Noname manuscript No.
(will be inserted by the editor)

Efficient Evaluation Methods of Elementary Functions Suitable for SIMD Computation

Naoki Shibata

Received: date / Accepted: date

Abstract Data-parallel architectures like SIMD (Single Instruction Multiple Data) or SIMT (Single Instruction Multiple Thread) have been adopted in many recent CPU and GPU architectures. Although some SIMD and SIMT instruction sets include double-precision arithmetic and bitwise operations, there are no instructions dedicated to evaluating elementary functions like trigonometric functions in double precision. Thus, these functions have to be evaluated one by one using an FPU or using a software library. However, traditional algorithms for evaluating these elementary functions involve heavy use of conditional branches and/or table look-ups, which are not suitable for SIMD computation. In this paper, efficient methods are proposed for evaluating the sine, cosine, arc tangent, exponential and logarithmic functions in double precision without table look-ups, scattering from, or gathering into SIMD registers, or conditional branches. We implemented these methods using the Intel SSE2 instruction set to evaluate their accuracy and speed. The results showed that the average error was less than 0.67 ulp, and the maximum error was 6 ulps. The computation speed was faster than the FPUs on Intel Core 2 and Core i7 processors.

Keywords SIMD, elementary functions

1 Introduction

Recently, computer architects have been trying to achieve high performance by adopting data-parallel architectures like SIMD (Single Instruction Multiple Data) or SIMT (Single Instruction Multiple Thread), rather than extracting more instruction-level parallelism. Intel's x86 architecture employs 128-bit-wide SIMD instructions, but it will be extended to 256 bit in the Sandy Bridge processors and 512 bit in the Larrabee processors[1]. Many supercomputers utilize AMD Opteron Processors as computing cores. IBM Roadrunner[2] also utilizes Cell Broadband Engines (Cell B.E.)[3], and both of these architectures have powerful SIMD engines. nVidia's CUDA-enabled GPGPUs use SIMT architecture, and can achieve 600 double-precision GFLOPS with a Tesla C2050 processor[4].

Utilizing SIMD or SIMT processing is becoming more and more important for many kinds of applications, and double-precision calculation capabilities are especially important for many scientific applications. However, those instruction sets only include basic arithmetic and bitwise operations. Instructions for evaluating elementary functions like trigonometric functions in double-precision are not provided, and we have to evaluate these functions with a conventional FPU or a software library. An FPU calculation is fast, but FPU instructions for evaluating the elementary functions are only available in a few architectures.

We need to take special care for efficient computation utilizing SIMD instructions. Gathering and scattering data from and into SIMD registers are expensive operations. Looking up different table entries according to the value of each element in a SIMD register can be especially expensive, since each table look-up may cause a cache miss. Also, the SSE instructions[5] in the x86 architecture have restrictions on data alignment, and an access to unaligned data is always expensive. Random conditional branches are expensive for processors with a long instruction pipeline. On the other hand, many SIMD architectures provide instructions for di-

Naoki Shibata
Department of Information Processing & Management,
Shiga University
1-1-1 Bamba, Hikone 522-8522, Japan
E-mail: shibata@biwako.shiga-u.ac.jp

vision and taking square root, and these calculations are now inexpensive. Thus, sometimes traditional algorithms are not suitable, and we need new algorithms for them.

In this paper, we propose new methods for evaluating the sine, cosine, arc tangent, exponential and logarithmic functions in double precision which are suitable for processors with a long instruction pipeline and an SIMD instruction set. Evaluations of these functions are carried out by a series of double-precision arithmetic and bitwise operations without table look-ups, scattering/gathering operations, or conditional branches. The total code size is very small, and thus they are also suitable for Cell B.E. which has only 256K bytes of directly accessible scratch pad memory in each SPE. The proposed methods were implemented using the Intel SSE2 instruction set for evaluating accuracy and speed. The results showed that the proposed methods were faster than FPU calculations on Intel Core 2 and Core i7 processors, and the average and maximum errors were less than 0.67 ulp (unit in the last place) and 6 ulps, respectively. The total code size was less than 1400 bytes.

The remainder of this paper is organized as follows. Section 2 introduces some related works. Section 3 presents the proposed methods with working C codes. Section 4 shows the results of the evaluation of accuracy, speed and code size. Section 5 presents our conclusions.

2 Related Works

Some software libraries have capabilities for evaluating elementary functions. A few libraries with capabilities for evaluating elementary functions using SIMD computation in single precision are available[6,7,8]. However, adding capabilities of double-precision evaluations to these libraries is not a straightforward task. Software modules like the x87 FPU emulator code in Linux operating system kernel[9] or the libm in GNU C Library[10] have capabilities for evaluating elementary functions in double precision. These modules do not utilize SIMD computation, and they involve heavy use of conditional branches. Some multiple-precision floating-point computation libraries have capabilities for evaluating elementary functions in multiple-precision[11,12]. The program structures of these libraries are very different from the proposed method.

There are many researches on evaluating elementary functions on hardware[13,14,15,16]. The restrictions of calculation are very different from the SIMD computations, and many of these methods use table look-ups.

There are many researches on accelerating applications using SIMD or SIMT computation[17,18]. Traditional irregular algorithms are sometimes difficult to parallelize on massively multi-threaded hardware, and many researchers are now working on new algorithms for such hardware.

3 Proposed Method

In this section, the details of the proposed methods are explained. The working C codes are provided which can be easily converted into a series of SIMD instructions. Having converted the codes, a target function can be evaluated with multiple inputs in a SIMD register simultaneously.

3.1 Trigonometric Functions

The proposed method evaluates the functions of sine and cosine on the argument of d at the same time. The proposed method consists of two steps. First, the argument d is reduced to within 0 and $\pi/4$ utilizing the symmetry and periodicity of the sine and cosine functions, shown as (1) and (2). Second, an evaluation of the sine and cosine function on the reduced argument s is performed assuming that the argument is within the range.

$$\cos x = \sin(\frac{\pi}{2} - x) = \sin(x + \frac{\pi}{2}) \tag{1}$$

$$\sin x = \cos(\frac{\pi}{2} - x) = -\cos(x + \frac{\pi}{2}) \tag{2}$$

For the first step, we first find s and an integer q in (3) so that $0 \leq s < \pi/4$.

$$d = s + \frac{\pi}{4}q \tag{3}$$

q can be found by dividing d by $\pi/4$ and applying the floor function. Then s can be found by multiplying q by $\pi/4$ and subtracting it from d. We need to take special care here, because a cancellation error can make the derived value of s inaccurate when d is a large number close to a multiple of $\pi/4$. Some implementations like FPUs on x86 architectures exhibit this problem. In this paper, a way for efficiently preventing this problem is proposed. The problem is in the last multiplication and subtraction, and the proposed method prevents the problem by calculating this part with extra precision utilizing properties of the IEEE 754 standard for floating-point arithmetic[19]. Here, q is assumed to be expressed in 26 bits, which is a half of the length of the mantissa in an IEEE 754 double-precision FP number. The value of $\pi/4$ is split into three parts so that

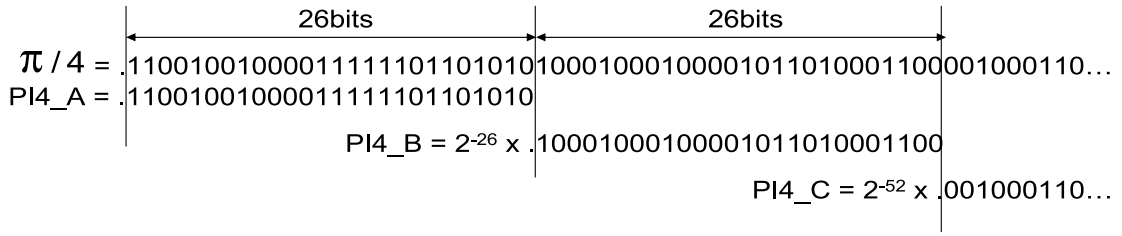

Fig. 1 Splitting $\pi/4$ into three parts

all of the bits in the lower half of the mantissa are zero, as shown in Fig. 1. Then, multiplying these numbers with q does not produce an error at all, and subtracting these numbers in descending order from d does not produce a cancellation error either.

For the second step, we evaluate the sine and cosine functions on the argument s. In many existing implementations, these evaluations are performed by simply summing up the terms of the corresponding Taylor series, shown as (4).

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots \tag{4}$$

However, if these terms are summed up using double-precision calculations, we cannot get an accurate result because of the accumulation of errors. In order to improve the accuracy while speeding up the evaluation, many implementations use another way of argument reduction [11]. The argument s is divided by a certain number, the terms of the Taylor series are evaluated for this value, and then the triple-angle formula (5) is used to obtain the final value. For example, in order to evaluate $\sin 0.27$, we first divide 0.27 by 9, evaluate the terms of the Taylor series for $\sin 0.03$, and then apply the triple-angle formula two times to find the value of $\sin 0.27$. Since the argument for the Taylor series is reduced, its convergence is accelerated, and we do not need to evaluate many terms to get an accurate value. In this way, we can reduce the accumulation of errors.

$$\sin 3x = 3\sin x - 4\sin^3 x \tag{5}$$

However, this way does not yield sufficient accuracy with double-precision calculations, because the difference between $3\sin x$ and $4\sin^3 x$ becomes large if $\sin x$ is small, and applying the triple-angle formula produces a rounding error. To prevent this error, we can use the double-angle formula (6) of cosine. In this case, the value of the sine function can be evaluated from the value of the cosine function using (7).

$$\cos 2x = 2\cos^2 x - 1 \tag{6}$$

$$\sin x = \sqrt{1 - \cos^2 x} \tag{7}$$

Now, we have another problem. If x is close to 0, finding the value of the sine function using (7) produces a cancellation error, since $\cos^2 x$ is close to 1. In this paper, instead of $\cos s$, the value of $1 - \cos s$ is found. Let $f(x)$ be $1 - \cos x$, and we get the following equations.

$$f(2x) = 4f(x) - 2f(x)^2 \tag{8}$$

$$\sin x = \sqrt{2f(x) - f(x)^2} \tag{9}$$

Since s is between 0 and $\pi/4$, the application of (8) or (9) does not produce cancellation or rounding errors. By evaluating the first five terms of the Taylor series and applying (8) three times, we can evaluate the functions of sine and cosine on the argument of s with sufficient accuracy. In order to improve the computational performance, we use the following technique. Let $g(x)$ be $2f(x)$ and we get the following equation:

$$g(2x) = (4 - g(x)) \cdot g(x) \tag{10}$$

By using (10) instead of (8), we can omit one multiplication each time we apply the double-angle formula. This also improves the accuracy by a little. To incorporate the result from the first step, we use the symmetry and periodicity of the trigonometric functions. This is performed by exchanging the derived values of $\sin s$ and $\cos s$, and multiplying those values by -1 according to the remainder q divided by 4. In order to evaluate the tangent function, we can simply use (11).

A working C code for the proposed method is shown in Fig. 2[1].

$$\tan x = \frac{\sin x}{\cos x} \tag{11}$$

[1] The code contains conditional branches for ease of reading. All of these conditional branches can be eliminated by unrolling loops or replacing them by combinations of comparisons and bitwise operations.

```c
#include <math.h>

typedef struct sincos_t { double s, c; } sincos_t;

#define N 3

sincos_t xsincos0(double s) { // 0 ≤ s < π/4
  int i;

  // Argument reduction
  s = s * pow(2, -N);

  s = s*s; // Evaluating Taylor series
  s = ((((s/1814400 - 1.0/20160)*s + 1.0/360)*s - 1.0/12)*s + 1)*s;

  for(i=0;i<N;i++) { s = (4-s) * s; } // Applying double angle formula
  s = s / 2;

  sincos_t sc;
  sc.s = sqrt((2-s)*s);  sc.c = 1 - s;
  return sc;
}

#define PI4_A .78539815545082092285156 25 // π/4 split into three parts
#define PI4_B .79466273561479283671360462903976440429687 5e-8
#define PI4_C .30616169978683829430651648306875026455243736 1480769e-16

// 4/π
#define M_4_PI 1.27323954473516254282117188267875462770462036 1328125

sincos_t xsincos(double d) {
  double s = fabs(d);
  int q = (int)(s * M_4_PI), r = q + (q & 1);

  s -= r * PI4_A; s -= r * PI4_B; s -= r * PI4_C;

  sincos_t sc = xsincos0(s);

  if (((q + 1) & 2) != 0) { s = sc.c; sc.c = sc.s; sc.s = s; }
  if (((q & 4) != 0) != (d < 0)) sc.s = -sc.s;
  if (((q + 2) & 4) != 0) sc.c = -sc.c;

  return sc;
}

double xsin(double d) { sincos_t sc = xsincos(d); return sc.s; }
double xcos(double d) { sincos_t sc = xsincos(d); return sc.c; }
double xtan(double d) { sincos_t sc = xsincos(d); return sc.s / sc.c; }
```

Fig. 2 Working C code for evaluating trigonometric functions

3.2 Inverse Trigonometric Functions

We cannot get an accurate result by simply summing up many terms of the Taylor series (12) using double-precision calculations because of an accumulation of errors. As far as we surveyed, no good way is known for reducing the argument for evaluating inverse trigonometric functions. In this paper, a new way of argument reduction for evaluating the arc tangent function utilizing the half-angle formula (14) of the cotangent function is proposed.

$$\arctan x = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} x^{2n+1} \tag{12}$$

$$\cot x = \frac{1}{\tan x} = \tan\left(\frac{\pi}{2} - x\right) \tag{13}$$

$$\cot \frac{x}{2} = \cot x \pm \sqrt{1 + \cot^2 x} \tag{14}$$

Let d be the argument and $\theta = \arctan d$. Thus, $d = \tan\theta$ holds. For simplicity, we now assume $d \geq 0$ and $0 \leq \theta < \pi/2$. From (13), the following equation holds.

$$\tan\left(\frac{\pi}{2} - \theta\right) = \frac{1}{d} \tag{15}$$

Thus, if $d \leq 1$, the argument can be reduced by evaluating the arc tangent function on the argument of $1/d$ instead of d, and subtracting the result from $\pi/2$. Thus, we can now assume $d > 1$ and $\pi/4 < \theta < \pi/2$. In order to reduce the argument $d = \tan\theta$, we can use (14) to enlarge $\cot\theta = 1/\tan\theta$. Please note that $\cot x/2$ is larger than $\cot x$ if $0 < x < \pi/2$. For example, assume that we are evaluating $\arctan 2$, and ϕ is a number such that $2 = \tan\phi$. We first apply (13) to get $\cot\phi = 1/2$. Next, we apply (14) to get $\cot\phi/2 = 1/2 + \sqrt{1 + (1/2)^2}$, and apply (13) to get $\tan\phi/2 = 1/(1/2 + \sqrt{5/4})$. Note that $1/(1/2 + \sqrt{5/4})$ is less than 2, thus we reduced the argument. Then, we evaluate several terms of (12) for the arc tangent of $1/(1/2 + \sqrt{5/4})$, which is $\phi/2$. Finally, we multiply 2 to the result to get $\phi = \arctan 2$.

By applying (14) two times and summing the first eleven terms of Taylor series, we can evaluate the arc tangent function with sufficient accuracy.

The arc sine and the arc cosine functions can be evaluated using the following equation.

$$\arcsin x = \arctan \frac{x}{\sqrt{1 - x^2}} \tag{16}$$

$$\arccos x = \arctan \frac{\sqrt{1-x^2}}{x} \tag{17}$$

However, these equations produce cancellation errors if $|x|$ is close to 1. These errors can be avoided by modifying the above equations as follows. Note that no cancellation error is produced by subtracting x from 1 in these equations. A working C code for the proposed method is shown in Fig. 3.

$$\arcsin x = \arctan \frac{x}{\sqrt{(1+x)(1-x)}} \tag{18}$$

$$\arccos x = \arctan \frac{\sqrt{(1+x)(1-x)}}{x} \tag{19}$$

```
#include <math.h>

#define M_PI 3.14159265358979323846
#define N 2

double xatan(double d) {
  double x, y, z = fabs(d);
  if (z < 1.0) x = 1.0/z; else x = z;

  int i; // Applying cotangent half angle formula
  for(i = 0;i < N;i++) { x = x + sqrt(1+x*x); }

  x = 1.0 / x;

  y = 0; // Evaluating Taylor series
  for(i=10;i>=0;i--) {
    y = y*x*x + pow(-1, i)/(2*i+1);
  }
  y *= x * pow(2, N);

  if (z > 1.0) y = M_PI/2 - y;
  return d < 0 ? -y : y;
}

double xasin(double d) {
  return xatan(d / sqrt((1+d)*(1-d)));
}

double xacos(double d) {
  return xatan(sqrt((1+d)*(1-d))/d) +
    (d < 0 ? M_PI : 0);
}
```

Fig. 3 Working C code for evaluating inverse trigonometric functions

3.3 Exponential Function

The proposed method for evaluating the exponential function consists of two steps. As the first step, the argument d is reduced to within 0 and $\log_e 2$. Then, as the second step, the exponential function is evaluated on the reduced argument s.

For the first step, in order to evaluate $\exp d$, we first find s and an integer q in (20) so that $0 \le s < \log_e 2$.

$$d = s + q \cdot \log_e 2 \tag{20}$$

Here, we have the same problem as we had in the case of the trigonometric functions. In order to find the value of s, we have to multiply q by $\log_e 2$ and subtract it from d, where we have a cancellation error when d is close to a multiple of $\log_e 2$. In order to prevent this, the value of $\log_e 2$ is split into two parts in a similar way as in the case of the trigonometric functions, and each part is multiplied by q and subtracted from d in turn.

For the second step, we can use (21) for reducing the argument, and then evaluate the terms of the Taylor series (22).

$$\exp 2x = (\exp x)^2 \tag{21}$$

$$\exp x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots \tag{22}$$

However, we have a problem if we simply use these two equations. If s is close to 0, the difference between the first term, which is 1, and the other terms in (22) is too large and produces rounding errors. In order to avoid this problem, the proposed method finds the value of $\exp s - 1$ instead of $\exp s$. Let $h(x)$ be $\exp x - 1$, and we derive the following equation from (21) and (22):

$$h(2x) = (2 + h(x)) \cdot h(x) \tag{23}$$

$$h(x) = \sum_{n=1}^{\infty} \frac{x^n}{n!} \tag{24}$$

By summing the first eight terms of (24), applying (23) four times, and then adding 1 to the result, we can evaluate the exponential function on the argument of s with sufficient accuracy. By incorporating the result from the first step, we get the final result by multiplying the result by 2^q as shown in (25). Calculation of 2^q from q can be performed by substituting $1023 + q$ for the exponent of 1, utilizing the properties of the IEEE 754 standard.

A working C code for the proposed method is shown in Fig. 4.

$$\begin{aligned} \exp(s + q \cdot \log_e 2) &= (\exp s) \cdot \exp(q \cdot \log_e 2) \\ &= (\exp s) \cdot 2^q \end{aligned} \tag{25}$$

3.4 Logarithmic Function

In order to evaluate the logarithmic function, we can use the series shown as (26) which converges faster than the Taylor series[20].

$$\log_2 x = 2 \sum_{n=0}^{\infty} \frac{1}{2n+1} \left(\frac{x-1}{x+1} \right)^{2n+1} \tag{26}$$

```
#include <math.h>

#define M_LN2 0.693147180559945309417232121458176 6 // log_e 2

// log_e 2 split into two parts
#define L2U .693147180559662956511601805686950683593 75
#define L2L .282352905630315771225884481750134360255254120 68e-12

#define N 4

double xexp(double d) {
  int q = (int)(d / M_LN2), i;
  double s = d - q * L2U - q * L2L, t;

  s = s * pow(2, -N); // Argument reduction

  // Evaluating Taylor series
  t = (((s/40320 + 1.0/5040)*s + 1.0/720)*s + 1.0/120)*s;
  t = ((((t + 1.0/24)*s + 1.0/6)*s + 1.0/2)*s + 1)*s;

  for(i=0;i<N;i++) { t = (2+t)*t; }

  return (t + 1) * pow(2, q);
}
```

Fig. 4 Working C code for evaluating exponential function

The proposed method consists of two steps. The first step is reducing argument by splitting the original argument into its mantissa and exponent. The second step is to evaluate (26) on the mantissa part as the argument.

As the first step, we split the original argument d into its mantissa m and an integer exponent e where $0.5 \leq m < 1$, as shown in (27), utilizing the property of the IEEE 754 standard. Since (26) converges fast if its argument is close to 1, we multiply 2 to m and subtract 1 from e if m is smaller than $\sqrt{2}/2$. Then m is in the range of $\sqrt{2}/2 \leq m < \sqrt{2}$.

$$x = m \cdot 2^e \tag{27}$$

As the second step, we simply evaluate the first 10 terms of the series of (26) to evaluate $\log m$. We can get the value of $\log d$ by adding $e \log 2$ to the value of $\log m$ to get the final result, as shown in (28).

A C code for this method is shown in Fig. 5

$$\begin{aligned} \log(m \cdot 2^e) &= \log m + \log 2^e \\ &= \log m + e \log 2 \end{aligned} \tag{28}$$

4 Evaluation

The proposed methods were tested to evaluate accuracy, speed, and code size. The tests were performed on a PC with Core i7 920 2.66GHz. The system is equipped with 3 GB of RAM and runs Linux (Ubuntu Jaunty with 2.6.28 kernel, IA32). The proposed methods were implemented in C language with Intel SSE2 intrinsics[2].

[2] An intrinsic is a function known by the compiler that directly maps to a sequence of one or more assembly language instructions.

```
#include <math.h>

double xlog(double d) {
  int e, i;

  double m = frexp(d, &e);
  if (m < 0.7071) { m *= 2; e--; }

  double x = (m-1) / (m+1), y = 0;

  for(i=19;i>=1;i-=2) { y = y*x*x + 2.0/i; }

  return e*log(2) + x*y;
}
```

Fig. 5 Working C code for evaluating logarithmic function

We used gcc 4.3.3 with `-msse2 -mfpmath=sse -O` options to compile our programs.

4.1 Accuracy

The accuracy of the proposed methods was tested by comparing the results produced by the proposed methods to the accurate results produced by 256 bit precision calculations using the MPFR Library[12]. We checked the accuracy by evaluating the functions on the argument of each value in a range at regular intervals, and we measured the average and maximum error in ulp. We measured the evaluation accuracy within a few ranges for each function. The ranges and the measurement results are shown in Tables 1, 2, 3, 4 and 5. Please note that because of the accuracy requirements of IEEE 754 standard, these results do not change between platforms comform to the standard.

The evaluation error did not exceed 6 ulps in every test. The average error of 0.28 ulp to 0.67 ulp can be considered to be good, since even if accurate numbers are correctly rounded, we have 0.25 ulp of average error.

Table 1 Average and maximum error for trigonometric functions (ulp)

Range	interval	sin avg	sin max	cos avg	cos max	tan avg	tan max
$0 \leq x \leq \pi/4$	1e-7	0.402	2.97	0.278	1.78	0.580	4.74
$-10 \leq x \leq 10$	1e-6	0.378	3.80	0.370	3.94	0.636	5.14
$0 \leq x \leq 20000$	1e-3	0.373	3.53	0.373	3.52	0.636	5.61
$10000 \leq x \leq 10020$	1e-6	0.374	3.79	0.373	3.35	0.639	5.86

Table 2 Average and maximum error for arc tangent functions (ulp)

Range	interval	arctan avg	arctan max
$-1 \leq x \leq 1$	1e-7	0.628	4.91
$0 \leq x \leq 20$	1e-6	0.361	4.68

Table 3 Average and maximum error for arc sine and arc cosine functions (ulp)

Range	interval	arcsin avg	arcsin max	arccos avg	arccos max
$-1 \leq x \leq 1$	1e-7	0.665	5.98	0.434	5.15

Table 4 Average and maximum error for exponential function (ulp)

Range	interval	exp avg	exp max
$0 \leq x \leq 1$	1e-7	0.295	1.97
$-10 \leq x \leq 10$	1e-6	0.299	1.93

Table 5 Average and maximum error for logarithmic function (ulp)

Range	interval	log avg	log max
$0 < x \leq 1$	1e-7	0.430	2.54
$0 < x \leq 1000$	1e-4	0.286	2.65

Table 6 Time taken for pair of evaluations (sec.)

	Core i7 2.66GHz (IA32)			Core 2 2.53GHz (x86_64)		
	Proposed	FPU	MPFR	Proposed	FPU	MPFR
sin+cos	3.29e-8	7.92e-8	3.53e-5	3.51e-8	8.87e-8	2.80e-5
sin	3.29e-8	7.04e-8	1.33e-5	3.51e-8	8.83e-8	1.08e-5
cos	3.29e-8	7.04e-8	2.14e-5	3.51e-8	8.92e-8	1.68e-5
tan	3.68e-8	1.82e-7	1.68e-5	4.12e-8	1.65e-7	1.33e-5
arctan	4.72e-8	1.82e-7	2.11e-5	4.98e-8	1.61e-7	1.33e-5
arcsin	6.64e-8	N/A	3.81e-5	7.35e-8	N/A	2.69e-5
arccos	6.73e-8	N/A	3.92e-5	7.68e-8	N/A	2.73e-5
exp	2.45e-8	N/A	1.59e-5	2.85e-8	N/A	1.33e-5
log	2.14e-8	4.59e-8	1.19e-5	2.69e-8	4.88e-8	8.14e-6

Table 7 Number of instructions and size of subroutines

	sin+cos+tan	arctan+arcsin+arccos	exp	log
Number of instructions	94	94	52	49
Subroutine size (byte)	426	416	253	254

4.2 Speed

In this test, we compared the speed of the proposed methods for evaluating the functions to those of an FPU and the MPFR Library[12]. Besides the Core i7 system mentioned above, we conducted the speed test on a PC with Core 2 Duo E7200 2.53GHz with 2GB of RAM running Linux x86_64 (Debian Squeeze) with a 2.6.26 kernel. There was no register spill in our methods. In these tests, we used the `gettimeofday` function to measure the time for each subroutine to execute 100 million times in a single thread. In order to measure the time for an FPU to evaluate each function, we wrote subroutines with an inline assembly code for executing the corresponding FPU instruction. These FPU subroutines execute two corresponding FPU instructions per call, since the subroutines for the proposed methods evaluate the target function on the argument of each element in a SIMD register, which contains two double precision numbers. For sin+cos measurement, we measured the time for an FPU to execute the `sincos` instruction to evaluate the sine and cosine functions simultaneously. For the MPFR Library, we set 53 bit as its precision, and measured the time for each subroutine to execute 1 million times. The measured times in second for each subroutine to perform one pair of evaluations on the two systems are shown in Table 6.

The execution speed of the subroutines of the proposed methods was more than twice as fast as the equivalent FPU instructions.

4.3 Code Size

We measured the number of instructions and code size for each subroutines of our methods by disassembling the executable files for an IA32 system using gdb and counting these numbers in the subroutines. Please note that each instruction in the subroutines is executed exactly once per call, since they do not contain conditional branches. The results are shown in Table 7.

5 Conclusion

We proposed efficient methods for evaluating elementary functions in double precision without table lookups, scattering from, or gathering into SIMD registers, or conditional branches. The methods were implemented using the Intel SSE2 instruction set to check the accuracy and speed. The average and maximum errors were less than 0.67 ulp and 6 ulps, respectively. The speed was faster than the FPUs on Intel Core 2 and Core i7 processors.

References

1. L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan : " Larrabee: A many-core x86 architecture for visual computing," *Proc. of ACM SIGGRAPH 2008*, pp. 1–15, 2008.
2. K. Barker, K. Davis, A. Hoisie, D. Kerbyson, M. Lang, S. Pakin and J. Sancho : "Entering the petaflop era: the architecture and performance of Roadrunner," *Proc. of the 2008 ACM/IEEE conference on Supercomputing*, pp. 1–11, 2008.
3. M. Gschwind, H. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe and T. Yamazaki : "Synergistic Processing in Cell's Multicore Architecture," *IEEE Micro*, Vol. 26, No. 2, pp. 10–24, 2006.
4. "Tesla C2050 and Tesla C2070 Computing Processor Board," `http://www.nvidia.com/docs/IO/43395/BD-04983-001_v01.pdf`
5. S. Thakkar and T. Huff : "Internet Streaming SIMD Extensions," *Computer*, Vol. 32, No. 12, pp. 26–34, 1999.
6. Approximate Math Library 2.0, `http://www.intel.com/design/pentiumiii/devtools/AMaths.zip`.
7. Simple SSE and SSE2 optimized sin, cos, log and exp, `http://gruntthepeon.free.fr/ssemath/`.
8. L. Nyland and M. Snyder: "Fast Trigonometric Functions Using Intel's SSE2 Instructions," *Tech. Report*.
9. Linux Kernel Version 2.6.30.5, `http://www.kernel.org/`.
10. GNU C Library Version 2.7, `http://www.gnu.org/software/libc/`.
11. R. Brent : "Fast Algorithms for High-Precision Computation of Elementary Functions," *Proc. of 7th Conference on Real Numbers and Computers (RNC 7)*, pp. 7-8, 2006.
12. The MPFR Library, `http://www.mpfr.org/`.
13. J. Detrey, F. Dinechin and X. Pujul : "Return of the hardware floating-point elementary function," *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, pp. 161–168, 2007.
14. I. Koren and O. Zinaty : "Evaluating Elementary Functions in a Numerical Coprocessor Based on Rational Approximations," *IEEE Transactions on Computers*, Vol. 39, No. 8, pp.1030-1037, 1990.
15. M. Ercegovac, T. Lang, J. Muller and A. Tisserand : "Reciprocation, Square Root, Inverse Square Root, and Some Elementary Functions Using Small Multipliers," *IEEE Transactions on Computers*, Vol. 49, No. 7, pp. 628–637, 2000.
16. E. Goto, W.f. Wong, "Fast Evaluation of the Elementary Functions in Single Precision," IEEE Transactions on Computers, Vol. 44, No. 3, pp. 453-457, 1995.
17. D. Scarpazza and G. Russell : "High-performance regular expression scanning on the Cell/B.E. processor," *Proc. of the 23rd international conference on Supercomputing*, pp. 14–25, 2009.
18. M. Rehman, K. Kothapalli and P. Narayanan : "Fast and scalable list ranking on the GPU," *Proc. of the 23rd international conference on Supercomputing*, pp. 235–243, 2009.
19. D. Goldberg : "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys*, Vol. 23, No. 1, pp. 5–48, 1991.
20. M. Abramowitz and I. Stegun : "Handbook of Mathematical Functions: with Formulas, Graphs, and Mathematical Tables," Dover Publications, 1965.