operations are performed componentwise.

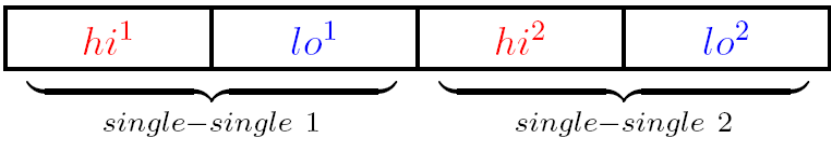
Table 1 Cell-specific instructions

Instruction	*Explanation*
$d = spu\_add(a, b)$	$d = a + b$.
$d = spu\_sub(a, b)$	$d = a - b$.
$d = spu\_mul(a, b)$	$d = a \times b$.
$d = spu\_madd(a, b, c)$	$d = a \times b + c$.
$d = spu\_msub(a, b, c)$	$d = a \times b - c$.
$d = spu\_nmsub(a, b, c)$	$d = -(a \times b - c)$.
$d = spu\_nmadd(a, b, c)$	$d = -(a \times b + c)$.
$d = spu\_re(a)$	For each element of vector a, an estimate of its floating-point reciprocal is computed. The resulting estimate is accurate to 12 bits.
$d = spu\_cmpabsgt(a, b)$	The absolute value of each element of vector a is compared with the absolute value of the corresponding element of vector b. If the element of a is greater than the element of b, all bits of the corresponding element of vector d are set to one; otherwise, all bits of the corresponding element of d are set to zero.
$d = spu\_sel(a, b, pattern)$	For each bit in the 128-bit vector *pattern*, the corresponding bit from either vector a or vector b is selected. If the bit is 0, the bit from a is selected; otherwise, the bit from b is selected. The result is returned in vector d.

5.1 Representation

A single-single is a pair of two floating point numbers so each vector of 128 bits contains two single-singles (Figure 5). Hence, the 128-bit register containing two single-single numbers could be seen as a vector of four floating point numbers.

Figure 5 A vector of 2 single-singles (see online version for colours)



5.2 Implementation of the error-free transformations

The EFT for the product is implemented simply by two instructions:

```
1  Two-Prod-FMA  (a,b)
2     p = spu_mul(a,b)
3     e = spu_msub(a, b, p)
4  return (p,e)
```

The algorithm of the EFT for the sum begins with a test and a swap. This test limits the possibility of parallelism. Hence, we first have to eliminate this test by the following procedure:

- evaluation of the condition. The result is a vector comp of type `unsigned int`, in which a value of zero means the condition holds, and a value of `FFFFFFFF` means opposite
- computation of the values of the two branches `val_1` (if the condition is satisfied) and `val_2` (if not)
- selection of the correct value according to the vector of condition by using the bit selection function:

$$d = spu\_sel(val\_2,\ val\_1,\ comp).$$

For each bit in the 128-bit vector *comp*, the corresponding bit from either vector val_2 or val_1 is selected. If the bit is 0, the bit from val_2 is selected; otherwise, the bit from val_1 is selected. The result is returned in vector *d*.

For example, the test and the *swap* can be coded as follows:

```
1    comp = spu_cmpabsgt (b, a)
2    hi = spu_sel(a, b, comp)
3    lo = spu_sel(b, a, comp)
```

Figure 6 gives a concrete example of this exchange.

Figure 6 Example of the exchange of two vectors

a	$a1$	$a2$	$a3$	$a4$
b	$b1 > a1$	$b2 < a2$	$b3 = a3$	$b4 > a4$
$comp = spu\_cmpabsgt(b, a)$				
	FFFFFFFF	00000000	00000000	FFFFFFFF
$hi = spu\_sel(a, b, comp)$				
	$b1$	$a2$	$a3$	$b4$
$lo = spu\_sel(b, a, comp)$				
	$a1$	$b2$	$b3$	$a4$

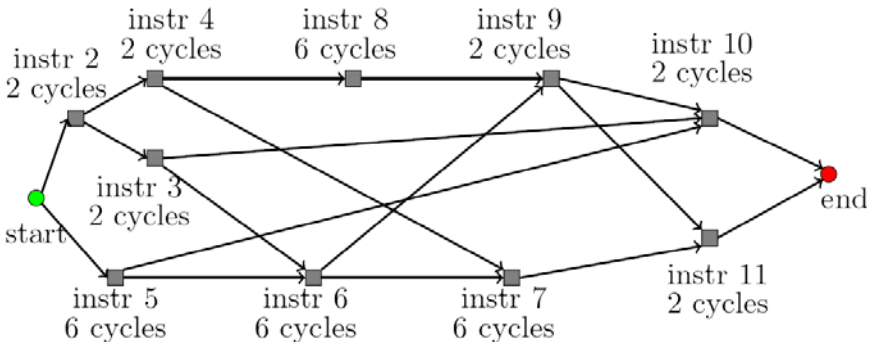
The `spu_cmpabsgt` and `spu_sel` instructions cost two clock cycles each. Moreover, since the instructions of lines 2 and 3 of this code are independent, they can be pipelined. Hence, these three instructions cost only five clock cycles, which is less than a single precision operation (six clock cycles for the FMA).

Applying the same procedure for the last conditional test of the algorithm `Two-Sum-toward-zero2`, this algorithm can be rewritten as follows:

```
1   Two-Sum-toward-zero2  (a,b)
2       comp = spu_cmpabsgt (b, a)
3       hi = spu_sel(a, b, comp)
4       lo = spu_sel(b, a, comp)
5       s = spu_add(a , b)
6       d = spu_sub(s, hi)
7       e = spu_sub(lo, d)
8       tmp = spu_mul(2, lo)
9       comp = spu_cmpabsgt (d, tmp)
10      s = spu_sel(s, hi, comp)
11      e = spu_sel(e, lo, comp)
12  return  (s, e)
```

Note that the addition of a and b does not change after the exchange. Hence, we choose to use $a + b$ instead of $hi + lo$ to avoid the instruction dependencies. More precisely, the three first instructions for the test and the exchange are independent of the instruction of line 5 which costs six cycles, so, they can be executed in parallel[2]. Figure 7 emphasises the full independencies of instructions. This algorithm costs 20 clock cycles, which is a little bit more than the execution time of three sequential single precision operations.

Figure 7 The dependencies between instructions of algorithm `Two-Sum-toward-zero` (see online version for colours)



5.3 Renormalisation

The implementation of algorithm `Renormalise2-toward-zero` is similar to the `Two-Sum-toward-zero2` algorithm but without the conditional test and the exchange at the end.

```
1  Renormalise2-toward-zero (a,b)
2      s = spu_add(a, b)
3      comp = spu_cmpabsgt (b, a)
4      hi = spu_sel(a, b, comp)
5      Lo = spu_sel(b, a, comp)
6      d = spu_sub(s, hi)
7      e = spu_sub(lo, d)
8  return (s, e)
```

With the same analysis as `Two-Sum-toward-zero2`, `Renormalise2-toward-zero` costs only 18 clock cycles. Now we will use these two functions to implement the arithmetic operators of single-singles.

5.4 Version 1

The natural version of single-single operations computes one operation on TWO single-singles. The SIMD processor allows us to manipulate simultaneously four 32-bit floating point numbers at the same time. When applying to vectors of single-singles, we can manipulate both the high and low components of these single-singles.

Using the `Two-Sum-toward-zero2` presented above, we calculate the sums and the rounding errors of two pairs of high components and also of two pairs of low components in the same time. The rounding errors of these two pairs of low components is computed, but not used by the algorithm.

Moreover, in the algorithms, it is necessary to compute operations between high and low components. This requires some extra operations to shuffle those components. Hence, the first version does not take full advantage of the SIMD processor. We have implemented the first version for the sum and the product of single-singles which are `add_ds_ds_2`, `mul_ds_ds_2` for two single-singles and cost 50 cycles and 49 cycles, respectively.

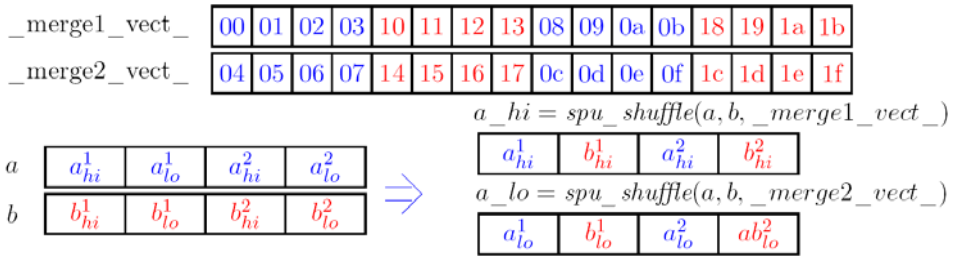
As we will see later in Section 6, these first versions as well as the versions presented below compute correct results except for overflow and underflow cases.

5.5 Version 2

The second version computes one operation on FOUR single-singles. It separates the high and the low components into two separate vectors (see Figure 8) by using the function `spu_shuffle` of SPE which costs four clock cycles. This solution makes possible a better optimisation of the pipelined instructions.

Then, the operators can be implemented by applying directly the algorithms presented above on four operands separated into four vectors.

Figure 8 Merging of two vectors (see online version for colours)



The intermediate result of these algorithms is also a pair of vectors which contain respectively the four high parts and the four low parts of the result. At the end of the algorithm, the result vectors should be built by shuffling the high and the low components.

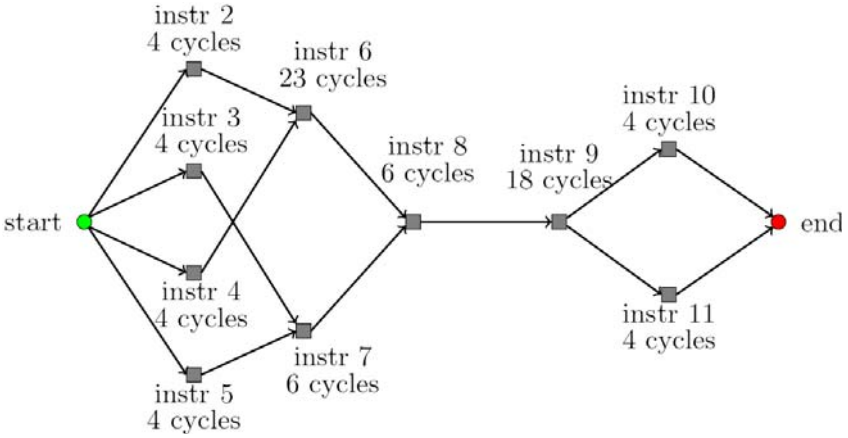
For example, version 2 for the sum of single-singles is[3]

```
1   add_ds_ds_4 (vect_a1, vect_a2, vect_b1, vect_b2)
2      a_hi = spu_shuffle (vect_a1, vect_a2, _merge1_vect_)
3      a_lo = spu_shuffle (vect_a1, vect_a2, _merge2_vect_)
4      b_hi = spu_shuffle (vect_b1, vect_b2, _merge1_vect_)
5      b_lo = spu_shuffle (vect_b1, vect_b2, _merge2_vect_)
6     (s, e) = Two-Sum-toward-zero (a_hi, b_hi)
7      t1 = spu_add(a_lo, b_lo)
8      tmp = spu_add(t1, e)
9      (hi, lo) = Renormalise2-toward-zero (s , tmp)
10     vect_c1 = spu_shuffle (hi, lo, _merge1_vect_)
11     vect_c2 = spu_shuffle (hi, lo, _merge2_vect_)
12  return (vect_c1, vect_c2)
```

Figure 9 shows the dependencies between instructions of this function. By using the tool `spu_timing` of IBM, the execution time of this function is **64 clock cycles** for four single-singles.

It is the same for the product of single-singles. We have successfully implemented version 2 of the product of single-singles, called `mul_ds_ds_4`, with an execution time of **60 clock cycles** for four single-singles. Although the first versions of the addition and the product are nearly equal (50 and 49 clock cycles, respectively), the product takes less operations than the addition so there are more idle clock cycles in the product implementation. Hence, when implementing the second version, we can save more clock cycles with the product operator.

Figure 9 The dependencies between instructions of `add_ds_ds_4` (see online version for colours)



The implementation of the division is more complicated. As described in the previous section, the division of single-singles `div_ds_ds` is based on the division in single precision, although the Cell processor does not support this kind of operation. It provides only a function to estimate the inverse of a floating point number called `spu_re`, which allows us to obtain a result precise up to 12 bits. Hence, to implement the division of single-singles, we first have to implement division in single precision.

The procedure to calculate the division of two 32-bit floating point numbers a and b is as follows:

1 calculate the inverse of b, and

2 multiply the inverse of b with a.

To improve the precision of the inversion, we use the iterative Newton's method, $inv_{i+1} = inv_i + inv_i \times (1 - inv_i \times b)$. We also use Newton's method for the multiplication, with $a \times inverse(b)$ being the initial value. The division in single precision can be written as

```
1  div (a, b)
2      tmp0 = spu_re(b)
3      rerr = spu_nmsub(tmp0, b, 1)
4      inv = spu_madd(rerr, tmp0, tmp0)
5      Rerr = spu_nmsub(inv, b, 1)
6      eerr = spu_mul(rerr, inv)
7      tmp = spu_mul(eerr, a)
8      q = spu_madd(a, inv, tmp)
9  return q
```

The precision of the algorithm `div` is provided by the following theorem.

Theorem 9: *Let a and b be two floating point numbers in single precision, ε being the machine precision. The relative error of the algorithm* `div` *is bounded by*

$$\left| div(a,b) - a/b \right| < \left[\varepsilon + \mathcal{O}\left(\varepsilon^2\right) \right] \times \left| a/b. \right|$$

Using the newly implemented single-precision division operator and the algorithm of division of single-singles presented above, we have implemented the function `div_ds_ds_4` which calculates four single-single divisions at the same time, at a cost of **111 clock cycles**.

5.6 Optimised algorithms

The versions 2 of the single-single operators performs four operations at the same time, and they have taken full advantage of the SIMD processor which provides an important performance of calculation. By using the `spu_timing` tool of IBM, we recognised that there still left many non-used clock cycles in the process of calculation of each operator.

We can use these non-used clock cycles by increasing the number of operations executed at the same time.

With the restricted local storage (only 256 KB for both the code and data), we choose to implement operations on EIGHT single-singles. This third version is considered as the optimal version in our library. The third version of the sum, the product and the division are named `add_ds_ds_8`, `mul_ds_ds_8`, `div_ds_ds_8` and cost respectively **72 cycles**, **63 cycles**, and **125 cycles** for eight single-singles. In comparison with the version 2 with only some supplementary clock cycles (for example eight cycles for the sum and three cycles for the product) we can execute eight single-single operations instead of four. It means that we have achieved a coarse gain with the final version in terms of performance.

Almost every clock cycle being used, there would be no gain from dealing with sixteen single-singles.

5.7 Theoretical results

On a Cell processor with a frequency of 3.2 GHz, its theoretical performances (without memory access problems) of the single-single are presented in Table 2.

Table 2 Theoretical results of the single-single library

Function	*Number of operations*	*Execution time*	*Performance*
`add_ds_ds_2`	2	50 cycles	0.128 GFLOPs
`add_ds_ds_4`	4	64 cycles	0.2 GFLOPs
`add_ds_ds_8`	8	72 cycles	0.355 GFLOPs
`mul_ds_ds_2`	2	49 cycles	0.130 GFLOPs
`mul_ds_ds_4`	4	60 cycles	0.213 GFLOPs
`mul_ds_ds_8`	8	63 cycles	0.406 GFLOPs
`div_ds_ds_4`	4	111 cycles	0.115 GFLOPs
`div_ds_ds_8`	8	125 cycles	0.2048 GFLOPs

6 Numerical simulations

6.1 Experimental results

To test the performance of the single-single library, we created a program which performs the basic operators on two large vectors of single-single and also on two large double precision vectors of the same size. To achieve the peak performance of the library, we use the third version of each operator. Double-buffering is used to hide data transfer time.

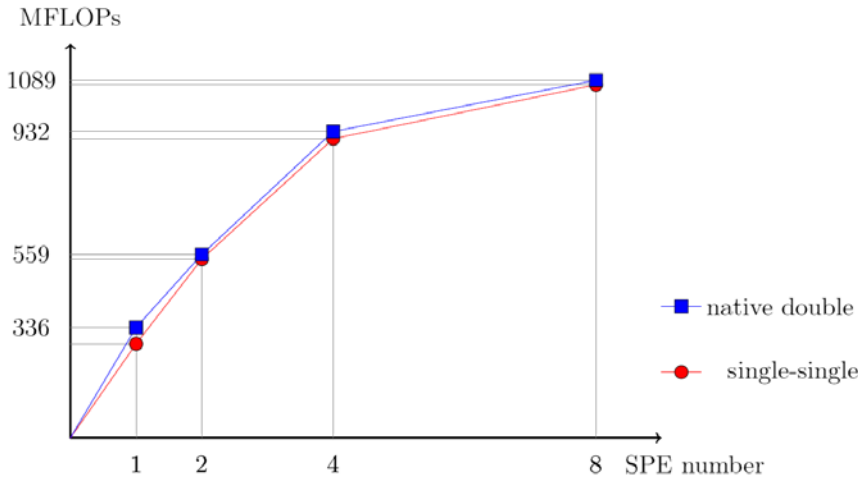
This program is executed on a IBM Cell Blade based at CINES, Montpellier, France. The CPU frequency is 3.2 GHz. The results obtained are listed in the Table 3.

Table 3 Real performances of the library single-single

Functions	*Theoretical performance*	*Experimental performance*
`Add_ds_ds_8`	355 MFLOPs	250.4 MFLOPs
`mul_ds_ds_8`	406 MFLOPs	287.2 MFLOPs
`div_ds_ds_8`	204 MFLOPs	166.4 MFLOPs

Figure 10 illustrates the performance of the addition on single-singles and on native double precision. Both have the same memory size. They are very close.

Figure 10 The performance of the library single-single: Addition (see online version for colours)



It is interesting to note that a performance limit is reached. The main limit for the performance is the memory access. The memory bandwidth on the Cell processor is equal to 25.6 GBytes/s, which corresponds to 6.4 GWords/s[4]. Each operation is performed on double words variables (single-single or double). That means that the bandwidth corresponds to a traffic of 3.2 G operands. For vector operations, there are two memory access to read the data and one memory access to write the data, so there are only 1.07 G operations. This explanation clearly shows that the peak performance is not reachable on large vectors operations due to the bandwidth limit.

The maximum performance with 64-bit floating point is not reached. In this case, the program measures mainly the memory transfer time. The native double operations are completely hidden. For the single-singles, the computing time of one operation is on the same order as the transfer memory necessary for one operation.

To have another comparison, another program is created which executes a large number of basic operators on a small number of data generated within the SPE without any data transfer. The execution time of the program is exactly the time of calculation. The results are presented in Table 4. The peak performance for the multiplication on the Cell processor is achieved for native double precision.

Table 4 Performance of the single-single library and of the double precision of the machine, without data transfer

Functions	*Theoretical performance (1 SPE)*	*Experimental performance (1 SPE)*	*Experimental performance (8 SPEs)*
`add_ds_ds_8`	355 MFLOPs	266 MFLOPs	2133 MFLOPs
`mul_ds_ds_8`	406 MFLOPs	320 MFLOPs	2560 MFLOPs
`div_ds_ds_8`	204 MFLOPs	172 MFLOPs	1383 MFLOPs
sum in double precision	914 MFLOPs	914 MFLOPs	7314 MFLOPs
product in double precision	914 MFLOPs	914 MFLOPs	7314 MFLOPs
division in double precision	(not supported)	86 MFLOPs	691 MFLOPs

With the single-singles numbers, it is not possible to achieve the same performance as with the native double precision. This is mainly due to two factors:

- the cost of the function call
- the transfer from the local memory to the registers.

6.2 Exactness

Let $a = (a_h, a_l)$ be a single-single. Following the definition of a single-single, a_h and a_l have the same sign, and that $|a_l| < ulp(a_h)$ which leads to a relation between their exponents: $EXP(a_h) \geq EXP(a_l) + 24$. Here, $EXP(\circ)$ denotes the exponent of a floating point number. Since the range of exponent for single precision floating point numbers is from −126 to 127, there will be 8515 possible pairs of exponents for a single-single.

Moreover, for each pair of exponents, there are two possible signs and $2^{23} * 2^{23}$ possible pairs of mantissas. Hence, it is really difficult to test thoroughly the exactness of the library.

To get an acceptable test, and to cover numerical phenomenons, such as overflow, underflow, and cancellation, we go through all possible pairs of exponents. For each exponent, we take only 16 values of the mantissa: four minimum values, four maximum values and eight medium values. Each pair of exponents will be tested for both two signs.

Calculation results of single-singles will be tested against calculation results of native double of the Cell processor.

Let $a = (a_h, a_l)$ and $b = (b_h, b_l)$ be two single-singles. Let $c = (c_h, c_l)$ be the result of the calculation (addition, product or division) over a and b. The corresponding reference double result rd is calculated by:

1 Addition: $rd = ((double)a_h + (double)b_h) + ((double)a_l + (double)b_l)$

2 Product: $rd = ((double)a_h*(double)b_h + ((double)a_h*(double)b_i + ((double)a_i*(double)b_h + ((double)a_l * (double)b_l))))$,

3 Division: $rd = ((double)a_h + (double)b_h)/((double)a_l + (double)b_l)$.

The absolute error and relative error are calculated by:

$$diff = (double)c_h + (double)c_l - rd,$$
$$rel\_diff = diff / rd.$$

A calculation result greater than or equal to 2^{128} will be considered as overflow and will be ignored. Furthermore, the test program behaves like following:

1 Addition: if $rel\_diff < 6 * 2^{-46}$ then the result is considered to be exact and returned value is the absolute value of the relative error *rel_diff*. Else, we calculate $rel\_diff2 = diff/(a_l + b_l)$ and compare this value to 2^{-23}. If $rel\_diff2 < 2^{23}$ than the result is normal too, and the returned value is *rel_diff2*. Finally, if these two conditions do not hold, than the result is abnormal and the returned value is the absolute error *diff*.

2 Product: if $rel\_diff < 2^{-43}$ then the result is considered to be exact and returned value is the absolute value of the relative error *rel_ diff*. If not, the result is considered as abnormal and returned value is the absolute value of absolute error *diff*.

3 Division: if $rel\_diff < 2^{-42}$ then the result is considered to be exact and returned value is the absolute value of the relative error *rel_ diff*. If not, the result is considered as abnormal and returned value is the absolute value of absolute error *diff*.

The three versions of each operation (addition, product and division) give the same results, which are summarised in Table 5.

Table 5 The exactness of single-single library

Operation	*Max relative error*	*Max absolute error*	*Max diff/($a_l + b_l$)*
Sum	8.603330e-14	1.175494e-38	1.249775e-07
Product	1.145182e-13	3.526483e-38	–
Division	2.138291e-13	1.689773e-38	–

As Table 5 reveals, the maximum absolute errors of non exact results of both three operations are of order 10^{-38}, which can be considered as errors incurred by underflow phenomenon.

These results allow us to state that, except for overflow and underflow exceptions, our library provides correct answers which are accurate to 42 bits with faithful rounding mode.

7 Conclusions and perspectives

This paper is based mostly on work of Hida et al. (2001) with some adaptations to the rounding mode toward zero and to the implementation environment of the Cell processor. First we propose an algorithm for the error-free transformation of the sum which is proved to be implemented effectively on the Cell processor. Then, we introduce method to develop the extended precision of single-single with basic operators sum, product and division. A large part of this paper is dedicated to the implementation of this library in exploiting the specific characteristics of the Cell processor, among which the most important are the truncation rounding, the SIMD processor and the fully pipelined instruction set. The performance and the precision of the implemented library are tested by running test programs on a real Cell processor with a frequency of 3.2 GHz.

In the future, this library could be completed by the treatment of numeric exceptions, by the binary operations, algebraic operations and transcendental operations.

Anticipating for the next Cell generation, we are developing the quad-single precision library. With the next generation of the Cell processor, we will be able to get easily:

- the quad precision implemented with double-double numbers with the methods of the single-single library, and
- the quad-double precision implemented with four double numbers with the methods of the quad-single library.

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. Also, we wish to thank the referees for their careful work, in particular, for their remark about exactness test, which led to great efforts to improve test results.

References

Dekker, T.J. (1971) 'A floating-point technique for extending the available precision', *Numerische Mathematik*, Vol. 18, pp.224–242.

Gschwind, M., Hofstee, H.P., Flachs, B., Hopkins, M., Watanabe, Y. and Yamazaki, T. (2006) 'Synergistic processing in Cell's multicore architecture', *IEEE Micro*, Vol. 26, No. 2, pp.10–24.

Hida, Y., Li, X.S. and Bailey, D.H. (2001) 'Algorithms for quad-double precision floating point arithmetic', *Proceedings of 15th IEEE Symposium on Computer Arithmetic*, IEEE Computer Society Press, Los Alamitos, CA, USA, pp.155–162.

IEEE (1987) *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*, Vol. 22, No. 2, pp.9–25.

Jacobi, C., Oh, H-J., Tran, K.D., Cottier, S.R., Michael, B.W., Nishikawa, H., Totsuka, Y., Namatame, T. and Yano, N. (2005) 'The vector floating point unit in a synergistic processor element of a Cell processor', *ARITH '05: Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, IEEE Computer Society, Washington, DC, USA, pp.59–67.

Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R. and Shippy, D. (2005) 'Introduction to the Cell multiprocessor', *IBM Journal of Research and Development*, Vol. 49, Nos. 4–5, pp.589–604.

Knuth, D.E. (1998) *The Art of Computer Programming*, *Volume 2*, *Seminumerical Algorithms*, 3rd ed., Addison-Wesley, Reading, MA, USA.

Nguyen, H.D. (2007) *Calcul précis et efficace sur le processeur Cell*, Master report. Available online at: http://www-pequan.lip6.fr/~graillat/papers/rapport_Diep.pdf

Priest, D.M. (1992, November) *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*, PhD Thesis, Mathematics Department, University of California, Berkeley, CA, USA. Available online at: ftp://ftp.icsi.berkeley.edu/pub/theory/priest-thesis.ps.Z

Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P. and Yelick, K. (2006) 'The potential of the Cell processor for scientific computing', *CF'06: Proceedings of the 3rd conference on Computing frontiers*, ACM Press, New York, NY, USA, pp.9–20.

Notes

1 http://crd.lbl.gov/~dhbailey/dhbtalks/dhb-pelz.pdf

2 On the SPE, there are two pipelines. The first one is devoted to numerical operations, the second one is for control and logical operations. The two pipelines can be used in parallel.

3 The SIMD unit computes on 128-bit vectors. The four single-singles values of a and b are cut into two parts to keep the register organisation.

4 One word is equal to four bytes. In our case, one word is a single floating point number.