Vol.2021-HPC-180 No.6
2021/7/20

AVX2を用いたマルチコンポーネント型多倍長精度直接法の性能評価

幸谷 智紀[1,a)]

概要：多倍長精度浮動小数点演算が必要な程度に悪条件な連立一次方程式を安定的に解くためには，直接法が第一に試されるべきである．我々はすでに AVX2 を用いてマルチコンポーネント型多倍長精度行列乗算が高速化できることを示した．本講演ではその応用として，直接法がどの程度の性能を持つのか，複数の x86_64 環境下でベンチマークテストを行いその結果を示す．

キーワード：SIMD，AVX2, マルチコンポーネント方式，多倍長精度浮動小数点演算，直接法，LU 分解

Performance evaluation of multi-component-type multiple precision direct methods with AVX2

Abstract: Direct methods should be firstly applied to ill-conditioned linear equations that we must use multiple precision floating-point arithmetic. We have already shown that the application of AVX2 instructions to multi-component-type matrix multiplication is effective. In this talk, we will describe that how much the acceleration with AVX2 is also effective to LU decomposition in direct method through benchmark tests on x86_64 computational environment.

Keywords: SIMD instruction, AVX2, multi-component type, multiple precision floating-point arithmetic, direct method, LU decomposition

1. 目的

連立一次方程式は科学技術計算の基本であり，高速な解導出方法の追求は高性能計算研究の主要テーマである．特に，密行列を係数行列として持つ場合は，LU 分解と前進・後退代入から成る直接法の適用が最初に考えられる．直接法は有限回で必ず計算が終了し，枢軸選択を行うことで丸め誤差に対する頑健性を持つことがよく知られており，ことに IEEE754 倍精度 (binary64) では不十分な，多倍長精度浮動小数点演算 (以下「多倍長演算」と略記）を使用せざるを得ない悪条件問題に対しては，直接法の実装が必要不可欠である．

多倍長演算の実装方法のうち，広く使用されているライブラリに使われているのは現在２種類に絞られている．一つはハードウェアがサポートする binary64 等の浮動小数点数を複数組み合わせて実装するマルチコンポーネント方式で，Bailey らの QD[1] およびその派生ライブラリが多数使用されている．もう一つは，整数演算をベースに浮動小数点演算を構築する多数桁方式であり，GNU MP[13] の多倍長自然数演算 (MPN) カーネルを土台として任意仮数部長の多倍長演算を実現した MPFR[11] や，藤原による exflib[17] が広く使用されている．

現在，多倍長演算をサポートした直接法は，QD や MPFR を土台とする MPLAPACK[9] がサポートする Rgesv もしくは Rgetrf(LU 分解)+Rgetrs(前進・後退代入) があるが，LAPACK[7]/BLAS[2] のリファレンスコードがベースになっているため，ATLAS[12]，OpenBLAS[10], oneMKL[8] のように最適化されたライブラリに比べると計算速度に改善の余地がある．マルチコア CPU 上での高速化を考えた場合，アセンブラに頼らない技法としては，AVX2[3] 等の SIMD(Single Insturuction Multiple Data) 命令セット

[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

を利用して 1 コア内での並列化を行いつつ，OpenMP や Pthread を用いて複数スレッドによる並列化を行うことが現在では主流である．これを最初にマルチコンポーネント方式の多倍長精度行ったのは菱沼ら [6] であり，現在では Lis[14] および MuPAT[15] といったオープンソースのライブラリとして利用することができる．

我々は菱沼らの手法を用いて，binary64 を 2 つ組み合わせた Double-double(DD) 精度だけでなく，3 つ組み合わせた Triple-double(TD) 精度，4 つ組み合わせた Quadruple-double(QD) 精度の基本線形計算と行列乗算の高速化を行った．その結果は既に報告済み [16] であるが，それをベースに，今回は直接法，特に計算量の多い LU 分解の並列化を行い，高速化を図った．その結果，DD 精度では不十分ではあるが，TD 精度，QD 精度では相応の高速化を達成することができた．

本稿ではまず AVX2 によるマルチコンポーネント型多倍長精度基本線形計算の高速化の手法と成果について簡単に示す．次に，今回使用した LU 分解のアルゴリズムとその並列化方法について示し，ベンチマークテストによってその性能評価を行う．また，DD 精度と QD 精度については MPLAPACK の計算時間と比較を行い，AVX2 と OpenMP による並列化によって，Rgetrf より我々の実装した LU 分解が高速にできることを示す．最後に結論と今後の課題を述べる．

2. AVX2 による基本線形計算の高速化

マルチコンポーネント型の多倍長精度浮動小数点演算を用いた基本線形計算が SIMD 命令を使うことで高速化できることは既に菱沼ら [6] の研究によって明らかにされているが，TD, QD 精度演算含めた網羅的なベンチマークテストを伴ったものは我々のもの [16] 以外には見当たらないようである．ここではマルチコンポーネント型多倍長精度 LU 分解実装にあたり，使用した DD, TD, QD 精度の演算の一部と，AVX2 の Load/Store 関数の利用に適したベクトル・行列のデータ構造についてまとめて紹介する．

なお，本稿で使用する 2 つの x86_64 環境，Corei9 と EPYC は下記のものである．MPLAPACK 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て使用している．

Corei9 Intel Core i9-1090X (3.6GHz, 8 cores), 16GB RAM Ubuntu 20.04.02, GCC 9.3.0

EPYC AMD EPYC 7402P (2.8GHz, 24 cores), 64GB RAM, Ubuntu 18.04.5, GCC 7.5.0

2.1 DD, TD, QD 精度演算と AVX2 化

今回我々は，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, AVX2TwoSum, AVX2TwoProd-FMA 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．

DD 精度の加算と乗算

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 1 $r[2] :=$ 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 2 $r[2] :=$ 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)

TD 精度の加算と乗算

最適化された 3 倍精度浮動小数点演算は Fabiano ら [4] によって提唱されたものである．VecSum と VSEB(k) (VecSum with Error Branch) 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．ただし，加算については高速化の余地がないため，QD 精度の加算において $x[3] = y[3] = 0$ 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．

QD 精度の加算と乗算

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 と ThreeSum2，一部 AVX2 化した Renorm 関数を用いて AVX2QDadd(Algorithm 5) と AVX2QDmul(Algorithm 6) を実装した．

Algorithm 3 $r[3] :=$ 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 $(r[0], r[1], r[2])$

Algorithm 4 $r[3] :=$ 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 $(r[0],\ r[1],\ r[2])$

Algorithm 5 $r[4] :=$ 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 6 $r[4] :=$ 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 $(r[0], r[1], r[2], r[3])$

2.2 行列・ベクトルの実装

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．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 binary64 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．

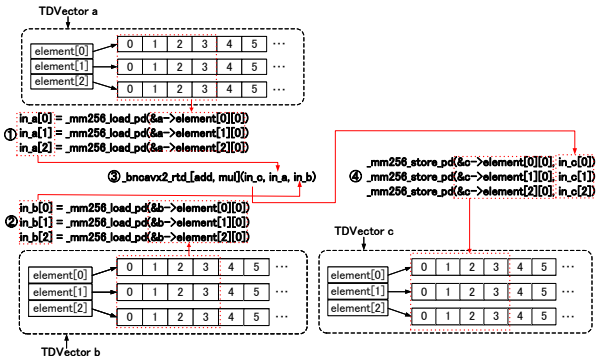

図 1　Load/Store 命令で呼び出せる TD ベクトルデータ型

このデータ構造を用いたベクトル要素単位の加算・乗算のベンチマークテスト結果の詳細については [16] を参照されたい．

3. LU 分解のベンチマークテスト

現在，LAPACK 等で使用されている LU 分解のアルゴリズムは行列乗算を使用している [5]．我々は既に前述の AVX2 命令を用いて最適化した多倍長精度の行列乗算を実装しており，Block 行列乗算，Strassen 行列乗算，いずれのアルゴリズムでも既存の多倍長行列乗算より高速に計算ができることを示した [16]．今回はこれらを用いて LU 分解を実装し，かつ並列化することでどの程度の性能が得られるかをベンチマークテストによって示す．また，比較のために一行単位で計算を行う LU 分解を実装し，併せて AVX2 化して行列乗算利用の LU 分解との比較も行う．

使用する連立一次方程式 (1) は，

$$A\mathbf{x} = \mathbf{b} \tag{1}$$

であり，$A \in \mathbb{R}^{n\times n}$, $\mathbf{b} \in \mathbb{R}^n$ とする．

3.1 行列乗算を用いた LU 分解のベンチマークテスト

行列乗算を用いた LU 分解過程の一部を図 2 に示す．

予め設定した列幅 K 単位で LU を次のように計算していく．

(1) A を，$A_{11} \in \mathbb{R}^{K\times K}$, $A_{12} \in \mathbb{R}^{K\times(n-K)}$, $A_{21} \in \mathbb{R}^{(n-K)\times K}$, $A_{22} \in \mathbb{R}^{(n-K)\times(n-K)}$ に分割する．

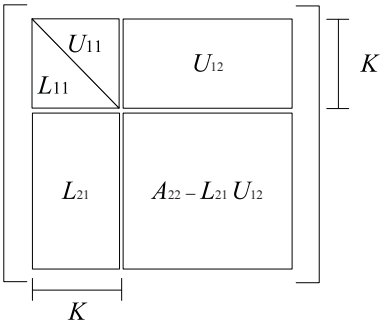

図 2　行列乗算を用いた LU 分解

(2) A_{11} を $L_{11}U_{11}(=A_{11})$ と LU 分解した後，A_{12} を U_{12} に，A_{21} を L_{21} に LU 分解する．
(3) 最後に，$A_{22}^{(1)} := A_{22} - \underline{L_{21}U_{12}}$ としてこの段階の計算を終える．

$A := A_{22}^{(1)}$ と代入し，$n - K \geq 0$ であるうちは計算を続行する．

今回は乱数行列として A を設定し，各要素 a_{ij} は $[-1, 1]$ 区間の一様乱数として与える．次元数は $n = 1024$ とする．

真の解は $\mathbf{x} = [0\ 1\ ...\ n-1]^T$ とし，定数ベクトルは $\mathbf{b} := A\mathbf{x}$ として求める．条件数 $\kappa_1(A)$ は $n = 1024$ で $O(10^6)$ 程度である．

最小計算時間を得られる K を探索するため，$K = \alpha n_{min}$ $(\alpha = 1, 2, ..., 15)$ とし，列幅の刻み単位を $n_{min} = 32$ とした．各 K に対して計算時間（秒）と得られた数値解の各成分の相対誤差の最大値を確認する．図 3 と図 4 に Corei9 と EPYC 環境におけるすべての行列乗算使用型 LU 分解のベンチマーク結果を示す．

全体的な傾向を下記にまとめる．

相対誤差について

DD 精度, TD 精度, QD 精度いずれも条件数分程度の精度減少結果となっており，どの K であっても，また Strassen, Block 乗算いずれに対しても顕著な精度の違いは見られない．

AVX2 化による性能向上について

DD 精度, TD 精度, QD 精度いずれの LU 分解においても，AVX2 化による性能向上が見られる．ただし，後述するように並列化によってその向上幅が減少していく．また，TD 精度における性能向上率は Corei9, EPYC いずれにおいても DD 精度，QD 精度より低めに推移する．

並列化性能について

TD 精度，QD 精度については Corei9, EPYC いずれの環境下でも並列化によって計算時間を低減させることができている．ただし，Strassen 行列乗算の利用や AVX2 化による性能向上の割合も減る．

最も顕著なのは DD 精度の並列化による性能悪化である．ここでは並列なし (Serial)，8 スレッド，24 スレッド (EPYC のみ) の結果のみ示しているが，OpenMP による並列性能が一番悪いのが 2 スレッド実行時であり，そこからスレッドを増やすことで計算時間は減少していくものの，8 スレッド程度では並列化なしの結果に追いつかない．この原因は不明であるが，今の所は DD 精度 LU 分解は並列化なしの利用をお願いする他ないと考えている．

3.2 一行単位の LU 分解と MPLAPACK との比較

現在標準的な行列乗算を用いた LU 分解と比較するため，一行単位の消去を行う LU 分解（Rowwise, R と略記) も実装し，併せてこの消去過程を AVX2 化し (R+A と略記)，OpenMP による並列化も行った．図 3 と図 4 の最小計算時間 (とその K 値) を抜粋した結果をまとめた．並列化なしの結果を MPLAPACK の Rgetrf の結果 (M と略記）と併せて表 1 に示す．当該精度における最小の経産時間には下線を引いてある．

表 1 シリアル計算時の最小計算時間 (秒)

Corei9

$n = 1024$		R	R+A	B	B+A	S	S+A	M
DD	min s	1.6	<u>0.6</u>	2.7	0.98	2.3	0.94	2.0
	K			64	32	96	32	–
TD	min s	11.3	<u>5.8</u>	11.6	7.3	10.2	6.8	N/A
	K			64	32	96	64	–
QD	min s	36.9	<u>8.8</u>	39.2	14.0	31.5	14.4	24.7
	K			64	32	160	32	–

EPYC

$n = 1024$		R	R+A	B	B+A	S	S+A	M
DD	min s	2.1	<u>0.6</u>	3.1	0.93	2.4	0.86	2.5
	K			480	32	160	32	–
TD	min s	13.4	<u>7.5</u>	13.5	9.5	10.6	7.9	N/A
	K			96	96	160	224	–
QD	min s	46.6	<u>13.9</u>	44.2	17.2	35.7	16.3	28.2
	K			480	32	288	64	–

並列化なしの結果は次のようにまとめられる．

(1) AVX2 化した LU 分解は，Corei9，EPYC 両環境下において，DD 精度，TD 精度，QD 精度いずれにおいても MPLAPACK の Rgetrf 関数より高速である．
(2) 行列乗算を用いた LU 分解は，AVX2 下によって最小計算時間となる K が小さくなる傾向にある．併せて，Strassen 行列乗算による計算時間低減効果も低く抑えられる．
(3) AVX2 化した一行単位計算を行った LU 分解は，行列乗算を用いた LU 分解よりいずれも高速である．ただし，実行時間のブレも大きい．

並列化ありの結果を表 2 に示す．

並列化した LU 分解の結果は次のようにまとめられる．

(1) TD 精度，QD 精度の LU 分解が並列化，AVX2 化によって高速化しているのに対し，DD 精度では並列化による性能低下が目立つ．また一行単位の LU 分解では DD 精度のときのみ AVX2 化によって低速になる現象も見られる．
(2) 並列化なしの結果と異なり，行列乗算を用いた場合に最小計算時間となる場合が増える．

4. 結論と今後の課題

連立一次方程式に対する直接法において，最も計算時間を要する LU 分解を AVX2 化した多倍長精度演算を利用す

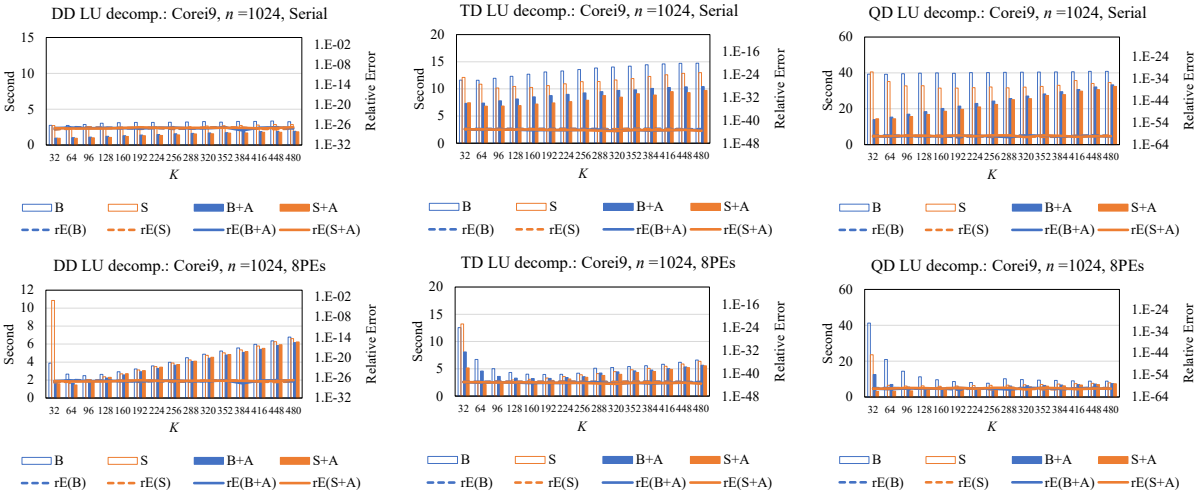

図 3 Corei9 における DD, TD, QD 精度 LU 分解の計算時間と数値解の最大相対誤差: シリアル計算 (上段) と 8 スレッド並列計算 (下段)

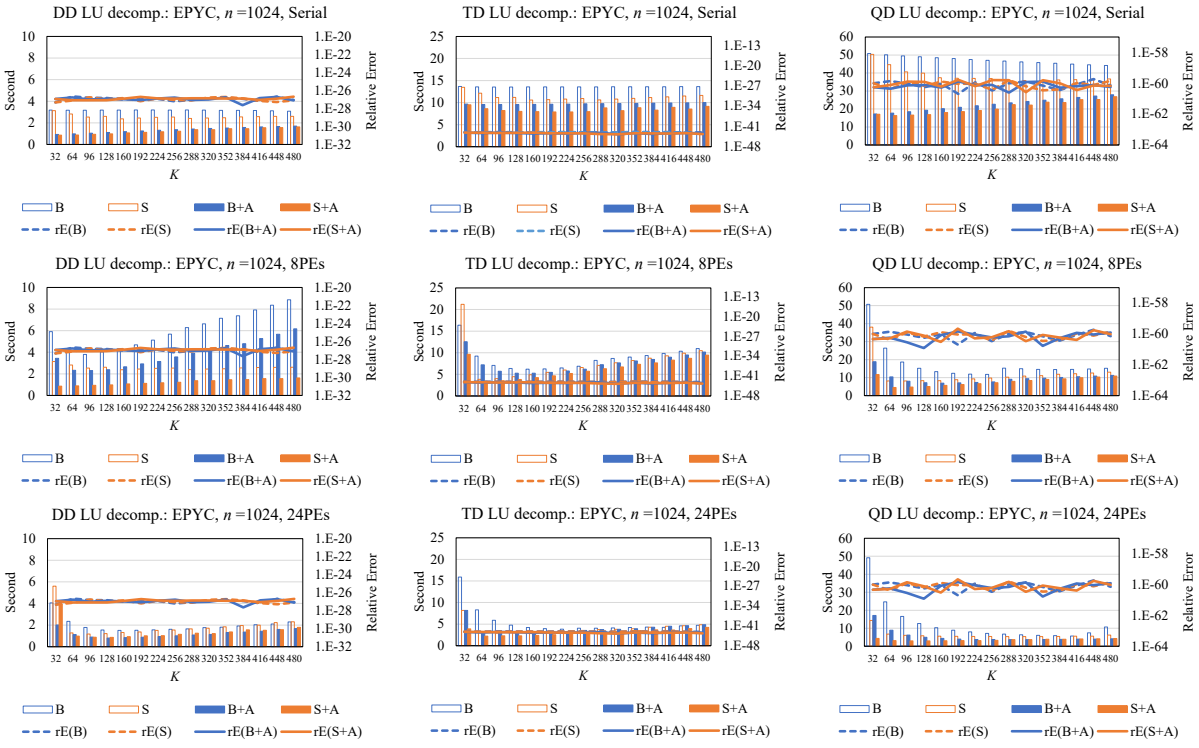

図 4 EPYC における DD, TD, QD 精度 LU 分解の計算時間と数値解の最大相対誤差: シリアル計算 (上段), 8 スレッド並列計算 (中段), 24 スレッド並列計算 (下段)

ることで，どの程度の性能向上が図れるかを検証した．結果として，並列化なしの場合は既存のものより AVX2 化することで高速化することができ，特に一行単位の計算を行う LU 分解が最も高速であることが判明した．また，並列化によって TD 精度，QD 精度の LU 分解が高速化できることは明確に示すことができたものの，DD 精度においては性能低下減少が見られ，現状の実装ではコア数の少ない環境下の利用が適さないことも判明した．

今後の課題としては，DD 精度 LU 分解の性能向上を目指すとともに，Python 等，様々な環境から利用しやすい高性能な多倍長精度 BLAS ライブラリとして整備し，公開を目指すことが挙げられる．

謝辞 本研究は，科研費 20K11843 の助成を受けたものである．また 2021 年度は静岡理工科大学の助成も得た．関係各位に感謝する．

表 2 並列計算時の最小計算時間 (秒)

Corei9, 8PEs

$n = 1024$		R	R+A	B	B+A	S	S+A
DD	min s	1.9	0.24	2.5	1.66	1.65	1.45
	K			96	64	64	64
TD	min s	5.34	3.75	3.96	3.21	2.70	2.14
	K			192	160	64	64
QD	min s	13.4	2.9	7.7	4.8	6.0	3.1
	K			256	192	160	32

EPYC, 8PEs

$n = 1024$		R	R+A	B	B+A	S	S+A
DD	min s	1.9	7.5	3.8	2.3	2.4	0.9
	K			96	96	160	32
TD	min s	10.7	5.1	6.2	5.3	3.8	3.4
	K			160	192	64	64
QD	min s	25.9	6.9	11.8	7.0	8.1	4.5
	K			256	160	64	64

EPYC, 24PEs

$n = 1024$		R	R+A	B	B+A	S	S+A
DD	min s	1.1	1.3	1.5	0.8	1.2	0.9
	K			160	128	96	128
TD	min s	3.7	2.0	3.8	2.9	2.9	2.2
	K			224	96	96	64
QD	min s	7.8	2.5	5.7	3.7	5.4	2.9
	K			416	288	352	128

参考文献

[1] D.H. Bailey. QD. https://www.davidhbailey.com/dhbsoftware/.
[2] BLAS. http://www.netlib.org/blas/.
[3] Intel Corp. The intel intrinsics guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/.
[4] 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.
[5] G.H.Golub and C.F.van Loan. *Matrix Computations (4th ed.)*. Johns Hopkins University Press, 2013.
[6] T.Hishinuma, A.Fujii, T.Tanaka, and H.Hasegawa. Avx acceleration of dd arithmetic between a sparse matrix and vector. *Parallel Processing and Applied Mathematics*, pp. 622–631, 2014. Springer.
[7] LAPACK. http://www.netlib.org/lapack/.
[8] Intel Math Kernel Library. http://www.intel.com/software/products/mkl/.
[9] MPLAPACK/MPBLAS. Multiple precision arithmetic LAPACK and BLAS. http://mplapack.sourceforge.net/.
[10] OpenBLAS. http://www.openblas.net/.
[11] MPFR Project. The MPFR library. https://www.mpfr.org/.
[12] ATLAS: Automatically Tuned Linear Algebra Software. http://math-atlas.sourceforge.net/.
[13] T.Granlaud and GMP development team. The GNU Multiple Precision arithmetic library. https://gmplib.org/.
[14] T.Kotakemori, S. Fujii, H. Hasegawa, and A. Nishida. Lis: Library of iterative solvers for linear systems. https://www.ssisc.org/lis/.
[15] H. Yagi, 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.
[16] 幸谷智紀. 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. 第 178 回 HPC 研究会, 2021.
[17] 藤原宏志. exflib. http://www-an.acs.i.kyoto-u.ac.jp/~fujiwara/exflib/.