Vol.2020-HPC-175 No.3
2020/7/30

SIMD 命令を用いた 3 倍精度行列乗算の性能評価

幸谷 智紀[1,a)]

概要：マルチコンポーネント方式の 3 倍精度演算は，Bailey らの QD ライブラリでサポートされている DD(double-double) 精度と QD(Quad-double) 精度のちょうど中間の計算精度をサポートし，計算量的にも中間に位置するものである．我々は既に倍精度ベースの 3 倍精度演算 (Triple-double, TD 精度) を C のインライン関数を用いて実装し，行列乗算によってその性能評価を行った．本稿では更に SIMD 命令を用いて高速化を試みた結果について報告する．

キーワード：SIMD 命令，3 倍精度浮動小数点演算，行列乗算

Performance evaluation of triple-double precision matrix multiplication

Abstract: Triple precision arithmetic is categorized in multi-component way of implementation of multiple precision floating-point arithmetic. It can provide the middle precison and performance between DD(double-double) and QD(quad-double) precision supported by Bailey et al.'s QD library. We have already evaluated the performance of TD(triple-double) precision matrix multiplication constructed by C inline functions. In this paper, we report the results of further attempts to accelerate it using the SIMD instruction.

Keywords: SIMD instruction, triple-double precision floating-point arithmetic, matrix multiplication

1. 初めに

科学技術計算の大規模化は常に進む一方であり，ユーザの要求を満足する出力結果の精度に到達するために必要な最低限の浮動小数点数については IEEE754 倍精度 (binary64) 以上，いわゆる多倍長浮動小数点演算が求められることが増えてきている．その中核を担う基本線型計算は，非線型問題のパーツとして重要な物であり，入力値に含まれる初期誤差の影響を極力少なくするためにも binary64 以上の桁数を持つ多倍長精度浮動小数点形式のサポートがなくてはならない．それ故に，様々な多倍長精度浮動小数点演算ライブラリが作られてきたが，現在では既存のハードウェアベースでサポートされる binary64 等の浮動小数点数を束ねて使用するマルチコンポーネント方式のものと，整数演算ベースの多数桁方式のものに 2 分されてるようになっている．前者の代表的な実装が Bailey らの QD ライブラリ [10] であり，後者の代表が GNU MP[13] の任意長自然数演算カーネルを土台とする MPFR[11] である．前者は binary64 の数倍程度の精度で充分な用途に使用され，それ以上の精度が必要な時は後者を使用するという使い分けが一般的である．

LAPACK/BLAS のインターフェースを使用し，多倍長精度化した線型計算を実現した MPLAPACK[14] は，この使い分けを実行できる最初の実用的な実装系と言える．QD ライブラリがサポートする DD(double-double) 精度と QD(quadruple-double) 精度，IEEE754 の 4 倍精度 (`__float128`)，MPFR の提供する任意精度の全てに対応している上に，MPFR ベースの部分は OpenMP による並列化もサポートしており，多倍長精度線形計算のパフォーマンスが高い．しかし，binary32(IEEE754 単精度) や binary64 用の LAPACK や BLAS は，よりマシンアーキテクチャに最適化され高速化した ATLAS, OpenBLAS, Intel Math Kernel ライブラリを用いることで高速になることが知られている．MPLAPACK においても，特に BLAS レベルでの高速化はまだ行う余地が十分残っている．

一方，計算を担うハードウェアそのものの高速化については，消費電力の増大を抑えるため，動作周波数の頭打ち

[1] 静岡理工科大学
Shizuoka Institute of Science and Technology, Fukuroi, Shizuoka 437–8555, Japan
[a)] kouya.tomonori@sist.ac.jp

状態が続いており，1 命令当たりの単純な高性能化は難しくなっている．故に，処理全体の高速化のためには，コア単位では SIMD 命令の多様化，CPU・GPU 単位ではコア数の増加といった様々なレイヤーでの並列化技法をソフトウェア的に使いこなして行う必要がある．

長谷川らはマルチコンポーネント型の DD（double-double, binary64 二つ分) 精度を中心に，AVX を用いた SIMD 活用と OpenMP によるマルチコア環境上での並列化を組み合わせ，疎行列もサポートした多倍長精度線型計算の最適化を行い，その成果物は Lis[3] や MuPAT[2] にまとめている．一方，多数桁方式の多倍長精度整数計算は高橋ら [5] が行っている．GPU 上での多倍長精度計算は椋木ら [4] や，中里ら [6] が行っており，これらも GPU が備える多数のコアを生かした並列性を生かしたものになっている．CAMPARY[7] という CPU, GPU 両用の任意精度演算ライブラリもある．

今回取り上げるマルチコンポーネント方式の 3 倍精度演算 [1] は，Bailey らの QD ライブラリでサポートされている DD 精度と QD 精度のちょうど中間の計算精度をサポートし，計算量的にも中間に位置するものである．我々は既に倍精度ベースの 3 倍精度演算 (Triple-double, TD 精度, binary64 三つ分) を C のインライン関数を用いて実装し，三重ループを用いた単純行列乗算 (Simple と略記)，ブロック化した行列乗算，分割統治法を用いた行列乗算のベンチマークテストを行ってその有効性を確認した [15]．

本稿では更に SIMD 命令，現在販売されているコンシューマ用 CPU では普通にサポートされている AVX2 と FMA[12] を用いて，DD，　TD, QD 精度の行列乗算の高速化を試みた結果について報告する．

2. SIMD 化したマルチコンポーネント型多倍長精度演算

前述した通り，マルチコンポーネント型の多倍長精度浮動小数点数は，既存のハードウェアベースの IEEE754 浮動小数点数を複数繋ぎ合わせて多倍長精度を実現する．この演算に際しては無誤差変換技法と呼ばれる，上位桁を表現する浮動小数点数の丸め誤差を下位桁で救い上げる技法を組み合わせて使用される．最初に無誤差変換技法を利用したマルチコンポーネント方式の多倍長浮動小数点演算を明確に示したのは Dekker[8] であり，任意精度にまで拡張可能であることを示したのは Priest[9] である．ライブラリとしては Bailey らの QD ライブラリ [10] が著名であり，これは DD 精度と QD 精度を C++のクラスライブラリとして実装したものである．インライン関数を使用しており，計算性能も比較的高い．これをベースとした派生ライブラリも多数存在する．

一方，SIMD(Single Insturuction, Multiple Data) はその名の通り，複数のデータ型に対する演算などの処理を一つの機械語命令で実行することができるもので，コンシューマ向けの x86_64 アーキテクチャ CPU では 256bit 幅のレジスタを備え，そこにフィットする複数のデータ型を利用できる AVX2 が 2020 年現在では AMD, Intel の CPU では一般的に利用できる．今回我々はこのうち，binary64 を 4 つまとめた`_m256d` データ型を用い，これに対する四則演算命令を C から利用できる`_mm256_[add, sub, mul, div]_pd` 関数や FMA(Fused Multiply-Add) に相当する`_mm256_fmadd_pd` 関数を使用して無誤差変換技法の主要機能である QuickTwoSum，TwoSum, TwoProd-FMA 関数を SIMD 化し，それぞれ AVX2QuickTwoSum(Algorithm 1), AVX2TwoSum(Algorithm 2), AVX2TwoProd-FMA(Algorithm 3) 関数として利用した．

Algorithm 1 $(s[4],\ e[4])$:= AVX2QuickTwoSum($a[4]$, $b[4]$)

s := `_mm256_add_pd`(a, b)
e := `_mm256_sub_pd`(b, `_mm256_sub_pd`(s, a))
return (s, e)

Algorithm 2 $(s,\ e)$:= AVX2TwoSum(a, b)

s := `_mm256_add_pd`(a, b)
v := `_mm256_sub_pd`(s, a)
e := `_mm256_add_pd`(`_mm256_sub_pd`(a, `_mm256_sub_pd`(s, v)), `_mm256_sub_pd`(b, v))
return(s, e)

Algorithm 3 $(p[4],\ e[4])$:= AVX2TwoProd-FMA($a[4]$, $b[4]$)

p :=`_mm256_mul_pd`(a, b)
e :=`_mm256_fmadd_pd`(a, b, $-p$)
return(p, e)

DD 精度演算についてはこれらの無誤差変換機能の単純な組み合わせて構築されているため，行列乗算に利用する加算と乗算は素直に SIMD 化して実装できるが，下記に述べる TD 精度演算や，QD 精度演算については，正規化処理の内部に条件分岐が含まれており，そのままでは SIMD 化できない箇所が残る．そのため，正規化処理部分は 4 回ループとして実装し，それ以外の所は極力 SIMD 化したものを用いて実装するようにした．

2.1 Triple-double(TD) 精度演算と SIMD 化

3 倍精度浮動小数点演算は，既存の浮動小数点数を 3 つ使用して表現した浮動小数点数を用いて実行する．ここでは 3 倍精度浮動小数点数を $x := (x_0, x_1, x_2)$ として表現する．

3 倍精度浮動小数点演算のためには演算結果を正規化する必要がある．Fabiano らは VecSum(Algorithm 4) と VSEB(k) (VecSum with Blanch, Algorithm 5) を組み合わせて，演算結果を正規化するようにしている．

Algorithm 4 $(e_0, ..., e_{n-1}) := \mathrm{VecSum}(x_0, ..., x_{n-1})$

$s_{n-1} := x_{n-1}$
for $i = n-2$ to 0 **do**
  $(s_i, e_{i+1}) := \mathrm{TwoSum}(x_i, s_{i+1})$
end for
$e_0 := s_0$
return $(e_0, ..., e_{n-1})$

Algorithm 5 $(y_0, ..., y_{k-1}) := \mathrm{VSEB}(k)(e_0, ..., e_{n-1})$

$j := 0$
$\epsilon_0 := e_0$
for $i = 0$ to $k-3$ **do**
  $(r_i, \epsilon^t_{i+1}) := \mathrm{TwoSum}(\epsilon_i, e_{i+1})$
  if $\epsilon^t_{i+1} \neq 0$ **then**
    $y_j := r_i$
    $\epsilon_{i+1} := \epsilon^t_{i+1}$
    $j := j + 1$
  else
    $\epsilon_{i+1} := r_i$
  end if
end for
$(y_j, y_{j+1}) := \mathrm{TwoSum}(\epsilon_{k-2}, e_{k-1})$
$y_{j+2} := 0, ..., y_{k-1} := 0$
return $(y_0, ..., y_{k-1})$

VecSum と VSEB(n) のうち SIMD 化できる倍精度四則演算や無誤差変換を AVX2 関数を用いて書き換えたものをそれぞれ AVX2VecSum，AVX2VSEB(n) と書くことにする．前者は完全に SIMD 化できるが，後者は要素毎の if 文があり，今回はこの部分は SIMD 化していない．

加算 (TDadd) は，$x := (x_0, x_1, x_2)$, $y := (y_0, y_1, y_2)$ の和，即ち $r = (r_0, r_1, r_2) := x + y$ を求めるものである．まず最初に x と y をマージソートしてから VecSum で正規化し，しかる後に VSEB(3) で 3 倍精度浮動小数点数として正規化して r を返す．

これを SIMD 化すると，$x[4] := (x_0[4],\ x_1[4],\ x_2[4])$, $y[4] := (x_0[4],\ x_1[4],\ x_2[4])$ に対してこの和である $r[4] := (r_0[4],\ r_1[4],\ r_2[4])$ を返す AVX2TDSum 関数となる．前述の通り，このアルゴリズムのうち，完全に SIMD 化できているのは VecSum 関数のみで，Merge 関数は全く，VSEB(n) 関数はごく一部を除き今回は SIMD 化できていないことから，加算に関してはほぼ TDsum 関数をほぼそのまま使用していることになる．

Algorithm 6 $r[4] := \mathrm{AVX2TDsum}(x[4],\ y[4])$

$(z_0, ..., z_5) := \mathrm{Merge}(x_0, x_1, x_2, y_0, y_1, y_2)$
$(e_0, ..., e_5) := \mathrm{AVX2VecSum}(z_0, ..., z_5)$
$(r_0, r_1, r_2) := \mathrm{AVX2VSEB(3)}(e_0, ..., e_5)$
return $(r_0,\ r_1,\ r_2)$

Fabiano らは Accurate 乗算と，演算数の少ない Fast 乗算の二つを提唱している．我々は後者の乗算を TDmul として実装し，VSEB 関数以外を SIMD 化した AVX2TDmul 関数を実装した．

Algorithm 7 $r[4] :=$ AVX2TDmul($x[4]$, $y[4]$)

$(z_{00}^{\mathrm{up}}, z_{00}^{\mathrm{lo}}) :=$ AVX2TwoProd-FMA(x_0, y_0)
$(z_{01}^{\mathrm{up}}, z_{01}^{\mathrm{lo}}) :=$ AVX2TwoProd-FMA(x_0, y_1)
$(z_{10}^{\mathrm{up}}, z_{10}^{\mathrm{lo}}) :=$ AVX2TwoProd-FMA(x_1, y_0)
$(b_0, b_1, b_2) :=$ AVX2VecSum$(z_{00}^{\mathrm{lo}}, z_{01}^{\mathrm{up}}, z_{10}^{\mathrm{up}})$
$c :=$ `_mm256_fmadd_pd`(x_1, y_1, b_2)
$z_{31} :=$ `_mm256_fmadd_pd`$((x_0, y_2, z_{10}^{\mathrm{lo}})$
$z_{32} :=$ `_mm256_fmadd_pd`$(x_2, y_0, z_{01}^{\mathrm{lo}})$
$z_3 :=$ `_mm256_add_pd`(z_{31}, z_{32})
$s_3 :=$ `_mm256_add_pd`(c, z_3)
$(e_0, e_1, e_2, e_3) :=$ AVX2VecSum$(z_{00}^{\mathrm{up}}, b_0, b_1, s_3)$
$r_0 := e_0$
$(r_1, r_2) :=$ AVX2VSEB(2)(e_1, e_2, e_3)
return (r_0, r_1, r_2)

従って，現状では DD 精度演算，QD 演算に比べ，TD 演算の，特に加算に関しては処理時間のかかる部分の SIMD 化ができておらず，後述するような性能向上のボトルネックの一因がここにあるものと思われる．

3. 単純行列乗算のベンチマークテスト

DD 演算，TD 演算，QD 演算の加算と乗算の演算量を比較すると，TD 演算は DD 演算の 3.8 倍，QD 演算の 0.3 倍の演算量となる．また，演算ごとの相対誤差もそれぞれ $O(2^{-106}) = O(10^{-32})$, $O(2^{-159}) = O(10^{-48})$, $O(2^{-212}) = O(10^{-64})$ となることから，DD 演算では精度が足りず，QD 演算までは必要のない，ちょうど中間の精度で十分な場合に有効な演算と言える．前述したように我々は既に SIMD 化しない行列乗算に関して詳細なベンチマークを行い，演算量に比例した計算時間になることを確認している．

今回は SIMD 化のための試みの実装であるため，3 重ループに基づく単純行列乗算アルゴリズム，即ち，実正方行列 $A, B \in \mathbb{R}^{n\times n}$ に対して，行列 $C := AB$ を，各成分 c_{ij} に対して

$$c_{ij} := \sum_{k=1}^{n} a_{ik} b_{kj}$$

として求める方法のみ使用した．この際，A, B は次のものを利用してベンチマークテストを行っている．

$$A = \left[\sqrt{2}\,(i+j-1)\right]_{i,j=1}^{n},\ B = \left[\sqrt{3}\,(i+j-1)\right]_{i,j=1}^{n}$$

要素全てが正の実数であるため，桁落ちは殆ど起きない．実際，以下で述べる行列乗算の計算結果は，どの精度の計算においても使用する計算桁数より 10 進 1〜2 桁の精度低下がみられる程度であり，C の各成分の相対誤差の最大値は，DD 精度行列乗算で $O(10^{-32})$〜$O(10^{-31})$, TD 精度で $O(10^{-48})$〜$O(10^{-47})$，QD 精度で $O(10^{-65})$〜$O(10^{-64})$ であった．

同じ AVX2 を用いた行列乗算 C プログラムは下記の Ryzen7 と Corei7 の 2 環境で実行し，n = 64, 128, 256, 512, 1024, 2048, 4096 の実正方行列の行列乗算ベンチマークテストを行った．比較対象は，AVX2 を使用しない C のインライン関数による独自のマルチコンポーネント型ライブラリを用いたものと，これを AVX2 を用いて 256bits にパッケージングしたインライン関数ライブラリを用いたものである．DD 精度演算，QD 精度演算は QD ライブラリと同じアルゴリズムを用いており，TD 精度演算は前述のアルゴリズムに基づく実装である．

Ryzen7 AMD Ryzen 2700 (2.4 GHz), 16 GB RAM, Ubuntu 18.0.4 LTS x86_64, GCC 7.5.0

Corei7 Intel Core i7-9700K (3.6GHz), 16 GB RAM, Ubuntu 18.04.4 LTS x86_64, GCC 7.5.0

C options `gcc -O3 -mavx2 -mfma`

行列要素の格納方式は行優先 (Row-major) であるが，AVX2 による実装では，各要素のコンポーネントを分割して添え字ごとに行列単位でまとめたものを使用している．例えば DD 精度の行列 A であれば，各要素は $a_{ij} = (a_{ij}[0], a_{ij}[1])$ となるが，これを $A[0] = [a_{ij}[0]]_{i,j=1,2,\ldots,n}$, $A[1] = [a_{ij}[1]]_{i,j=1,2,\ldots,n}$ と分割して格納したものを使用した．こうすることで，行列要素の読み込み時には AVX2 の読み込み関数を各コンポーネント単位でまとめて実行することができる．

以上の環境下で単純行列乗算の計算時間を，DD, TD, QD 精度演算で行った結果を示す．Corei7 環境下で実施したものが表 1 で，Ryzen7 環境下で行ったものが表 2 である．AVX2 を用いた SIMD 化を行ったことで高速化されたかどうかを確認するため，AVX2 なしの計算時間を AVX2 使用時の計算時間で割った計算速度向上比を各表の右側に付加してある．AVX2 で高速化できなかった場合はこの比が 1 未満になるが，その箇所には下線を引いて明示した．

Corei7 の場合，DD 精度，QD 精度共に AVX2 による高速化の寄与が大きく，前者が 1.33〜3.37 倍，後者が 4.35〜6.81 倍高速化されている．それに対し，TD 精度は AVX2 による高速化の寄与が低く，0.75〜1.69 倍と，幾つかの行列サイズではかえって低速になっていることが分かる．結果として，多くの行列サイズにおいて，TD 精度より QD 精度の方が計算時間が短くなっていることも分かる．

Ryzen7 の場合においても QD 精度において AVX2 による高速化の寄与が大きく，2.00〜3.67 倍となっている．DD 精度，TD 精度はあまり高速化されておらず，それぞれ 0.65〜3.11 倍，0.85〜1.41 倍の高速化に留まっている．全体的に，Corei7 に比べて高速化の寄与が低いことが分かる．

表 1 Computational time: Corei7

	Computational Time (Unit: second)						Speedup ratio		
n	DD	DD AVX2	TD	TD AVX2	QD	QD AVX2	DD/AVX2	TD/AVX2	QD/AVX2
64	0.0015	0.0004	0.0106	0.0109	0.05	0.01	3.37	0.98	5.00
128	0.0121	0.0036	0.0844	0.0894	0.46	0.07	3.33	0.94	6.57
256	0.1	0.05	0.75	0.7	3.61	0.53	2.00	1.07	6.81
512	0.79	0.34	5.98	5.26	29	4.62	2.32	1.14	6.28
1024	16.66	12.52	50.07	67	292.42	67.16	1.33	0.75	4.35
2048	169.17	124.26	844.88	574.74	2406.14	547.93	1.36	1.47	4.39
4096	1548.07	956.83	7848.64	4642.62	20105.05	4273.8	1.62	1.69	4.70

表 2 Computational time: Ryzen7

	Computational Time (Unit: second)						Speedup ratio		
n	DD	DD AVX2	TD	TD AVX2	QD	QD AVX2	DD/AVX2	TD/AVX2	QD/AVX2
64	0.0014	0.0005	0.012	0.012	0.02	0.01	2.88	1.01	2.00
128	0.0115	0.0037	0.096	0.092	0.22	0.06	3.11	1.04	3.67
256	0.0925	0.0669	0.75	0.71	1.73	0.48	1.38	1.06	3.60
512	0.73	1.12	5.96	6.74	17.48	6.59	0.65	0.88	2.65
1024	19.69	14.97	61.32	72.33	194.54	64.61	1.32	0.85	3.01
2048	150.52	122.78	1013.02	719.46	1568.07	736.03	1.23	1.41	2.13
4096	1329.48	1996.58	8300.01	6002.56	12736.02	5428.45	0.67	1.38	2.35

このように QD 精度計算の AVX2 化の効果が大きいのは，AVX2 化できる計算部分が多く，正規化部分の非 AVX2 化部分のボトルネックが生じていないことが原因と考えられる．また TD 精度計算の AVX2 化の効果が薄いことは，AVX2 化していない部分においてボトルネックが生じていることが一因として考えられる．

また DD 精度，TD 精度共通のボトルネックとしては，行列要素の読み込み部分に`_mm256_set_pd`関数のみ使用していることが大きいと考えられる．メモリアラインメントを固定して`_mm256_load_pd`関数を多数使用することができれば，この点の解消も図れ，計算量が軽い DD 精度, TD 精度の計算時間の低減がある程度は見込めると思われる．

4. 結論と今後の課題

今回我々は DD 精度，TD 精度，QD 精度の加算と乗算をそれぞれ SIMD 化し，単純行列乗算アルゴリズムを用いて高速化を図った．その結果，TD 精度計算の高速化が予想以上に図れていないことが分かり，逆に，QD 精度計算よりも計算時間が長くなるという結果を招いている．

Ryzen7，Corei7，いずれの環境においても，本文中に述べたように，TD 精度計算における SIMD 化が不十分であり，特に加算の SIMD 化が要因として挙げられる．DD 精度よりも複雑であり，最適化された演算になっていることも要因として考えられる．

いずれにせよ，今回はキャッシュヒット率の悪い単純行列乗算アルゴリズムを用いていることから，ブロック化アルゴリズムや分割統治法を用いた場合に比べて低速である．

今後はこれらのアルゴリズムに今回実装した SIMD 化マルチコンポーネント型多倍長演算を組み込み，OpenMP による並列化も行ってより高速な多倍長精度行列乗算ライブラリを構築することを目指したい．

参考文献

[1] N. Fabiano and J. Muller and J. Picot, Algorithms for Triple-Word Arithmetic, IEEE Trans. on Computers, Vol.68, No.11, pp.1573–1583, 2019.
[2] H. Yagi, and E. Ishiwata and H. Hasegawa, Acceleration of Interactive Multiple Precision Arithmetic Toolbox MuPAT Using FMA, SIMD, and OpenMP, Advances in Parallel Computing Vol.36, pp.431 - 440, 2020.
[3] 小武守 恒, 藤井 昭宏, 長谷川 秀彦, 西田 晃, `https://www.ssisc.org/lis/`
[4] 椋木大地, 高橋大介, GPU における 3 倍・4 倍精度浮動小数点演算の実現と性能評価, 情報処理学会 ACS 論文誌, Vol.6, No.1, pp.66-77, 2013.
[5] T. Edamatsu and D. Takahashi, Acceleration of Large Integer Multiplication with Intel AVX-512 Instructions, 2018 IEEE 20th International Conference on High Performance Computing and Communications, pp.211-218, 2018.
[6] 中村光典, 中里直人, OpenCL による四倍精度行列積の高速化, 情報処理学会研究報告 HPC-133, No.27, 2012.
[7] M. Jolders and J-. M. Muller and V. Popescu and W.Tucker, CAMPARY: Cuda mutiple precision arithmetic library and applications, 5th ICMS,2016.
[8] T.J.Dekker, A floating-point technique for extending the available precision, Numer. Math. 18, 224–242 (1971).
[9] D. M. Priest, Algorithms for arbitrary precision floating point arithmetic, Proceedings 10th IEEE Symposium on Computer Arithmetic, pp. 132-143, 1991.
[10] D.H. Bailey, QD, `http://crd.lbl.gov/~dhbailey/mpdist/`.
[11] MPFR Project, The MPFR library, `http://www.mpfr.org/`.
[12] Intel Corp., The Intel Intrinsics Guide, `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`
[13] T.Granlaud and GMP development team, The GNU Multiple Precision arithmetic library. `http://gmplib.org/`.
[14] M.Nakata, MPLAPACK(MBLAS), `https://github.com/nakatamaho/mplapack`.
[15] 幸谷智紀, 3 倍精度行列乗算の性能評価, 第 173 回 HPC 研究会, 2020.