

パイプライン化された演算器を生かす指数関数実装

小泉 透^{1,a)} 入江 英嗣¹ 坂井 修一¹

概要：指数関数等の超越関数を計算する浮動小数点数演算命令は、複雑なことから命令セットに含まれないことが多い。そのため、CPU に存在する命令のみを用いたソフトウェア実装が使われることが一般的である。効率の良いソフトウェア実装が研究されているが、その評価基準は演算数によるものが多い。しかし、最近の浮動小数点数演算器はパイプライン化されているため、演算数が多くとも並列性の高い実装の方が高速に動作することがある。本稿では、パイプライン化されている最近の浮動小数点数演算器を生かすことのできる、新たな指数関数実装を提案する。Intel Architecture Code Analyzer を用いた評価では、提案手法は指数関数の実行レイテンシを 15%削減できることを明らかにした。

1. はじめに

指数関数 $\exp(x)$ は、機械学習や科学技術計算など、広範なアプリケーションで使用される。使用頻度は高いが複雑なため、CPU には指数関数を計算する命令が存在しないことが多い。そのため、CPU に存在する命令のみを用いたソフトウェア実装が開発されている。

浮動小数点数は実数を正確に表すことができないため、計算結果には丸め誤差が発生する。浮動小数点数で表せる最も近い値に丸める場合、丸め誤差は最大で仮数部の最終桁の重みの半分となる。仮数部の最終桁の重みは Unit in the Last Place (ULP) と呼称される。これを用いて、「丸め誤差は最大でも 0.5 ULP を保証」などと表記する。

指数関数は超越関数であり、丸める前の値を正確に求めることが難しい。特に丸める前の値が、離散的に存在する浮動小数点数で表せる数の中間付近に存在する場合、どちらに丸めるべきかを判定することが困難である。最近接の値に丸めることを保証せず、最近接あるいは二番目に近い値に丸める、と制約を緩めることで計算コストを大幅に下げられる。この場合は丸め誤差は最大で 1 ULP となる。

これまでの多くの研究の対象は、演算数が少なく、かつ丸め誤差が 1 ULP に収まる実装であった。例えば、計算結果を再利用することにより演算回数を減らす手法が広く用いられている。しかしこれは依存が深くなることを意味し、レイテンシが長くなる。

最近の CPU はアウトオブオーダー実行するため、レイテンシの長さは問題にならないと考えられがちである。しかし長期間インフライトである命令が存在すると、リオー

ダーバッファ容量を使い切ってしまう、レイテンシを隠す命令をフェッチできなくなってしまう。よって演算回数だけではなく、レイテンシの短さも考慮したソフトウェア実装が重要となる。

本稿では以下を示す。

- 指数関数を計算する上で桁落ちが問題となる部分について、積和演算を用いて精度を維持しつつ演算数を減らしレイテンシも短くする方法
- 指数関数の近似多項式を計算する部分について、演算回数は増えるものの並列に計算することでレイテンシを短くする方法
- 上記二種の手法を用いた、低レイテンシかつ最大丸め誤差 1 ULP を保証する指数関数実装

2. 本稿における記法

本稿においては、断りなく書いた数式は実数に関する事実を示すことと約束する。近似値を意図している記号は、真の値を表す記号にチルダをつけて表す。例えば、 r の近似値を意図した変数は $\tilde{r}$ と表し、 $u(r)$ の近似関数は $\tilde{u}(r)$ と表す。また、IEEE754-2008 [1] で定義された binary64 (倍精度浮動小数点数) で表せる範囲に計算結果を最近接丸めで丸めることを、 $\llbracket x + y \rrbracket$ のように二重の角かっこで示す。例えば、 $\llbracket 1234.56 + 3 \times 2^{51} \rrbracket = 1235 + 3 \times 2^{51}$ である。

サンプルコードは、C99 もしくは C++17 で動作するものとなっている。ただし、`double` が binary64 である処理系を仮定している。また、`uint64_t` が定義されている処理系を仮定している。`0x1.23ABp+3` などといった表記は、十六進浮動小数点数リテラルである。`fma` 関数は、`<math.h>` で定義されている、積和演算を行う関数である。積和演算は、

¹ 東京大学 大学院情報理工学系研究科

^{a)} koizumi@mtl.t.u-tokyo.ac.jp

第一引数と第二引数の積に第三引数を加えたものを正確に計算し、その結果を一度だけ丸める演算である。例えば、以下のコードは $\llbracket (1+3 \times 2^{51}) \times (1-3 \times 2^{51}) + 9 \times 2^{102} \rrbracket$ を計算し、1を得る。

```
double r = fma( 1+0x3p+51, 1-0x3p+51, 0x9p+102 );
```

一方、以下のコードは $\llbracket (1+3 \times 2^{51}) \times (1-3 \times 2^{51}) \rrbracket + 9 \times 2^{102}$ を計算するため、桁落ちにより0が得られる点に注意したい。

```
double r1 = (1+0x3p+51) * (1-0x3p+51);  
double r = r1 + 0x9p+102;
```

3. 既存の指数関数実装

3.1 既存の実装に共通する手法

指数関数は、引数 x が 709.78271289338397 以上であると結果が binary64 で表せる最大の浮動小数点数を超えてしまう。また、引数が -708.39641853226408 以下である場合、binary64 の正規化数で表せる最小の正の数を下回る。その他、0, -0, INF, NaN 等の特殊な引数に対しては特別扱いが必要であり、分岐処理が行われる。以下ではそのような特殊なパターンは考慮せず、浮動小数点数演算が支配的な引数値での挙動を確認する。

指数関数のソフトウェア実装では、以下の三点を使った実装を用いることが一般的である。入力 x は s と r の和に分解される。この時、 $|r|$ はなるべく小さく、 s は $\exp(s)$ が簡単に求められる形に分割する。

- 加法定理 $\exp(s+r) = \exp(s) \times \exp(r)$
- スケール変換 $\exp(s) = 2^{\frac{s}{\log 2}}$
- 有理関数近似 $\exp(r) \simeq 1 + r + r^2/2$ (一例)

以下では、複数の既存実装について詳しく手順を確認する。

3.2 Sun Microsystems による実装

Sun Microsystems による実装 [2] では、以下の手順により指数関数を計算している。

- (1) $k \in \mathbb{Z}, |r| < 0.34658 \sim \frac{\log 2}{2}$ を満たすように $x = k \log 2 + r$ と分解する
- (2) $c(r) = 2 - \frac{2r}{\exp(r)-1}$ を $\tilde{c}(r) = r + C_1 r^2 + C_2 r^4 + C_3 r^6 + C_4 r^8 + C_5 r^{10}$ と多項式近似して計算する
- (3) $\exp(x) = 2^k \left(1 + r + \frac{rc(r)}{2-c(r)}\right)$ により指数関数の結果を求める

以下で詳しく手順を説明する。

まず k を求める。 $k = \text{round}\left(\frac{x}{\log 2}\right)$ とすればよい。この時、 $r = \left|\frac{x}{\log 2} - k\right| < 0.34658$ を満たしていれば十分のため、 $\frac{x}{\log 2}$ の計算は多少不正確に行ってもよい。よって事前に計算された binary64 の定数である $\llbracket \frac{1}{\log 2} \rrbracket$ を用いて

$$k = \text{round}\left(\left\lfloor x \times \left\lfloor \frac{1}{\log 2} \right\rfloor \right\rfloor\right) \text{ と計算する。}$$

次に $s = k \log 2$ を用いて、 $r = x - s$ とする。ここで、 x と s の値が近いことに注意する。 $\tilde{r} = \left\lfloor x - \left\lfloor k \times \left\lfloor \log 2 \right\rfloor \right\rfloor \right\rfloor$ と計算してしまうと桁落ちが発生し、指数関数の計算結果への影響が大きい。そこで、 $\log 2$ の近似値を $a1 = 0x1.62E42FEE00000p-1$ と $a2 = 0x1.A39EF35793C76p-33$ の二数の和に分解して保持する。 $a1+a2 \simeq \log 2 + 1.16 \times 10^{-26}$ である。ここで、 $|k| \leq 710$ であること、 $a1$ は下 21bit が 0 であることから、 $r_1 = x - k \times a1$ は正確に計算される。次いで $\tilde{r} = \left\lfloor r_1 - \left\lfloor k \times a2 \right\rfloor \right\rfloor$ と計算することで、 $\tilde{r}$ を十分な精度で求めることができる。

最後に、 $\exp(r)$ を Remetz 近似により求める。 $|r|$ が大きい場合計算すべき項数が多く、しかも低速な浮動小数点数除算が含まれる。

$\exp(x)$ は $\exp(r)$ に 2^k を乗ずれば求まる。

この手法では、 $\exp(x)$ の誤差は $\exp(r)$ の計算誤差及び丸め誤差に由来する。最大の丸め誤差は 1 ULP 未満であることが保証される。

3.3 ARM による実装

ARM による実装 [3] では、以下の手順により指数関数を計算している。

- (1) $k \in \mathbb{Z}, |r| \lesssim \frac{\log 2}{256}$ を満たすように $x = \frac{k}{128} \log 2 + r$ と分解する
- (2) $k_1, k_2 \in \mathbb{Z}, 0 \leq k_2 < 128$ を満たすように $k = 128k_1 + k_2$ と分解する
- (3) $\exp\left(\frac{k_2}{128}\right) = h \times (1 + t + \delta_t), |t| < 2^{-53}, |\delta_t| < 2^{-53} |t|$ を満たす binary64 の定数 h, t を表引きで求める
- (4) $c(r) = \exp(r) - 1$ を $\tilde{c}(r) = r + C_2 r^2 + C_3 r^3 + C_4 r^4 + C_5 r^5$ と多項式近似して求める
- (5) $u(r) = (1 + t + \delta_t)(1 + c(r)) - 1$ を $\tilde{u}(r) = t + c(r)$ と近似して求める
- (6) $\exp(x) = 2^{k_1} h (1 + \tilde{u}(r))$ により指数関数の結果を求める

以下で詳しく手順を説明する。

まず k を求める。やはり正確に求める必要はないため、事前に計算された binary64 の定数である $\llbracket \frac{128}{\log 2} \rrbracket$ を用いて

$$k = \text{round}\left(\left\lfloor x \times \left\lfloor \frac{128}{\log 2} \right\rfloor \right\rfloor\right) \text{ と計算する。}$$

次に r を求める。やはり桁落ちが問題となるが $|k| < 2^{17}$ であることが利用できる。つまり、 $\frac{\log 2}{128}$ の近似値を二つの浮動小数点数の和に分解し、片方は下 17bit が 0 であるようにする。具体的には、 $a1 = 0x1.62E42FEFA00000p-8$ と $a2 = 0x1.CF79ABC9E3B3Ap-47$ の二数に分解する。これを用いて $\tilde{r} = \left\lfloor (x - k \times a1) - \left\lfloor k \times a2 \right\rfloor \right\rfloor$ と計算する。この時、 $x - k \times a1$ は正確に計算される。なお、 $128(a1+a2) \simeq \log 2 + 1.01 \times 10^{-28}$

である。

それと並行して、表引きにより $\exp\left(\frac{k_2}{128}\right) \simeq h \times (1+t)$ を満たす binary64 の定数 h, t を求め、 $2^{k_1}h$ を計算する。実際には h は浮動小数点数レジスタ上にロードするのではなく、そのビット表現を整数レジスタ上にロードする。さらに整数演算を用いて $2^{k_1}h$ を binary64 で表現したときのビット表現を求める。その後、ビット表現をそのままに浮動小数点数レジスタに転送する。

続いて $(1+t+\delta_t)\exp(r)-1$ を $\tilde{u}(r) = t+r+C_2r^2+C_3r^3+C_4r^4+C_5r^5$ と近似し、 $\tilde{r}$ を使ってこれを求める。

最後に、 $\exp(x)$ を $2^{k_1}h(1+\tilde{u}(r))$ で求める。この時、 $|u(r)|$ が小さいため、 $2^{k_1}h \times \llbracket 1 + \llbracket \tilde{u}(r) \rrbracket \rrbracket$ としてしまうと丸め誤差の影響を大きく受けてしまう。そこで $\llbracket \llbracket 2^{k_1}h \times \llbracket \tilde{u}(r) \rrbracket \rrbracket + 2^{k_1}h \rrbracket$ と変形することで丸め誤差の影響を極力回避する。なお、この部分では二回丸めを行っているが、複数回の丸めは丸め誤差を増大させる。積和演算を用いれば正確な丸めが一回だけ行われ、丸め誤差の影響をさらに減らすことができる。

この手法では、 $\exp(x)$ の誤差は主に以下の二点からなる。

- 最終演算での丸め誤差、積和演算を使えば最大で 0.5 ULP
- $\tilde{u}(r)$ の計算誤差、最終結果への影響は小さい

最大の丸め誤差は 1 ULP 未満であることが保証される。

3.4 fast math library の実装

fast math library の実装 [4] では、精度を犠牲にしているものの、以下の手順により高速に指数関数を計算している。

- (1) $k \in \mathbb{Z}, |r| \leq \frac{\log 2}{4096}$ を満たすように $x = \frac{k}{2048} \log 2 + r$ と分解する
 - (2) $k_1, k_2 \in \mathbb{Z}, 0 \leq k_2 < 2048$ を満たすように $k = 2048k_1 + k_2$ と分解する
 - (3) $h = \llbracket \exp\left(\frac{k_2}{2048}\right) \rrbracket$ を表引きで求める
 - (4) $c(r) = \exp(r)$ を $\tilde{c}(r) = 1 + r + C_0r^2(C_1 + r)$ と多項式近似して求める
 - (5) $\exp(x) \simeq 2^{k_1}hc(r)$ により指数関数の結果を求める
- 以下で詳しく手順を説明する。

まず、他の手法と同様、 $\llbracket x \times \llbracket \frac{2048}{\log 2} \rrbracket \rrbracket$ を丸めることで k を求める。この時、丸め命令を使うのではなく、大きな数を加算することで丸める技法を用いる。具体的には、 $k^+ = \llbracket \llbracket x \times \llbracket \frac{2048}{\log 2} \rrbracket \rrbracket + 3 \times 2^{51} \rrbracket$, $k = k^+ - 3 \times 2^{51}$ と計算する。

次に、 r を $\tilde{r} = \llbracket x - \llbracket \frac{\log 2}{2048} \rrbracket k \rrbracket$ と計算する。この計算方法では桁落ち誤差が発生するが、高速化を優先し、誤差の補正は行っていない。その後 $\exp(r)$ を三次の多項式

$\tilde{c}(r) = 1 + r + C_0r^2(C_1 + r)$ で近似し、 $\tilde{r}$ を用いてこれを求める。

これと並行して、表引きにより $\exp\left(\frac{k_2}{2048}\right) \sim h$ を満たす binary64 の定数 h を求め、 $2^{k_1}h$ を計算する。これを行うため、まず k_2 を整数レジスタ上に作成する。具体的には、まず k^+ のビット表現をそのままに整数レジスタに転送する。 k^+ のビット表現を 64bit 符号なし整数として解釈した値は $k + 2151 \times 2^{51}$ に等しい。よって、下 12bit を取り出せば k_2 が得られる。これを用いて 2048 エントリーのテーブルを引き、 h のビット表現を整数レジスタ上にロードする。 k_1 も同様の手順で作成し、整数演算を用いて $2^{k_1}h$ を binary64 で表現したときのビット表現を求める。その後、ビット表現をそのままに浮動小数点数レジスタに転送する。

最後に、 $2^{k_1}h$ と $\exp(r)$ をかけ合わせることで $\exp(x)$ が求まる。

この手法で計算される $\exp(x)$ には、以下に挙げる複数の原因に由来する誤差が含まれるため精度は低い。

- r を計算するときの桁落ち誤差、 $|x|$ が大きいほど大きい
- h の精度、最終結果に最大で 1 ULP の誤差
- $\tilde{c}(r)$ の精度、最終結果に最大で 1 ULP を超える誤差
- 最終演算での丸め誤差、最大で 0.5 ULP

4. 提案手法

4.1 r を高速かつ正確に求める手法

指数関数を計算するとき、 r の値が判明しないことには $\exp(r)$ の近似多項式の計算に取り掛かれない。よって r を高速に求める手法が有用である。もちろん、精度を犠牲にすれば高速に求めること自体は可能である。精度を犠牲にせず r を高速に求める提案手法を以下に示す。 $k \in \mathbb{Z}, r = x - \frac{k}{32} \log 2, |r| \leq \frac{\log 2}{64}$ を満たすように選ぶ時の例を示す。 $|x| < 710$ のみを考えているため、 $|k| \leq 2^{15}$ が成立する。

本手法では、 $\frac{\log 2}{32}$ を二つの binary64 の積で近似する。ここで、片方は下 15bit が 0 であるように選ぶ。具体的には、 $a_1 = 0x1.6EAF5B2E10000p-6$ と $a_2 = 0x1.EF885A58C056Dp-1$ を用いる^{*1}。 $32a_1a_2 \simeq \log 2 - 1.11 \times 10^{-27}$ である。 a_1 の下 15bit が 0 であること、また $|k| < 2^{15}$ が成立していることから、 k を乗じた ka_1 は binary64 で誤差なく表現できる。

以下、 ka_1 を高速かつ正確に求める具体的な手法を示す。 $k = \text{round}\left(x \times \llbracket \frac{32}{\log 2} \rrbracket\right)$ であるが、大きな値を足して丸

^{*1} a_1 の指数部は自由に取れる。また、 a_1 が決まれば最適な a_2 は自動的に決まる。これを利用し、 a_1 の仮数部としてありうる 2^{37} 通りについて全探索を行った。Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz (32 スレッド) を用いて 202 秒で結果が得られた。

める技法を用いれば $k = \left\lfloor x \times \left\lfloor \frac{32}{\log 2} \right\rfloor + 3 \times 2^{51} \right\rfloor - 3 \times 2^{51}$ となる。両辺に a_1 を乗ずると、 $ka_1 = a_1 \times \left\lfloor x \times \left\lfloor \frac{32}{\log 2} \right\rfloor + 3 \times 2^{51} \right\rfloor - 3a_1 \times 2^{51}$ となり、これは積和演算が使える形である。なお、 $3a_1 \times 2^{51}$ は binary64 で正確に表現できる。

この二つを組み合わせ、 $\tilde{r} = \left\lfloor x - (ka_1) \times a_2 \right\rfloor$ のように計算すると、 $\tilde{r}$ を三回の積和演算で求めることができる。また、丸めが発生するのは最後の演算だけであり、非常に正確に求めることができる。

```
static const double a1 = 0x1.6EAF5B2E10000p-6;
static const double a2 = 0x1.EF885A58C056Dp-1;
static const double inv_ln2 = 0x1.71547652B82FEp+0;
static const double a1p = a1 * 0x3p+51;
double kp = fma( x, inv_ln2, 0x3p+51 );
double r = fma( fma( kp, a1, -a1p ), -a2, x );
```

4.2 $\exp(t) - 1$ の 6 次多項式近似を低レイテンシで求める手法

r の多項式を低レイテンシで求めるためには Knuth の Adaptation of coefficients [5] という技法を用いるのが一般的である。この技法を用いて以下のように 6 次多項式を変形すると、4 演算レイテンシで求めることができる。

$$\left((Ar^2 + Br + C)(r^2 + D) + E \right) r^2 + Fr + G$$

この式を求める際、最も早く取り掛かるべきなのは $Ar + B$ の計算である。 r を算出してから $Ar + B$ の計算に取り掛かるのではなく、 r の計算とオーバーラップして計算することでレイテンシを短くする手法を以下に示す。

$r = x - k \log 2$ とすると、 $Ar + B = Ax + B - kA \log 2$ である。ここで $A \log 2$ は定数であり、事前に求めておくことができる。さらに、演算器がパイプライン化されている場合、 k を求めている間に並列して $Ax + B$ を求めることが可能である。これらを利用すると、 $r = x - k \log 2$ が算出されるのと同一のタイミングで $Ar + B = (Ax + B) - kA \log 2$ を算出可能である。

r と $Ar + B$ が求まっている場合、そこから $\left((Ar^2 + Br + C)(r^2 + D) + E \right) r^2 + Fr + G$ を求めるのは 3 演算レイテンシで可能である。

4.3 低レイテンシかつ最大丸め誤差 1 ULP を保証する指数関数実装

4.3.1 設計

提案手法は、以下の手順により指数関数を計算する。

- (1) $k \in \mathbb{Z}, |r| \lesssim \frac{\log 2}{64}$ を満たすように $x = \frac{k}{32} \log 2 + r$ と分解する
- (2) $k_1, k_2 \in \mathbb{Z}, 0 \leq k_2 < 32$ を満たすように $k = 32k_1 + k_2$ と分解する

(3) $\exp\left(\frac{k_2}{32}\right) = h \times (1 + t + \delta_t), |t| < 2^{-53}, |\delta_t| < 2^{-53} |t|$ を満たす binary64 の定数 h, t を表引きで求める

(4) $c(r) = \frac{\exp(r) - r - 1}{r^2}$ を $\tilde{c}(r) = (Ar^2 + Br + C)(r^2 + D) + E$ と多項式近似して求める

(5) $u(r) = (1 + t + \delta_t)(1 + r + r^2 c(r)) - 1$ を $\tilde{u}(r) = t + r + r^2 c(r)$ と近似して求める

(6) $\exp(x) = 2^{k_1} h (1 + u(r))$ により指数関数の結果を求める

以下で詳しく手順を説明する。

4.1 節で述べた手法により、まず $q = ka_1$ を求める。その途中で得られる $k^+ = \left\lfloor x \times \left\lfloor \frac{32}{\log 2} \right\rfloor + 3 \times 2^{51} \right\rfloor$ は、そのビット表現をそのままに整数レジスタへ転送し、fast math library の実装と同様に k_1, k_2 の計算に役立てる。続いて、4.2 節で述べたように、 q から $\tilde{r} = \left\lfloor x - q \times a_2 \right\rfloor$ と $l_r = \left\lfloor \left\lfloor \left\lfloor A \right\rfloor \times x + \left\lfloor B \right\rfloor \right\rfloor + q \times \left\lfloor \left\lfloor a_2 A \right\rfloor \right\rfloor \right\rfloor \simeq Ar + B$ を求める。

それと並行して、ARM による実装と同様、 k_1 と k_2 を用いて、表引きにより $\exp\left(\frac{k_2}{32}\right) \simeq h \times (1 + t)$ を満たす binary64 の定数 h, t を求め、 $2^{k_1} h$ を計算する。

次に、 $s_3 = \left\lfloor \left\lfloor l_r \times \tilde{r} + \left\lfloor C \right\rfloor \right\rfloor \times \left\lfloor \tilde{r}^2 + \left\lfloor D \right\rfloor \right\rfloor + \left\lfloor E \right\rfloor \right\rfloor$ を計算する。これは $\tilde{c}(r)$ の近似値となっている。その後、 $p = \left\lfloor \left\lfloor t + \tilde{r} \right\rfloor + \left\lfloor \tilde{r}^2 \right\rfloor \times s_3 \right\rfloor$ を計算する。これは $(1 + t + \delta_t) \exp(r) - 1$ の近似値となっている。

最後に、 $\exp(x)$ を $\left\lfloor 2^{k_1} h \times p + 2^{k_1} h \right\rfloor$ として求める。

この手法では、 $\exp(x)$ の誤差は主に以下の二点からなる。

- 最終演算での丸め誤差、最大で 0.5 ULP
- $\tilde{c}(r)$ の計算誤差、最終結果への影響は小さい

最大の誤差は 1 ULP 未満であることが保証される。

4.3.2 C/C++コード

$-708.39641853226408 < x < 709.78271289338397$ を満たす x に対する指数関数を計算するコードは以下のとおりである。

```
#include <stdint.h>
#include <string.h>
#include <math.h>

uint64_t bit_cast_d2u( double x ) {
    uint64_t u;
    memcpy( &u, &x, sizeof u );
    return u;
}

double bit_cast_u2d( uint64_t u ) {
    double x;
    memcpy( &x, &u, sizeof x );
    return x;
}

const double h[32] = {
    0x1.0000000000000p+0,
```

```

0x1.059b0d3158574p+0,
0x1.0b5586cf9890fp+0,
0x1.11301d0125b51p+0,
0x1.172b83c7d517bp+0,
0x1.1d4873168b9aap+0,
0x1.2387a6e756238p+0,
0x1.29e9df51fdee1p+0,
0x1.306fe0a31b715p+0,
0x1.371a7373aa9cbp+0,
0x1.3dea64c123422p+0,
0x1.44e086061892dp+0,
0x1.4bfdad5362a27p+0,
0x1.5342b569d4f82p+0,
0x1.5ab07dd485429p+0,
0x1.6247eb03a5585p+0,
0x1.6a09e667f3bcdp+0,
0x1.71f75e8ec5f74p+0,
0x1.7a11473eb0187p+0,
0x1.82589994cce13p+0,
0x1.8ace5422aa0dbp+0,
0x1.93737b0cdc5e5p+0,
0x1.9c49182a3f090p+0,
0x1.a5503b23e255dp+0,
0x1.ae89f995ad3adp+0,
0x1.b7f76f2fb5e47p+0,
0x1.c199bdd85529cp+0,
0x1.cb720dcef9069p+0,
0x1.d5818dcfba487p+0,
0x1.dfc97337b9b5fp+0,
0x1.ea4afa2a490dap+0,
0x1.f50765b6e4540p+0,
};

const double t[32] = {
    0x0p+0,
    0x1.cd2523567f613p-55,
    0x1.79aa65d837b6ap-54,
    -0x1.556522a2fbd10p-54,
    -0x1.01b15eaa5934ap-55,
    0x1.aecf73e3a2f61p-54,
    0x1.68efde3a8a893p-54,
    0x1.2f7e16d09ab2dp-55,
    0x1.34d754db0abb7p-55,
    -0x1.24aedcc4b5069p-54,
    0x1.59f48a72a4c6bp-55,
    0x1.363ed60c2ac2cp-59,
    0x1.690cebb7aafb0p-56,
    -0x1.8dec6bd0f386ap-56,
    0x1.063e1e21c540ap-54,
    -0x1.c33c53bef4da7p-55,
    -0x1.3b3efbf5e2228p-54,
    -0x1.81f647e5a3ed0p-56,
    -0x1.b32dcb94da517p-56,
    -0x1.369b6f13b3734p-54,
    0x1.db72fc1f0eab7p-55,
    -0x1.da9b88b6c1e48p-58,
    0x1.1affc2b91ce22p-56,
    -0x1.1bbd1d3bcb15p-54,

```

```

0x1.c1a7792cb3389p-55,
-0x1.8d6f438ad9351p-57,
0x1.36eae30af0cb0p-56,
0x1.76b2c6c92196dp-57,
0x1.4a385a63d07a9p-56,
-0x1.2d52107b43e21p-55,
-0x1.ffa7128fd391f1p-55,
0x1.a64a931d185edp-55,
};

inline double exp_main( double x ) {
    static const double A = 1 / 720.0;
    static const double B = 1 / 120.0;
    static const double C = 1 / 72.0;
    static const double D = 20.0;
    static const double E = 1 / 4.5;

    static const double a1 = 0x1.6EAF5B2E10000p-6;
    static const double a1p = a1 * 0x3.p+51;
    static const double a2 = 0x1.EF885A58C056Dp-1;
    static const double c0 = 1 / a1 / a2;
    static const double a2A = a2 * A;

    const double kp = fma( x, c0, 0x3.p+51 );
    const double q = fma( kp, a1, -a1p );
    const double r = fma( q, -a2, x );
    const double lx = fma( A, x, B );
    const double lr = fma( q, -a2A, lx );
    const double s1 = fma( lr, r, C );
    const double s2 = fma( r, r, D );
    const double s3 = fma( s1, s2, E );
    const double rr = r * r;
    const uint64_t k1 = bit_cast_d2u( kp ) >> 5;
    const uint64_t k2 = bit_cast_d2u( kp ) & 0x1f;
    const double det = t[k2] + r;
    const double poly = fma( s3, rr, det );
    const uint64_t tableau
        = bit_cast_d2u( h[k2] ) + ( k1 << 52 );
    const double table = bit_cast_u2d( expsu );

    return fma( table, poly, table );
}

```

5. 評価

ソースコードのコンパイルは全て gcc 9.1 を用いて行った。最適化オプションは -O3 -march=skylake である。

5.1 レイテンシ

Intel(R) Architecture Code Analyzer (IACA) を用いたシミュレーションを行い、レイテンシの評価をした。想定アーキテクチャは Skylake とした。Skylake アーキテクチャにおける主要な命令のレイテンシは表 1 に記載した。

シミュレーションの結果、3.3 節で述べた ARM による既存実装のレイテンシは 33 サイクルであった。一方、提

コードでの 変数名	占有ポート (一例)	アセンブリコード	4	8	12	16	20	24	28
		vmovapd xmm2, xmm0							
kp	p0	vfmadd132sd xmm2, xmm12, xmm13	vfmadd						
		vmovapd xmm1, xmm2							
q	p1	vfmadd132sd xmm1, xmm10, xmm11		vfmadd					
	p0	vmovq rcx, xmm2		vmovq					
k2	p5	and ecx, 0x1f		and					
	p0	vmovq rax, xmm2		vmovq					
k1	p6	shr rax, 0x5		shr					
		vmovapd xmm14, xmm1							
r	p1	vfmadd132sd xmm14, xmm0, xmm9		vfmadd					
lx	p1	vfmadd132sd xmm0, xmm7, xmm8	vfmadd						
	p6	shl rax, 0x34		shl					
tableu	p2 + p5	add rax, qword ptr [rsi+rcx*8]		qword ptr [rsi+rcx*8]	add				
lr	p0	vfmadd231sd xmm0, xmm1, xmm6		vfmadd					
		vmovapd xmm1, xmm14							
s2	p1	vfmadd132sd xmm1, xmm4, xmm14				vfmadd			
s1	p0	vfmadd132sd xmm0, xmm5, xmm14				vfmadd			
s3	p1	vfmadd132sd xmm0, xmm3, xmm1					vfmadd		
rr	p0	vmulsd xmm1, xmm14, xmm14				vmulsd			
det	p3 + p1	vaddsd xmm14, xmm14, qword ptr [rdi+rcx*8]		qword ptr [rdi+rcx*8]	vaddsd				
poly	p0	vfmadd132sd xmm0, xmm14, xmm1					vfmadd		
table	p5	vmovq xmm1, rax				※			
	p1	vfmadd132sd xmm0, xmm1, xmm1							vfmadd

図 1 提案実装の機械語コードがパイプラインを流れる様子

表 1 Skylake における主要命令のレイテンシ

整数命令	1 cycle
浮動小数点数レジスタ→整数レジスタ転送命令	2 cycle
整数レジスタ→浮動小数点数レジスタ転送命令	1 cycle
浮動小数点数命令 (加算・乗算・積和演算)	4 cycle
ロード命令	5 cycle

案手法のレイテンシは 28 サイクルであり、ARM による既存実装より 15% 短いレイテンシを達成した。

図 1 は、IACA の出力結果を整形した、提案手法の機械語命令列がパイプラインを流れる様子である。

5.2 精度

精度の評価は乱数を用いて行った。提案手法の `exp_main` 関数に乱数を入力し、関数の返り値と真の値との差を求めた。真の値を求めるためには、精度保証演算ライブラリである `kv` [6] を用いた。

`exp_main` 関数に入力する乱数は、次のように作成した。まず、メルセンヌツイスタを用いて、64bit のビット列を一樣ランダムに生成した^{*2}。そして、そのビット表現を binary64 で表現された浮動小数点数だと解釈した際、 -708.39641853226408 より大きく 709.78271289338397 より小さい浮動小数点数のみを抽出した。これを繰り返し、条件を満たす乱数を一億個用意した。

この実験を行った結果、全体の 0.008% にあたる、7911 ケースについて、真の値との差が 0.5 ULP を超えた。最大の誤差は $-0x1.6B4D3128456B1p-7$ を入力したときの 0.549 ULP であった。

6. まとめ

指数関数 $\exp(x)$ は多くの応用アプリケーションで用いられるが、CPU 命令で計算できないことが多い。CPU 命

令のみを用いるソフトウェア実装が研究されてきたが、その評価軸は演算回数であった。

本稿では、レイテンシを短くすることに役立つ二つの手法を提案した。一つ目は、 $\log 2$ の近似値を二つの binary64 の和として保持するのではなく、二つの binary64 の積として保持する手法である。この表現を用いることで、 $r = x - k \log 2$ の計算に積和演算を利用することができ、 r を正確かつ高速に求めることができる。二つ目は、式変形により依存の連鎖を短くする手法である。計算結果の再利用ができなくなり演算数が増加するものの、演算器がパイプライン化されている場合はレイテンシを削減することに役立つ。

さらに、それらを用いたレイテンシ指向の新たな実装を示した。本稿で提案した実装は、最大丸め誤差 1 ULP 未満を保証しつつ、従来と比べ 15% 短いレイテンシを達成した。

参考文献

- [1] Zuras, D., Cowlishaw, M., Aiken, A., Applegate, M., Bailey, D., Bass, S., Bhandarkar, D., Bhat, M., Bindel, D., Boldo, S., *et al.*: IEEE standard for floating-point arithmetic, *IEEE Std 754-2008*, pp. 1–70 (2008).
- [2] Sun Microsystems, Inc.: `exp(x)`, <http://git.musl-libc.org/cgit/musl/tree/src/math/exp.c?id=9b0fcb441a44456c7b071c7cdaf90403f81ec05a>
- [3] Arm Limited: Double-precision e^x function., <https://git.musl-libc.org/cgit/musl/tree/src/math/exp.c?id=5fc43798250255455e4b5f9b08000bd3102274d9>
- [4] herumi: fast math library for float, <https://github.com/herumi/fmath/blob/master/fmath.hpp>
- [5] Knuth, D. E.: *The Art of Computer Programming*, Vol. 2, Addison-Wesley, 3rd edition (1998).
- [6] Masahide, K.: `kv` - a C++ Library for Verified Numerical Computation version 0.4.47, <https://github.com/mskashi/kv>

^{*2} C++標準ライブラリに搭載されている `std::mt19937` と `std::uniform_int_distribution<uint64_t>` を用いた。