Vol.2022-HPC-185 No.4
2022/7/27

NumPy 超越関数処理の Arm SVE 向け最適化

山田芙夕楓 [1, a)]　川上健太郎 [1]　栗原康志 [1]　松田一仁 [1]　田原司睦 [1]

概要：Arm プロセッサ用の HPC 向け命令セットである SVE が開発され，スーパーコンピュータ「富岳」では，この命令セットを採用した Fujitsu Processor A64FX が使用されている．Python の数値演算ライブラリである NumPy は SVE に対応しておらず，A64FX の性能を十分に引き出せていない．そこで今回 NumPy を SVE 対応したので報告する．特に配列に対する超越関数の処理では，A64FX の性能を引き出すように①SIMD 化と②ループ展開数の最適化の 2 つの工夫を行い，配列データに対する超越関数処理を 24 倍高速化した．

Optimization of NumPy Transcendental Function for Arm SVE

YAMADA FUYUKA[1, a)]　KAWAKAMI KENTARO[1]　KURIHARA KOUJI[1]
MATSUDA KAZUHITO[1]　TABARU TSUGUCHIKA[1]

Abstract: SVE which is an instruction set for HPC for the Arm processor was developed, and the supercomputer Fugaku adopts Fujitsu Processor A64FX which adopted this instruction set. NumPy, the fundamental package for scientific computing with Python, does not support SVE and does not fully exploit the performance of the A64FX. Then, we added SVE implementations to NumPy, and we will report on it in this paper. In particular, in transcendental functions processing for the array, we made two improvements to bring out the performance of the A64FX. The first is SIMD, and the second is optimizing the number of loop unrolling. As a result, transcendental functions processing for array data was made 24 times faster.

1. はじめに

SVE(Scalable Vector Extension)命令とは Arm が HPC 用途に定義した命令セットである[1]．ベクトルレジスタサイズは 128 ビット単位で，最大 2048 ビットまで定義されており，CPU の製造ベンダが選択することができる． Fujitsu A64FX[2] は，SVE 命令セットに対応しており，ベクトル長は 512 ビットを採用している．2021 年 3 月から正式共用が開始されているスーパーコンピュータ「富岳」は，この A64FX CPU を採用している[3].

NumPy[4] とは，Python 上での数値/論理演算を高速化する演算ライブラリであり，AI・機械学習など種々のアプリケーションで用いられている．NumPy は様々な CPU アーキテクチャ・命令セットに対応している．Arm アーキテクチャに対しては 128 ビットベクトル長の処理が可能な NEON 命令に対応しているが， SVE 命令セットには対応していない．そのため，NumPy を A64FX 上で使用した場合, A64FX の性能を十分に引き出せない. 本稿では, NumPy の SVE 対応について報告する. その中でも特に配列データに対する超越関数処理の高速化について述べる.

本稿の構成は以下の通りである. 2 章で NumPy の実装の仕組み, SVE への対応方法について述べる. 3 章では NumPy の SVE 向け超越関数の実装について述べる. 従来の NumPy を A64FX で動かした場合，超越関数は GNU C Library の libm.so 数学ライブラリが使われるため，処理が SIMD 化されていなかった．これを既存 OSS の SLEEF[5][6]を利用して SIMD 化した．またループアンローリングを用いてさらなる高速化を行った. pragma によるループアンローリングとソースコードとして記述した実装によるループアンローリングを比較する．4 章で本稿のまとめを述べる.

2. NumPy for SVE の実装

1 章で述べたとおり，NumPy は広く使われている Python の数値演算ライブラリである．インストールには 2 つの方法があり， Python のパッケージ管理ツールである pip[7] などを用いてコンパイル済みパッケージをインストールする方法と, またはソースコードからビルドする方法がある. NumPy のソースコードはオープンソースソフトウェア(OSS)として公開されており, 誰でも使用や改変が可能である.

従来の NumPy のソースコードは，Python の実装と C の実装の階層構造になっている（図 1）. 配列データに対する基本的な四則演算や論理演算は C で実装されており，コンパイラによる自動ベクトル化を期待して実装された部分と, SIMD 命令を使うように intrinsic を用いて直接的に実装された部分がある．intrinsic の名前は CPU 命令セットごとに異なるため，npyv_で始まる識別子を#define することで抽象化されている．したがって，SVE 命令セットに対応する intrinsic と npyv_*の対応関係を実装することで，四則演算

1 富士通株式会社
Fujitsu Limited.
a) yamada.fuyuka@fujitsu.com

や論理演算については SVE 対応を実現することができる．

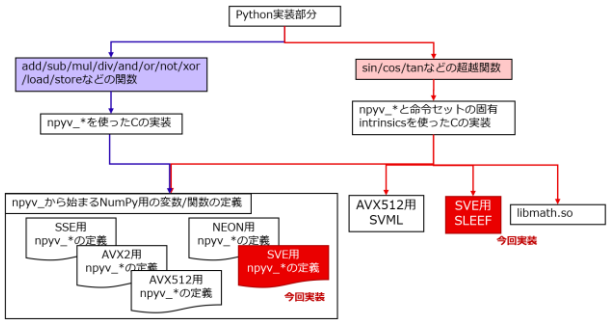

図 1 NumPy の実装イメージ

表 1 に各命令セットの intrinsic 関数や，SIMD データ型と NumPy のソースコード上で抽象化された型の対応の例を示す．npyv_f64 は 64 ビット浮動小数のベクトルデータを表す型を抽象化したデータ型であり，npyv_add_f64 はこのデータを加算する intrinsic 関数を抽象化した関数である．表 1 に示した例の他に，32 ビット浮動小数，符号付き/符号なしの 8/16/32/64 ビット整数のベクトルデータの型と，これらに対する四則演算，論理演算，ロード/ストア命令に対応する intrinsic 関数が定義されている．今回，NumPy を SVE 対応するにあたり，この npyv_から始まる NumPy の変数/関数の定義を SVE 向けに実装した．SVE の C/C++言語向け intrinsic である ACLE（Arm C Language Extensions）[8]を用いた．

表 1 NumPy の変数/関数と命令セットの変数/関数の対応例

	命令セット		
抽象化されたベクトルデータの型，intrinsic	AVX512	NEON	SVE
npyv_f64	__m512d	float64x2_t	svfloat64_t
npyv_add_f64(a, b)	_mm512_add_pd(a, b)	vaddq_f64(a, b)	svadd_f64_x(svptrue_b64(), a, b)

SVE 命令では AVX512 命令と同様に，SIMD レーンごとに演算を行う/行わないを制御することができる．SVE 向けの intrinsic の多くでは第 1 引数としてこの情報を与える必要がある．この情報はプレディケートと呼ばれ，svptrue_b64()関数はすべての SIMD レーンを有効にするプレディケートを与える．扱うデータ型によって関数名が異なり，64 ビットデータのベクトルデータを扱う場合は svptrue_b64()，32 ビットデータのベクトルデータの場合は svptrue_b32()となる．

NumPy では四則演算などを行う関数は入力配列をベクトルに変換する部分と，npyv_add_f64 などのベクトル演算を行う部分が別々に実装されている．一方，sin/cos などの超越関数は複数命令を組み合わせて計算する必要があるため，から始まる抽象化された関数が定義されておらず，入力の配列をベクトルに変換する部分と，実際にベクトル計算を行う部分が複数行にわたって実装されている．従来の NumPy の超越関数には，AVX512 用には AVX512 の intrinsic を直接用いた実装がされている（図 1）が， SVE 向けにビルドした場合，libm ライブラリの超越関数がリンクされ使用される．AVX512 の場合，超越関数の種類によってはベクトル算術関数ライブラリである SVML（Short Vector Mathematical Library）組み込み関数[9]をリンクして使用する実装がされており，超越関数の SIMD 化がされている．今回，SVE 向けに SLEEF を用いた SIMD 化実装を追加した．

SVE 用に実装した関数を NumPy で使用するには，ビルド時のチェックでターゲットの命令セット向けの機能が追加される必要がある．図 1 の「C の実装」には，ソースコードのヘッダー部分に命令セットのキーワード（AVX512, NEON など）が記載されており，ビルドターゲットとする命令セットに対応する C のソースコードが収集され，コンパイル・リンクが行われる．

今回の NumPy for SVE の実装では，AVX512 や NEON 向けの実装を参考に，プログラムのヘッダー部分に「SVE」のキーワードの追加や，SVE のコンパイルチェックプログラムの追加を行い，NumPy ビルドの SVE 対応を行った．

3. SVE 向け超越関数処理の高速化

NumPy でサポートされている種々の超越関数の実装を SVE 向けに新規に開発するのは工数が大きくなる．ベースとなる実装として，SLEEF のソースコードを流用することで，開発期間の短縮を目指した．

3.1 SLEEF の特徴

SLEEF（SIMD Library for Evaluating Elementary Functions）は C 言語で書かれた OSS の SIMD 演算ライブラリである[5][6]．ライセンスは Boost ソフトウェア・ライセンス 1.0（BSL-1.0）である．SLEEF は Intel アーキテクチャ，Arm アーキテクチャ，RISC-V アーキテクチャなどを含むマルチアーキテクチャ対応で実装されており，ACLE で記述された SVE 命令セット向けの実装も含まれている．

3.2 配列データに対する超越関数処理の実装

配列データに対する超越関数を実装した方法について述べる．2 章で説明したように，NumPy には AVX512 向けに SVML を用いて SIMD 対応された超越関数が実装されている．その実装では，以下のように実装されている．まず入力配列と出力配列のメモリのオーバーラップのチェックを行う．

- オーバーラップしていない場合，入力データをスカラー型からベクトル型へ変換し，超越関数の SIMD 演算を行うループを実行する．この際，超越関数の演算には SVML をリンクして使用する．
- オーバーラップしている場合は，スカラー演算用のループを実行する．この際， libm ライブラリを使用する．

SVML は 1 回の関数 call で SIMD レーン数分のデータの処理を行うため，AVX512 向け実装のように SVML ライブラリをリンクして使用する方法だと，1 ベクトルの計算ごとに関数 call が発生し，オーバーヘッドが大きくなることが欠点である．SVE 用超越関数の実装は，AVX512 向け実装の SVML を呼び出す部分で SLEEF を呼び出すように行ったが，ライブラリをリンクするのではなく，インライン関数としてソースコードを埋め込む形で実装した．これにより，AVX512 向け実装とは異なり，任意長の配列に対する超越関数の処理を 1 関数で実行できる．

SLEEF の超越関数処理部分のソースコードは，SLEEF のビルド時にオプションを追加すると入手できる．具体的には，SLEEF のソースコードを git から clone してビルドを行い，cmake 時に”-DBUILD_INLINE_HEADERS=TRUE”とオプションを付けると，”sleefinline_sve.h”というヘッダファイルが生成される．

3.3 SVE 向け超越関数の実験方法と結果

実験では，SVE 命令セットが使用可能な CPU の A64FX が搭載された FX700 を使用した．NumPy のオリジナルの超越関数処理と SLEEF を実装した log10 の関数処理を 1 コアで実行し，実行時間を比較した．

実験方法：1,024 個の double 型データを持つ配列に対して，要素ごとに log10 の計算を 100 万回繰り返すプログラムの処理時間で比較した．結果を図 2 に示す．SIMD 化したことでオリジナルの NumPy より 6.0 倍の高速化効果が確認できた．

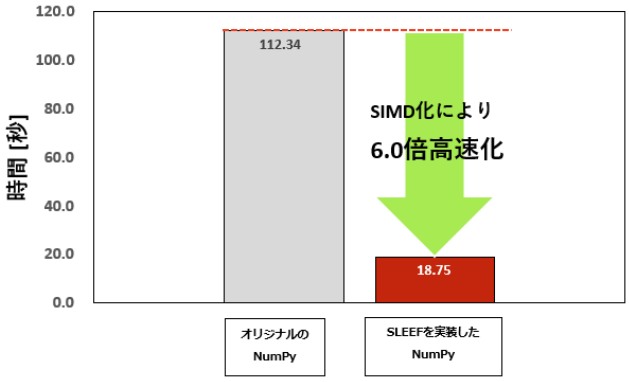

図 2 log10 関数の実行時間比較
(左)オリジナル，(右)SIMD 化したもの

富士通製プロファイラの fapp を用いて処理時間の内訳を取得した．今回実装した配列に対する超越関数計算部分のみを対象としたプロファイルを取得したいため，当該処理部分のみを抽出した C のプログラムを作成し，それを用いてプロファイリングを行った．結果が図 3(a)の棒グラフである．

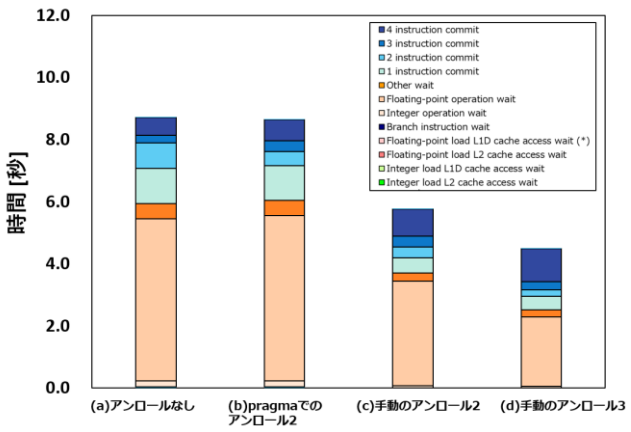

図 3 fapp のプロファイリング結果

命令の実行完了を待っている時間が全体に占める割合は 68%で，1 つ以上の命令を実行結果がコミットされる時間の割合は 32%である．表 2 に，この結果から求められる浮動小数点演算の理論上限値に対して今回の結果が占める割合を示す．実効効率は 28%である．これはプログラム実行中に平均して A64FX の演算器を 28%しか使用できていないということになり，A64FX の性能を最大限に引き出せているとは言えない．この結果から，まだ改善の余地があると判断したため，次の 3.4 では更なる高速化に取り組んだ．

3.4 ループ処理の工夫による高速化

図 3(a)の結果から，3.3 の実装はレジスタ依存のある命令が連続し実効効率が低くなっていることが考えられる．これを解決するため，この章ではループ処理の高速化技術であるループアンローリングを使って更に高速化を行う方法について検討する．ループアンローリング（ループ展開）は，ループの反復回数を減らすようにループ内の処理を複製し，一度のループで複数レジスタ分のデータ処理を行う．プログラムのサイズは大きくなるが，命令スケジューリングの自由度が増え，並列実行可能な命令が増えることで，演算器の使用効率が高まることが期待できる．今回は，pragma によるループアンローリングと，ソースコード自体を書き替えて，処理をループアンローリング化させた 2 種類の方法を評価した．

3.4.1 実験環境・方法

実験では，SVE 命令セットが使用可能な CPU の A64FX が搭載された FX700 を使用し，1 コアで実行した結果を用いた．コンパイラについては，今回行った log10 関数のアンローリングの実験では全体を通して fcc（富士通コンパイラ）が gcc よりも実行時間・高速化率が優れていたため，

fcc の trad モード[(a]での結果を記載している．fcc のオプションは-Kfast を使用した．

2 種類のループアンローリングの実験は，C のプログラムで行った．3.3 で実装した NumPy の log10 関数の計算を再現する C のプログラムを用意して，ループのアンロール数の指定を pragma で行った．実験には 3.3 と同じく，1,024 要素の配列に対して，要素ごとに log10 の計算を行う処理を 100 万回繰り返すプログラムの処理時間を測定した．

3.4.2 pragma によるループアンローリング

コンパイラによる自動ループアンローリングを行う．例えばアンロール数 2 としてループアンローリングした実行コードを生成させたい場合，for ループ文の前に，「#pragma loop unroll 2」と 1 行追加する．実際に使用した C プログラムのソースコードの一部を図 4 に示す．

```
const int n = 1024;
const int l = 1000000;
double src[n];
double dst[n];
int vstep = 8; /* num of SIMD lanes */
const svbool_t tp = svptrue_b64();
for (int k=0; k<l; k++){
  int j=0;
  #pragma loop unroll 2
  for (int i=n; i > vstep; i -= vstep, j += vstep) {
    svfloat64_t x = svld1_f64(tp, src+j);
    /* calculate log10(x) */
    x = Sleef_log10dx_u10sve(x);
    svst1_f64(tp, dst+j, x);
  }
}
```

図 4 pragma でループアンロールするプログラム例

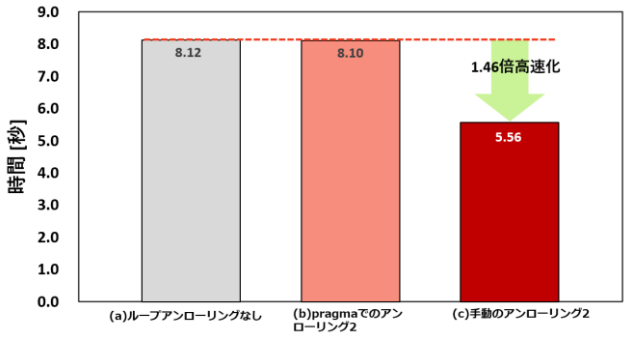

図 5 log10 関数の処理時間
(左)アンローリングなしの場合，
(中央)コンパイラによる自動アンローリング(アンロール数 2)，
(右)ループアンローリングするように作成したソースコード(アンロール数 2)

Sleef_log10dx_u10sve 関数は 64 ビット浮動小数点型のベクトルデータに対して，要素ごとに log10 の計算を行う inline 関数であるる．コンパイラが，これを呼び出す for ループ 2 回分を 1 回のループで実行できる実行コードを生成し，さらに極力レジスタ依存のある命令が連続しないように命令スケジューリングすることができれば，高い実効効率を実現する実行コードが生成できることを期待できる．

図 5(b)に処理時間の実測結果を示す．併せて，fapp で分析した結果を図 3(b)に示す．ループアンローリングなしの場合と比べて，アンロール数 2 での高速化は 0.2%に留まり，ほとんど処理時間に変化はない．fapp の結果を見ると，1 サイクルに 4 命令分実行結果をコミットしている時間（紺色）が増えており，2 命令分実行結果をコミットしている時間（濃い水色）が減少している．今回の pragma でアンロール数 2 のループアンロールを行った場合の，実行コードの逆アセンブル結果を図 6 に示す．

```
.L1594
(5命令省略)
ld1d    {z5.d}, p0/z, [x0, 0, mul vl] /* 1つ目のベクトルをロード */
fmul    z16.d, z5.d, z16.d
(94命令省略)                          /* 1つ目のベクトルのlog10計算 */
st1d    {z5.d}, p0, [x0, 0, mul vl]   /* 1つ目のベクトルをストア */
add x0, x2, 8
(1命令省略)
ld1d    {z5.d}, p0/z, [x1, 0, mul vl] /* 2つ目のベクトルをロード */
fmul    z16.d, z5.d, z16.d
(94命令省略)                          /* 2つ目のベクトルのlog10計算 */
st1d   {z5.d}, p0, [x0, 0, mul vl]    /* 2つ目のベクトルをストア */
add x2, x2, 16                        /* 2ベクトル分のループカウント */
(2命令省略)
bge    .L1594
```

図 6 pragma でアンロール数 2 を指定した場合のアセンブラの例

図 5 をみると，1 つ目のベクトルの log10 計算を行った後，2 つ目のベクトルの log10 計算を行うアセンブラ命令列になっている．この命令列だと，z5 レジスタにロードする ld1d 命令と z5 レジスタの値を使用する fmul 命令が連続していて，z5 レジスタがロードされるまでの待ち時間を考慮した命令並び替えが行われていない．今回の pragma でループアンローリングを行った意図は，1 つ目ベクトルの処理と 2 つ目ベクトルの処理は依存関係がないため，z5 レジスタの値をロードしている間に，2 つ目ベクトルの値のロードを行うなど，レイテンシを考慮した命令スケジューリングが行われることを予想していた．

今回実験した log10 の計算において，pragma でループアンローリングする方法だと，レジスタの依存関係を考慮したスケジューリングを行うのは難しいと判断した．この結果は gcc でも同様であった．

3.4.3 手動でのループアンローリング

3.4.2 の結果より，pragma よるループアンローリングでは，レジスタ依存のある命令のレイテンシを考慮した命令スケジューリングが難しいと判断した．そのため本節では，依存関係のある命令同士が離れるように，アンローリングしたソースコードを手で作成した．作成したソースコードの一部を図 7 に示す．

(a 今回使用した fcc の trad モードでは，ACLE の使用は公式には非対応であるため，fcc の trad モードで ACLE を使用される場合はその点をご了承願いたい．

アンローリングなし

```
const int n = 1024;
const int l = 1000000;
double src[n];
double dst[n];
int vstep = 8; /* num of SIMD lanes */
const svbool_t tp = svptrue_b64();
for (int k=0; k<l; k++){
  int j = 0;
  for (int i=n; i > vstep; i -= vstep, j += vstep) {
    svfloat64_t x = svld1_f64(tp, src+j);
    x = Sleef_log10dx_u10sve(x)
    svst1_f64(tp, dst+j, x);
}}
```

手動アンローリング2

```
const int n = 1024;
const int l = 1000000;
double src[n];
double dst[n];
int vstep = 8; /* num of SIMD lanes */
const svbool_t tp = svptrue_b64();
for (int k=0; k<l; k++){
  int j = 0;
  for (int i=n; i > 2*vstep; i -= 2*vstep, j += 2*vstep) {
    svfloat64_t xa = svld1_f64(tp, src+j);
    svfloat64_t xb = svld1_f64(tp, src+j+vstep);
    /* log10の計算 start */
    ... (9x2行省略)
    svfloat64_t ua = vadd_vd_vd_vd_sve_sleef(tp, xa)
    svfloat64_t ub = vadd_vd_vd_vd_sve_sleef(tp, xb)
    ...(47x2行省略)
    /* log10の計算 end */
    svst1_f64(tp, dst+j, xa);
    svst1_f64(tp, dst+j+vstep, xb);
}}
```

図 7 アンロールを実装したソースコード

ループアンローリングするように実装したソースコードのプログラムの処理時間を 図 5(c)に示す．アンローリングなしの場合と比べて 1.43 倍高速化しており，pragma によるアンローリングよりも明らかに速くなっている． fapp で分析した結果が図 3(c)である．アンロールなしの場合と比べて，浮動小数点演算の待ち時間が減っており，実効効率が向上していることがわかる．

アンロールを手で実装したプログラムの逆アセンブル結果を図 8 に示す．

```
.L2438
(9命令省略)
ld1d    {z1.d}, p0/z, [x8, 0, mul vl]  /* 1つ目のベクトルをロード */
ld1d    {z0.d}, p0/z, [x1, 0, mul vl]  /* 2つ目のベクトルをロード */
(5命令省略)                 /* 1つ目と2つ目のベクトルのlog10計算start */
fmul    z2.d, z1.d, z9.d
(1命令省略)
fmul    z1.d, z0.d, z9.d
(173命令省略)               /* 1つ目と2つ目のベクトルのlog10計算end */
st1d    {z0.d}, p1, [x8, 0, mul vl]    /* 1つ目のベクトルをストア */
st1d    {z1.d}, p1, [x3, 0, mul vl]    /* 2つ目のベクトルをストア */
add x3, x3, 128                        /* 2ベクトル分のループカウント */
bne     .L2438
```

図 8 手でアンロール数 2 を実装したプログラムの逆アセンブル結果

1 つ目と 2 つ目のベクトルの log10 を並列に計算している．図 6 と比べて，依存関係のある ld1d（ロード)命令と fmul（乗算）命令の間にいくつかの命令が存在しており，距離が離れていることも確認できる．

これらの結果から，今回の log10 を計算する処理において，アンローリングするようにソースコードを記述する方法が高速化に有効だとわかった．

3.4.4 アンロール数による比較

図 8 のアンロール数 2 の実験では, z1 レジスタを利用する ld1d 命令と fmul 命令の間は 6 命令離れているが，ld1d 命令のレイテンシは 11 であり, レイテンシ分の距離は確保できてはいない． そのためアンロール数 2 では A64FX の演算器を十分に使いきれていない可能性がある. 一般的に，アンロール数を増やすと依存関係のない並列実行可能な命令が増えるため，演算器を使いきれていない場合には，アンロール数を増やすことによって高速化が期待される．そのため, 複数のアンロール数で処理時間を測定･比較した．結果を図 9 に示す．

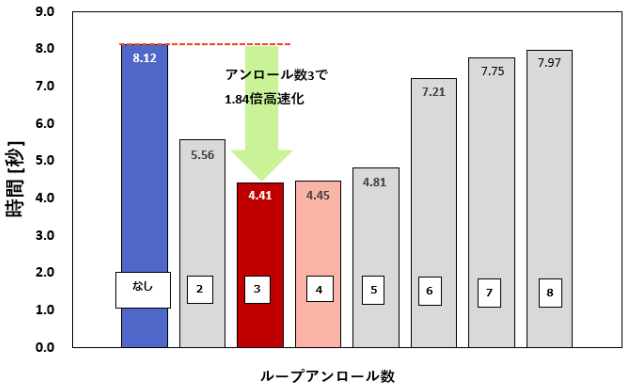

図 9 アンロール数の違いによる log10 関数の処理時間の比較

アンロールなしの場合と比べて，アンロール数 3 が 1.84 倍の高速化で一番処理時間が短いことがわかる．アンロール数 4 の場合も処理時間はほとんどアンロール数 3 と同じである．アンロール数 3 と 4 で実行アセンブラを比較すると，アンロール数 4 の場合にはレジスタの数が足りずレジスタスピルが起こり，データのメモリ退避を行っていることがわかった．他方，アンロール数 3 よりもループ反復回数が減って分岐命令の実行回数が減るため最終的な処理時間はアンロール数 3 の場合とほとんど変わらなかったと考えられる．アンロール数が 5 より大きくなると，レジスタスピルの頻度が増えアンロール数が増えるほど処理時間は悪化していく．今回実験した log10 の計算において，適したアンロール数は 3 と 4 であり，どちらも同等の高速化効果が得られるとわかった．

アンロール数 3 の場合の fapp で分析した結果を図 3(d)に示す．アンロールなしの場合に比べて待ち時間が減少していることがわかる．fapp で取得した，プログラム実行中の浮動小数点演算の理論値に対する実測値を表 2 に示す．A64FX の CPU の動作周波数は 2GHz，1CPU に 2 つの 512 ビット幅 SIMD 演算器を備え， 1 つの SIMD 演算器では倍精度浮動小数点演算が 8 並列に行えるため，2*2*8=32 で，倍精度浮動小数点演算の理論値は 32[GFLOPS(倍精度)]となる.アンローリングを行うことで，GFLOPS（倍精度）が 8.85 から 17.17 まで向上し，実行効率が 1.93 倍になっている．

表 2 アンロールによる倍精度浮動小数点演算性能の理論値と実測値の比較

	倍精度浮動小数点演算回数[GFLOPS(倍精度)]		
	実行効率（実測値/理論値）	実測値	理論値
ループアンローリングなし	28%	8.85	32
アンロール数 3	54%	17.17	32

今回の log10 関数ではアンロール数 3 や 4 の場合に高速化が効きやすいことがわかったが，計算に使用するレジスタ数は関数によって異なるため，レジスタスピルが起こらない程度のアンロール数は関数ごとに異なる．そのため，ループ処理をループアンローリングで高速化したい場合，処理ごとに最適なアンロール数を調査し適応することが重要である．

3.5 C のプログラム上での SVE 対応の結果

3.3 の SLEEF による SIMD 化による高速化の結果と，3.4 のループアンローリングによる高速化の結果をまとめた比較を行う．3.4 では C のプログラムで実験・比較を行ったが，これはループアンローリングによる高速化を NumPy にまだ実装できていないためである．そのため，C のプログラムでオリジナル NumPy の log10 関数相当のプログラムを用意し，比較を行った．その結果を図 10 に示す．

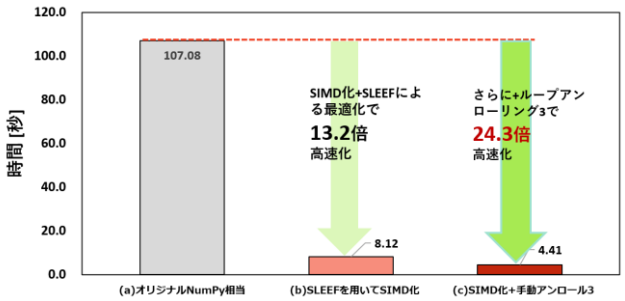

図 10 C プログラムで SIMD 化とアンローリングを行った結果

4. まとめ

NumPy は SVE に対応しておらず，A64FX では十分な性能を発揮できていなかった．そのため今回，NumPy の関数を SVE 向けに実装した．本稿では，その中でも超越関数の SVE 向け実装について述べた．

超越関数の SVE 向け実装で行った工夫は以下の 2 点である．

- SIMD 化
- ループアンローリング

一つ目の SIMD 化については，SLEEF ライブラリのソースコードを流用して NumPy に実装し，オリジナルの NumPy と比べて log10 関数の 6 倍の高速化を確認した．

二つ目のループアンローリングについては，ループアンローリングするように記述した C のプログラムを作成し，様々なループアンロール数で実験した．その結果，log10 の計算においては，アンロール数 3 と 4 が高速化の効果が高いことが分かった．アンロール数 3 でループアンローリングした場合，しなかった場合と比べて 1.84 倍の高速化を確認できた．

今後の課題として，ソフトウェアパイプライニングを適用したさらなる実効効率改善が挙げられる．開発成果は，NumPy コミュニティにアップストリームを行っていく予定である． SVE 命令を採用した CPU は現時点では A64FX だけであるが，Graviton3[10] など SVE を採用した CPU が増えていくため，本開発成果は様々な計算機システムで利用できる．

5. 参考文献

[1] Nigel Stephens, et al., “The ARM Scalable Vector Extension”, in CoRR, 2018
[2] A64FX (online), https://github.com/fujitsu/A64FX (accessed 2022-06-10).
[3] Fujitsu and RIKEN Complete Joint Development of Japan’s Fugaku, the World’s Fastest Supercomputer (online), https://www.fujitsu.com/global/about/resources/news/press-releases/2021/0309-02.html (accessed 2022-06-10)
[4] NumPy (online), https://numpy.org/ (accessed 2022-06-10).
[5] SLEEF (online), https://sleef.org/ (accessed 2022-06-10)
[6] N. Shibata, et al., “SLEEF: A Portable Vectorized Library of C Standard Mathematical Functions,” in IEEE Trans. on Parallel and Distributed Systems, Dec. 2019.
[7] pip・PyPI (online), https://pypi.org/project/pip/ (accessed 2022-06-27)
[8] ARM C Language Extensions for SVE (online), https://developer.arm.com/Architectures/Arm%20C%20Language%20Extensions (accessed 2022-06-24)
[9] Intrinsics for Short Vector Math Library Operations (SVML), https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-short-vector-math-library-ops.html (accessed 2022-06-23)
[10] Join the Preview – Amazon EC2 C7g Instances Powered by New AWS Graviton3 Processors (online), https://aws.amazon.com/jp/blogs/aws/join-the-preview-amazon-ec2-c7g-instances-powered-by-new-aws-graviton3-processors/ (accessed 2022-06-27)