Vol.2015-HPC-151 No.15
2015/10/1

京・FX10における倍々精度演算の高速化

佐々木 信一[1,a)]　菱沼 利彰[2]　藤井 昭宏[1,b)]　田中 輝雄[1,c)]　椋木 大地[3]　今村 俊幸[3]

概要：倍々精度演算は Bailey らが考案した”Double-Double”精度のアルゴリズム [1] を用いて，倍精度演算の組合せにより四倍精度演算を行う手法である．倍々精度乗算は FMA（積和）演算器を搭載するプロセッサでは積和演算の中間結果を高精度で保持できアルゴリズム的に演算量を削減できる（FMA アルゴリズム）．本研究ではスーパコンピュータ京 [6] および FX10 に搭載されている FMA 演算器と SIMD（Single Instruction Multiple Data）演算機構を用いて，これらに最適なベクトル演算および疎行列ベクトル積（疎行列部は倍精度）の高速化手法を提案する．実験の結果，ベクトル演算ではキャッシュ容量に収まる範囲では本手法適用前に比べ約 5 倍，収まらない範囲ではメモリ性能に制約を受けるまで高速化できること，疎行列ベクトル積では高速化手法によりフロリダ行列に対しては最大 4.5 倍の高速化したことを確認した．

1. はじめに

物理シミュレーションの核となる反復解法は丸め誤差によって収束が停滞あるいは発散することがある．これらの解決には高精度演算が有効であることが確認されている [2]．高精度演算は演算時間が多くかかる．Bailey らが提案した倍精度変数 2 つを組合せて 1 つの四倍精度変数の値を保持する倍々精度演算と呼ばれる高速な高精度演算手法がある．倍精度演算の SIMD 命令がサポートされているアーキテクチャであれば倍々精度演算においても SIMD 命令が利用できる．

一方，大規模物理シミュレーションにおいて京のようなスーパコンピュータが用いられる．京は浮動小数点レジスタが 128bit あることや FMA 演算器が搭載されている等の特徴がある．ソフトウェアの性能をハードウェアの理論性能に近づけるためには，それらの機構を最大限有効活用する必要がある．

我々は反復解法の核となるベクトル演算や疎行列ベクトル積 (SpMV : Sparse matrix and vector product) を対象にスーパコンピュータ京や FX10 上での倍々精度演算の高速化手法を提案する．

2. 倍々精度演算（DD 演算）

倍々精度演算とは，Bailey が提案した“Double-Double”精度のアルゴリズム [1] を用いて，倍精度変数を 2 つ組合せて四倍精度演算を実装する手法である．

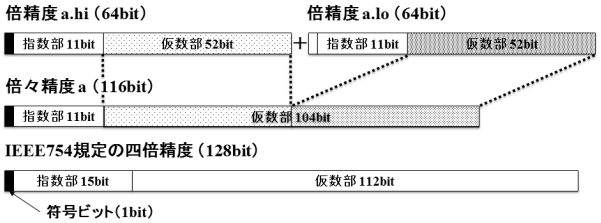

図 1　倍々精度のビット数

“Double-Double”精度のアルゴリズムでは Knuth が示した丸め誤差のない倍精度加算のアルゴリズム [3] と Dekker が示した丸め誤差のない倍精度乗算のアルゴリズム [4] が用いられており，倍精度の加算と乗算の組合せのみで実装できるため SIMD 命令を用いて高速化が可能である．

倍々精度変数と IEEE754 規定の四倍精度変数のデータ構造を**図 1** に示す．倍々精度変数 a を構成する上位 a.hi と下位 a.lo はそれぞれ倍精度でデータを保持する．倍々精度の仮数部は 52×2=104bit であり，指数部は 11bit のままである．一方，IEEE754 規定の四倍精度変数の仮数部は 112bit，指数部は 15bit であるため，倍々精度は IEEE754 規定の四倍精度に対して仮数部は 8bit，指数部は 4bit 少ない．倍々精度演算は，精度は劣るが IEEE754 規定の四倍精度よりも高速に実行できる [5]．

倍々精度加算の擬似コードを**図 2** に，倍々精度乗算の擬似コードを**図 3** に示す．また，FMA 演算に対応しているアーキテクチャでは FMA 命令を用いることで積和演算の中間結果を高精度で保持できるため，**図 4** に示す疑似コードのように書き換え演算量を削減することができる．

[1] 工学院大学
[2] 筑波大学
[3] 国立研究開発法人理化学研究所 計算科学研究機構
[a)] em14009@ns.kogakuin.ac.jp
[b)] fujii@cc.kogakuin.ac.jp
[c)] teru@cc.kogakuin.ac.jp

```
DD_ADD(a, b, c)
{ //a = b + c
  sh = b.hi + c.hi;
  th = sh - b.hi;
  tl = sh - th;
  th = c.hi - th;
  tl = b.hi - tl;
  eh = tl + th;
  eh = eh + b.lo;
  eh = eh + c.lo;
  a.hi = sh + eh;
  a.lo = a.hi - sh;
  a.lo = eh - a.lo;
}
```

図 2　倍々精度加算

```
DD_MUL(a, b, c)
{ //a = b * c
  sp = 134217729.0;
 p1 = b.hi * c.hi;
 tq = sp * b.hi;
 bh = tq - (tq - b.hi);
 bl = b.hi -  bh;
 tq = sp * c.hi;
 ch = tq - (tq - c.hi);
 cl = c.hi - ch;
 p2 = bh * ch - p1;
 p2 = bh * cl + p2;
 p2 = bl * ch + p2;
 p2 = bl * cl + p2;
 p2 = b.hi * c.lo + p2;
 p2 = b.lo * c.hi + p2;
 a.hi = p1 + p2;
 a.lo = p2 - (a.hi - p1);
}
```

図 3　倍々精度乗算 (noFMA)

```
DD_MUL(a, b, c)
{ //a = b * c
 p1 = b.hi * c.hi;
 p2 = b.hi * c.hi - p1;
 p2 = b.hi * c.lo + p2;
 p2 = b.lo * c.hi + p2;
 a.hi = p1 + p2;
 a.lo = p2 - (a.hi - p1);
}
```

図 4　倍々精度乗算（FMA）

3. 京・FX10 における DD 演算の高速化手法

3.1 京・FX10 のアーキテクチャ構造の利用

スーパコンピュータ京や FX10 のノードプロセッサに関して，以下の点に着目してその効率的な利用方法を述べる．

- FMA 演算器を 2 個搭載
- 256 本の浮動小数点レジスタ
- HPC-ACE

図 5 は FX10 のノードプロセッサのブロック図である．図の FLA と FLB が浮動小数点演算器であり，それぞれ独立に FMA 演算および加算と乗算を行える．そのため，命令に依存関係がなければ，1step で 4 つの倍精度演算が行える（FMA 命令 1 回で 2 つの演算としてカウント）．加えて，128bit の浮動小数点レジスタが 256 本ある．これにより，専用の HPC-ACE[9] と呼ばれる SIMD 組み込み関数

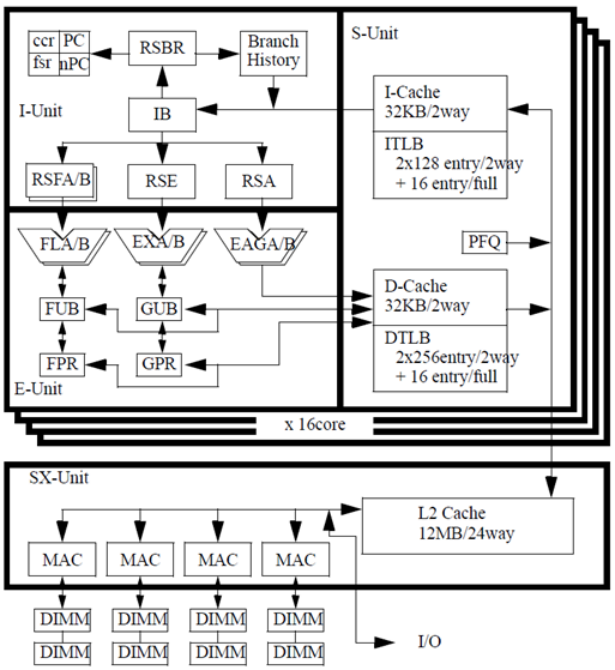

図 5　FX10 のブロック図 出典:SPARC64^{TM} IXfx Extensions[7]

を用いることで，1 命令あたり 2 回の倍精度演算を同時に行うことができる．これらの構成は京も同様である．

従来の FMA 演算器のないアーキテクチャ向けのアルゴリズムでの倍々精度乗算は，図 3 で示すように，倍精度の加算 15 回と乗算 9 回で構成されており命令の依存関係から，京・FX10 なら 1 回あたり 15step 必要であった．積和演算の中間結果を高精度で保持できる FMA 演算を用いると，丸め誤差のなく乗算の結果を加算に利用できる．この性質を利用すると，倍々精度乗算は倍精度加算命令 3 回と乗算命令 1 回，FMA 命令 3 回で構成されるアルゴリズムに変更でき，1 回あたり 7step にできる．この変更後のアルゴリズムを以降，FMA アルゴリズムと呼ぶ．これらより，倍々精度乗算部分は FMA アルゴリズムを用いると，step 数が 15 から 7 へと削減でき，約 2.1 倍（=15/7）の高速化が期待できる．

なお，倍々精度加算は依存関係のある 11 回の倍精度加算のみで構成されているため，FMA 演算器を用いてもアルゴリズムは変わず，1 命令あたり 11step である．

また，倍々精度加算および乗算はすべて倍精度乗加算で構成されているため，SIMD 命令を用いれば約 2 倍の高速化が期待できる．

FMA 演算器が 2 個あるため，効率的に演算器を動かすためには，FLA と FLB に常に命令が割り当てられるようにする必要があり，解決方法として最適化手法の一つループアンロールが挙げられる．たとえば，x86 のように浮動小数点レジスタが 16 本程のアーキテクチャでは，倍々精度演算は 1 命令あたりの step 数が長いため 1 命令だけでレジスタを使い切り，メモリへの一時退避のための move

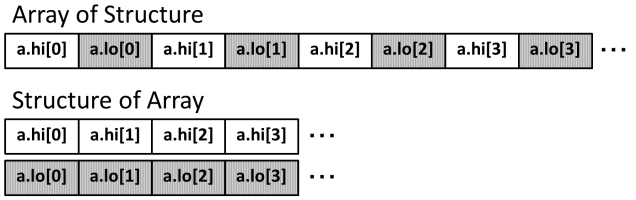

図 6 倍々精度データ構造

命令によりループアンローリングの効果は低い．しかし，FX10 には浮動小数点レジスタが 256 本あるため，この問題を解消し効果が期待できる．

3.2 データ構造

倍々精度演算を SIMD を用いて高速化する際に，メモリを効率よく使うために倍々精度変数のデータ構造を考慮する必要がある．**図 6** に示すように，構造体配列（AoS : Array of Structure）と配列構造体 (SoA : Structure of Array) の 2 種の実装方法が考えられる．一般に SIMD 化を前提としている場合には，SoA が適している [11].

特に，倍々精度演算の場合には，擬似コードで示したように倍々精度変数 A と B で計算を行う際に必ず A.hi と B.lo や A.lo と B.hi という組合せで乗加算が発生する．AoS で実装し 128bit の浮動小数点レジスタ環境で SIMD 化を行うと，shuffle 命令と呼ばれるレジスタ内の上位ビットと下位ビットを入れ替えて計算する命令が発生し計算速度の低下につながる．SoA では shuffle 命令は発生しない．

本研究では，SoA で実装された倍々精度演算をサポートしている反復解法ライブラリ Lis[10] をベースに京や FX10 向けの最適化を検証する．なお，比較対象には AoS で倍々精度演算を実装した京や FX10 の標準ライブラリ “高速四倍精度基本演算ライブラリ fast_dd[8]” を用いる．

4. 数値実験

4.1 実験環境

実験はスーパコンピュータ京および FX10 の 1 ノード上で行った．

京の環境は SPARC64^{TM}VIIIfx@2.0 GHz (8 cores, L1 Cache 32KiB, L2 Cache 6MiB)，メモリは DDR3 SDRAM （Bandwidth 64GB/s），FX10 の環境は SPARC64^{TM}IXfx@1.848 GHz (16 cores, L1 Cache 32KiB, L2 Cache 12MiB)，メモリは DDR3 SDRAM (Bandwidth 85GB/s) である．コンパイラは富士通社製 C コンパイラである．最適化は O3，並列化は OpenMP，プリフェッチはコンパイラに任せ，アラインメントは明示的に揃え，最適化時に命令順序を並び替えないようにするため，オプションは “-Kopenmp -Kprefetch_conditional -Kdalign -Knoeval -O3” である．fma を無効にする場合に “-no-fma” をつける．

表 1 実験環境

	K	FX10
Processor	SPARC64^{TM} VIIIfx	IXfx
Frequency	2.0 GHz	1.848GHz
Number of Core	8	16
Number of Register	256	256
L1 Cache per core	32KB	32KB
L2 Cache	6MB	12MB
Memory	DDR3 SDRAM	
Memory Size	16GB	32GB
Memory Bandwidth	64GB/s	85GB/s
Compiler	Fujitsu Compiler (fccpx)	
Vectorization	HPC-ACE	
Options	-Kopenmp -Kprefetch_conditional -Kdalign -Knoeval -O3	
Options (nofma)	-Kopenmp -Kprefetch_conditional -Kdalign -Knoeval -O3 -no-fma	

表 2 京・FX10 における倍々精度ベクトル演算

Name	Operation	Load	Store	step nofma (fma)
axpy	$\boldsymbol{y} = \alpha\boldsymbol{x} + \boldsymbol{y}$	2	1	26 (18)
axpyz	$\boldsymbol{z} = \alpha\boldsymbol{x} + \boldsymbol{y}$	2	1	26 (18)
xpay	$\boldsymbol{y} = \boldsymbol{x} + \alpha\boldsymbol{y}$	2	1	26 (18)
scale	$\boldsymbol{x} = \alpha\boldsymbol{x}$	1	1	15 (7)
dot	$val = \boldsymbol{x} \cdot \boldsymbol{y}$	2	0	26 (18)
nrm2	$val = \|\|\boldsymbol{x}\|\|_2$	1	0	22 (16)

4.2 倍々精度ベクトル演算

実験はスーパコンピュータ京の 1 ノード上で OpenMP を用いて 8 スレッドで行った．

$\boldsymbol{x}$，$\boldsymbol{y}$，$\boldsymbol{z}$ をベクトル値，α，val をスカラ値として**表 2** に示す 6 種類の演算を倍々精度演算で実装した．

SIMD 演算の有無と FMA アルゴリズムの有無の 4 つの組合せの演算時間を比較し，L2 キャッシュに収まるデータ量の結果を**表 3** に，収まらない場合の結果を**表 4** に示す．表の lis が従来の Lis，lisbased が lis をベース高速化したもの，fastdd が高速四倍精度基本演算ライブラリ fast_dd である．scalar は逐次演算，simd は SIMD 演算，nofma は従来のアルゴリズム，fma は FMA アルゴリズムによる実装を示しており，各種法の対応状況を**表 5** に示す．括弧内の数値は lis_scalar_nofma が基準の相対性能である．

scale を例に取ると，FMA アルゴリズムを用いると演算が 15step から 7step へと減少するため約 2.1 倍の高速化が期待できる．実測値が 2.17 倍と予想を上回った理由は，おそらくアルゴリズム変更に伴いメモリレイテンシとスループットが改善されたためである．

SIMD の効果について 2 倍の高速化が期待できるが，実測値は 2.64 倍であった．また，京の標準ライブラリの実測値の fastdd_simd が fastdd_scalar より 2.8 倍速いことよりほぼ同等の効果であった．なお，fast_dd よりも Lis ベース

表 3　L2 キャッシュに収まるデータサイズでの演算時間 (データサイズ 6MiB)

	Time [ms] (speed up ratio)					
	①lis_scalar_nofma	②lis_scalar_fma	③lisbased_simd_nofma	④lisbased_simd_fma	⑤fastdd_scalar	⑥fastdd_simd
scale	0.51 (1.00)	0.24 (2.11)	0.19 (2.62)	0.09 (5.98)	0.96 (0.53)	0.34 (1.48)
axpy	0.48 (1.00)	0.32 (1.52)	0.18 (2.75)	0.11 (4.61)	1.19 (0.41)	0.43 (1.14)
xpay	0.48 (1.00)	0.32 (1.50)	0.18 (2.70)	0.10 (4.64)	1.12 (0.43)	0.42 (1.13)
axpyz	0.48 (1.00)	0.32 (1.51)	0.17 (2.77)	0.10 (4.71)	1.11 (0.43)	0.42 (1.13)
dot	0.95 (1.00)	0.66 (1.44)	0.37 (2.59)	0.22 (4.29)	2.17 (0.44)	0.86 (1.11)
nrm2	0.84 (1.00)	0.65 (1.28)	0.31 (2.70)	0.18 (4.63)	2.06 (0.41)	0.80 (1.04)

表 4　L2 キャッシュから溢れるデータサイズでの演算時間 (データサイズ 1GB)

	Time [s] (speed up ratio)					
	①lis_scalar_nofma	②lis_scalar_fma	③lisbased_simd_nofma	④lisbased_simd_fma	⑤fastdd_scalar	⑥fastdd_simd
scale	0.08 (1.00)	0.04 (2.17)	0.03 (2.64)	0.03 (3.06)	0.15 (0.53)	0.05 (1.49)
axpy	0.10 (1.00)	0.07 (1.51)	0.04 (2.74)	0.03 (3.79)	0.24 (0.42)	0.09 (1.13)
xpay	0.10 (1.00)	0.07 (1.50)	0.04 (2.74)	0.03 (3.73)	0.23 (0.42)	0.09 (1.13)
axpyz	0.10 (1.00)	0.07 (1.51)	0.04 (2.71)	0.03 (3.69)	0.23 (0.42)	0.09 (1.12)
dot	0.15 (1.00)	0.10 (1.45)	0.06 (2.60)	0.03 (4.37)	0.34 (0.44)	0.13 (1.12)
nrm2	0.13 (1.00)	0.10 (1.34)	0.05 (2.70)	0.04 (3.70)	0.33 (0.40)	0.13 (1.04)

の演算が速い理由は，保持する倍々精度変数のデータ配列の形式が異なるためである．

計測結果より FMA アルゴリズムと SIMD を合わせると 5.7（$= 2.17 \times 2.64$）倍の高速化が期待でき，キャッシュ容量に収まる場合は 5.98 倍と順当な値となった．一方，収まらない場合は 3.06 倍であった．これはメモリ帯域幅に制約を受けたためである．

Vector Size 1.0×10^2〜1.0×10^8 の scale 計算時の性能を**図 7** に示す．なお，この性能は倍々精度乗算を 1 回で 1 flop と換算しているため，$performance[GFolps] = VectorSize/time \times 10^{-9}$ としている．データサイズが L2 キャッシュ（6MB）と等しくなる付近 (Vector Size $= 6.0 \times 10^5$) でピークを迎え，12MB 付近 (Vector Size $= 1.2 \times 10^6$) で 1.5GFlops まで急速に低下し，500MB (Vector Size $= 3.0 \times 10^7$) でさらに使用率 1.2GFlops まで低下し以降はその値を維持する．scale だけでなく Store 命令の発生する axpy，xpay，axpyz でも同じ傾向が見られた．dot や nrm2 が L2 キャッシュサイズを超えても倍率に変化が見られないのはメモリ帯域使用率を維持し続けているためである．なお，FX10 においても同様な傾向が得られた．また，キャッシュからあふれるデータサイズではメモリ性能に制約を受けたため，アンロールを行っても性能の向上は見られなかった．

4.3　CRS 形式の倍々精度疎行列ベクトル積

これまでの結果から FMA アルゴリズムおよび SIMD 化の効果が有用であることが確認できたため，SpMV においては FMA アルゴリズムと SIMD 化した上で，アンロール段数を変更して性能の変化を検証していく．また，FX10 上で 16 スレッドを用いて確認する．

表 5　各実装手法の特徴

	SoA	AoS	fma	nofma	simd	scalar
①	✓			✓		✓
②	✓		✓			✓
③	✓			✓	✓	
④	✓		✓		✓	
⑤		✓		✓		✓
⑥		✓		✓	✓	

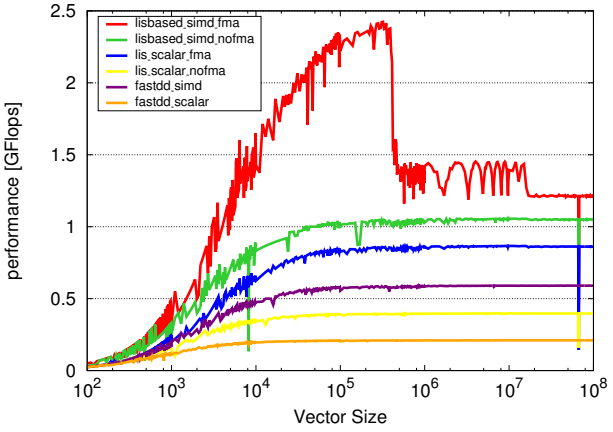

図 7　scale($\boldsymbol{x} = \alpha\boldsymbol{x}$) の性能

疎行列の圧縮方法の一つに非零要素のみを格納する CRS(Compressed Row Storage) 形式 [12] がある．疎行列 A の非零要素数を nnz，行数を row とし，以下の 3 本の配列で構成することによりデータ量を減らしている．

- value：非零要素の値を納める倍精度配列
- index：非零要素の列番号を納める整数配列
- pointer：各行の先頭 index の番号を納める整数配列

value と index の長さは nnz，pointer の長さは row ＋ 1

```
DD_SpMV(A, x, y)
{ //y = A * x
  for(i=0:i<A.row;++i)
  {
    js = A.ptr[i];
    je = A.ptr[i+1];
    vy = _mm_setzero_pd();
    for(j=js;j<je;j+=2)
    {
      va = _mm_load_pd(&A.val[j])
      vx = _mm_set_pd(x[A.index[j+1]] , x[A.index[j]])
      DD_MUL(tmp, va, vx,);
      DD_ADD(vy, vy, tmp,);
    }
    y[i] = redction(vy);
  }
  fraction_padding()
}
```

図 8　疎行列ベクトル積の SIMD コード

である．SpMV において $\boldsymbol{x}$ を参照する場合には，index 配列を参照してから value 配列を間接参照するためキャッシュヒット率は悪い．加えて，疎行列データは一回の疎行列ベクトル積演算で一度しか使われないためキャッシュ再利用率も悪い演算である．これを SIMD 化すると，**図 8** のようなコードになる．$\boldsymbol{x}$ をレジスタへ読み込むときに SET 命令を用いるためランダムアクセスが発生しやすい．

反復解法において，与えられる疎行列データは倍精度であると想定して，倍々精度演算で SpMV を実装するに当たり，倍精度疎行列 A_D と倍々精度ベクトル $\boldsymbol{x}_{DD}$ の積 $\boldsymbol{y}_{DD}$=$\mathrm{A}_D\boldsymbol{x}_{DD}$ を DD-SpMV とした．このとき，倍々精度加算および乗算の FMA アルゴリズムにおいて A に関しては A.hi のみを用いるため，命令数は 17step になる．

SIMD 化すると，行方向に対して 2 つずつデータを処理するため行あたりの非零要素数が 2 の倍数でないときは各行最後の計算で端数を考慮しなければならない．本実装では SET 命令と呼ばれる浮動小数点レジスタの上位ビットと下位ビットに直接倍精度データを格納する命令を用いて，浮動小数点レジスタに要素数が 2 個になるように 0 を格納しており，関数 fraction_processing() として定義する．加えて，レジスタ内の値の総和を $\boldsymbol{y}$ へ足し込む必要があり，各行で水平加算を reduction() と定義する．

分析には行列の構造が単調で評価が行いやすいテスト行列と実問題より作られた The Univ. of Florida Sparse Matrix Collection[13] (フロリダ行列) のうち row が 10^5 以上の実数かつ正方対称の疎行列を計 425 個用いた．

テスト行列 A は

- if(0 ≦ j-i ≦帯幅)A[i][j] = value
- else A[i][j] = 0

を満たす正方の疎行列である．また，実験結果には 500 回反復計算したものの平均を用いた．性能の算出方法は，$performance[Flops] = 2 \times nnz/time$ と定義する．

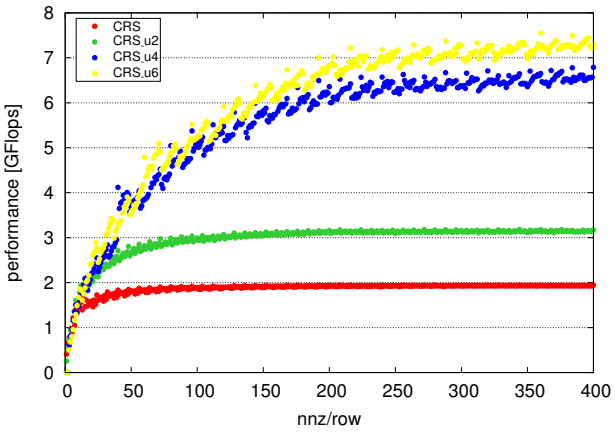

図 9　アンロール段数毎の DD-SpMV の性能 (テスト行列)

図 9 にテスト行列におけるアンロール段数 1,2,4,6 の DD-SpMV の性能を示す．テスト行列の row を 10^5 に固定し，行あたりのに零要素数を 1 から 400 まで変動させて計測した．どのアンローリング段数でも nnz/row の増加に伴い性能が高くなり，一定の性能に達すると飽和する．飽和する点がアンロール段数が増えるにしたがって nnz/row の増加方向にずれていくのは端数処理の影響である．

アンローリングについては図 8 の最内側ループの段数を増加させた．DD-SpMV の核となる倍々精度積和演算 2 回における各 step の FLA/FLB への倍精度命令の Load,Store,Brunch を除いた理想的な演算器への割り当てを**図 10** に示す．アンローリングを行わない場合には図の左のように，倍々精度積和演算を逐次処理するため，命令の依存関係から片方の演算器がほとんど働かない．そのため計 28step 必要となる．一方，2 段アンローリングを行った場合には，FLA と FLB がそれぞれ独立に動作することにより，17step で演算が完了する．これらより，2 段アンローリングを行うことにより，約 1.6 倍の高速化が期待できる．実測値を確認すると，アンローリングを行わない DD-SpMV に対して，2 段アンローリングの DD-SpMV (CRS_u2) の性能は 1.6 倍であり予測通りの結果であった．

更にアンローリング段数を増やしていくと，アンローリング段数が 6 段で最高値となり，以降は性能が低下していった．CRS_u2 に対して CRS_u6 の性能が 3 倍近くまで向上した理由は，ソフトウェアパイプラインによりレイテンシを隠蔽したことに加え，条件分岐が削減できたためだと予想している．なお，8 段以上になると DD-SpMV 内部での fraction_processing() と reduction() の占める割合が無視できなくなるために性能が劣化した．

実問題を想定して計測したフロリダ行列における DD-SpMV の性能を**図 11** に示す．425 個のフロリダ行列を対象に計測し，アンローリングを行わない DD-SpMV (CRS)

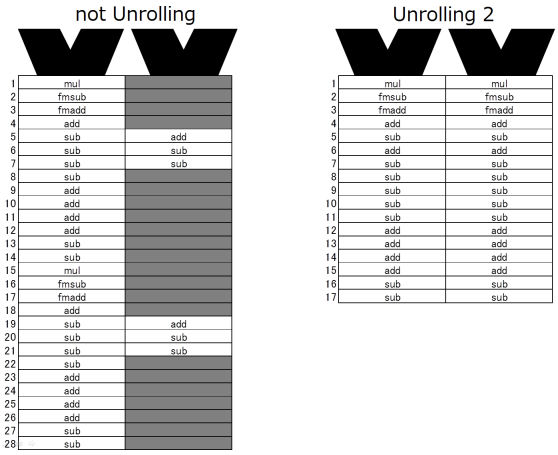

図 10 依存関係による演算器の働き
（左：アンローリングなし　右：2 段アンローリング）

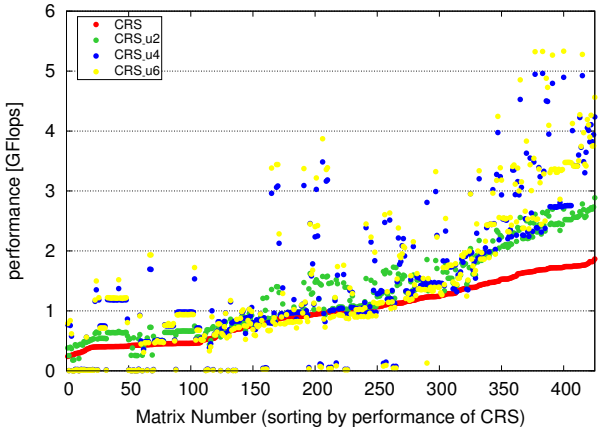

図 11 アンロール段数毎の DD-SpMV の性能 (フロリダ行列)

の性能順にソートしてある．絶対性能でみた場合には多くの場合 CRS_u6 が最も高い性能を示すことが多く，問題によってテスト行列に近い性能を示しているものもある．しかし，アンロール段数を増すことにより，問題の構造によって CRS よりも性能が劣化する場合がある．

図 11 の性能の図を各問題に対して CRS の性能を 1 としたときの相対性能の図に変換したものを図 **12** に示す．CRS の相対性能順にソートしている．最も性能差が開いている問題では CRS_u6 は CRS に対して最大 4.5 倍の性能を示している．一方で問題によっては 0.1 倍以下と性能が著しく劣化する場合がある．このように，段数を増やすことにより性能が劣化する問題として，テスト行列での実測結果と同様に nnz/row が少なく，常に端数処理が発生するような構造の問題という傾向があった．段数を増やすことにより，問題構造への影響が強くなるため一概にアンロール段数を増やせば良いわけではない．しかし，テスト行列のように nnz/row が一定な問題に対しては段数を増やすと性能が向上することからも，問題に応じて使い分けることを視野に入れれば有用である．

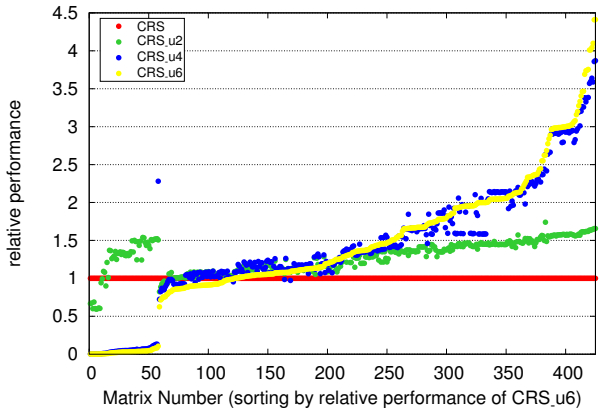

図 12 アンロール段数毎の DD-SpMV の相対性能 (フロリダ行列)

5. まとめ

スーパコンピュータ京・FX10 において FMA アルゴリズムと SIMD を用いて倍々精度演算を実装し，FMA アルゴリズムを利用することにより削減した命令数だけ性能が向上すること（1.2〜2.1 倍），SIMD 化をすることにより約 2 倍の性能向上すること，両手法を適用することで，キャシュ容量に収まる範囲においては 4 倍近い性能向上を確認した．特に，ベクトル演算においては提案手法適用前に比べ，キャシュ容量に収まる範囲では約 5 倍，収まらない範囲ではメモリ性能に制約を受けるまで高速化した．一方で，メモリアクセスが一部ランダムなアクセス（$\boldsymbol{x}$ への参照）になる疎行列ベクトル積においては，京・FX10 はレジスタが非常に多いため最適化手法ループアンローリングが有効であり性能が向上した．テスト要素行列では，6 段アンロールにより最大 3.6 倍，実問題を想定したフロリダ行列においては 6 段アンロールにより最大 4.5 倍の性能向上が確認できた．しかし，nnz/row の小さいフロリダ行列においては，最悪 0.1 倍まで性能が劣化するケースも見られたため，一つの課題である．

謝辞　スーパコンピュータ京は “AICS HPC 計算科学インターンシップ (2014 年度)” によって利用した．理化学研究所大規模並列数値計算技術研究チームの皆様にさまざまなご教授を頂いたことを深謝する．

本研究の一部は JSPS 科学研究費 25330144 の助成を受けた．

参考文献

[1] Bailey,D,H., “High-Precision Floating-Point Arithmetic in Scientic Computation.”, computing in Science and Engineering, pp.54-61 (2005).
[2] Hasegawa, H., “Utilizing the Quadruple-Precision floating-Point Arithmetic Operation for the Krylov Subspace Methods”, The 8th SIAM Conference on Applied Linear Algebra (2003).
[3] Knuth,D,E., “The Art of Computer Programming:

Seminumerical Algorithms”, Vol. 2, Addison-Wesley (1969).
[4] Dekker,T., “A floating-point technique for extending the available precision”, Numerische Mathematik, Vol.18, pp.224-242 (1971).
[5] 菱沼 利彰, 藤井 昭宏, 田中 輝雄, 長谷川 秀彦, AVX2 を用いた倍精度 BCRS 形式疎行列と倍々精度ベクトル積の高速化, 情報処理学会論文誌コンピューティングシステム (ACS), Vol.7, No.4, pp.25-33 (2014).
[6] FUJITSU, “Super Computer K.”, http://jp.fujitsu.com/about/tech/k/.
[7] FUJITSU, “SPARC64^{TM} IXfx Extensions.”, http://img.jp.fujitsu.com/downloads/jp/jhpc/sparc64ixfx-extensionsj.pdf.
[8] FUJITSU,“高速 4 倍精度基本演算ライブラリ 使用手引書” (2012).
[9] FUJITSU,“C++言語使用手引書（PREMEHPC FX10 用）”, pp.148-160 (2013).
[10] SSI, “反復解法ライブラリ Lis.”, http://www.ssisc.org/lis/index.ja.html.
[11] マイケル・マックール 他, “構造化並列プログラミング.”, カットシステム, pp.229-234 (2013).
[12] R. Barrett et al.,“Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods”, SIAM pp.57-65 (1994).
[13] “The University of Florida Sparse Matrix Collection.”, http://www.cise.ufl.edu/research/sparse/matrices/.