Benchmark of an MPFR emulation of Binary128 arithmetic

September 13, 2013

Abstract

This paper describes the measure of execution time for two implementations of the Binary128 arithmetic with the MPFR library.

1 Introduction

Quad-precision arithmetic is easily emulated with the MPFR library but with some overhead with respect to floating-point types which supported by hardware. We present here an experimental measure of this overhead when compared to double arithmetic.

The quad-precision is defined as the Binary128 type in the IEEE-754 standard. The mantissa is 113-bit large and the exponent range lies from -16382 to $+16383$.

In our experiments, the precision is set to 113, but the exponent range is much larger as it defaults to the MPFR's one. While MPFR interface provides some facilities to emulate subnormal numbers and subnormal arithmetic, this would only add more overhead, and no attempt was made in this way.

2 Experimental Setup

The timings are measured on the platform that is detailed in Table 1.

platform1	Intel(R) Core(TM)2 Quad CPU Q9400 @ 2.66GHz

Table 1: Platform description

The compiler and the version of the MPFR library are displayed in Table 2. The compiler options were set to `-O2`.

Platform1:
gcc (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3
GNU MPFR version 3.1.0-p3

Table 2: Tools description

Input vectors of size n are chosen uniformly in $[0, 1]^n$ with the `rand48` function. Each displayed timing is the arithmetic mean of 20 measurements. For functions with `mpfr_t` array as input, the random `double` vector is converted in an `mpfr_t` vector in 53-bit precision and the conversion time is not taken into account in the corresponding timings.

3 Summation

We present here algorithms for the summation of n floating-point numbers in double precision and their experimental timing measurements.

The straightforward code used for the summation of a double precision vector in double precision arithmetic is shown in Listing 1.

The code used for the summation of a double precision vector in 113-bit precision arithmetic is shown in Listing 2. It uses the `mpfr_add_d` function which mixes `mpfr_t` and `double` types.

Another possibility to implement the summation of a double precision vector with Binary128 arithmetic for intermediate computations is to use a 53-bit `mpfr_t` vector as input and a `mpfr_t` variable in 113-bit precision for accumulator, as shown in Listing 3.

```
double
dsum_53 (const unsigned int n, const double
    *x)
{
  int i;
  double s = 0;
  for (i = 0; i < n; i++) {
    s += x[i];
  }
  return s;
}
```

Listing 1: Code for summation in Binary64 arithmetic

```
double
dsum_113 (const unsigned int n, const double
     *x)
{
  int i;
  double s;
  mpfr_t mps;
  mpfr_init2 (mps, 113);
  mpfr_set_ui (mps, 0, MPFR_RNDN);
  for (i = 0; i < n; i++) {
    mpfr_add_d (mps, mps, x[i], MPFR_RNDN);
  }
  s = mpfr_get_d (mps, MPFR_RNDN);
  mpfr_clear (mps);
  return s;
}
```

Listing 2: Code for summation in Binary128 arithmetic with double precision input and mixed `mpfr_t-double` functions

The timing measurements of the three summation functions with platform and random data set detailed in section 2 are presented in Figure 1. The execution time in terms of processor clock cycles is shown in the top graph, and the time ratios of the Binary128 emulations compared to the time ratio of the double precision version are displayed in the bottom graph.

4 Product

We present here algorithms for computation of the product of n floating-point numbers in double precision and their experimental timing measurements.

```
double
mpsum_113 (const unsigned int n, mpfr_t *x)
{
  unsigned int i;
  double s;
  mpfr_t mps;
  mpfr_init2 (mps, 113);
  mpfr_set_ui (mps, 0, MPFR_RNDN);
  for (i = 0; i < n; i++) {
    mpfr_add (mps, mps, x[i], MPFR_RNDN);
  }
  s = mpfr_get_d (mps, MPFR_RNDN);
  mpfr_clear (mps);
  return s;
}
```

Listing 3: Code for summation in Binary128 arithmetic with mpfr input of 113-bit precision

The straightforward code used for the product of a double precision vector in double precision arithmetic is shown in Listing 4.

```
double
dprod_53 (const unsigned int n, const double
     *x)
{
  int i;
  double s = 1;
  for (i = 0; i < n; i++) {
    s *= x[i];
  }
  return s;
}
```

Listing 4: Code for product in double precision arithmetic

The code used for the product of a double precision vector in 113-bit precision arithmetic is shown in Listing 5. It uses the `mpfr_mul_d` function which mixes `mpfr_t` and `double` types.

The code with a 53-bit precision `mpfr_t` input and 113-bit precision `mpfr_t` accumulator is shown in Listing 6.

The timing measurements of the three product functions with platform and random data set detailed in section 2 are presented in Figure 2. The execution time in terms of processor clock cycles is shown in the top graph, and the time ratios of the Binary128

```
double
dprod_113 (const unsigned int n, const
    double *x)
{
  int i;
  double s;
  mpfr_t mps;
  mpfr_init2 (mps, 113);
  mpfr_set_ui (mps, 0, MPFR_RNDN);
  for (i = 0; i < n; i++) {
    mpfr_mul_d (mps, mps, x[i], MPFR_RNDN);
  }
  s = mpfr_get_d (mps, MPFR_RNDN);
  mpfr_clear (mps);
  return s;
}
```

Listing 5: Code for product in Binary128 arithmetic with double precision input and mixed `mpfr_t-double` functions

```
double
mpprod_113 (const unsigned int n, mpfr_t *x)
{
  int i;
  double s;
  mpfr_t mps;
  mpfr_init2 (mps, 113);
  mpfr_set_ui (mps, 0, MPFR_RNDN);
  for (i = 0; i < n; i++) {
    mpfr_mul (mps, mps, x[i], MPFR_RNDN);
  }
  s = mpfr_get_d (mps, MPFR_RNDN);
  mpfr_clear (mps);
  return s;
}
```

Listing 6: Code for product in Binary128 arithmetic with mpfr input of 113-bit precision

emulations compared to the time ratio of the double precision version are displayed in the bottom graph.

5 Dot Product

We present here algorithms for computation of the dot product of floating-point vectors of dimension n and their experimental timing measurements.

The straightforward code used for the product of a double precision vector in double precision arithmetic is shown in Listing 7.

```
double
ddot_53 (const unsigned int n, const double
    *x, const double *y)
{
  int i;
  double s = 0;
  for (i = 0; i < n; i++) {
    s += x[i]*y[i];
  }
  return s;
}
```

Listing 7: Code for dot product in double precision arithmetic

Listing 8 shows the version with mixed `mpfr_t-double` functions.

```
double
ddot_113 (const unsigned int n, const double
     *x, const double *y)
{
  int i;
  double s;
  mpfr_t mps;
  mpfr_t mpp;
  mpfr_init2 (mps, 113);
  mpfr_init2 (mpp, 113);
  mpfr_set_ui (mps, 0, MPFR_RNDN);
  for (i = 0; i < n; i++) {
    mpfr_set_d (mpp, x[i], MPFR_RNDN);
    mpfr_mul_d (mpp, mpp, y[i], MPFR_RNDN);
    mpfr_add (mps, mps, mpp, MPFR_RNDN);
  }
  s = mpfr_get_d (mps, MPFR_RNDN);
  mpfr_clear (mps);
  mpfr_clear (mpp);
  return s;
}
```

Listing 8: Code for dot product in Binary128 arithmetic with double precision input and mixed `mpfr_t-double` functions

MPFR-type only function is shown in Listing 9.

The MPFR library also provides a Fused-Multiply-Add (FMA) function which can be used for dot product computation. Listing 10 shows such an implementation of the dot product with Binary128 arithmetic in intermediate calculations.

```
double
mpdot_113 (const unsigned int n, mpfr_t *x,
    mpfr_t *y)
{
  int i;
  double s;
  mpfr_t mps;
  mpfr_t mpp;
  mpfr_init2 (mps, 113);
  mpfr_init2 (mpp, 113);
  mpfr_set_ui (mps, 0, MPFR_RNDN);
  for (i = 0; i < n; i++) {
    mpfr_mul (mpp, x[i], y[i], MPFR_RNDN);
    mpfr_add (mps, mps, mpp, MPFR_RNDN);
  }
  s = mpfr_get_d (mps, MPFR_RNDN);
  mpfr_clear (mps);
  mpfr_clear (mpp);
  return s;
}
```

Listing 9: Code for dot product in Binary128 arithmetic with mpfr input of 113-bit precision

```
double
mpfmadot_113 (const unsigned int n, mpfr_t *
    x, mpfr_t *y)
{
  int i;
  double s;
  mpfr_t mps;
  mpfr_init2 (mps, 113);
  mpfr_set_ui (mps, 0, MPFR_RNDN);
  for (i = 0; i < n; i++) {
    mpfr_fma (mps, x[i], y[i], mps,
        MPFR_RNDN);
  }
  s = mpfr_get_d (mps, MPFR_RNDN);
  mpfr_clear (mps);
  return s;
}
```

Listing 10: Code for dot product in Binary128 arithmetic with `mpfr_fma` function

The timings for the previous functions are displayed in Figure 3.

6 Conclusions

We can remark that, on the particular test platform we used, the time ratios tend to be stable beyond the dimension $n \approx 1000$.

The cost of the `sum_113` function is about 250-fold the cost of `dsum_53`; for product, `dprod_113` is 120 times more expensive than `dprod_113`; and, `ddot_113`, which contains a conversion from `double` to `mpfr_t`, a multiplication and an addition costs about 400 times more than `ddot_53`.

For functions with `mpfr_t` input, costs lessen somewhat: the sum (respectively the product, the dot product) is about 70-fold (resp. 7-fold, 120-fold) the corresponding double precision algorithm. So exposing the use of `mpfr_t` type to the user yields a 3-fold speedup factor (17-fold factor for product) since `double`-to-`mpft_t` conversions do no more occur at each step.

In contrast to a hardware-supported FMA, the dot product implemented with `mpfr_fma` is always slower than `ddot_113`, costing about 180 times the cost of the double precision algorithm.

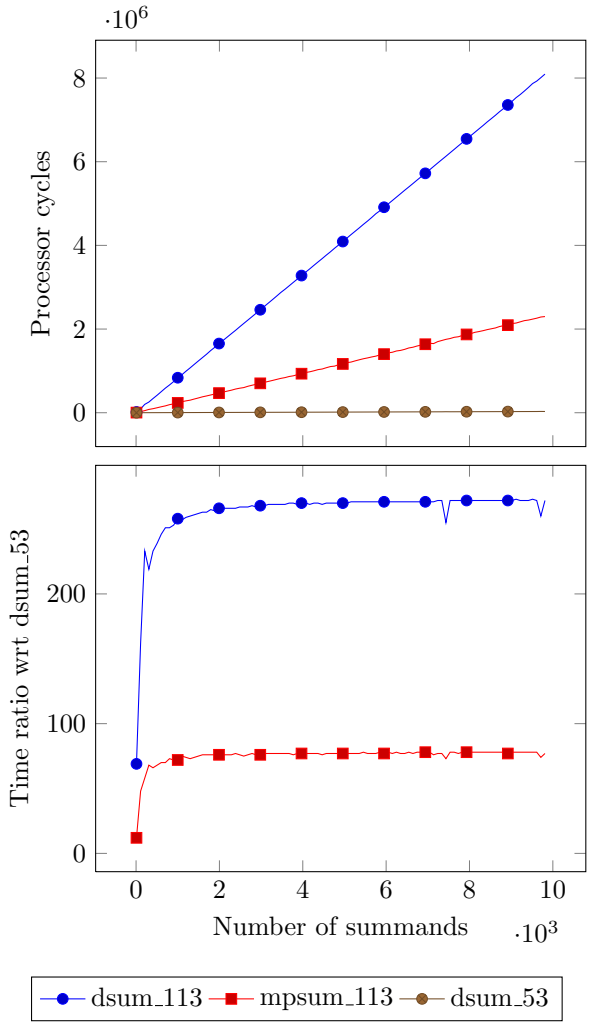

Figure 1: Summation algorithms – Top: Execution time, Bottom: Ratio execution time / time for dsum_53

,

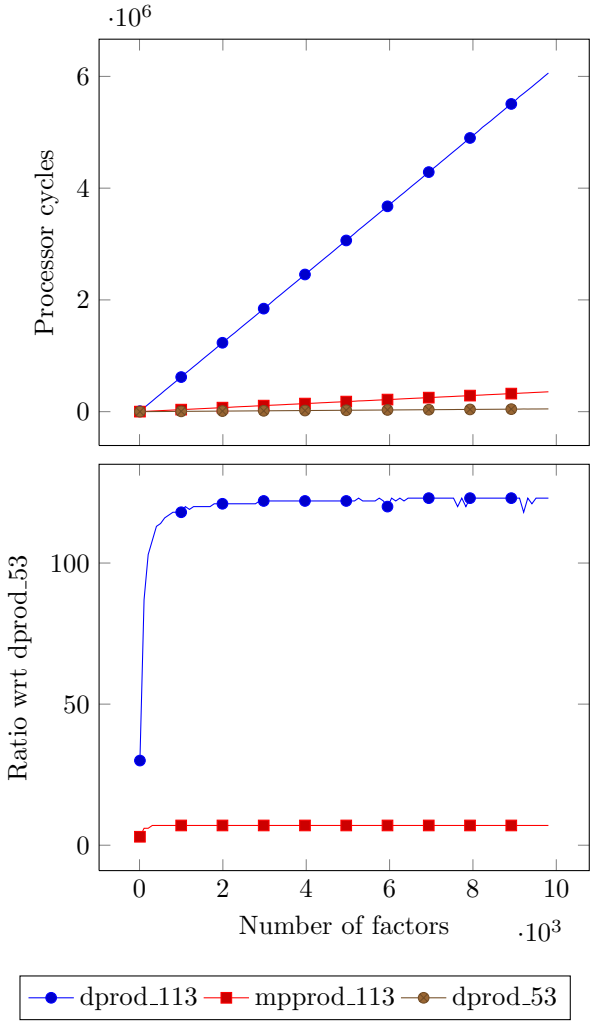

Figure 2: Product algorithms – Top: Execution time, Bottom: Ratio execution time / time for dprod_53

,

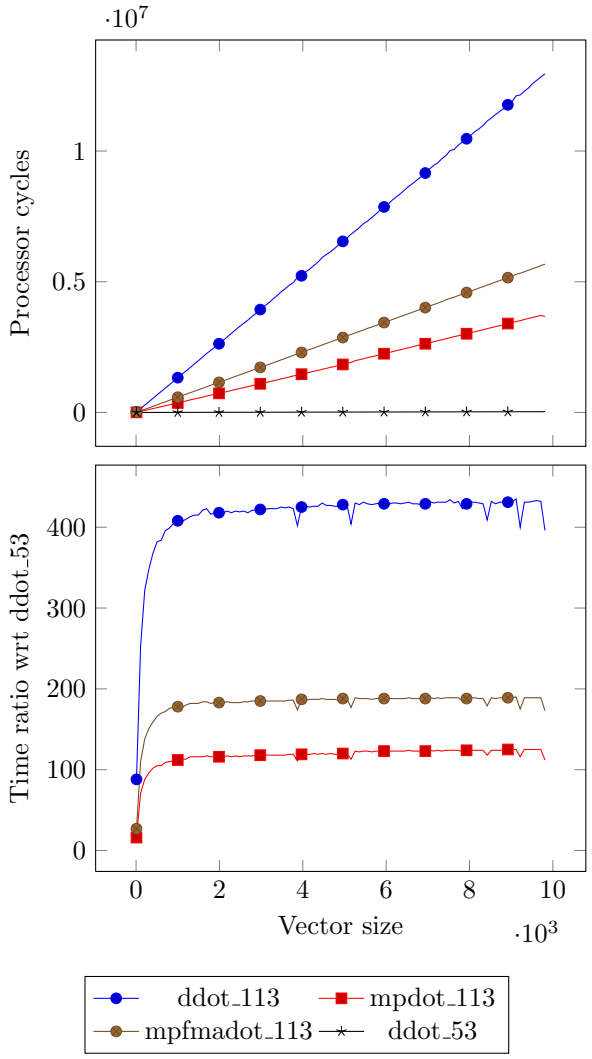

Figure 3: Dot product algorithms – Top: Execution time, Bottom: Time / time for ddot_53