AVX2 を用いた Python プログラミング環境における 多倍長精度線形計算高速化の試み

Trial acceleration of multiple precision linear computation with AVX2 on Python programming environment

幸谷 智紀*

Tomonori KOUYA

Abstract: Current Python programming enviroment in 2020 does not have any reliable and effieicent multiple precision floating-point (MPF) arithmetic except the “mpmath” package based on GNU MP(GMP) and MPFR libraries. Although it is well known that multi-component type MPF library can be utilized for middle length precision arithmetic under 200 bits, they are not widely used in Python environment. In this paper, we describe our BNCmatmul library, which is more accelerated MPF linear computation library with AVX2 techniques and can be combined with RDD.py as shown on our paper in 2020, and demonstrate more efficient MPF linear computation than only with RDD.py through bechmark tests on Core i9 environment.

1. 初めに

Python[1] はスクリプト言語の一種であり，今や科学技術計算だけでなく，深層学習・機械学習等，幅広い領域で使用される汎用環境，所謂「Python エコシステム」として広く全世界に普及しつつある．そのエコシステムを下支えする NumPy, SciPy といった高速性を求められるパッケージやモジュール群は，既存のコンパイラ言語である C/C++によってライブラリ化されたものが使用される．スクリプトをその都度解釈して実行するインタプリタはどうしても動作的にコンパイラが吐き出す機械語コードに比べると低速になる．Julia[2] のようにそれを克服しようという野心的なプロジェクトはあるものの，慣れ親しんだ信頼性の高い Python エコシステムを使い続けたいというニーズは根強い．故に，任意精度浮動小数点演算を必要とする高性能計算のためには，土台部分に Python スクリプトではなく，コンパイラ言語を用いた信頼性の高いライブラリを使用する必要がある．

2020 年現在の Python 環境には MPFR や GNU MP(GMP) をベースとした任意精度浮動小数点演算が可能な mpmath パッケージ以外，有望な多倍長精度演算パッケージが存在しない．比較的精度が低い場合は，IEEE754 binary32 や binary64 を複数使用するマルチコンポーネント方式に基づく多倍長浮動小数点演算が有用であることが知られているが [10]，これらはまだ Python 環境で広く使われているとは言い難い．本論文では昨年度示したマルチコンポーネント型多倍長精度演算ライブラリ RDD.py[12] と併用でき，更に AVX2 を用いて高速化した多倍長精度線形計算ライブラリ BNCmatmul[8] が Python 環境でも使用でき，かつ RDD.py より高速な線形計算が可能であることを示す．

2. AVX2 による多倍長精度線形計算の高速化

コンピュータの処理速度を上げるには，ハードウェアの動作周波数を上げるか，同時に使用できるハードウェア資源を増やして並列化する他ない．現状，ハードウェアの集積度が頭打ちになりつつあり，消費電力を減らす社会的圧力が高いことから前者を推し進めるのは困難な状況にあり，後者の方法が広く使われるようになっている．Fig.1 に示すよう，スーパーコンピュータでは計算ノード (computing node) を高速なネットワークで束ね，計算ノード内でも多数のコアを持つ CPU や GPU を複数と搭載し，CPU コアや GPU 内でのミクロレベルの並列化も行い，基本線形計算や画像処理等，高速化に向いた手法を複数組み合わせ，MPI, CUDA, OpenCL, OpenMP 等を使ってソフトウェアレベルで最適化を行うことが，処理能力を高めるためには必須のタスクとなっている．

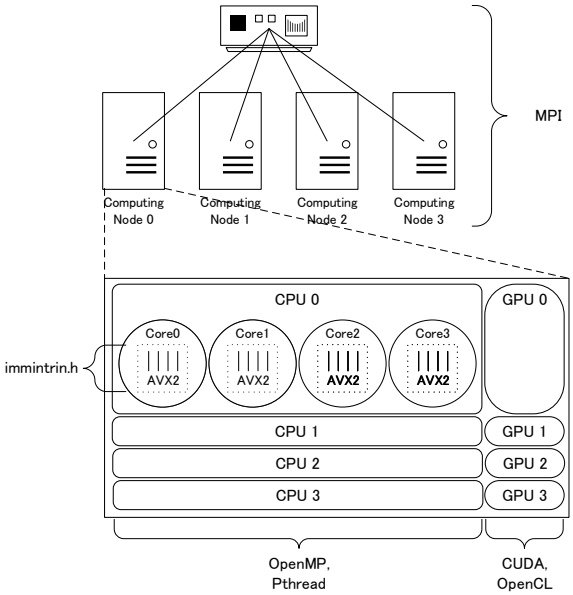

Fig. 1: 並列レベルとプログラミング手法

我々は既に MPI を用いた多倍長精度計算ライブラリ BNCpack[5][6] をリリースしており，OpenMP を用いた並列化も様々な多倍長精度行列乗算アルゴリズムに適用して高速化を行った BNCmatmul ライブラリ [8] を公開している．唯一，CPU コア内で，1 命令で複数の計算を同時に行う SIMD(Single Instruction, Multiple Data) 命令を使用した高速化は行ってこなかっ

2021 年 5 月 27 日 受理

*情報学部 コンピュータシステム学科

た．我々が使用している任意精度演算ライブラリ MPFR(Fig.2 上) を下支えする GNU MP(GMP) の MPN ライブラリでは SIMD 命令を使った最適化を行ってはいるが，完全に SIMD 命令の能力を生かしたものではないという指摘もある．

また，Dekker[4] の無誤差変換技法 (error-free transformation) を用いたマルチコンポーネント方式の固定精度演算 (Fig.2 下) には，小武守らの Lis[7] が実装した Double-double(DD) 精度の SIMD 化が既にあり，AVX2(Advanced Vector eXtensions 2)[3] を使用することで，疎行列・ベクトル乗算に対して 3 倍程度の高速化が達成でき，MATLAB でも有用であることが示されている [9] が，3 倍々精度 (Triple-double, TD)，4 倍々精度 (Quadruple-double, QD) 向けの SIMD 化は広く公開されたものが存在していない．

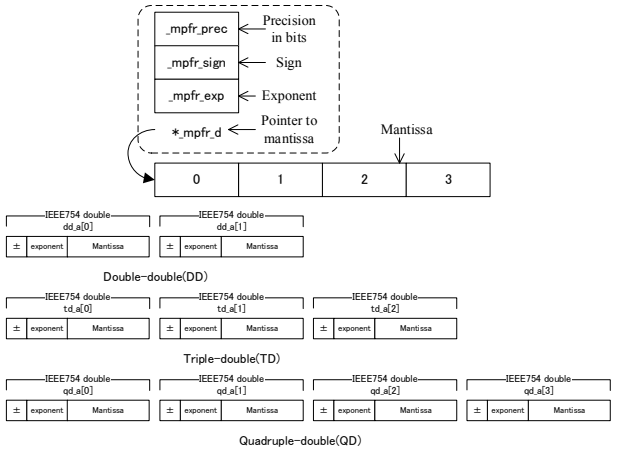

Fig. 2: MPFR, DD, TD, QD のデータ型

DD 精度演算については先行研究 [7] 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，今回我々は正規化処理部分は AVX2 のデータ型 `_m256d` を 4 つの `double` 型変数を格納した配列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．

以下，基本線形計算で使用する DD, TD, QD 精度の加算・乗算の SIMD 化したアルゴリズムとパフォーマンスについて触れる．本論文で示すベンチマークテストはすべて下記の Corei9 環境で行ったものである．

Intel Core i9-10900X (3.6GHz, 10 cores), 16GB RAM Ubuntu 18.04.2, GCC 7.3.0

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．

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)
```

FMA を利用する TwoProd 関数は Algorithm 3 のように 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 (Algorithm 4) と AVX2DDmul (Algorithm 5) として実装した．

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 (Algorithm 6) 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 (Algorithm 7) を使用した．

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

Algorithm 4 $r[2] := \mathrm{AVX2DDadd}(x[2], y[2])$

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

Algorithm 5 $r[2] := \mathrm{AVX2DDmul}(x[2], y[2])$

$(p_1, p_2) := \mathrm{AVX2TwoProd-FMA}(x[0], y[0])$
$w_1 :=$ `_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)
$(r[0], r[1]) := \mathrm{AVX2QuickTwoSum}(p_1, p_2)$

Algorithm 6 $r[3] := \mathrm{AVX2TDadd}(x[3],\ y[3])$

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

Algorithm 7 $r[3] := \mathrm{AVX2TDaddq}(x[3], y[3])$

$s_0 :=$ `_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`$(w_2,\ u_2)$
$(s_1, t_0) :=\mathrm{AVX2TwoSum}(s_1,\ t_0)$
$(s_2, t_0, t_1) :=\mathrm{AVX2ThreeSum}(s_2,\ t_0,\ t_1)$
$t_0 :=$ `_mm256_add_pd`$($`_mm256_add_pd`$(t_0,\ t_1),\ t_2)$
$(r[0], r[1], r[2]) :=\mathrm{AVX2Renorm3}(s_0,\ s_1,\ s_2,\ t_0)$
return $(r[0], r[1], r[2])$

実装し，VSEB 関数以外を SIMD 化した AVX2TDmul 関数を実装した．

Algorithm 8 $r[3] := \mathrm{AVX2TDmul}(x[3],\ y[3])$

$(z_{00}^{\mathrm{up}}, z_{00}^{\mathrm{lo}}) := \mathrm{AVX2TwoProd-FMA}(x[0], y[0])$
$(z_{01}^{\mathrm{up}}, z_{01}^{\mathrm{lo}}) := \mathrm{AVX2TwoProd-FMA}(x[0], y[1])$
$(z_{10}^{\mathrm{up}}, z_{10}^{\mathrm{lo}}) := \mathrm{AVX2TwoProd-FMA}(x[1], y[0])$
$(b_0, b_1, b_2) := \mathrm{AVX2VecSum}(z_{00}^{\mathrm{lo}}, z_{01}^{\mathrm{up}}, z_{10}^{\mathrm{up}})$
$c :=$ `_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(`$c,\ z_3$`)`
$(e_0, e_1, e_2, e_3) := \mathrm{AVX2VecSum}(z_{00}^{\mathrm{up}}, b_0, b_1, s_3)$
$r[0] := e_0$
$(r[1], r[2]) := \mathrm{AVX2VSEB(2)}(e_1, e_2, e_3)$
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，一部 AVX2 化した Renorm 関数を用いて AVX2-QDadd(Algorithm 11) と AVX2QDmul(Algorithm 12) を実装した．

Algorithm 9 $(a, b, c) := \mathrm{AVX2ThreeSum}(x, y, z)$

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

これらの QD 演算では必要となる，3 つの浮動小数点数 x, y, z の加算を TwoSum を用いて行い，その誤差も得られる ThreeSum も AVX2 化したものを使用する。3 つの返り値 $a \approx x + y + z$, b, c を返す ThreeSum(Algorithm 9) と，2 つの返り値 $a \approx x + y + z$, b を返す ThreeSum2(Algorithm 10) の 2 つを QD 演算で使用する。

Algorithm 10 $(a, b) := \mathrm{AVX2ThreeSum2}(x, y, z)$

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

無誤差変換技法と，2 種の ThreeSum を使用して実装した AVX2 化した加算 (AVX2QDadd) を Algorithm 11 に示す。

同様に，AVX2 化した乗算 (AVX2QDmul) を Algorithm 12 に示す。

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 $r[4] := \mathrm{AVX2QDadd}(x[4], y[4])$

$s_0 :=$ `_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)
$(s_1, t_0) := \mathrm{AVX2TwoSum}(s_1, t_0)$
$(s_2, t_0, t_1) := \mathrm{AVX2ThreeSum}(s_2, t_0, t_1)$
$(s_3, t_0) := \mathrm{AVX2ThreeSum2}(s_3, t_0, t_2)$
$t_0 :=$ `_mm256_add_pd`$($`_mm256_add_pd`$(t_0, t_1), t_3)$
$(r[0], r[1], r[2], r[3]) := \mathrm{AVX2Renorm}(s_0, s_1, s_2, s_3, t_0)$
return $(r[0], r[1], r[2], r[3])$

Algorithm 12 $r[4] := \mathrm{AVX2QDmul}(x[4], y[4])$

$s_0 :=$ `_mm256_add_pd`$(x[0],\ y[0])$
$(p_0, q_0) := \mathrm{AVX2TwoProd}(x[0], y[0])$
$(p_1, q_1) := \mathrm{AVX2TwoProd}(x[0], y[1])$
$(p_2, q_2) := \mathrm{AVX2TwoProd}(x[1], y[0])$
$(p_3, q_3) := \mathrm{AVX2TwoProd}(x[0], y[2])$
$(p_4, q_4) := \mathrm{AVX2TwoProd}(x[1], y[1])$
$(p_5, q_5) := \mathrm{AVX2TwoProd}(x[2], y[0])$
$(p_1, p_2, q_0) := \mathrm{AVX2ThreeSum}(p_1, p_2, q_0)$
$(p_2, q_1, q_2) := \mathrm{AVX2ThreeSum}(p_2, q_1, q_2)$
$(p_3, p_4, p_5) := \mathrm{AVX2ThreeSum}(p_3, p_4, p_5)$
$(s_0, t_0) := \mathrm{AVX2TwoSum}(p_2, p_3)$
$(s_1, t_1) := \mathrm{AVX2TwoSum}(q_1, p_4)$
$s2 :=$ `_mm256_add_pd`(q_2, p_5)
$(s_1, t_0) := \mathrm{AVX2TwoSum}(s_1, t_0)$
$s2 :=$ `_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)
$(r[0], r[1], r[2], r[3]) := \mathrm{AVX2Renorm}(p_0, p_1, s_0, s_1, s_2)$
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`_m256d` 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`_m256d` 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`_m256d` 型データの連続呼び出し (Load/Store 関数) が使えず，binary64(IEEE 2 進倍精度浮動小数点数型) 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 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，Fig.3 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 Fig.4，Fig.5，Fig.6 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,Fig.3 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, Fig.3 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．

Fig.4 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．

Fig.5 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．

Fig.6 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

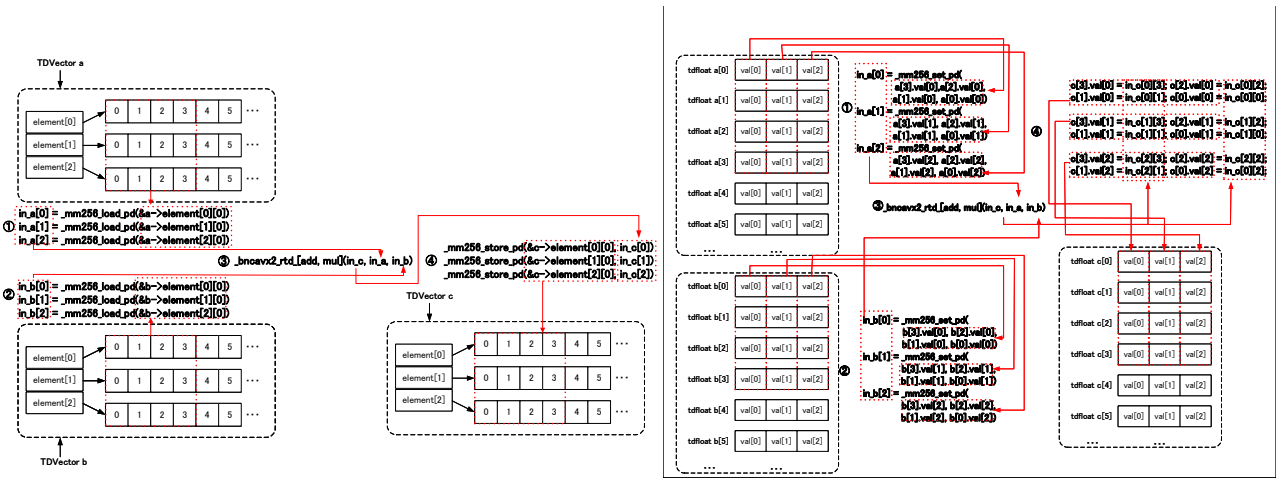

Fig. 3: Load/Store 命令で呼び出せる TD ベクトルデータ型 (左), set 命令のみ利用する TD 配列 (右)

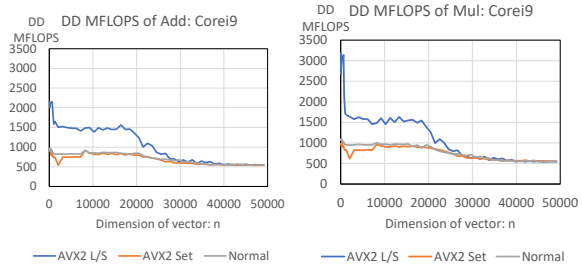

Fig. 4: DD 精度ベクトル要素の加算と乗算の DD MFLOPS の推移

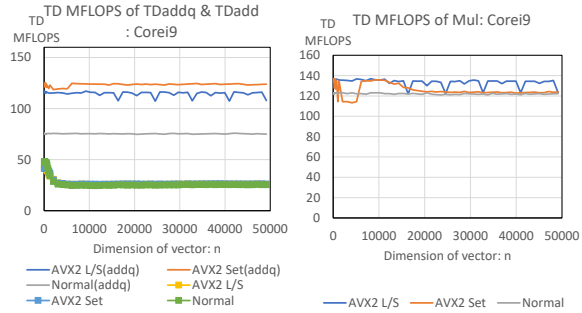

Fig. 5: TD 精度ベクトル要素の加算と乗算の TD MFLOPS の推移

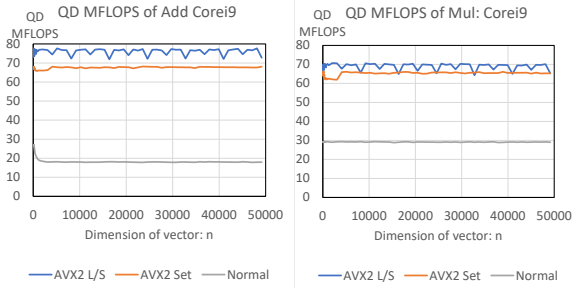

Fig. 6: QD 精度ベクトル要素の加算と乗算の QD MFLOPS の推移

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．ベクトル演算だけでなく，行列・ベクトル乗算や行列乗算においても，この形式を用いることで高速化できることが予想される．

3. Python 環境における多倍長精度線形計算の実装

我々は既に Python プログラミング環境においても，QD 精度以下であれば gmpy2 が提供する MPFR より高速な計算が可能であることを示した [12]．しかしながら，これは単純行列乗算アルゴリズムだけの実装であり，キャッシュメモリのヒット率の向上や演算量を減らすブロック化アルゴリズムや Strassen アルゴリズムを使って更に高速化した行列乗算に比べると圧倒的に低速である．

そこで，今回我々は AVX2 を用いて高速化を行った BNCmatul ライブラリを使用し，これらのアルゴリズムと計算時間の比較を Corei9 環境で行った．ソフトウェア構成は Fig.7 に示した通りである．

我々の DD, TD, QD 精度基本演算関数は c_dd_qd.h に C の inline 関数として実装されたものがベースになっており，これを rdd.h から MPFR 関数と同様の引数順に修正したマクロを介して呼び出して使用する．これを rdd.c に C の静的関数として定義したものを DLL(Dynamic Linking Library) にした

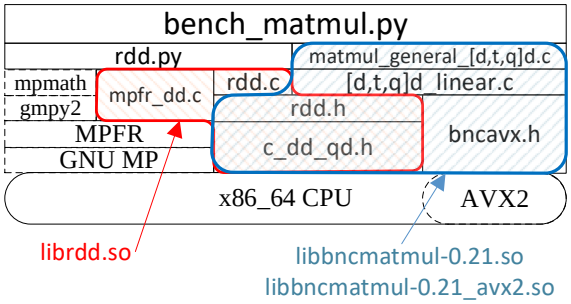

Fig. 7: Python 環境における多倍長精度線形計算のソフトウェア階層

ものが librdd.so であり，Python のクラスはこの DLL 内の関数を呼び出して実装されている．我々は既に，この Python クラスの性能は mpmath 並で，gmpy2 の MPFR に比べて著しく低速であり，行列乗算用の C 関数を rdd.c で定義して使用した方が高速であることを示している [12]．

BNCmatmul も，rdd.h を用いて実装された行列乗算ライブラリであるが，前述したようにブロック化アルゴリズムと Strassen アルゴリズムを実装しており，これを使用するだけで単純行列乗算より高速になることは既に示している [11]．今回 AVX2 を用いて実装したのは，DD, TD, QD 精度の基本線形計算で，ブロック化，Strassen アルゴリズムの実装そのものは全く変更しておらず，AVX2 の高速化の恩恵は同じ関数を使用しながら受けることができる．

今回新たに BNCmatmul の行列，ベクトル演算を使用するために定義したデータ型と関数の定義は下記に示すようなものである．これは TD 精度のベクトル (tdvector) と行列 (tdmatrix) であるが，DD 精度，QD 精度のベクトルや行列も同様の定義を行っている．

Listing 1: TDVector と TDMatrix データ型と DLL 関数の定義

```
# TDVector class
# TD vector
#typedef struct {
# long int dim; // dim <= real_dim
# long int real_dim; // multiplier of
    _BNC_D_WIDTH
# double *element[TDSIZE];
#} tdvector;
#typedef tdvector *TDVector;
class tdvector(ct.Structure):
  _fields_ = [
    ("dim", ct.c_long),
    ("real_dim", ct.c_long),
    ("element", ct.ARRAY(ct.POINTER(ct.c_double),
        3))
  ]

# init_tdvector
init_tdvector = libbncmatmul.init_tdvector
init_tdvector.argtype = [ct.c_long]
init_tdvector.restype = ct.POINTER(tdvector)

# free_tdvector
free_tdvector = libbncmatmul.free_tdvector
free_tdvector.argtype = ct.POINTER(tdvector)
free_tdvector.restype = None

# dd array to tdvector
# TDVector vec -> tdfloat array
# void set_tdfloat_tdvec(tdfloat ret[], int
    ret_dim, TDVector vec);
set_tdfloat_tdvec = libbncmatmul.
    set_tdfloat_tdvec
set_tdfloat_tdvec.argtype = [ct.POINTER(ct.
    c_double), ct.c_int, ct.POINTER(tdvector)]
set_tdfloat_tdvec.restype = None

# tdfloat array -> TDVector ret
# void set_tdvector_tdfloat(TDVector ret, tdfloat
     array[], int array_dim);
set_tdvector_tdfloat = libbncmatmul.
    set_tdvector_tdfloat
set_tdvector_tdfloat.argtype = [ct.POINTER(
    tdvector), ct.POINTER(ct.c_double), ct.c_int]
set_tdvector_tdfloat.restype = None

# TDMatrix class
# TD matrix
#typedef struct{
# long int row_dim, col_dim;
# long int real_row_dim, real_col_dim; //
    multiplier of _BNC_D_WIDTH
# double *element[TDSIZE];
#} tdmatrix;
class tdmatrix(ct.Structure):
  _fields_ = [
    ("row_dim", ct.c_long),
    ("col_dim", ct.c_long),
    ("real_row_dim", ct.c_long),
    ("real_col_dim", ct.c_long),
    ("element", ct.ARRAY(ct.POINTER(ct.c_double),
        3))
  ]

# init_tdmatrix
init_tdmatrix = libbncmatmul.init_tdmatrix
init_tdmatrix.argtype = [ct.c_long, ct.c_long]
init_tdmatrix.restype = ct.POINTER(tdmatrix)

# free_tdmatrix
free_tdmatrix = libbncmatmul.free_tdmatrix
free_tdmatrix.argtype = ct.POINTER(tdmatrix)
free_tdmatrix.restype = None

# td array to tdvector
# TDMatrix mat -> tdfloat array
# void set_tdfloat_tdmat(tdfloat ret[], int
    ret_dim, TDMatrix mat);
set_tdfloat_tdmat = libbncmatmul.
    set_tdfloat_tdmat
set_tdfloat_tdmat.argtype = [ct.POINTER(ct.
    c_double), ct.c_int, ct.POINTER(tdmatrix)]
set_tdfloat_tdmat.restype = None

# tdfloat array -> TDmatrix ret
# void set_tdmatrix_tdfloat(DDMatrix ret, tdfloat
     array[], int array_dim);
set_tdmatrix_tdfloat = libbncmatmul.
    set_tdmatrix_tdfloat
set_tdmatrix_tdfloat.argtype = [ct.POINTER(
    tdmatrix), ct.POINTER(ct.c_double), ct.c_int]
set_tdmatrix_tdfloat.restype = None
```

以上で定義した BNCmatmul(Fig.7) の関数は，DLL 化した libbncmatmul-0.21.so と AVX2 による高速化した libbncmatmul-0.21_avx2.so，どちらを呼び出しても同じ形式で使用できる．

4. 行列ベクトル乗算と行列乗算の性能評価

以上の準備を行って用意した RDD.py ベースの行列乗算 (MM, RDD)，rdd.c に施した C による実装 (MM, C_PTR) の計算時間を Table 1 に示す．今回はこれに加え，行列・ベクトル乗算 (MV, RDD) と C による実装 (MV, C_PTR) も併せて示す．

使用した $n \times n$ 実行列 A，B と n 次実ベクトル $\mathbf{x}$ は次の通りである．

$$A = \left[\sqrt{2}\,(i+j-1)\right]_{i,j=1}^{n},\ B = \left[\sqrt{3}i\right]_{i,j=1}^{n},\ \mathbf{x} = \left[\sqrt{5}i\right]_{i=1}^{n}$$

Table 1: Python コードによる実装比較 (単位:秒)

$n = 128$	MM,RDD	MM,C_PTR	(R/C)	MV, RDD	MV,C_PTR	(R/C)
DD	4.00	0.015	(266.7)	0.0305	0.000139	(219.4)
TD	4.55	0.073	(62.3)	0.0372	0.000577	(64.5)
QD	5.15	0.206	(25.0)	0.0386	0.00165	(23.4)
MP(212)	3.11			N/A		
MPFR(212)	0.89			N/A		
$n = 1024$						
DD	2216.6	15.5	(142.7)	2.03	0.008	(267.1)
TD	2582.0	75.6	(34.2)	2.33	0.035	(66.6)
QD	2865.0	164.3	(17.4)	2.57	0.101	(25.4)
MP(212)	2042.5			N/A		
MPFR(212)	668.5			N/A		

これらを用いて，行列・ベクトル乗算 $\mathbf{b} := A\mathbf{x}$ と，行列乗算 $C := AB$ を計算した．

既に示したとおり，Corei9 環境でも MPFR(212 bits) 計算に比べ，C による QD 実装は約 4 倍高速である．rdd.c による C 実装は DD 精度で 143〜267 倍，TD 演算で 34〜67 倍，QD 演算で 17〜25 倍高速である．従って，AVX2 による高速化した行列乗算，行列ベクトル乗算は，rdd.c による C 実装と比較して行うことにする．

AVX2 を用いて高速化した行列乗算，行列・ベクトル乗算がどの程度高速になったかを，MM, C_PTR の計算時間との比，すなわち高速化率で示した図を Fig.8 と Fig.9 に示す．

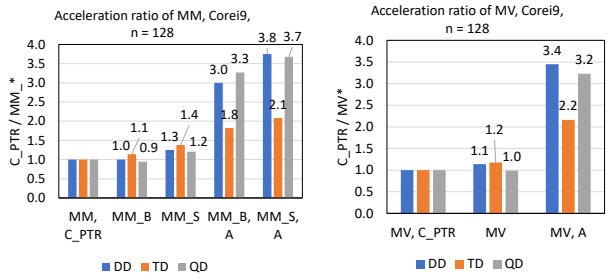

Fig. 8: C による単純実装と比較しての AVX2 利用による高速化率 ($n = 128$): 行列乗算 (MM, 左図)，行列・ベクトル乗算 (MV, 右図)，

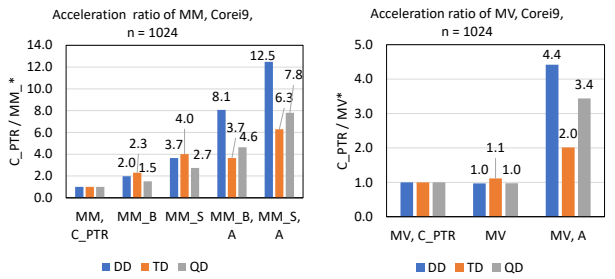

Fig. 9: C による単純実装と比較しての AVX2 利用による高速化率 ($n = 1024$): 行列乗算 (MM, 左図)，行列・ベクトル乗算 (MV, 右図)，

DD 精度で行列乗算は最大 12.5 倍，QD 精度で最大 7.8 倍の高速化を実現できたことが分かる．Strassen アルゴリズムは分割統治法に基づいて演算量を減らしたものであり，その効果も相まって高速化率が高くなっている．

反面，TD 演算は Fig.5 で分かるように AVX2 でも乗算の高速化はあまり行えず，そのせいもあって行列乗算，行列ベクトル乗算どちらにしても AVX2 による高速化率は低く，最大 6.3 倍に留まっている．

5. まとめと今後の展開

AVX2 による高速化の結果，DD, TD, QD 精度行列乗算，行列ベクトル乗算いずれも高速化でき，Python プログラミング環境でも十分性能を発揮できていることが示された．

今後の課題としては，OpenMP によるマルチコア向けの並列化を行ってさらなる高速化を行うとともに，直接法 (LU 分解)，Hessenberg リダクション，QR 分解等，広く利用されている各種線形計算アルゴリズムに対して高性能化を行い，実用的な多倍長精度線形計算環境を整備していく．

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) Python Software Foundation, `https://python.org/`.
2) Julia, `https://julialang.org/`.
3) Intel Corp. The intel intrinsics guide. `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`.
4) T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, Vol. 18, No. 3, pp. 224–242, Jun 1971.
5) Tomonori Kouya. BNCpack. `https://na-inet.jp/na/bnc/`.
6) Tomonori Kouya. A Tutorial of BNCpack. `https://na-inet.jp/tutorial/`.
7) T.Kotakemori, S. Fujii, H. Hasegawa, and A. Nishida. Lis: Library of iterative solvers for linear systems. `https://www.ssisc.org/lis/`.

8) T.Kouya. BNCmatmul. http://na-inet.jp/na/bnc/bncmatmul-0.2.tar.bz2.

9) 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.

10) 幸谷智紀. 多倍長精度数値計算. 森北出版, 2019.

11) 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.

12) 幸谷智紀. Python プログラミング環境における多倍長精度数値計算について. 静岡理工科大学紀要, Vol. 28, pp. 23–31, 2020.