Extended precision with a rounding mode toward zero environment. Application on the CELL processor

Hong Diep Nguyen (hong.diep.nguyen@ens-lyon.fr), Stef Graillat (stef.graillat@lip6.fr), Jean-Luc Lamotte (jean-luc.lamotte@lip6.fr)
CNRS, UMR 7606, LIP6,University Pierre et Marie Curie, 4 place Jussieu, 75252 Paris cedex 05, France

Abstract. In the field of scientific computing, the exactitude 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.

CELL processor is a microprocessor architecture jointly developed by a Sony, Toshiba, and IBM. Although its first major commercial application of Cell was in Sony's PlayStation 3 game console, it can provide a great potential for scientific computing with a peak single precision performance of 204.8 Gflop/s.

In this paper, we will do the study on how to implement the double working precision library, named single-single, on the SPEs (Synergistic Processing Element), the workhorse processors of the CELL. The methodology of this implementation is based on the paper of Yozo Hida, Xiaoye S. Li, and David H. Bailey, titled "Algorithms for quad-double precision floating point arithmetic".

To improve the performance, the FPU of the SPE supports only the truncation rounding. So all the floating point operations used in the implementation of the library can only use this rounding mode, which requires to make some modifications to the algorithms. That increases the complexity of the implementation. However by taking advantage of the characteristics of the SPE processor, among which the most important are the fully pipelined set of instructions in single precision and the FMA (Fused Multiplier-Add) function, we have managed to implement the error-free transformations very effectively, even more quickly than the ones used in the paper (Hida et al., 2001). With the SIMD characteristic, we can perform 4 operations at the same time. We also prove the exactitude of our modified error-free transformation, and the precision of our floating-point arithmetics by providing error bounds.

Even though the theoretical peak performance of the library is much less than the performance of the real double precision of the machine, which is about 2.7 Gflop/s in comparison with the 14.4 Gflop/s of the real double precision, the results of our test show that it is not such that bad. In the best case, the performance of our library and the performance of the real double are nearly equal. With the same approach, in the future, we will promote our work to the quad-single precision, which is very promising because the CELL processor does not support the quad precision.

Keywords: extended precision, rounding mode toward zero, CELL processor

1. Introduction

The CELL processor jointly developed 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. The goal of our work is to develop extended precision libraries for this architecture.

In this paper we will study how to implement the double working precision library named single-single on the SPEs (Synergistic Processing Element) which are the workhorse processors of the CELL. Our approach is similar to those used in (Hida et al., 2001) for the quad-double precision arithmetic in the rounding mode to the nearest. The next CELL generation will provide powerfull 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 $(+, -, \times, /)$ 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.

2. Introduction to 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" (SPE). The PPE and SPEs are linked together by an internal high speed bus called "Element Interconnect Bus" (EIB).

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.

The SPE is composed of an "Synergistic Processing Unit" (SPU) and a "Memory Flow Controller" (MFC) which is devoted to the 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 at the same time 4 operations in single precision or 2 operations in double precision. It supports all the 4 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. So with a frequency of 3.2 GHz, each SPE can achieve the performance of $2 \times 4 \times 3.2 = 25.6$ 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 7 cycles. So the peak performance of each SPE for the double precision is: $2 \times 2 \times 3.2/7 = 1.8$ GFLOPs.

Each SPE has a “Local Storage” (LS) of 256 KB for both data and code. In the opposite of the cache memory managment, there is no mechanism to load data in the LS. It is up to the programmer to explicit data transfer via DMA function call. The SPE possess a large set of registers (128 128-bits 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 methodology 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 $\mathbb{F}$ is the most frequently used to represent real numbers. A binary floating-point number is described as follows:

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

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 $ulp(x) = 2^{e-p+1}$.

Let x, y be two floating-point numbers, $\circ$ be a floating-point operation ($\circ \in \{+, -, \times, /\}$). 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 between $(x \circ y)$ and $fl(x \circ y)$ 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 2 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 2 floating-point numbers whose sums are equal. To ensure a unique representation, a_h and a_l should have the same sign and require to satisfy:

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

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

3.1. The error-free transformations (EFT)

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

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

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 (Knuth, 1998) which uses 6 standard operations, or the algorithm proposed by Dekker (Dekker, 1971) which uses only 3 standard operations but with the assumption on the order between the absolute values of two input numbers.

In this paper, our work focuses only on the rounding mode toward zero. So, it is necessary to adapt these algorithms. Priest (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 1. 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 exactitude of the newly proposed algorithm is provided in the following theorem.

Theorem 1. Let a, b be two floating-point numbers. The result of `Two-Sum-toward-zero2` (s, e) in applying on a, b satisfies:

$$\begin{aligned} s + e &= a + b, \\ |e| &< ulp(s). \end{aligned}$$

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 then the classic algorithm for the product can be used.

Algorithm 2. 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 libray implements this algorithm.

4. Basic operations of single-single

4.1. Renormalisation

Using the EFTs toward zero, 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:

```
Renormalise2-toward-zero (a,b)
    if (|a| < |b|)
        swap(a,b)
    s = fl(a + b)
    d = fl(s - a)
    e = fl(b - d)
return (s,e)
```

It is interesting to note that the 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 2. Let a, b be two floating-point numbers. The result returned by `Renormalise2-toward-zero` is a couple of two floating-point numbers (s, e) which satisfies:

– s, e have the same sign and $|e| < ulp(s)$,

– $a + b = s + e + \delta$, where δ is error of normalisation and $|\delta| \leq \frac{1}{2}\varepsilon^2|a + b|$.

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 4.1.

4.1.1. *Addition*

The figure 2 represents the algorithm for the addition of two single-singles a, b. The source code is as follows:

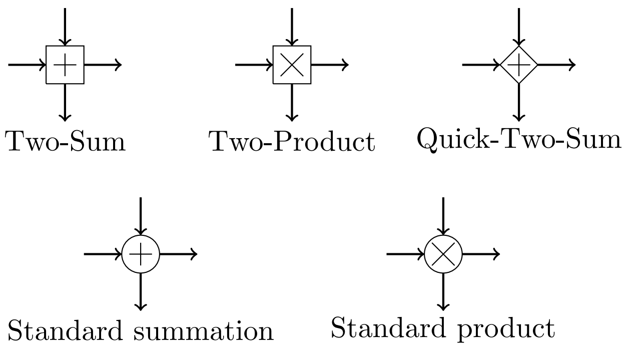

Figure 1. Notations

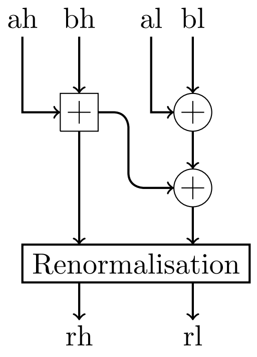

Figure 2. Algorithm for the addition of two single-singles

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

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 3. 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 produced by this algorithm δ satisfies:

$$r_h + r_l = (a_h + a_l) + (b_h + b_l) + \delta,$$

$$|\delta| < max(2^{-23} * |a_l + b_l|, 2^{-43} * |a_h + a_l + b_h + b_l|) + 2^{-45} * |a_h + a_l + b_h + b_l|.$$

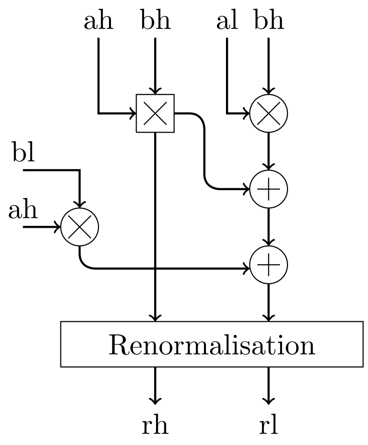

Figure 3. Algorithm for the product of two single-singles

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 between two single-singles a and b can be considered as the product of two sum $a_h + a_l$ and $b_h + b_l$ so the exact product has 4 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 1 term $\mathcal{O}(1)$, 2 terms $\mathcal{O}(2)$ and 1 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 2 floating-point numbers of orders $\mathcal{O}(1)$ and $\mathcal{O}(2)$ respectively. So 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 easily translated into the following algorithm:

```
mul_ds_ds (ah, al, bh, bl)
        (th, tl) = Two-Product-FMA (ah, bh)
        tll = fl(al * bh)
        tll = fl(ah * bl + tll)
        tl = fl(tl + tll)
```

```
      (rh, rl) = Renormalise2-toward-zero (th, tl)
      return (rh, rl)
```

This algorithm is described in figure 3.

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

Theorem 4. 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 of this algorithm called δ satisfies:

$$|(r_h + r_l) - (a_h + a_l) \times (b_h + b_l)| < 2^{-43} \times |(a_h + a_l) \times (b_h + b_l)| + 9 \times 2^{-68} \times |(a_h + a_l) \times (b_h + b_l)|.$$

4.1.4. *The division*

The division of two single-singles is calculated by using 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, at first we calculate the approximate quotient by: $q_h = a_h/b_h$.

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

It can be written in detail as follows:

```
div_ds_ds(a,b)
    ph = fl(ah / bh)
    tmp1 = fl(ah - qh * bh)
    tmp2 = fl(al - qh * bl)
    r = fl(tmp1 + tmp2)
    pl = fl(r / bh)
    (qh,ql) = Renormalise2-toward-zero(ph, pl)
    return (qh,ql)
```

We also provide the following theorem to estimate the error of this algorithm.

Theorem 5. 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:

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

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

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

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 it has a great impact on the performance.

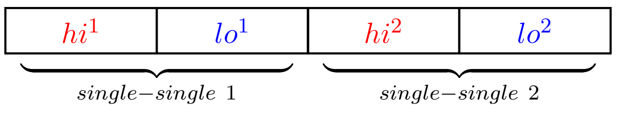

Figure 4. A vector of 2 single-singles

5. Implementation

The SPE (Synergistic Processor Element) of CELL processor contains a 32-bit 4-way SIMD processor together with a large set of 128 128-bits 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 vectorial 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 vectorial operations for the single-singles.

5.1. Representation

A single-single is a couple of two floating-point numbers so each vector of 128 bits contains two single-singles (figure 4). So, the 128 bits register containing two single-single numbers could be seen as a vector of 4 floating points numbers.

5.2. Implementation of the error-free transformations

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

```
Two-Prod-FMA (a,b)
    p = spu_mul(a,b)
    e = spu_msub(a, b, p)
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. So, we have to first 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` for the opposite case.
- computation of the values of the two branches `val_1` (if the condition is satisfied) and `val_2` (if not).
- selection of the right value according to the vector of condition by using 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*.

a	$a1$	$a2$	$a3$	$a4$
b	$b1 > a1$	$b2 < a2$	$b3 = a3$	$b4 > a4$

$comp = spu\_cmpabsgt(b, a)$

FFFFFFFF	00000000	00000000	FFFFFFFF

$hi = spu\_sel(a, b, comp)$

$b1$	$a2$	$a3$	$b4$

$lo = spu\_sel(b, a, comp)$

$a1$	$b2$	$b3$	$a4$

Figure 5. Example of the exchange of two vectors

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 5 gives a concrete example of this exchange.

Each `spu_cmpabsgt` costs 2 clock cycles and the `spu_sel` too. Moreover, since the instructions of lines 2, 3 of this code are independent, they can be pipelined. So these 3 instructions cost only 5 clock cycles, which is less than a single precision operation (6 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. So we choose to use $a + b$ instead of $hi + lo$ to avoid the instructions dependencies. More precisely the 3 first instructions for the test and the exchange are independent of the instruction of line 5 which costs 6 cycles. So, they

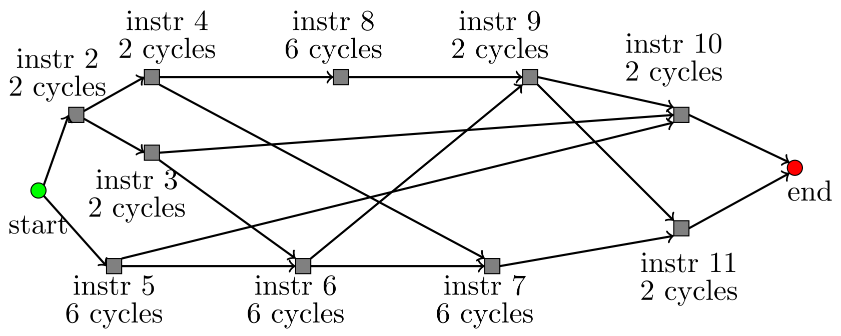

Figure 6. The dependencies between instructions of algorithm `Two-Sum-toward-zero`

can be executed in parallel[1]. The figure 6 emphasis the full independencies of instructions. This algorithm costs 20 clock cycles, which is a little bit more than the execution time of 3 consecutive double precision operations.

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.

```
Renormalise2-toward-zero (a,b)
    s = spu_add(a , b)
    comp = spu_cmpabsgt(b,a)
    hi = spu_sel(a, b, comp)
    lo = spu_sel(b, a, comp)
    d = spu_sub(s , hi)
    e = spu_sub(lo , d)
    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 on single-single operations computes one operation on TWO single-singles. The SIMD processor allows us to manipulate simultaneously four 32 bits 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.

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

Using the `Two-Sum-toward-zero2` presented above we calculate the sums and the rounding errors of two couples of high components and also of two couples of low components in the same time. Note that the rounding errors of these two couples 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. So 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` and cost 50 cycles and 49 cycles respectively for two single-singles.

5.5. Version 2

The second version computes one operation on FOUR single-singles. It separates the high and the low components into two separated vectors (see figure 7) by using the function `spu_shuffle` of SPE which costs 4 clock cycles. This solution makes it possible a better optimisation of the pipelined instructions

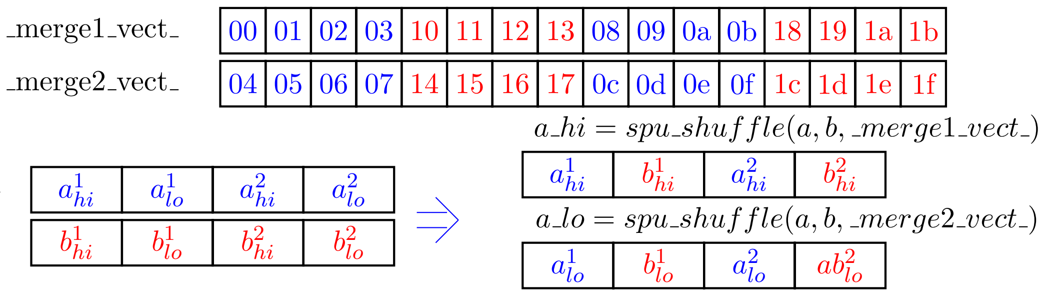
_merge1_vect_
00 01 02 03 10 11 12 13 08 09 0a 0b 18 19 1a 1b
_merge2_vect_
04 05 06 07 14 15 16 17 0c 0d 0e 0f 1c 1d 1e 1f
$a\_hi = spu\_shuffle(a, b, \_merge1\_vect\_)$
a
a^1_{hi} a^1_{lo} a^2_{hi} a^2_{lo}
b
b^1_{hi} b^1_{lo} b^2_{hi} b^2_{lo}
a^1_{hi} b^1_{hi} a^2_{hi} b^2_{hi}
$a\_lo = spu\_shuffle(a, b, \_merge2\_vect\_)$
a^1_{lo} b^1_{lo} a^2_{lo} ab^2_{lo}

Figure 7. Merging of two vectors

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

The intermediate result of these algorithms is also two 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, the version 2 for the sum of single-singles is written as follows[2]

```
    add_ds_ds_4 (vect_a1 , vect_a2 , vect_b1 , vect_b2)
        a_hi = spu_shuffle(vect_a1 , vect_a2 , _merge1_vect_)
        a_lo = spu_shuffle(vect_a1 , vect_a2 , _merge2_vect_)
        b_hi = spu_shuffle(vect_b1 , vect_b2 , _merge1_vect_)
        b_lo = spu_shuffle(vect_b1 , vect_b2 , _merge2_vect_)
        (s , e) = Two-Sum-toward-zero (a_hi , b_hi)
```

[2] The SIMD unit computes on 128-bits vectors. The 4 single-singles values of a and b are cut into two parts to keep the register organisation.

```
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 8 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.

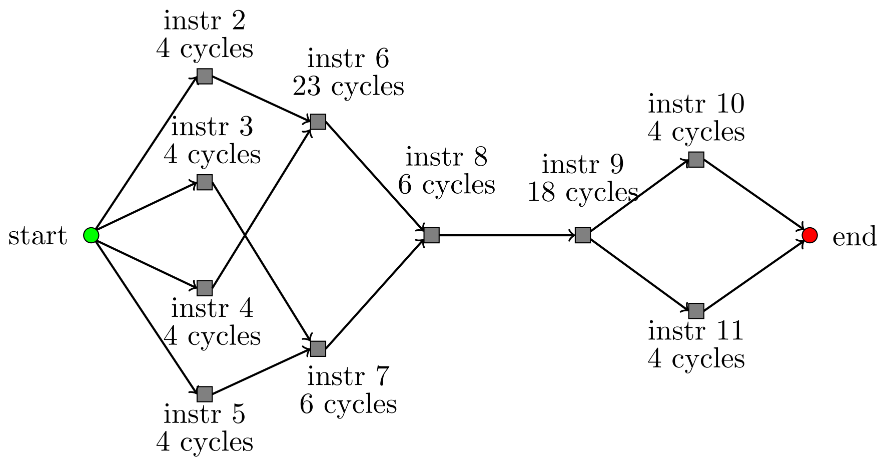

Figure 8. The dependencies between instructions of `add_ds_ds_4`

It is the same for the product of single-singles. We have successfully implemented the 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.

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, meanwhile 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. So in order to implement the division of single-singles, we first have to implement the division in single precision.

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

1. calculate the inverse of b,
2. multiply the inverse of b with a.

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

```
    div (a, b)
        tmp0 = spu_re(b)
        rerr = spu_nmsub(tmp , b , 1)
        inv = spu_madd(rerr , tmp0, tmp0)
        rerr = spu_nmsub(inv , b , 1)
        eerr = spu_mul(rerr , inv)
        tmp = spu_mul(eerr , a)
        q = spu_madd(a , inv , tmp)
    return q
```

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

Theorem 6. Let a, b be two floating-point numbers in single precision, ε being the machine precision. The relative error of the algorithm `div` is bounded by:

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

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 fours single-single divisions at the same time. This function costs **111 clock cycles** for four single-singles.

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. But using the `spu_timing` tool of IBM we recognized 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 8 cycles for the sum and 3 cycles for the product) we can execute 8 single-single operations instead of 4. It means that we have achieved a coarse gain with the final version in terms of performance.

Almost every clock cycles being used, there would be no gain to deal with sixteen single-singles.

Table I. 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

5.7. Theoretical results

On a CELL processor with a frequency of 3.2 GHz, its theoretical performances (without memory access problem) of the single-single are presented in table I.

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 3.2GHz. The results obtained are listed in the table II.

Figure 9 illustrates the performance of the addition on single-singles and on native double precision. Both have the same memory size. They are very close. It is interesting to note that the maximum performance with 64 bits floating point is not reached. In this case the program measures mainly the memory transfer time. The native double operation 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. This kind of program benefits for our library.

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

Table II. 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

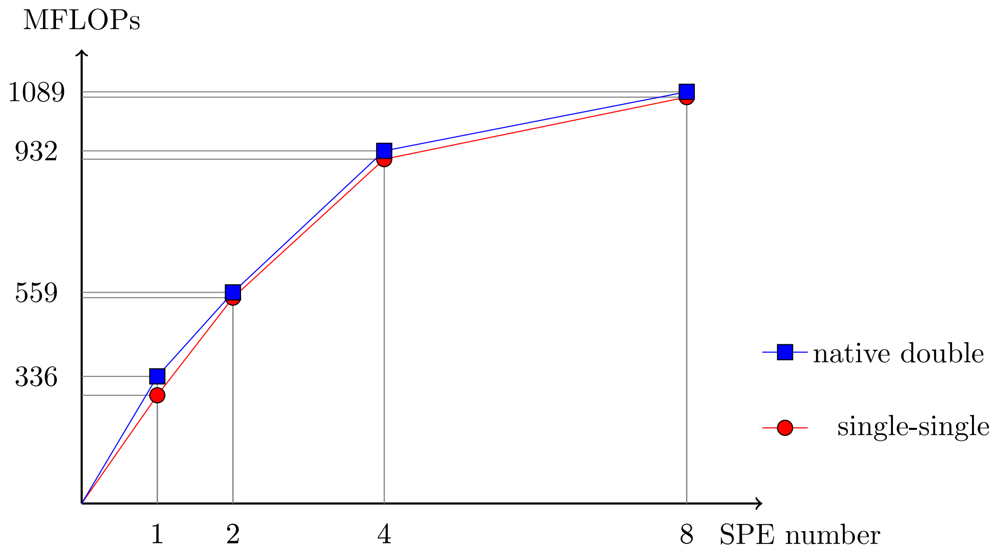

Figure 9. The performance of the library single-single: The addition

in table III. The peak performance for the multiplication on the CELL processor is achieved for native double precision.

With the single-singles numbers, it is not possible to achieve the same performance than 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. Exactitude

The exactitude of the library is tested by performing a large number of operations on random values of single-single and their corresponding double precision values. With 2^{24} comparisons, the results are summarized in table IV.

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

Functions	**Theoretical performance (1SPE)**	**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

Table IV. The exactitude of single-single library

Operation	**Max difference**	**Mean difference**
Sum	0.0e+00	0.00e+00
Product	2.964e-14	1.425e-16
Division	2.373e-14	1.758e-15

7. Conclusions and perspectives

This paper is based mostly on (Hida et al., 2001) with some adaptations to the rounding mode toward zero and to the implementation environment of CELL processor. First we propose an algorithm for the error-free transformation of the sum which is proved to be effectively implemented on the CELL processor. Then, we introduce the methodology to develop the extended precision of single-single with such basic operators that the sum, the product and the division. A large part of this paper is dedicated to the implementation of this library in exploiting the specific characteristics of 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 is tested by running test programs on a real CELL processor with a frequency of 3.2GHz.

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

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

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

These precisions are needed by more and more current applications.

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

Dekker, T. J.: 1971, 'A floating-point technique for extending the available precision'. *Numer. Math.* **18**, 224–242.

Gschwind, M., H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki: 2006, 'Synergistic Processing in Cell's Multicore Architecture'. *IEEE Micro* **26**(2), 10–24.

Hida, Y., X. S. Li, and D. H. Bailey: 2001, 'Algorithms for Quad-Double Precision Floating Point Arithmetic'. In: *Proc. 15th IEEE Symposium on Computer Arithmetic*. pp. 155–162, IEEE Computer Society Press, Los Alamitos, CA, USA.

Jacobi, C., H.-J. Oh, K. D. Tran, S. R. Cottier, B. W. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, and N. Yano: 2005, 'The Vector Floating-Point Unit in a Synergistic Processor Element of a CELL Processor'. In: *ARITH '05: Proceedings of the 17th IEEE Symposium on Computer Arithmetic*. Washington, DC, USA, pp. 59–67, IEEE Computer Society.

Kahle, J. A., M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy: 2005, 'Introduction to the cell multiprocessor'. *IBM J. Res. Dev.* **49**(4/5), 589–604.

Knuth, D. E.: 1998, *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. Reading, MA, USA: Addison-Wesley, third edition.

Priest, D. M.: 1992, '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. `ftp://ftp.icsi.berkeley.edu/pub/theory/priest-thesis.ps.Z`.

Nguyen, H. D.: 2007, 'Calcul précis et efficace sur le processeur CELL'. Master report, LIP6, UPMC (P. and M. Curie University), Paris, France `http://www-pequan.lip6.fr/~graillat/papers/rapport_Diep.pdf`.

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