Vol.2019-HPC-172 No.7
2019/12/18

AVX2機能を使った分割FFTを利用した多倍長数の乗算

平山 弘[1,a)]

概要：AVX2 の機能を利用して、倍精度浮動小数点数を 2 個または 4 個の配列を並列に計算できる FFT プログラムを作成した。多倍長数を 4 分割し 4 個の数値にして、この 4 並列 FFT プログラムを利用して、乗算プログラムを作成した。この方法によって、乗算をこれまでのプログラムと比較して 1.5 倍から 2.1 倍の高速化を行うことが出来た。この方法は、倍精度浮動小数点数を 2 個利用した 4 倍精度数の 4 並列の FFT も作成した。この方法を使えば、倍精度を利用した場合の限界を超えた高精度数の計算も行うことができる。

Multiplication of Multi-precision number by division FFT with AVX2 function

Hiroshi Hirayama[1,a)]

Abstract: Using the function of AVX2, we created an FFT program that can compute 2 or 4 arrays in parallel with double precision floating point numbers. The high precision numbers were divided into four and made four numerical values, and a multiplication program was created using this 4 parallel FFT program. By this method, the multiplication can be speeded up from 1.5 times to 2.1 times compared with the previous program.
In this method, four parallel FFTs of quad precision number using two double precision floating point numbers were also created. By using this method, it is possible to calculate a high precision number exceeding the limit accuracy when double precision is used.

Keywords: AVX, FFT, High Precision number, Quad precision number

1. はじめに

最近のパーソナル・コンピュータは、1 個の CPU でも複数の計算機機能 (コア) を持ち、1 個のコアには、AVX(Intel Advanced Vector Extensions)[5] と呼ばれるベクトル演算機能をもっているものがごく普通に見られるようになって来ている。最新のベクトル機能には、8 個の倍精度浮動小数点数 (512 ビット) を同時に処理できる AVX512 と呼ばれるものが現れているが、このでは良く普及している 4 個の倍精度浮動小数点数 (256 ビット) を同時に処理できる AVX2 の機能を使用した。

[1] 神奈川工科大学創造工学部自動車システム開発工学科
Department of Vehicle System Engineering, Faculty of Creative Engneering, Kanagawa Institute of Technology, Shimo-Ogino 1030, Atsugi, Kanagawa, 243-0292, Japan
[a)] hirayama@kanagawa-it.ac.jp

このように、高速計算できる機能を持っているが、コンパイラがこの機能を自動的に十分に性能が発揮できるコードを生成できないためかこの機能はあまり利用されていないように思える。これらの機能を使って、いろいろな計算問題を高速にしようと試みた。

本論文では、特別な CPU 等を準備しないで、普通に利用されている CPU を使って、高精度数の演算の高速化を試みた。1 個の CPU で、複数のコアを持ち、各コアは 64 ビットの倍精度浮動小数点数を 4 個分が入る 256 ビットのレジスタを持つ AVX2 の機能を持つものと仮定して、問題の高速化を行った。ここでは特に、高精度数の乗算について調べた。

ここで取り上げた計算例の実行時間は、主に Intel i7-8700K CPU 3.7GHz で実行した時間である。

2. ベクトル機能による FFT の計算

現在のパーソナル・コンピュータは、複数のコアを持ち、それぞれのコアはベクトル演算機能 (AVX) を持っている。計算を高速化するには、複数のコアを使い、それぞれのコアのベクトル機能を使うのが最も効果的と思われる。複数のコアを使うには、OpenMP 機能を使いマルチスレッド化し、各スレッドベクトル機能を使うことが考えられる。マルチスレッド化とベクトル機能を比較するために、4 個のデータ列の FFT(高速フーリエ変換) の計算をマルチスレッド法とベクトル化法で計算した。FFT のプログラムとして、大浦 [8] の基数 8 のプログラムを変更し使用した。データ数 N の倍精度 FFT の計算を 1 個、2 個と 4 個のコアをを使用し OpenMP を使いマルチスレッド化して計算した。これらの FFT は高精度数の乗算に使うことを想定しているので、N 個のデータの内、添え字の小さい半分には、4 桁の数値の最大値 9999 を与え、残りの添え字の大きい半分はゼロとした。このデータは、FFT を使って多倍長数の乗算を行う場合の限界を求める計算になる。マルチスレッド化を行う場合、ハイパースレッドを無効化して計算した。その結果は、表 1 に示す。

表 1 マルチスレッド倍精度数 FFT の実行時間 (単位 μ sec)

N	1 thread	2 threads	4 threads
256	1.169	1.228	1.277
512	2.506	2.793	2.706
1024	5.439	5.341	5.604
2048	11.621	16.084	15.217
4096	24.382	24.600	24.175
8192	60.347	68.773	67.502
16384	121.802	122.647	165.023
32768	283.059	283.659	430.601
65536	602.300	580.184	814.229
131072	1749.353	1711.241	1788.504
262144	3622.963	3275.480	4040.878
524288	7384.084	7138.707	12443.257

表 1 の 1 thread の計算は、通常のスカラー計算に相当するものである。2 thread の計算は 2 コアを使って並列に計算することを意味する。4 thread の計算は 4 コアを使い 4 並列に計算したときの時間である。オペ-レーティングシステム (OS) は、Windows 10 を使用した。この OS は、すべての同じ機能を持つ計算機 (コア) であると仮定するシンメトリックな OS であるので、見かけ上コアの数の 2 倍のコアがあるように見えるハイパースレッド機能があると十分な性能が発揮出来ない可能性がある。OS から見ると 1 個のコアは 2 個のコアに見えるのである。1 個のコアには、1 個の浮動小数点数の演算機能しかないので、複数のスレッドが 1 個のコアに割り当てられると当然実行を待たされることになる。このような事態を想定し、ハイパースレッド機能を切ってその評価を行い、その結果を表にまとめたものである。

表 2 ベクトル使用した倍精度数 FFT の実行時間 (単位 μ sec)

N	scalar	2 double	4 double
256	1.628	1.216	1.499
512	3.424	2.582	3.161
1024	7.222	5.491	6.609
2048	15.200	11.959	18.570
4096	31.910	27.232	35.014
8192	126.528	89.314	74.612
16384	147.520	117.979	192.110
32768	334.002	323.406	395.253
65536	732.776	684.535	822.078
131072	1666.147	1397.940	1968.692
262144	3156.316	3295.401	5133.666
524288	7443.317	9382.738	19777.267

AVX2 と呼ばれるベクトル機能を利用して、マルチスレッド化したプログラムと同様な計算を行った。その結果を表 2 に示す。4 個の倍精度浮動小数点数の演算を同時に出来るベクトル機能であるが、上の評価を見るとベクトル機能を使うより、マルチスレッド化が効率的に思える。マルチスレッド化して、多数のコアを使うことは、多くのキャシュメモリを使うことができるためか、効率的に計算が出来るようである。

ベクトル化は、効率的に動作させるためには、ベクトルデータを 32 バイト語境界に配置する必要がある。このように配置する方法は、最近の C++言語では多数提案されているが、コンパイラによっては、サポートされていなかったり、動作が遅かったりする問題があるように思える。

この問題を解決するために、次の 3 種類の方法で配置して、動作させることができた。

使用したマイクロソフト社の C++では、単純にベクトルデータを宣言すると、32 バイトの語境界に配置されていない可能性があるとの警告がでる。警告が出ない場合でも、32 バイト語境界に配置されるとの保証はないようである。

データの配置の問題だけでなく、プログラム作成するとき、並列にベクトル計算出来るようにデータの宣言をしなければならないなどの制限があるため、プログラムが難しくなるという問題がある。

表 2 の scalar は通常のスカラー計算で、表 1 の 1 thread の計算と同じものである。時間が異なるのは、メモリーやキャッシュの状態が異なるために時間が違って出たものと思われる。他の計算時間もこの程度の差があると思われる。

この結果を見ると 2 組のデータの FFT の計算と 4 組のデータの FFT の計算にはあまり時間に違いがないことがわかる。また、スレッド計算より、ベクトル計算の方が遅い場合が多いことがわかる。

この結果から、今回は、4 分割したデータの FFT をベク

トル計算とマルチスレッド化で行うことにした。

2.1 AVX2 を利用したベクトル計算

AVX2 におけるベクトル計算とは、4 個の倍精度浮動小数点の集まりを 1 つのベクトルとして、そのベクトル間の演算である。ベクトル間の演算だけでなく倍精度間の演算も含む。これらの演算を定義すれば、いろいろな計算が可能である。容易にベクトルの配列の FFT も計算できる。

ベクトル演算プログラムは Visual C++に付属するインテル社製の dvec.h を使用して作成した。このファイルの中では、4 個の倍精度浮動小数点の集まりのベクトルを F64vec4 と定義されており、次の例のように演算が定義されている。

```
F64vec4 a, b, c; // 4 個の倍精度数の要素を持つ
                 // ベクトル a,b,c を宣言
a=F64vec4(3.6);  // a の 4 要素をすべて 3.6 にする。

b=F64vec4(1.0,2.0,3.0,4.0);// b の 4 個の要素に
          // 数値{1.0,2.0,3.0,4.0}を代入
c=a + b  ; // a と b をベクトル的に加算し c に代入
c=a - b  ; // a と b をベクトル的に減算し c に代入
c=a * b  ; // a と b をベクトル的に乗算し c に代入
c=a / b  ; // a と b をベクトル的に除算し c に代入
c=round(a) ; // a の各要素を丸め c に代入
c=floor(a) ; // a の各要素の floor を求め c に代入
c=sqrt(a) ;  // a の各要素の平方根を計算する。
cout<<c[0]<<endl ;// c の 0 番目の要素 c[0] を出力
```

例えば、$f(x) = \sqrt{x} + x$ としたとき、x が 2,3,4,5 の場合の $f(x)$ の値を計算するとき、プログラムは次のようになる。

```
 1: #include <dvec.h>
 2: #include <iostream>
 3: using namespace std ;
 4: int main()
 5: {
 6:     F64vec4 x, y ;
 7:     x=F64vec4( 2.0, 3.0, 4.0, 5.0 ) ;
 8:     cout <<"x="<<x[3]<<" "<<x[2]<<" " ;
 9:     cout <<x[1]<<" "<<x[0]<< endl ;
10:     y=sqrt(x)+x ;
11:     cout <<"y="<<y[3]<<" "<<y[2]<<" " ;
12:     cout <<y[1]<<" "<<y[0]<< endl ;
13: }
```

計算結果は次のようになる。各ベクトルの成分毎に計算されていることがわかる。

```
x=2 3 4 5
y=3.41421 4.73205 6 7.23607
```

ファイル dvec.h では、F64vec4 と整数や倍精度浮動小数点等の数値との加算等の四則演算が定義されていないのでこのような定義を行えばより使いやすくなる。

本論文で示したベクトル演算プログラムではこのような変更がなされたインクルード・ファイルを作成し使用している。また、このファイルでは定義されていない関数もあるので、Intel 社のマニュアル [4] を使って追加して利用している。例えば fma(a,b,c) などの関数である。

3. ベクトル機能を使った FFT による高精度数の乗算

4 個のベクトル機能を使って高精度計算にを行う方法として、高精度数を 4 個の同じ桁数の数値に分割する方法が考えられる。この方法は、高橋等 [7] で述べられているように、高精度数の乗算法としては効率の良い方法であることが知られている。4 個の倍精度浮動小数点数が入るベクトル機能である AVX2 を持つ CPU ならば、4 分割が最も適切な方法と思われる。今回は、乗数と被乗数をそれぞれを 4 分割し 8 個のデータ列にする方法を行った。乗数と被乗数を 4 分割し、同時にこれらの数値の FFT を AVX2 の機能しか持たない CPU でも効率的に FFT を行えるからである。

乗数と被乗数を 4 分割することを考える。乗数と被乗数の数値の桁数が一般的には異なるが、ここでは、同じ程度の桁数の高精度数であると仮定し、その中で桁数の多い数の桁数を r 進数 m 桁とする。$m \leq 2^p$ を満たす整数 p を求め、$M = 2^p$ とする。また、乗数を a、被乗数を b とする。a の下位 $M/4$ 桁を $a3$、次の $M/4$ 桁を $a2$、さらに次の $M/4$ 桁を $a1$、残りの最上位を $a0$ にする。同様に b も a と同様に 4 分割する。$a0, a1, a2, a3$ の下位桁に 0 を付加して、m 桁の数値にする。この 4 つの数値列をベクトル機能を使って同時にフーリエ変換する。

```
                a0   a1   a2   a3
 x              b0   b1   b2   b3
 -------------------------------------------
              a0b3 a1b3 a2b3 a3b3
         a0b2 a1b2 a2b2 a3b2
    a0b3 a1b2 a2b3 a3b3
```

上のように 4 分割し、通常の数値のように計算する。axby は、a の x 番目の部分と b の y 番目の積を意味する。その積の計算には次のようにフーリエ変換を行う。

フーリエ変換は、通常実フーリエ変換と呼ばれるプログラムを利用する。共役な複素数は同じ領域を使用することによってデータ領域を節約し、計算の高速化をはかることが出来るからである。フーリエ変換された数値列間の乗算を行う。この計算は、4 個同時に計算できるから、例えば上のような例では横一列 (a0b3,a1b3,a2b3,a3b3) を同時に計算できる。これを 4 回すれば、上の計算を完了させることができる。

このデータ列を逆フーリエ変換して、元に戻す。得られ

た数値列は、数値列の個数に比例した倍率 $(M/2)$ になっているので、この数値で割ることよって、正規化する。この計算は、浮動小数点数の計算なので、誤差が生じる。この誤差は、計算結果は整数になるはずであるから、計算された数値を丸め処理を行って、整数に変換する。もし誤差が 0.5 より小さいならば、この処理によって正確な計算ができることになる。

この誤差は、最後の丸め処理によって厳密な値になるためには、次のような関係式を満たさなければならない。この式は Henrici[1] によって導かれた誤差解析である。r 進数 l 桁の数値を厳密に乗算できるには、計算精度の相対誤差を ϵ (マシン・イプシロン) とすると

$$\epsilon < \frac{1}{(192l^2 \log l)r^2} \tag{1}$$

を満たさなければならない。この式から基数 r が大きいほど要求精度が高くなることがわかる。桁数 l も大きくなると要求精度が高くなることがわかる。この式は、相対誤差の 2 乗以上の高次の項を省略する方法で、誤差を評価しているので、十分条件になる。現実にはもっと高い精度でも計算可能である。たとえば、上の式で 、$r = 10000, \epsilon = 2.22 \times 10^{-16}$ (IEEE 方式の倍精度浮動小数点）の場合、この式で計算すると計算可能桁数は 428 桁となる。実際には 1000 万桁以上の数も計算可能である。

ここで示した 4 分割法で計算した結果と従来の分割しない方法 [3] との比較を表 3 に示す。

表 3 従来の方法と 4 分割法の乗算実行時間 (単位 msec)

計算桁数	従来の方法	4 分割	倍率
2048	0.072298	0.044149	1.6
4096	0.103197	0.062602	1.6
8192	0.166722	0.10891	1.5
16384	0.322814	0.20859	1.5
32768	0.614781	0.40173	1.5
65536	1.416	0.83120	1.6
131072	2.912	1.7086	1.7
262144	5.981	3.4684	1.7
524288	11.372	6.9938	1.6
1048576	24.772	14.659	1.7
2097152	50.534	30.035	1.7
4194304	121.944	58.325	2.1

この結果を見ると、4 分割法を行うことによって、分割しない方法と比較すると 1.5〜2.1 倍高速化できることがわかる。2 分割法とあまり変わらない結果であった。ここでの問題は、配列をいかに効率的に配置するかの問題である。配列を効率的に配置させることは、OS の問題で、OS を以下に効率利用するするかの問題になると思われる。

Henrici の式 (1) は、各桁の数値 r が大きな数値になっているとき、計算可能桁数は最も小さくなることを示している。

高精度数を 4 桁毎に分割したとき、各桁に入れられる最大値は 9999 となる。この数値を使って、フーリエ変換して、乗算を行うと、桁数は正確に計算できる数の最大桁数は、16777216 桁の数値であることがわかる。このときの最大誤差は 0.4375 である。この倍の 33554432 桁で計算すると、計算結果は正しく計算出来なかった。この数値は十分条件で多くの場合、この倍の 3355 万桁まで可能である。円周率計算プログラムで 3355 万桁まで計算できるのはこのためである。経験的には、さらにその倍の 6710 万桁までの計算が可能な場合がよくある。

4 分割すると、各乗算が 16777216 桁が限界になるので、全体としては計算の限界は 6710 万桁となる。高精度数を 3 桁毎に分割すると、計算可能な桁数は、約 8 億桁 (805306368) となる。この計算に 4 分割法を適用すると、計算可能な桁数は約 32 億桁になる。

4. 4 倍精度を使った高精度数の乗算

前節からわかるように、FFT を使った乗算で倍精度数を超える精度の浮動小数点数を使えると計算効率は非常に良くなる。ここでは、4 倍精度数値で効率的に計算出来る double-double 型 [2] の数値を使う。この 4 倍精度数 (real16) は、次のように定義する。

```
class real16
{
    double m0, m1 ;
} ;
```

この様に定義されている 4 倍精度数は、次のような簡単なプログラムで加減算 [9] および乗算が出来る。

```
 1: real16 add( const real16 &a,
 2:                const real16 &b )
 3: {
 4:   real16 c ;
 5:   double sh, eh, v, ss ;
 6:   sh = a.m0 + b.m0 ;
 7:   v = sh - a.m0 ;
 8:   eh = (a.m0 -(sh-v)) + (b.m0 - v) ;
 9:   eh = eh + a.m1 + b.m1 ;
10:   c.m0 = sh + eh ;
11:   c.m1 = eh-(c.m0-sh) ;
12:   return c ;
13: }
```

4 倍精度数の乗算は次のように簡単に書ける。

```
1: real16 operator*( const real16 &a,
2:                       const quad &b )
3: {
4:    quad c ;
5:    double  s ;
6:    c.m0 = a.m0 * b.m0 ;
```

```
 7:    c.m1  = fma( a.m0, b.m0, -c.m0 ) ;
 8:    c.m1 = c.m1+(a.m0*b.m1+a.m1*b.m0) ;
 9:    s = c.m0 +c.m1 ;
10:    c.m1 =c.m1 - (s - c.m0) ;
11:    c.m0 =s ;
12:    return c ;
13: }
```

加減算および乗算も簡単に書けるだけでなく条件文が入っていない。このため、これらのプログラムは、容易にベクトル機能を使って計算出来ることがわかる。

4 個の 4 倍精度数をベクトルを使って次のように表す。

```
class ddvec4
{
   F64vec4   m0, m1 ;
} ;
```

このように定義すると、4 倍精度のベクトル化された数の加減算のプログラムは 4 倍精度数の加減算を変更することによって容易に作成できる。4 倍精度数の加算プログラムで、real16 を ddvec4 に、double を F64vec4 に変更するだけで簡単に、書き換えることが出来る。次のようになる。

```
 1: ddvec4 operator+( const ddvec4 &a,
 2:                       const ddvec4 &b )
 3: {
 4:   ddvec4 c ;
 5:   F64vec4 sh, eh, v, ss ;
 6:   sh = a.m0 + b.m0 ;
 7:   v = sh - a.m0 ;
 8:   eh = (a.m0 -(sh-v)) + (b.m0 - v) ;
 9:   eh = eh + a.m1 + b.m1 ;
10:   c.m0 = sh + eh ;
11:   c.m1 = eh-(c.m0-sh) ;
12:   return c ;
13: }
```

同様に 4 倍精度の乗算 [6] も容易に ddvec4 のプログラムに変更できる。

```
 1: ddvec4 operator*( const ddvec4 &a,
 2:                       const ddvec4 &b )
 3: {
 4:    ddvec4 c ;
 5:    F64vec4  s ;
 6:    c.m0 = a.m0 * b.m0 ;
 7:    c.m1  = fma( a.m0, b.m0, -c.m0 ) ;
 8:    c.m1 = c.m1 + (a.m0 * b.m1 +a.m1 * b.m0) ;
 9:    s = c.m0 +c.m1 ;
10:    c.m1 =c.m1 - (s - c.m0) ;
11:    c.m0 =s ;
12:    return c ;
13: }
```

これらのプログラムを使って FFT のプログラムを作成し、そのプログラムの性能を調べた。この結果を表 4 に示す。N 個の 4 倍精度の FFT の実行時間 (real16)、N 個の 4 倍精度数の並列に同時 2 個の FFT を実行した時間（ddvec2) および 4 個並列に行った時の実行時間 (ddvec4) である。

表 4 N 個データの 4 倍精度 FFT の実行時間 (単位 msec)

N	real16	ddvec2	ddvec4
256	0.0199449	0.0404735	0.0508795
512	0.0571573	0.0645286	0.0597811
1024	0.126532	0.101082	0.119589
2048	0.207546	0.256954	0.253994
4096	0.429991	0.45861	0.555058
8192	0.807299	0.96501	1.17684
16384	1.71799	2.05834	2.49793
32768	3.52126	4.97552	5.24787
65536	7.37798	8.80023	10.5823
131072	14.9756	18.5223	23.2506
262144	29.9614	38.0686	50.0738
524288	63.0259	80.7332	104.9700

基数 8 の倍精度用の FFT プログラムを使うために、FFT のプログラム中の定数を 4 倍精度に変更した。この変更と宣言を変えるだけで、容易に FFT のプログラムを作成することができた。

この FFT プログラムを使って、多倍長精度の乗算ルーチンを作成した。ここで問題なのは、丸め処理である。

$$\mathrm{round}(x) = \mathrm{floor}(x + 0.5)$$

丸め処理関数は floor 関数を使って容易に作成できるので、次のような 4 倍精度ベクトル用の floor 関数を作成した。4 倍精度用 floor 関数は次のようにベクトル化することができる。

```
 1: ddvec4 floor( const ddvec4 & x )
 2: {
 3:     ddvec4  z = x ;
 4:     z.m0 = floor(x.m0) ;
 5:     F64vec4 t = cmp_eq( z.m0, x.m0 ) ;
 6:     z.m1 &= t ;
 7:     z.m1 = floor( z.m1 ) ;
 8:     t = z.m0+z.m1 ;
 9:     z.m1 = z.m1-(t-c.m0) ;
10:     z.m1 = t ;
11:     return z ;
12: }
```

これを使うと、1 億桁を超える高精度数の乗算が可能になる。以下にその性能を表 5 に示す。二つの N 桁の高精度数を 2 分割したものを 4 並列で FFT を行う。このとき 4 倍精度を使って 4 並列の FFT を行う。この時の乗算実行時間である。

表 5　4 倍精度 2 分割法の乗算実行時間 (単位 sec)

計算桁数	計算時間
1048576	0.0602338
2097151	0.122876
4194304	0.26572
8388608	0.544921
16777216	1.13814
33554432	2.34131
67108864	4.87678
134217728	9.98846
268435456	20.3998
536870912	42.4536

ベクトル機能を使う計算では、ベクトル変数を、16 バイトの境界や 32 バイトの境界に配置しないとその性能は著しく落ちたり、計算が出来ない場合あることが知られている。今回のこの計算では、このような語境界の問題 (アライメントの問題) が発生する。この問題に対応する関数など準備されているようであるが、その使い方等の文献はあまりないようで、その解決は難しい問題ある。現在のコンパイラーは、ベクトル変数を通常の変数 double 型変数のように扱うことができないため、アライメントの問題が発生して、プログラムは非常に不安定になる場合がある。このためすでに作成してある高精度プログラムにこれらの乗算ルーチンを組み込むことが出来なかった。

5. まとめ

マルチスレッドによる並列化とベクトル機能を使った並列化を比較すると同等の性能であるがマルチスレッド機能を使った並列化がキャッシュメモリが多く使えるためか少し速かった。このため、多倍長数を 4 分割して、FFT を並列に適用すると、多くの精度で、1.5 倍程度の性能を発揮することができたが期待よりかなり遅い結果となった。

この方法は、FFT を使った計算の精度の限界を 4 倍に引き上げる効果もある。また、この方法は、double-double 型 4 倍精度も同様にベクトル並列処理が可能である。これを使えば、1 億桁を超える高精度数も容易に計算出来る。1 億桁を超えた場合、メモリの使用効率が良くなるため同じメモリを持つ計算機を使用した場合、高い精度の計算が行える。

現在メモリの配置の問題が起こるため、同じプログラムを実行しても実行時間が大きく変化する場合がある。

参考文献

[1] Henrici P., Applied and Computational Complex Analysis, Vol. 3, Chap. 13, John Wiley & Sons, New York(1986)
[2] 長谷川 秀彦, 高精度演算を用いた混合精度反復法, 応用数理学会三部会連携「応用数理セミナー」資料集,(2013),4–35
[3] 平山 弘, C++ 言語による高精度計算パッケージの開発、日本応用数理学会論文誌，5 (1995),307–318
[4] Intel 社,Intel 64 and IA-32 Architectures Developer's Manual, https://software.intel.com/en-us/articles/intel-sdm
[5] 北山洋幸,AVX 命令入門, カットシステム,(2015)
[6] 小武守, 長谷川, 藤井, 西田, 反復法ライブラリ向け 4 倍精度演算の実装と SSE2 を用いた高速化, 情報処理学会誌 コンピューティングシステム,1(2008),73–84
[7] 高橋 大介, 金田 康正, 多倍長平方根の高速計算法, 情報処理学会研究報告，97(1995-HPC-058)(1995),51–56
[8] 大浦 拓哉, 汎用 FFT(高速フーリエ/コサイン/サイン変換) パッケージ,http://http://www.kurims.kyoto-u.ac.jp/ ooura/fft-j.html
[9] 山田進, 佐々成正, 今村俊幸, 町田昌彦, 4 倍精度基本線形代数ルーチン群 QPBLAS の紹介とアプリケーションへの応用, 情報処理学会研究報告,vol.2012-HPC-137,No.23(2012)
[10] Yozo Hida, Xiaoye S. Li, David H. Bailey, Library for Double-Double and Quad-Double Arithmetic, Proc. 15th Symposium on Computer Algorithmetic, (2007),155–162
[11] Yozo Hida, Xiaoye S. Li, David H. Bailey, Algorithms for Quad-Double Precision Floating Point Arithmetic, Lawrence Berkeley National Laboratory, (2000)