Vol.2021-HPC-178 No.8
2021/3/15

AVX2を用いたマルチコンポーネント型多倍長精度行列乗算の高速化

幸谷 智紀[1,a)]

概要：本稿では，Binary64(double) 精度の浮動小数点数を複数組み合わせて多倍長精度化するマルチコンポーネント型多倍長精度の行列乗算の高速化を，AVX2 を用いて行った結果について報告する．対象となるのは double-double(DD)，triple-double(TD)，quad-double(QD) の 3 種類の精度計算であり，それぞれ無誤差変換技法を組み合わせて四則演算を実行できる．今回我々は，4 つの double 型浮動小数点数を同時に操作できる x86_64 上の SIMD 命令である AVX2 を用いて無誤差変換技法を構築し，その上に DD, TD, QD の加算と乗算を実装した．また，行列要素読み書き時の高速性を保つため，AVX2 の load/store 命令を使って行えるようベクトル・行列の構造体を構築した．その結果，ブロッキングした行列乗算，Strassen/Winograd 行列乗算それぞれにおいて最大 3 倍程度の高速化を達成し，併せて OpenMP を用いた並列化効率の向上にも成功した．

キーワード：多倍長精度計算, 行列乗算, SIMD, 並列化

Acceleration of multi-component type multiple precision matrix multiplication using AVX2

1. 初めに

科学技術計算の大規模化により，IEEE754-1985 が規定する 2 進単精度 (Binary32)，倍精度浮動小数点演算 (Binary64) では，ユーザの要求する精度の数値解が得られない悪条件問題が多数現れている．そのためにはソフトウェアによって仮数部の桁数を Binary64 以上に設定できる多倍長精度浮動小数点演算が不可欠である．

現状，多倍長精度浮動小数点演算は，Binary32 や Binary64 を複数組み合わせ，無誤差変換技法 [8] を使って演算を構築するマルチコンポーネント方式と，主として整数演算による任意精度浮動小数点演算を構築する多数桁方式の，2 方式によって構築されたものが主流である．前者を代表するものとして Bailey らの QD ライブラリ [10] があり，後者は GNU MP[13] の MPN(Multiple Precision Natural number) カーネルを土台とする MPFR[11] がある．どちらも C/C++ソースコードを土台として構築され，後者は MPN カーネル部分を各種 CPU アーキテクチャ向けに高速化したアセンブラルーチンが強みである．これら両方のライブラリを使用して多倍長精度線形計算ライブラリとしては MPLAPACK/MPBLAS(or MPACK)[14] があり，使用する精度桁数に応じたライブラリが使用できるようになっている．

マルチコンポーネント方式の場合，AVX2 等の SIMD 命令を使用することで，基本線型計算が高速化できることは良く知られており，藤井・長谷川らの構築した Lis[3] はこの高速化を行った線型計算を利用した多倍長精度もサポートした疎行列ライブラリである．AVX2 を使用することで 3 倍を超える高速化を達成したことも報告されている．

本稿では，藤井らの研究を参照しつつ，Binary64(double) 精度の浮動小数点数を複数組み合わせて多倍長精度化するマルチコンポーネント型多倍長精度の行列乗算の高速化を，AVX2 を用いて行った結果について報告する．対象となるのは double-double(DD) だけでなく，triple-double(TD)，quad-double(QD) の 3 種類の計算である．

今回我々は，4 つの double 型浮動小数点数を同時に操作

[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

できる x86_64 上の SIMD 命令である AVX2 を用いて無誤差変換技法を構築し，その上に DD, TD, QD の加算と乗算を実装した．TD については加算の性能を上げるため，QD ベースの 3 倍精度加算を使用している．

また，行列乗算読み込み時の高速性を保つため，AVX2 の load/store 命令を使って行えるようベクトル・行列の構造体を新たに構築した．その結果，ブロッキングした行列乗算，Strassen 行列乗算それぞれにおいて最大 3 倍程度の高速化を達成し，いずれの精度計算でも MPBLAS の Rgemm より高速な計算が可能にであることが判明した．併せて OpenMP を用いた並列化効率の向上にも成功している．

しかしながら，再帰呼び出しを用いた Strassen のアルゴリズム実装は，シンプルなブロッキング行列乗算に対して並列化による速度向上に限界があることも新たに判明した．

2. AVX2 による高速化

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．

DD 精度演算については先行研究で示されている通り，条件分岐がなくスムーズに SIMD 化できるが，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．そこで，正規化処理部分は AVX2 のデータ型`_m256d`が 4 つの `double` 型変数として使用できることから，バラして実装することで処理の一貫性を保つようにした．

2.1 無誤差変換技法と DD 精度加算・乗算

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．

以下，a, b, c, d は`_m256d` データ型であり，それぞれ 4 つの binary64 浮動小数点数 $a = (a_0, a_1, a_2, a_3)$, $b = (b_0, b_1, b_2, b_3)$, $s = (s_0, s_1, s_2, s_3)$, $e = (e_0, e_1, e_2, e_3)$ を持つものとする．

QuickTwoSum と TwoSum は次のように書き換えられる．

FMA を利用する TwoProd 関数は Algorithm 3 のよう

Algorithm 1 $(s, e) :=$ AVX2QuickTwoSum(a, b)

$s :=$ `_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)

に AVX2TwoProd 関数に書き替えられる．

Algorithm 3 $(p, e) :=$ AVX2TwoProd(a, b)

$p :=$`_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．今回はこれらを AVX2DDadd と AVX2DDmul として実装した．

Algorithm 4 $r[2] :=$ AVX2DDadd$(x[2], y[2])$

$(s, e) :=$ AVX2TwoSum$(x[0], y[0])$
$w :=$`_mm256_add_pd(`$x[1], y[1]$`)`; $e :=$`_mm256_add_pd(`e, w`)`
$(r[0], r[1]) :=$ AVX2QuickTwoSum(s, e)
return $(r[0], r[1])$

Algorithm 5 $r[2] :=$ AVX2DDmul$(x[2], y[2])$

$(p_1, p_2) :=$ AVX2TwoProd $-$ FMA$(x[0], y[0])$
$w_1 :=$`_mm256_mul_pd(`$x[0], y[1]$`)`
$w_2 :=$`_mm256_mul_pd(`$x[1], y[0]$`)`
$w_3 :=$`_mm256_add_pd(`w_1, w_2`)`; $p_2 :=$`_mm256_add_pd(`p_2, w_3`)`
$(r[0], r[1]) :=$ AVX2QuickTwoSum(p_1, p_2)

2.2 TD 精度加算と乗算

3 倍精度浮動小数点演算用の正規化手法として，Fabiano らは VecSum と VSEB(k) (VecSum with Blanch) を組み合わせて，演算結果を正規化するようにしていることから，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`_m256d` 変数の要素毎の比較を伴う if 文があり，今回はこの部分は SIMD 化していない．

加算 (TDadd) は，$x[3] = (x[0], x[1], x[2])$，$y[3] = (y[0], y[1], y[2])$ の和 $r[3] = (r[0], r[1], r[2])$ 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 化したものが下記の AVX2TDadd 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 関数は全くできておらず，VSEB(n) 関数はごく一部を除き SIMD 化できていない．

後述するように，これは全く高速化の余地がないため，QDadd において，$x[3] = y[3] = 0$ として 3 倍精度化した TDaddq を実装し，これを SIMD 化した AVX2TDaddq を使用した．

Algorithm 6 $r[3] :=$ AVX2TDadd$(x[3], y[3])$

$(z_0, ..., z_5) :=$ AVX2Merge$(x[0], x[1], x[2], y[0], y[1], y[2])$
$(e_0, ..., e_5) :=$ AVX2VecSum$(z_0, ..., z_5)$
$(r[0], r[1], r[2]) :=$ AVX2VSEB(3)$(e_0, ..., e_5)$
return $(r[0], r[1], r[2])$

Algorithm 7 $r[3] :=$ AVX2TDaddq$(x[3], y[3])$

$s_0 :=$`_mm256_add_pd(`$x[0], y[0]$`)`
$s_1 :=$`_mm256_add_pd(`$x[1], y[1]$`)`
$s_2 :=$`_mm256_add_pd(`$x[2], y[2]$`)`
$v_0 :=$`_mm256_sub_pd(`$s_0, x[0]$`)`
$v_1 :=$`_mm256_sub_pd(`$s_1, x[1]$`)`
$v_2 :=$`_mm256_sub_pd(`$s_2, x[2]$`)`
$u_0 :=$`_mm256_sub_pd(`s_0, v_0`)`
$u_1 :=$`_mm256_sub_pd(`s_1, v_1`)`
$u_2 :=$`_mm256_sub_pd(`s_2, v_2`)`
$w_0 :=$`_mm256_sub_pd(`$x[0], u_0$`)`
$w_1 :=$`_mm256_sub_pd(`$x[1], u_1$`)`
$w_2 :=$`_mm256_sub_pd(`$x[2], u_2$`)`
$u_0 :=$`_mm256_sub_pd(`$y[0], v_0$`)`
$u_1 :=$`_mm256_sub_pd(`$y[1], v_1$`)`
$u_2 :=$`_mm256_sub_pd(`$y[2], v_2$`)`
$t_0 :=$`_mm256_add_pd(`w_0, u_0`)`
$t_1 :=$`_mm256_add_pd(`w_1, u_1`)`
$t_2 :=$`_mm256_add_pd(`w_2, u_2`)`
$(s_1, t_0) :=$AVX2TwoSum$(s1, t0)$
$(s_2, t_0, t_1) :=$AVX2ThreeSum(s_2, t_0, t_1)
$t_0 :=$`_mm256_add_pd(_mm256_add_pd(`t_0, t_1`),` t_2`)`
$(r[0], r[1], r[2]) :=$AVX2Renorm3(s_0, s_1, s_2, t_0)
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 8 $r[3] :=$ AVX2TDmul($x[3]$, $y[3]$)

$(z_{00}^{\mathrm{up}}, z_{00}^{\mathrm{lo}}) :=$ 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 $(r[0],\ r[1],\ r[2])$

ちなみに，TDmul についても，QDmul の 3 倍精度版を作成してベンチマークテストを実施してみたが，TDmul よりもよいパフォーマンスを得られなかったことから，今回は TDaddq と TDmul の組み合わせで行列乗算を実装した．

2.3 QD 精度加算と乗算

QD 演算については，計算量の少ない Sloppy 版の加算と乗算に基づき，AVX2 化した ThreeSum(Algorithm 9)，ThreeSum2(Algorithm 10) と，一部 AVX2 化した Renorm 関数を用いて AVX2QDadd(Algorithm 11) と AVX2QDmul(Algorithm 12) を実装した．

Algorithm 9 $(a, b, c) :=$ AVX2ThreeSum(x, y, z)

$(t_1, t_2) :=$ AVX2TwoSum(x, y)
$(a, t_3) :=$ AVX2TwoSum(z, t_1)
$(b, c) :=$ AVX2TwoSum(t_2, t_3)
return $(a,\ b,\ c)$

Algorithm 10 $(a, b) :=$ AVX2ThreeSum2(x, y, z)

$(t_1, t_2) :=$ AVX2TwoSum(x, y)
$(a, t_3) :=$ AVX2TwoSum(z, t_1)
$b :=$ `_mm256_add_pd`(t_2, t_3)
return $(a,\ b)$

後述するように，正規化に当たる AVX2Renorm 関数以外では完全に AVX2 化できており，DD,TD より AVX2 による高速化が十分に達成できている．今回詳細は省くが，MPFR 212bits 精度より行列乗算の性能は格段に高速化されたことで，QD 以上のマルチコンポーネント型基本線形計算でも MPFR より高速化できる余地が広がったと言える．

Algorithm 11 $r[4] :=$ AVX2QDadd$(x[4], y[4])$

$s_0 :=$ `_mm256_add_pd`$(x[0], y[0])$
$s_1 :=$ `_mm256_add_pd`$(x[1], y[1])$
$s_2 :=$ `_mm256_add_pd`$(x[2], y[2])$
$s_3 :=$ `_mm256_add_pd`$(x[3], y[3])$
$v_0 :=$ `_mm256_sub_pd`$(s_0, x[0])$
$v_1 :=$ `_mm256_sub_pd`$(s_1, x[1])$
$v_2 :=$ `_mm256_sub_pd`$(s_2, x[2])$
$v_3 :=$ `_mm256_sub_pd`$(s_3, x[3])$
$u_0 :=$ `_mm256_sub_pd`(s_0, v_0)
$u_1 :=$ `_mm256_sub_pd`(s_1, v_1)
$u_2 :=$ `_mm256_sub_pd`(s_2, v_2)
$u_3 :=$ `_mm256_sub_pd`(s_3, v_3)
$w_0 :=$ `_mm256_sub_pd`$(x[0], u_0)$
$w_1 :=$ `_mm256_sub_pd`$(x[1], u_1)$
$w_2 :=$ `_mm256_sub_pd`$(x[2], u_2)$
$w_3 :=$ `_mm256_sub_pd`$(x[3], u_3)$
$u_0 :=$ `_mm256_sub_pd`$(y[0], v_0)$
$u_1 :=$ `_mm256_sub_pd`$(y[1], v_1)$
$u_2 :=$ `_mm256_sub_pd`$(y[2], v_2)$
$u_3 :=$ `_mm256_sub_pd`$(y[3], v_3)$
$t_0 :=$ `_mm256_add_pd`(w_0, u_0)
$t_1 :=$ `_mm256_add_pd`(w_1, u_1)
$t_2 :=$ `_mm256_add_pd`(w_2, u_2)
$(s_1, t_0) :=$ AVX2TwoSum(s_1, t_0)
$(s_2, t_0, t_1) :=$ AVX2ThreeSum(s_2, t_0, t_1)
$(s_3, t_0) :=$ AVX2ThreeSum2(s_3, t_0, t_2)
$t_0 :=$ `_mm256_add_pd`$($`_mm256_add_pd`$(t_0, t_1),\ t_3)$
$(r[0], r[1], r[2], r[3]) :=$ AVX2Renorm$(s_0, s_1, s_2, s_3, t_0)$
return $(r[0], r[1], r[2], r[3])$

Algorithm 12 $r[4] :=$ AVX2QDmul$(x[4], y[4])$

$s_0 :=$ `_mm256_add_pd`$(x[0],\ y[0])$
$(p_0, q_0) :=$ AVX2TwoProd$(x[0], y[0])$
$(p_1, q_1) :=$ AVX2TwoProd$(x[0], y[1])$
$(p_2, q_2) :=$ AVX2TwoProd$(x[1], y[0])$
$(p_3, q_3) :=$ AVX2TwoProd$(x[0], y[2])$
$(p_4, q_4) :=$ AVX2TwoProd$(x[1], y[1])$
$(p_5, q_5) :=$ AVX2TwoProd$(x[2], y[0])$
$(p_1, p_2, q_0) :=$ AVX2ThreeSum(p_1, p_2, q_0)
$(p_2, q_1, q_2) :=$ AVX2ThreeSum(p_2, q_1, q_2)
$(p_3, p_4, p_5) :=$ AVX2ThreeSum(p_3, p_4, p_5)
$(s_0, t_0) :=$ AVX2TwoSum(p_2, p_3)
$(s_1, t_1) :=$ AVX2TwoSum(q_1, p_4)
$s2 :=$ `_mm256_add_pd`(q_2, p_5)
$(s_1, t_0) :=$ AVX2TwoSum(s_1, t_0)
$s2 :=$ `_mm256_add_pd`$(s_2,$ `_mm256_add_pd`$(t_0, t_1))$
$s_1 :=$ `_mm256_add_pd`$(s_1,$ `_mm256_mul_pd`$($x[0], y[3]$))$
$s_1 :=$ `_mm256_add_pd`$(s_1,$ `_mm256_mul_pd`$($x[1], y[2]$))$
$s_1 :=$ `_mm256_add_pd`$(s_1,$ `_mm256_mul_pd`$($x[2], y[1]$))$
$s_1 :=$ `_mm256_add_pd`$(s_1,$ `_mm256_mul_pd`$($x[3], y[0]$))$
$s_1 :=$ `_mm256_add_pd`$(s_1,\ q_0)$
$s_1 :=$ `_mm256_add_pd`$(s_1,\ q_3)$
$s_1 :=$ `_mm256_add_pd`$(s_1,\ q_4)$
$s_1 :=$ `_mm256_add_pd`$(s_1,\ q_5)$
$(r[0], r[1], r[2], r[3]) :=$ AVX2Renorm$(p_0, p_1, s_0, s_1, s_2)$
return $(r[0], r[1], r[2], r[3])$

2.4 ベクトル型を用いたベンチマークテスト

ここでは今回使用したベクトル・行列演算のデータ型について解説する．通常，多倍長精度浮動小数点数は一つの構造体としてまとまっており，これを並べて配列として使用することが一般的である．しかし，同じ精度のマルチコンポーネント型浮動小数点数を AVX2 の mm256 型データ単位で扱う場合，同じ演算をまとめて 4 つ同時に実行することで効率を上げる必要がある．前述したように，DD,TD,QD 精度演算ではそれぞれ 2, 3, 4 つの mm256 型データを読み書きする必要があり，ベクトル・行列要素を配列一つにまとめて置く形式では，高性能な読み書きが期待できる mm256 型データの連続呼び出し (load/store 関数) が使えず，Binary64 データを個別に読み出し (set)・書き込む必要がある．

そこで，今回我々は DD, TD, QD 精度データをそれぞれ各コンポーネントごとに Bianryh64 データの一次元配列に分割してベクトル・行列要素を格納する形式を採用した．これにより，例えば TD 精度演算の場合，図 1 に示すように，3 回の load 命令を 2 セット実施することで AVX2TDadd 演算 (`rtd_[add, mul]` 関数) に必要な被演算データを渡すことができる．Binary64 ごとの読み書きを行うよりも多くのケースで高速な処理が必要になることが期待できる．

実際，DD, TD, QD 精度それぞれで，2 つの n 次元実ベクトル $\mathbf{a} = [a_1\ a_2\ ...\ a_n]^T$, $\mathbf{b} = [b_1\ b_2\ ...\ b_n]^T$ を生成し，各要素の加算 $(a_i + b_i)$ と乗算 $(a_i \cdot b_i)$ を行った時の各精度の演算回数 (DD MFLOPS, TD MFLOPS, QD MFLOPS) を Corei9 の環境で計測した結果を図 2，図 3，図 4 に示す．これらのグラフは，AVX2 Load/Store 命令を用いて AVX2 演算を行った場合 (AVX2 L/S, 図 1 左図)，一要素に全てのコンポーネントをまとめて配列としてベクトル要素を並べて AVX2 Set 命令を用いて AVX2 演算を行った場合 (AVX2 Set, 図 1 右図)，AVX2 を一切使用しない場合 (Normal) の 3 種類で MFLOPS 値を各グラフの縦軸に示している．EPYC でも同様の傾向になることを確認している．

図 2 より，DD 精度演算は計算量が少ないため，load/store 関数を利用すると CPU のキャッシュの影響が強く出ることが分かる．実際，全てのベクトルがキャッシュに収まる際の性能が最高に高く，1.5 倍程度の差が出てくる．キャッシュサイズを超える次元数になると，AVX2 化による影響がほぼなくなり，加算・乗算どちらもほとんど同じ DD MFLOPS に収斂していく．

図 3 では左図の加算ではオリジナルの 3 倍精度演算 (TDadd) による結果も併せて示してあるが，AVX2 化のメリットは全く見えず，約 26 TD MFLOPS しか出ていない．これは Merge 関数の性能が著しく低いために引き起こされている．従って，性能向上のために前述したように TDaddq を用いたところ，通常演算で 75 TD MFLOPS, AVX2 化すると Load/store 使用時で 115 TD MFLOPS, Set 使用時で 123 TD MFLOPS の性能を得られることが分かった．TDmul 演算では AVX2 化のメリットは薄く，せいぜい 20 TD MFLOPS 程度の性能向上に留まっている．なお，load/store 使用時に Set 使用より性能が下がったのはこの Corei9 の場合のみである．DD 演算よりキャッシュによる影響はごく少ないことも分かる．

図 4 では，キャッシュサイズによる影響はほとんど見られず，安定的に AVX2 化による性能向上が達成できていることが分かる．QDadd 演算では約 4 倍，QDmul 演算では約 2 倍の高速化が達成されている．QD 精度より長いマルチコンポーネント型の多倍長精度演算では，AVX2 化によりこの程度の性能向上が達成できるのではないかと予想される．

以上の結果より，load/store 命令を用いたマルチコンポーネント型多倍長精度演算が，ベクトルデータ型には性能向上に寄与できる要素が大きいことが示された．行列乗算においても，この形式を用いることで高速化されることが期待できる．

3. 行列乗算のベンチマークテスト

行列要素は行優先 (Row-major) 方式を使用する．

本稿で対象とする任意サイズの実行列乗算を $C := AB = [c_{ij}] \in \mathbb{R}^{m\times n}$ とする．ここで，$A = [a_{ij}] \in \mathbb{R}^{m\times l}$ and $B = [b_{ij}] \in \mathbb{R}^{l\times n}$ である．ここで C の要素 c_{ij} は

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

である．この定義式をそのまま計算する方法を，本稿では単純行列乗算 (Simple) と呼ぶ．我々のライブラリではその他，ブロック化アルゴリズム (Block) と Strassen アルゴリズム (Strassen)，Winograd アルゴリズム (Winograd) をサポートしており，全てのアルゴリズムで OpenMP による並列化を行っている．

3.1 ベンチマークテスト

性能評価のための計算機環境は以下の通りである．MPACK は Github から 2019 年 6 月中旬にダウンロードしたものをインストールした．

Corei9 Intel Core i9-1090X (3.6GHz, 8 cores), 16GB RAM Ubuntu 18.04.2, GCC 7.3.0

EPYC AMD EPYC , 64GB RAM

使用した実正方行列 A，B は次の通りである．

$$A = \left[\sqrt{5}\,(i+j-1)\right]_{i,j=1}^{n},\ B = \left[\sqrt{3}\,(n-i)\right]_{i,j=1}^{n}$$

要素全てが正の実数であるため，桁落ちは殆ど起きない．計算結果は，どの計算においても使用する桁数より 10 進

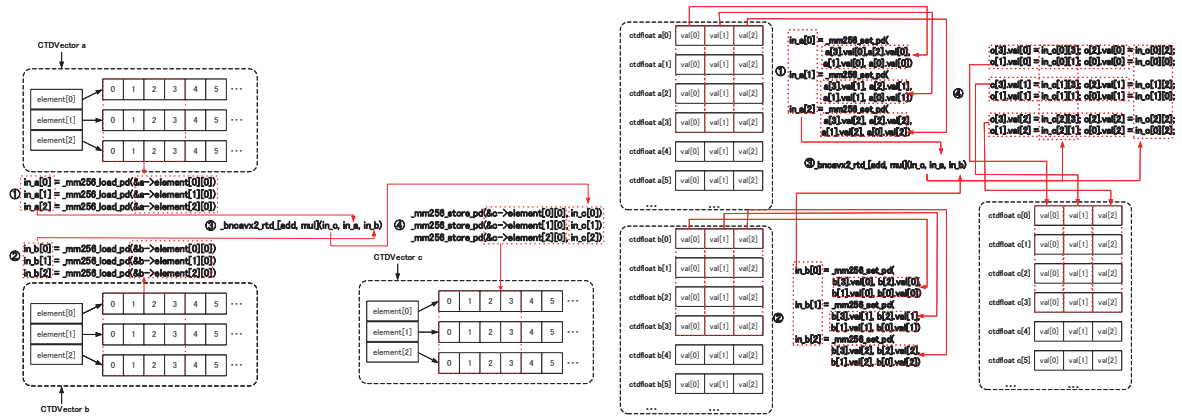

図 1　load/store 命令で呼び出せる TD ベクトルデータ型 (左), set 命令のみ利用する TD 配列 (右)

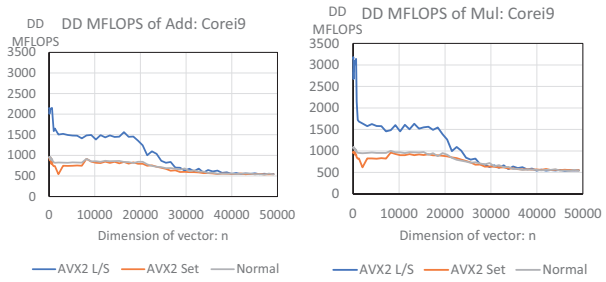

図 2　DD 精度ベクトル要素の加算と乗算の DD MFLOPS の推移

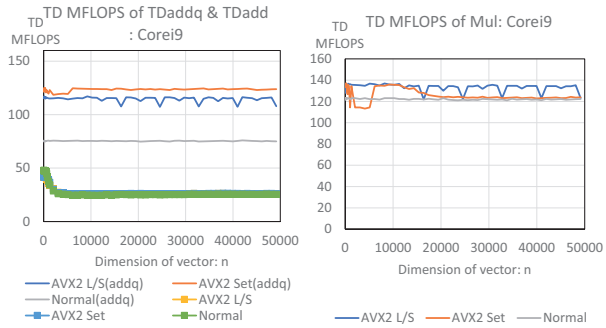

図 3　TD 精度ベクトル要素の加算と乗算の TD MFLOPS の推移

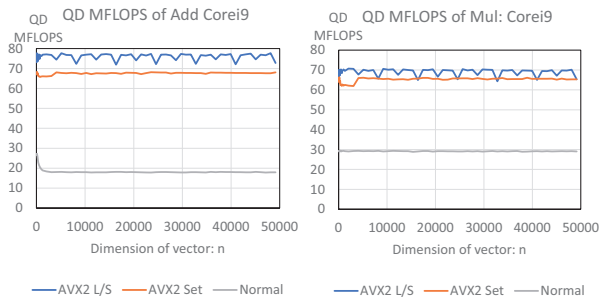

図 4　QD 精度ベクトル要素の加算と乗算の QD MFLOPS の推移

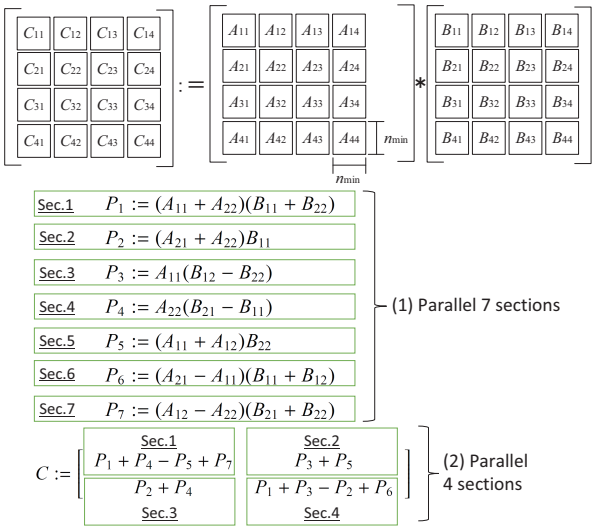

図 5　ブロック化行列乗算 (上) と Strassen 行列乗算

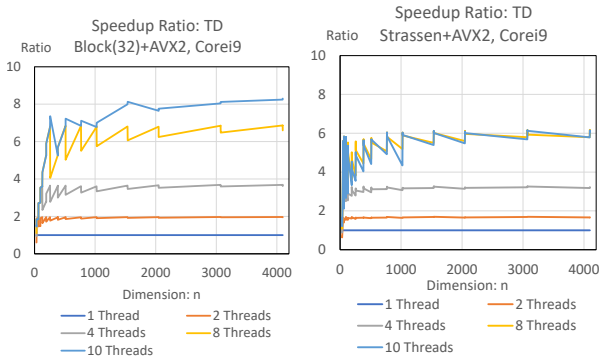

図 6 TD 精度行列乗算の並列化効率; Corei9

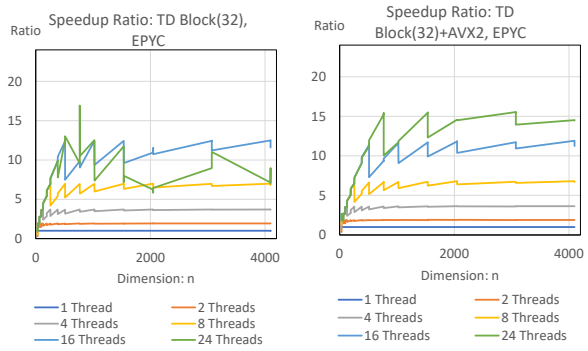

図 7 TD 精度ブロック化行列乗算の並列化効率改善:EPYC

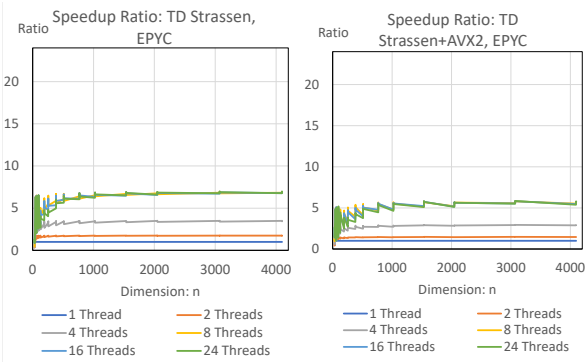

図 8 TD 精度 Strassen 行列乗算の並列化効率改善:EPYC

1〜2 桁の精度低下がみられる程度であった．

ベンチマークで使用したメイン関数を含むプログラムは全て C++で実装し，MBLAS の Rgemm 関数を直接呼び出せるようにしている．C++のコンパイルオプションは

```
g++ -O3 -std=c++11 -mavx2 -mfma -fopenmp
```

である．

3.2 シリアル計算: Rgemm(MPBLAS) との比較

Corei9 および EPYC 環境で，並列化を行わずに行列乗算の計算時間を計測した結果を表 1 に示す．ブロック化行列乗算 (B と略記)，Strassen 行列乗算 (S と略記)，そしてそれぞれ AVX2 化したものを B+A, S+A と略記したフィールド列に示している．併せて，MPBLAS がサポートする DD 精度，QD 精度演算の結果 (M と略記) も示してある．

既に報告済みであるが，Strassen 行列乗算を用いることで，一定以上の行列サイズに対しては MPBLAS(Rgemm 関数) より高速になる．今回 AVX2 化した行列乗算を実装したことで，ブロック化乗算でも MPBLAS より 2 倍以上の高速化を達成しており，行列サイズに関わらず高速になることが確認できた．MPBLAS がサポートしていない TD 演算では，全てのアルゴリズムと AVX2 化した計算時間が DD 精度と QD 精度の計算時間の中間に位置していることも確認できる．

3.3 並列化効率と計算時間

OpenMP による並列化の結果をここで示す．Corei9 の場合，AVX2 化による並列化効率への影響はあまり見られないことから，典型例として TD 精度の場合の並列化効率を図 6 に示す．ブロック化行列乗算では安定的に 10 スレッドまで高速化が行われていることが分かるが，Strassen 行列乗算の場合，8 スレッド以上使っても性能向上が 6 倍程度で頭打ちとなることがわかる．現状では再帰呼び出しする中でスレッドを生成しており，恐らくは 8 スレッド以上のスケジューリングがうまく行われていないと思われる．

EPYC 環境では，AVX2 化しない場合は，16 スレッド以上で並列化による性能向上にブレーキがかかることが判明した．DD 精度，QD 精度演算でも，一定以上の行列サイズになると並列化効率がガタ落ちする現象が見られる．しかし，AVX2 化することで安定的な並列化効率が得られることが分かる．これについても DD,QD 精度では同様のことが言える．

Strassen 行列乗算における並列化効率の頭打ち現象は EPYC 環境でも同様で，やはり 8 スレッド以上では性能向上が得られないことが分かる．AVX2 化することで，並列化効率は若干下がるが，頭打ち現象は AVX2 化によっても変わらない．現状の Strassen 行列乗算を抜本的に書き換え，再帰呼び出しを使用しないようにした並列化プログラムにしない限り，マルチコア環境での性能向上は見込めないと思われる．

以上のことから，実際にコア数と同じスレッド数で並列化を行った行列乗算の結果を表 2 に示す．Corei9 は 10 スレッド，EPYC は 24 スレッドでの実行時間を全てリストアップしてある．

ブロック化行列乗算の並列化効率が安定的であるのに対し，Strassen 行列乗算の並列化効率は 8 スレッドで頭打ちになることから，スレッド数が多くなればなるほど前者の方が高速になることが予想される．実際，10 スレッドの Corei9 では Strassen 行列乗算が最高速であるのに対し，24 スレッドの EPYC 環境ではブロック化行列乗算の方が高速になるケースが多い．おそらく，全ての精度計算において，32 コア以上の環境ではブロック化行列乗算が最高速に

表 1 行列乗算の計算時間: Corei9(左), EPYC(右)

DD : Corei9						DD : EPYC					
n	B	B+A	S	S+A	M	n	B	B+A	S	S+A	M
1023	7.84	2.46	4.35	1.57	5.98	1023	9.27	2.37	5.00	1.43	7.55
1024	7.86	2.46	4.34	1.55	6.00	1024	9.31	2.37	4.98	1.41	7.54
1025	8.61	2.68	4.40	1.59	6.01	1025	10.16	2.57	5.03	1.44	7.59
4095	507.77	162.73	212.25	74.47	390.70	4095	595.28	163.64	243.63	68.04	482.82
4096	509.13	161.83	212.49	74.07	390.67	4096	609.12	163.55	243.59	67.75	483.98
4097	518.03	161.68	213.28	74.94	390.95	4097	611.41	152.60	244.72	71.03	483.75
TD : Corei9						TD : EPYC					
1023	50.06	20.54	26.75	12.64	N/A	1023	61.12	21.19	32.18	12.97	N/A
1024	50.10	20.54	26.96	12.48	N/A	1024	61.15	21.16	31.98	12.78	N/A
1025	54.28	22.56	26.95	12.61	N/A	1025	66.46	23.17	32.20	12.89	N/A
4095	3202.43	1316.15	1276.04	619.45	N/A	4095	3918.16	1378.82	1555.08	632.29	N/A
4096	3205.11	1316.68	1276.27	618.64	N/A	4096	3933.92	1377.99	1554.19	631.47	N/A
4097	3272.34	1345.02	1286.50	620.45	N/A	4097	4002.87	1387.36	1561.32	634.16	N/A
QD : Corei9						QD:EPYC					
1023	102.82	31.41	54.90	19.55	73.76	1023	127.79	42.82	71.23	25.17	83.95
1024	102.87	31.40	54.71	19.41	76.04	1024	127.82	42.78	71.10	25.07	84.18
1025	111.76	34.41	55.23	19.63	74.12	1025	139.64	46.88	71.55	25.32	84.47
4095	6491.08	2012.78	2719.66	970.50	4728.68	4095	8177.30	2753.37	3440.47	1244.59	5383.92
4096	6493.10	2013.14	2720.92	968.58	4730.49	4096	8187.72	2753.73	3438.13	1242.58	5390.37
4097	6624.38	2059.28	2724.59	972.62	4727.34	4097	8370.84	2808.05	3449.23	1246.89	5393.45

表 2 並列行列乗算の計算時間: Corei9, 10 Threads(左), EPYC 24 Threads(右)

DD : Corei9 10 Threads					DD : EPYC 24 Threads				
n	B	B+A	S	S+A	n	B	B+A	S	S+A
1023	1.15	0.35	0.86	0.39	1023	0.78	0.20	0.80	0.59
1024	1.15	0.35	0.69	0.33	1024	0.73	0.20	0.77	0.54
1025	1.23	0.37	0.79	0.38	1025	0.79	0.22	0.84	0.57
4095	61.51	19.68	33.28	16.63	4095	160.68	10.74	35.67	25.43
4096	61.97	19.67	32.60	16.19	4096	191.24	10.72	35.24	24.90
4097	62.65	19.42	34.20	16.44	4097	169.09	10.28	36.26	25.23
TD : Corei9 10 Threads					TD : EPYC 24 Threads				
1023	7.65	3.03	4.57	2.91	1023	4.88	1.81	5.16	2.80
1024	7.67	3.02	4.11	2.07	1024	6.18	1.81	4.68	2.30
1025	8.12	3.22	4.30	2.13	1025	9.01	1.95	4.85	2.36
4095	409.94	159.68	202.34	107.08	4095	549.21	95.02	229.31	117.44
4096	410.48	159.66	196.19	100.55	4096	437.79	94.76	223.25	109.44
4097	416.29	162.04	198.99	101.92	4097	570.95	95.98	225.54	110.36
QD : Corei9 10 Threads					QD : EPYC 24 Threads				
1023	15.23	4.65	9.12	3.43	1023	26.95	3.35	8.91	4.07
1024	15.22	4.64	8.73	3.13	1024	28.98	3.37	8.40	3.50
1025	16.18	4.95	8.88	3.20	1025	27.35	3.60	8.53	3.56
4095	824.23	247.43	437.68	158.57	4095	863.46	171.27	415.35	179.00
4096	822.68	247.48	431.15	153.63	4096	865.12	171.13	407.80	170.27
4097	834.92	251.47	434.29	155.50	4097	882.51	172.95	410.26	171.72

なるものと予想される．

4. 結論と今後の課題

今回の実装により，DD, TD, QD 精度のブロック化行列乗算，Strassen 行列乗算いずれにおいても AVX2 化によって 2〜4 倍の性能向上が達成できることが判明した．これらの高速化は並列化によっても損なわれることはなく，EPYC 環境のように並列化効率の安定化にも寄与できることが示された．

今回の実装では Strassen 行列乗算に必要となる BLAS Level1, Level2 の演算も実装済みである．今後は主要な BLAS 演算の AVX2 化による性能評価をまとめ，これらの基本線型計算を用いた応用的な計算事例においても有用な活用ができることを示していきたい．

謝辞 本研究は，JSPS 科研費 JP20K11843 の助成を受けたものである．

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.