情報処理学会論文誌　コンピューティングシステム　Vol.7　No.4　25–33 (Dec. 2014)

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

菱沼 利彰[1,a)]　藤井 昭宏[1,b)]　田中 輝雄[1,c)]　長谷川 秀彦[2,d)]

受付日 2014年4月4日, 採録日 2014年7月16日

概要：高精度演算を用いることでKrylov部分空間法の収束を改善できるが，高精度演算はコストが高いことが知られている．高精度演算の1つに，倍精度を2つ組み合わせて4倍精度演算を行う倍々精度演算がある．我々は，Intelの SIMD 拡張命令であるAVX2を用いてBCRS形式の倍精度疎行列と倍々精度ベクトルの積（DD-SpMV）の高速化を行った．AVX2を用いたCRS形式のDD-SpMVでは，各行で端数処理などを必要とするが，BCRS形式は端数処理をなくし，メモリアクセスを改善できる．しかし，BCRS形式は演算量が増加する．本論文では，AVX2に適したBCRS形式のブロックサイズと，増加した演算量と端数処理の削減，メモリアクセスの改善効果のトレードオフについて示した．実験の結果，AVX2に最も適したブロックサイズは4×1であることが分かった．また，メモリアクセスの改善効果はサイズの大きい問題ほど有効で，行列サイズが10^5以上のとき，演算量が3.3倍以上になるケースにおいても，BCRS4×1にすることでCRS形式の実行時間を約45%に短縮できることを確認した．

キーワード：SIMD，疎行列ベクトル積，疎行列の格納形式，高精度演算

AVX2 Acceleration of Double Precision Sparse Matrix in BCRS Format and DD Vector Product

Toshiaki Hishinuma[1,a)]　Akihiro Fujii[1,b)]　Teruo Tanaka[1,c)]　Hidehiko Hasegawa[2,d)]

Received: April 4, 2014, Accepted: July 16, 2014

Abstract: High precision arithmetic can improve the convergence of Krylov subspace methods; however, it is very costly. One system of high precision arithmetic is Double-Double arithmetic, which uses two double precision variables to implement one quadruple precision variable. We accelerated double sparse matrix in BCRS format and DD vector product (DD-SpMV) using AVX2. DD-SpMV in CRS format using AVX2 needs fraction processing each row. BCRS format which aligns the SIMD register's length can eliminate fraction processing and improve memory access. However, it may increase operations. In this paper, we have shown that trade-off between increased operations and eliminated fraction processing and improving memory access. In experimental results, we concluded that the best BCRS block size is BCRS4×1. The effect of improving memory access in BCRS format depends on matrix sizes. When matrix size is more than 10^5, the number of computations also increased to 3.3 times, and the elapsed time of DD-SpMV in BCRS4×1 can be about 45% of that in CRS format.

Keywords: SIMD, sparse matrix and vector product, sparse matrix storage format, high precision arithmetic

1. はじめに

物理シミュレーションの核であるKrylov部分空間法は，丸め誤差の影響により収束に影響を受ける．収束の改善には高精度演算が有効だが，高精度演算は計算コストが高い[1]．高精度演算をする手法の1つに，倍精度変数を2つ用いて1つの4倍精度変数の値を保持し，4倍精度演算を実行する倍々精度演算という手法がある[2]．倍々精度演

[1] 工学院大学情報学部
Faculty of Informatics, Kogakuin University, Shinjuku, Tokyo 163–8677, Japan
[2] 筑波大学図書館情報メディア系
Faculty of Library, Information and Media Science, University of Tsukuba, Kasuga, Tsukuba 305–8550, Japan
a) em13015@ns.kogakuin.ac.jp
b) fujii@cc.kogakuin.ac.jp
c) teru@cc.kogakuin.ac.jp
d) hasegawa@slis.tsukuba.ac.jp

算を扱えるソフトウェアとして，Li ら [3] の XBLAS がある．これは倍々精度の密行列演算をサポートしており，入出力データを倍精度，内部の演算を倍々精度として実装している．

我々は，倍々精度演算を Intel の SIMD 拡張命令である Advanced vector extensions（AVX）[4] を用いて高速化している．入力行列を倍精度とし，ベクトルを倍々精度として内部で倍精度と倍々精度の混合演算を実装した．これまでの研究で，AVX を用いた Compressed row storage（CRS）形式 [5] の倍精度疎行列と倍々精度ベクトルの積（DD-SpMV）において，演算器性能がボトルネックであることが明らかになっている [6], [7]．また，AVX を用いた CRS 形式の DD-SpMV では，端数の処理などが性能劣化要因となっている．問題点を解決するために，我々は疎行列の格納形式に着目した．

疎行列の格納形式の 1 つである Block compressed row storage（BCRS）形式 [5] は，疎行列を 0 要素を含む $r \times c$ の小密行列（ブロック）の集合として格納するため，ロードの削減の効果があるが，ブロックが 0 要素を含むためにデータ量や演算量が増える [8]．我々は，ブロックサイズを SIMD レジスタサイズに合わせることで端数処理などをなくすことができると考えた．

Intel の SIMD 拡張命令である SSE2 は 1 命令で 2 つの倍精度演算を同時に実行でき，AVX は 4 つの倍精度演算を同時に実行できる．Intel Haswell アーキテクチャの AVX2 は 1 命令で 4 つの倍精度演算を Fused-Multiply-and-Add（FMA）命令で実行できる．FMA 命令を用いることで倍精度加算と乗算が 1 命令で実行できるため，AVX2 は理論上 SSE2 の 4 倍，AVX の 2 倍の性能が期待できる．

本論文では BCRS 形式の倍精度疎行列 $\boldsymbol{A}$ と倍々精度ベクトル $\boldsymbol{x}$ の積 $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$ を高速化し，端数処理などをなくしたことによる効果について分析を行った．以下 2 章で倍々精度演算のアルゴリズムと実装，3 章で AVX2 を用いた CRS，BCRS 形式の DD-SpMV の実装，4 章で数値実験，5 章でまとめを述べる．

2. 倍々精度演算

倍々精度演算は，Bailey が提案した “Double-Double” 精度のアルゴリズム [2] を用い，double-double 精度浮動小数 a を $a = a.hi + a.lo$，$\frac{1}{2}\mathrm{ulp}(a.hi) \geq |a.lo|$（上位 $a.hi$ と下位 $a.lo$ は倍精度浮動小数）とし，倍精度浮動小数 2 つを用いて 4 倍精度演算を実装する手法である．なお，$\mathrm{ulp}(\boldsymbol{x})$ は $\boldsymbol{x}$ の仮数部の “unit in the last place” を意味する．倍々精度の四則演算は，Dekker [9] と Knuth [10] の丸め誤差のない倍精度加算と乗算のアルゴリズムに基づき，倍精度の四則演算の組合せのみで実現できる．これから倍々精度演算は，倍精度，倍々精度間の混合精度演算や，精度の切替えが容易である．実装は小武守らの先行研究 [11] を基に，倍々精度変数 a を，2 つの倍精度変数 $a.hi$，$a.lo$ として持ち，倍々精度ベクトル $\boldsymbol{x}$ を 2 つの倍精度配列 $\boldsymbol{x}.hi$ と $\boldsymbol{x}.lo$ に格納することで，$\boldsymbol{x}.hi$ のみを用いれば，倍精度として扱うことができるようにしている．

倍々精度浮動小数は，符号部 1 bit，指数部 11 bit，仮数部 104（52×2）bit からなる．これは，符号部 1 bit，指数部 15 bit，仮数部 112 bit からなる IEEE754 準拠の 4 倍精度と比べ指数部が 4 bit，仮数部が 8 bit 少ない．簡単に 4 倍精度を利用する方法の 1 つに，Fortran REAL*16 がある．今回の実験環境において，Intel Fortran compiler 13.0.1 を用いて長さ 10^5 ベクトルの内積を 4 倍精度演算で計算するのにかかる時間は約 2.7 [ms] であるのに対し，倍々精度演算では約 0.64 [ms] で，倍々精度演算は Fortran REAL*16 の 4 倍精度と比べ約 4.2 倍高速であることを確認した．

実際の反復解法ライブラリにおいて，多くの場合倍精度の行列 $\boldsymbol{A}$ とベクトル $\boldsymbol{b}$ が与えられる．このとき，入力行列 $\boldsymbol{A}$ は反復解法中で繰り返し使われ，値の変更はない．我々のこれまでの研究 [6], [7] では，倍々精度ベクトルの内積計算などの性能は，メモリバンド幅がボトルネックになることが明らかになっている．メモリへの要求量を削減するために，疎行列 $\boldsymbol{A}$ を倍精度とし，倍精度疎行列 $\boldsymbol{A}$ と倍々精度ベクトル $\boldsymbol{x}$ の積 $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$; DD-SpMV を行った．これにより，データサイズを約半分にし，メモリへのデータ要求を減らすことができる．

CRS 形式の疎行列ベクトル積では，カーネル演算においてベクトル $\boldsymbol{x}$，$\boldsymbol{y}$ と，疎行列の列インデックス，要素の値をメモリに要求する．倍精度の疎行列ベクトル積は演算量が 2 flops（Floating point operations）で，ベクトルは倍精度，列インデックスは 4 バイト整数型，行列の要素の値は倍精度なので，1 命令あたりのメモリへの要求量は 28 (bytes)/2 (flops) = 14 byte/flop である．

ベクトルと行列の要素の値を倍々精度としたとき，倍々精度の積和演算の演算量は 21 flops である．このとき，1 命令あたりのメモリへの要求量は 52 (bytes)/21 (flops) = 2.48 byte/flop となる．ベクトルを倍々精度，行列の要素を倍精度にしたとき，倍々精度と積和演算は 19 flops からなり，1 命令あたりのメモリへの要求量は 44 (bytes)/19 (flops) = 2.32 byte/flop である．これは行列の要素を倍々精度とした場合と比べて約 7%少ない．

DD-SpMV の核である倍精度と倍々精度の積和演算は，倍々精度加算（DD_ADD）と，倍精度変数と倍々精度変数の乗算（DD_MULT）からなり，演算量は 19 flops である．DD_ADD は倍精度加減算のみでなり演算数は 11 flops である．DD_ADD のアルゴリズムを**図 1** に示す．DD_MULT は FMA 命令を用いた場合，倍精度加減算 3 回，倍々精度乗算 1 回，倍精度 FMA 命令 2 回からなり，命令数は 6 回，演算数は 8 flops になる．DD_MULT のアルゴリズムを**図 2** に示す．

```
DD_ADD(a.hi,a.lo,b.hi,b.lo,c.hi,c.lo){
    TWO_SUM(b.hi,c.hi,sh.eh);
    eh = eh + b.lo + c.lo;
    FAST_TWO_SUM(sh,eh,a.hi,a.lo);
}

TWO_SUM(x,y,s,e){
    s = x + y;
    v = s - x;
    e = (x - (s - v)) + (y - v);
}

FAST_TWO_SUM(x,y.s.e){
    s = x + y;
    e = y - (s - x);
}
```

図 1　倍々精度変数の加算
Fig. 1　Double-Double precision addition.

```
DD_MULT(a.hi,a.lo,b.hi,b.lo,c){
    TWO_PROD_FMA(b.hi,c,p1,p2);
    p2 += b.lo * c;
    FAST_TWO_SUM(p1,p2,a.hi,a.lo);
}

TWO_PROD_FMA(x,y,p,e){
    p = -x * y;
    e = x * y + p;
    p = -p;
}
```

図 2　倍精度変数と倍々精度変数の乗算
Fig. 2　Double and DD precision multiplication.

3. AVX2 を用いた DD-SpMV

3.1 Intel AVX2

AVX2 は，256 bit 長の SIMD レジスタを 16 本持ち，4 つの倍精度変数に対して同時に FMA 命令を使用できる [4]．AVX2 を用いた CRS，BCRS 形式の DD-SpMV の実装において，ロード，ストアに用いる AVX2 の組み込み関数命令を**表 1** に示す．

_mm256_set_pd (set) 命令はデータのメモリ配置が非連続な場合に使用する．これは 1 命令中に最大 4 回のランダムアクセスが発生する可能性がある．_mm256_load_pd (load) 命令はデータのメモリ配置が連続な場合に使用でき，ランダムアクセスが最大 1 回発生する可能性がある．_mm256_broadcast_sd (broadcast) 命令は 1 つの倍精度浮動小数を 256 bit の SIMD レジスタのすべての要素にロードする．これはランダムアクセスが最大で 1 回発生する可能性がある．

3.2 CRS 形式 DD-SpMV

CRS 形式 [5] は，疎行列の非零要素のみを圧縮して格納する．疎行列の行列サイズを N，非零要素数を the number of non-zero elements（nnz）としたとき，以下の 3 つの配列からなる．

表 1　DD-SpMV に用いる AVX2 のロード，ストア命令
Table 1　Intrinsics of AVX2 load and store instructions to implement SpMV.

Intrinsics	Description
_mm256_set_pd (set)	4 double precision elements from 4 source memory address
_mm256_load_pd (load)	4 way double precision load from source memory address
_mm256_broadcast_sd (broadcast)	double precision elements from source memory address to all elements of destination register
_mm256_store_pd (store)	store 256 bit data from source register to destination memory address

(1) 非零要素の値を格納する長さ nnz の倍精度配列 value
(2) 配列 value に格納された非零要素の列番号を格納する長さ nnz の整数配列 col_ind
(3) 配列 value と col_ind の各行の開始位置を格納する長さ N + 1 の整数配列 row_ptr

我々は，以下の 3 つの関数を用意した．

- 倍精度変数 a と倍々精度変数 x の乗算の結果を倍々精度変数 y に加算する “DD_ADD_MULT” 関数
- AVX2 において各行で発生する端数（1，2，3）を処理する “fraction_processing” 関数
- AVX2 のレジスタ内の 4 つの要素を DD_ADD を用いて倍精度変数に足し込む “reduction” 関数

“DD_ADD_MULT” 関数は，DD_ADD と DD_MULT からなる．“fraction_processing” は，set 命令の引数に対し，計算する要素数が 4 になるように 0 を代入し，“DD_ADD_MULT” 関数を用いて端数処理を行う．“reduction” は，DD_ADD を 3 回用いてトーナメント形式でレジスタ内の要素を総和する．

AVX2 を用いて CRS 形式の DD-SpMV を計算するコードを**図 3** に示す．AVX2 を用いた CRS 形式の DD-SpMV では，端数処理，$\boldsymbol{y}$ への総和計算が発生する．また，ベクトル $\boldsymbol{x}$ のロードには set 命令を使う．なお，_mm256_setzero_pd 命令は，レジスタの各要素に 0 を入れて初期化を行う命令である．

3.3 BCRS 形式 DD-SpMV

BCRS 形式は，疎行列を 0 を含むサイズ $r \times c$ の小密行列を用いて格納する．すべての要素が 0 となるブロックは作成しない．生成されるブロックの数を the number of blocks（blk）としたとき，BCRS 形式の疎行列は以下の 3 つの配列からなる．

(1) ブロック内の要素の値を格納する長さ blk $\times r \times c$ の倍

```
for(i=0;i<N;i++){
    yv = _mm256_setzero_pd(&y[i]);
    for(j=A->ptr[i];j<A->ptr[i+1]-3;j+=4){
      xv = _mm256_set_pd(&x[A->col_ind[j]]);
           x[A->col_ind[j+1]],
           x[A->col_ind[j+2]],
           x[A->col_ind[j+3]]);
      av = _mm256_load_pd(&A->value[j]);
      DD_ADD_MULT();
    }
    fraction_processing();
    reduction();
}
```

図 3　AVX2 を用いた CRS 形式の SpMV
Fig. 3　SpMV in CRS format using AVX2.

```
for(bi=0;bi<nr;bi++){
    i = bi*r; ii = 0; kk = Aout.bptr[bi];
    while(i+ii<n && ii<=nr-1){
      for(k=Ain.ptr[i+ii];k<Ain.ptr[i+ii+1];k++]){
        Aout.bindex[k] = Ain.index[k]/c;
        Aout.value[Ain.index[k] * r + ii] = Ain.value[k];
        kk = kk + 1;
      }
      ii = ii+1;
    }
    Aout.ptr[bi]=kk;
}
```

図 4　CRS 形式から BCRS 形式への変換
Fig. 4　Convert from CRS to BCRS format.

```
for(i=0;i<N;i++){
    yv = _mm256_setzero_pd(&y[i]);
    for(j=A->ptr[i];j<A->ptr[i+1]-3;j+=4){
      xv = _mm256_load_pd(&x[A->col_bind[j]]);
      av = _mm256_load_pd(&A->value[j]);
      DD_ADD_MULT();
    }
    reduction();
}
```

図 5　AVX2 を用いた BCRS1×4 形式 SpMV
Fig. 5　SpMV in BCRS1×4 format using AVX2.

精度配列 bvalue

(2) 配列 bvalue に格納されたブロックの開始列番号を格納する長さ nnz/c の整数配列 col_bptr

(3) 配列 bvalue と col_bind の各ブロック行の開始位置を格納する長さ $(N+1)/r$ の整数配列 row_bptr

CRS 形式を BCRS 形式に変換するアルゴリズムを**図 4** に示す．実装は，SPARSKIT [12] を参考にした．

BCRS 形式において，ブロックサイズを AVX2 のレジスタサイズに合わせた 4 にすることで，端数処理をなくし，レジスタブロッキング [13] の効果が得られる．

ブロックサイズが $r=1$，$c=4$ の BCRS1×4 のコードを**図 5** に示す．AVX2 を用いた BCRS1×4 の DD-SpMV は端数処理が発生しないが，$\boldsymbol{y}$ への総和計算が発生する．

```
for(i=0;i<N-3;i+=4){
    yv = _mm256_setzero_pd(&y[i]);
    for(j=A->ptr[i];j<A->ptr[i+1];j++){
      xv = _mm256_broadcast_sd(&x[A->col_bind[j]*4]);
      av = _mm256_load_pd(&A->value[j]);
      DD_ADD_MULT();
    }
    _mm256_store_pd(&y[i],yv);
}
```

図 6　AVX2 を用いた BCRS4×1 形式 SpMV
Fig. 6　SpMV in BCRS4×1 format using AVX2.

また，$\boldsymbol{x}$ のロードは load 命令で行われる．

次に，ブロックサイズが $r=4$, $c=1$ の BCRS4×1 のコードを**図 6** に示す．AVX2 を用いた BCRS4×1 の DD-SpMV は端数処理も $\boldsymbol{y}$ への総和計算も発生しない．また，$\boldsymbol{x}$ のロードは broadcast 命令で行われ，i ループが 4 段飛ばしになることで，$\boldsymbol{y}$ へのストアが CRS や BCRS1×4 の 1/4 となる．

3.4　BCRS 形式のブロックサイズごとの特徴

3.3 節のコードから，AVX2 を用いた BCRS 形式の DD-SpMV において BCRS 形式のブロックサイズを変化させたとき，以下のような特徴を持つことが分かる．

(1) $r \times c$ が 4 の倍数のとき，端数処理が不要

(2) c が 4 の倍数のとき，端数処理，$\boldsymbol{x}$ のロードに set 命令が不要

(3) r が 4 の倍数のとき，端数処理，$\boldsymbol{y}$ の足し込み，$\boldsymbol{x}$ のロードに set 命令が不要

CRS 形式と，$r \times c$ が 4 である BCRS1×4，2×2，4×1 の特徴を**表 2** に示す．ブロックサイズが大きいほどループアンローリングの効果が得られるが，ブロックに 0 を含むことにより発生する BCRS 形式の演算量の増加は，最大で $r \times c$ 倍になるため，演算量の増加を発生させやすくなる．

また，BCRS2×2 は，$r \times c$ が 4 だが，端数処理の削減とメモリアクセスの改善効果しかないため，効果はとぼしいと考えられる．

BCRS1×4 は，BCRS4×1 と比べ $\boldsymbol{y}$ への足し込みを各行で必要とするため BCRS4×1 と比べ性能が劣ると予想されるが，疎行列の構造によっては演算量の増加量が異なり，BCRS4×1 と比べ高速になる可能性がある．

4.　数値実験

4.1　実験環境

使用した CPU は Intel core i7 4770 K 3.4 GHz 4core 8threads（haswell），キャッシュサイズは 8 MB である．メモリは 16 GB DDR3-1600 dual channel で，メモリバンド幅は $12.8 \times 2 = 25.6$ GB/s である．

OS は CentOS6.4 で，コンパイラは Intel C/C++ Compiler 13.0.1，コンパイラオプションは最適化を行う “-O3”，AVX2 を有効にする “-xCORE-AVX2”，OpenMP を有効

表 2　各行列格納形式の特徴
Table 2　Feature of each matrix storage format.

	loading $\boldsymbol{x}$	fraction_processing	reduction	storing $\boldsymbol{y}$	maximum increasing computation
CRS	set	each row	each row	each row	none
BCRS1×4	load	none	each row	each row	×4
BCRS2×2	set	none	double space	double space	×4
BCRS4×1	broadcast	none	none	quadruple space	×4

表 3　BCRS 形式の効果 [ms]
Table 3　The effect of BCRS format [ms].

	CRS	BCRS1×4	BCRS4×1
test(32)	2.75	2.14	1.88
test(33)	3.15	2.33	2.11

化する “-openmp”，命令の並べ替えを抑制し精度を保つ “-fp-model precise” を用いた．OpenMP のスケジューリング方式は，“guided” を用い，実験はすべて 4 スレッドで行った．

実験には，2 種類の疎行列を用いた．1 つめは，The University of Florida Sparse Matrix Collection（フロリダコレクション）[14] から得た非対称，行列サイズ $N > 10^4$ を満たし，各問題の行列サイズ N，非零要素数 nnz，平均非零要素数 nnz/row がすべて異なる 100 問題である．

2 つめは，

- if($0 \leq j - i < m$) $a_{ij} = value$
- else $a_{ij} = 0$

を満たす 1 行あたりの非零要素数を m とした “test(m)” である．また，実験結果は各行列ごとに 100 回反復計測した平均を用いた．

4.2　端数処理，$\boldsymbol{y}$ への足し込みの影響

我々は，$N = 10^5$ の 2 つの行列：“test(32)”，“test(33)” を用いて，端数処理と $\boldsymbol{y}$ への足し込みにかかる時間を評価した．**表 3** に，CRS，BCRS4×1，BCRS1×4 の実行時間を示す．

“test(m)” において，BCRS 形式では最初の数行を除いたすべてのブロックが 0 を含まないため，演算量は CRS 形式とほぼ同一で，DD_ADD_MULT 関数が “test(32)” では 8×10^5 回，“test(33)” では 9×10^5 回発生する．それ以外の計算として，CRS 形式は 10^5 回の端数処理と $\boldsymbol{y}$ の足し込み，BCRS1×4 は 10^5 回の $\boldsymbol{y}$ の足し込みが発生する．

テスト用行列 “test(32)”，“test(33)” を用いた結果から，

- BCRS1×4 と CRS 形式における “test(32)” の結果から，メモリアクセスの改善効果は $2.75-2.14 = 0.61$ [ms]
- BCRS1×4 と CRS 形式における “test(32)” と “test(33)” から，DD_ADD_MULT 関数を除いた端数処理 10^5 回のオーバヘッドは $(3.15 - 2.75) - (2.33 - 2.14) = 0.21$ [ms]

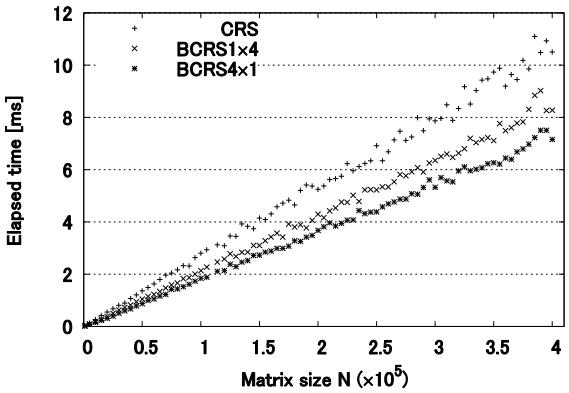

図 7　メモリアクセス性能
Fig. 7　Performance of memory access speed.

- “test(32)” における BCRS1×4 と BCRS4×1 の結果から，$\boldsymbol{y}$ への足し込み 10^5 回にかかる時間は $2.14-1.88 = 0.26$ [ms]

であると推定できる．これから，“test(33)” を用いた CRS 形式の DD-SpMV において端数処理，$\boldsymbol{y}$ の足し込みは全体の約 19%を占めていると考えられる．また，BCRS 形式を用いることでメモリアクセスの改善効果が得られ，“test(33)” において，BCRS4×1 は CRS と比べ 2.75 [ms]/3.15 [ms] = 約 67%まで実行時間を短縮できる．

4.3　メモリアクセスの影響

BCRS 形式の DD-SpMV におけるメモリアクセスの影響を分析するため，1 行あたり 32 の非零要素を持つ “test(32)” の行列サイズ N を 10^4 から 4.0×10^5 まで 5,000 ずつ変化させた．**図 7** に，CRS，BCRS1×4，BCRS4×1 の実行時間を示す．このとき，“test(32)” は 1.9×10^4 までキャッシュに収まる．

データはキャッシュが利用できるため 1 度しかメモリから読み込まないと仮定すると，CPU がメモリからロード・ストアしたデータ量は倍々精度ベクトル $\boldsymbol{x}$，$\boldsymbol{y}$，倍精度の value，4 バイト整数型の col_ind，row_ptr または col_bind，row_bptr である．

各行列サイズにおける計算時間と理論性能を比較する．本実験環境のメモリバンド幅は 25.6 GB/s である．ベクトル，行列の要素，インデックスのデータがキャッシュに収まる $N = 10^4$ のとき，CRS 形式，BCRS4×1 のデータサイズは約 3.6 [MB]，データがキャッシュに収まらな

い N = 4.0×10^5 のとき，CRS 形式のデータサイズは 168 [MB]，BCRS4×1 は約 167 [MB] である．

DD_ADD_MULT を構成する演算は，19 演算中 11 演算は加算であるため，2 つの FMA 演算器が並列に動作せず，演算に最大で 11 命令かかる．理論性能は，11 命令で 44 flops 計算できることを想定しているが，実際には 11 命令の間に 19 flops しか計算されないので，DD-SpMV の理論性能は 217.6 [GFLOPS] × 19/44 = 94 [GFLOPS] となる．CRS 形式は $\boldsymbol{y}$ への総和計算を必要とするため，演算量は N × 33 + nnz × 19，test(32) では BCRS4×1 の計算する要素数は CRS とほぼ等しいため，nnz × 19 と仮定して実行時間と比較すると，

データがキャッシュに収まるとき（$N = 10^4$）

- CRS 形式は理論上 0.067 [ms] で計算できる．実行時間は 0.28 [ms] で，理論性能の約 24%．データ転送速度は 3.6 [MB]/0.28 [ms] = 12.9 [GB/s] でメモリバンド幅の約 50%
- BCRS 形式は理論上 0.064 [ms] で計算できる．実行時間は 0.18 [ms] で，理論性能の 36%．データ転送速度は 3.6 [MB]/0.18 [ms] = 20.0 [GB/s] でメモリバンド幅の 78%

データがキャッシュに収まらないとき（$N = 4.0 \times 10^5$）

- CRS 形式は理論上 2.7 [ms] で計算できる．実行時間は 10.5 [ms] で，理論性能の約 26%．データ転送速度は 168 [MB]/10.5 [ms] = 16.0 [GB/s] でメモリバンド幅の約 63%
- BCRS 形式は理論上 2.5 [ms] で計算できる．実行時間は 7.2 [ms] で，理論性能の 35%．データ転送速度は 168 [MB]/7.2 [ms] = 23.2 [GB/s] でメモリバンド幅の 90%

となった．CRS，BCRS4×1 におけるキャッシュに収まるときと収まらないときのメモリ・演算器の理論性能との比の変化は小さく，性能は演算器がボトルネックになっていると考えられる．

次に，スレッド数を 1，2，4 に変化させ，メモリへの要求を増減させて評価を行った．BCRS4×1 において，1，2，4 スレッドにおけるキャッシュに収まるときの実行時間は 0.18 [ms]，0.33 [ms]，0.63 [ms] となった．2，4 スレッドのマルチスレッド化の効果は 1 スレッドと比べ 1.9 倍，3.5 倍である．キャッシュに収まらないときの実行時間は 24.5 [ms]，12.9 [ms]，7.2 [ms] となった．このとき，2，4 スレッドのマルチスレッド化の効果は 1 スレッドと比べ 1.9 倍，3.4 倍である．

キャッシュに収まる，収まらないにかかわらず，並列化の効果はスレッド数の増加に従って陽に増加している．このことからも，性能はメモリ性能に制約を受けず，演算器がボトルネックになっていると考えられる．

4.4　実問題への適用

フロリダコレクションの問題における BCRS1×4 の DD-SpMV の実行時間の CRS 形式との比を**図 8** に示す．BCRS1×4 は CRS 形式と比べ，最も時間のかかるもので 2.4 倍，最も時間のかからないもので 0.4 倍で，CRS 形式より実行時間が短縮できたものは 100 問題中 80 問題である．また，CRS と比べ最大で 3.9 倍の演算量となった．

BCRS4×1 の DD-SpMV の実行時間の CRS 形式との比を，増加した演算量順にソートしたものを**図 10** に，行列サイズ N 順にソートしたものを**図 9** に示す．BCRS4×1 は CRS 形式と比べ，最も時間のかかるもので 2.3 倍，最も時間のかからないもので 0.3 倍で，CRS 形式より実行時間が短縮できたものは 100 問題中 83 問題である．また，CRS と比べ最大で 3.9 倍の演算量となった．

2 番目に疎行列サイズが大きい N ≒ 5.1×10^6, nnz ≒ 10^8, nnz/row が 19.2 の疎行列では，BCRS4×1 において演算量が 3.3 倍に増加したが，実行時間は CRS 形式で 212.7 [ms], BCRS4×1 で 95.2 [ms] で，BCRS4×1 と CRS 形式の比は 0.45 倍と高い効果を得ている．このとき，1 行あたり平均

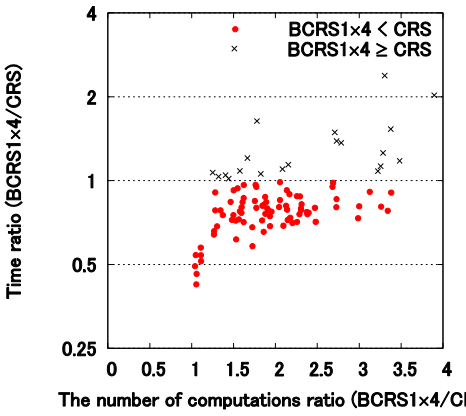

図 8　BCRS1×4 の効果（BCRS1×4 と CRS の時間比）

Fig. 8　The effect of BCRS1×4 (BCRS1×4 compared to CRS).

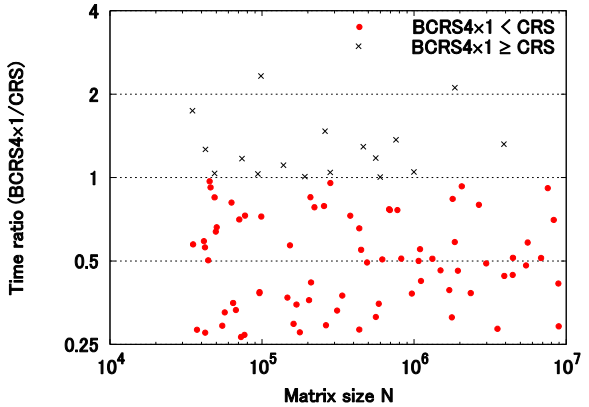

図 9　BCRS4×1 のメモリアクセスの改善効果（BCRS4×1 と CRS の時間比）

Fig. 9　The effect of smoothing memory access by BCRS4×1 (BCRS4×1 compared to CRS).

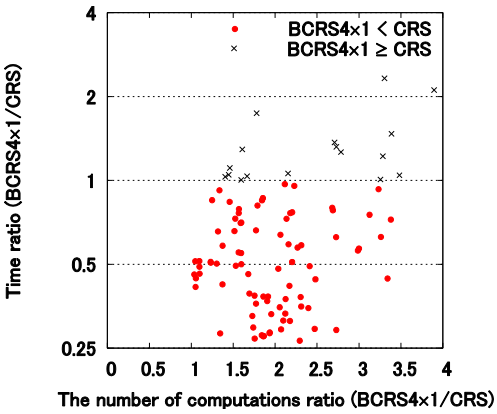

図 10　BCRS4×1 の効果（BCRS4×1 と CRS の時間比）
Fig. 10　The effect of BCRS4×1 (BCRS4×1 compared to CRS).

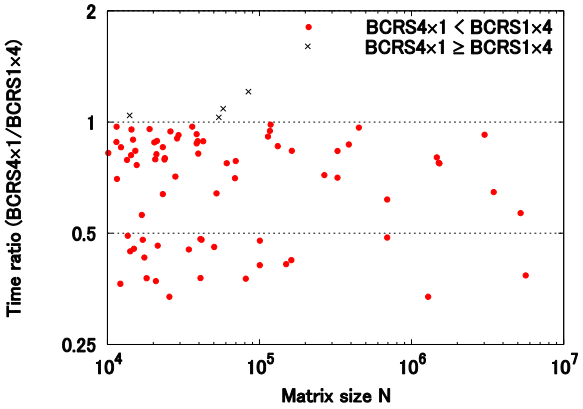

図 11　BCRS1×4 と BCRS4×1 の時間の比
Fig. 11　The elapsed time of BCRS4×1 compared to BCRS1×4.

で 45 の非零要素，11 回の DD_ADD_MULT が増えている．4.2 節の推定より，CRS 形式において端数処理と $\boldsymbol{y}$ への足し込みを 5.1×10^6 回行うのにかかる時間は約 24 [ms] であることから，実問題において，BCRS 形式にしたことによる効果の多くをメモリアクセスの改善効果が占めていると考えられる．

また，図 9 から，行列サイズが約 10^5 以上のとき，BCRS4×1 は CRS 形式よりも高速であることが分かる．このことから，CRS 形式においてサイズが大きい問題はランダムアクセスによって性能が低下しやすく，BCRS 形式のメモリアクセスの改善効果により高い効果を得られていると考えられる．

実問題において，BCRS1×4 と BCRS4×1 は，生成されるブロック数が異なるため，ブロック数によっては BCRS1×4 の方が高速な場合があると考えられる．BCRS1×4 と BCRS4×1 の時間の比を**図 11** に，演算量の比を**図 12** に示す．その結果，BCRS4×1 は，BCRS1×4 と比べ 100 問題中 94 問題で高速で，演算量は CRS と比べ最大で 1.12 倍，最小で 0.85 倍の演算量である．

次に，CRS 形式と BCRS 形式のブロックサイズの使い分けの必要性について考える．各格納形式のみで 100 問題を計測した場合の合計時間と，格納形式を最適に組み合わせた場合の合計時間を**表 4** に示す．この結果から，BCRS4×1 のみで計測した場合の合計時間と各形式を最適に組み合わせた場合の比は 1.01 と小さい．

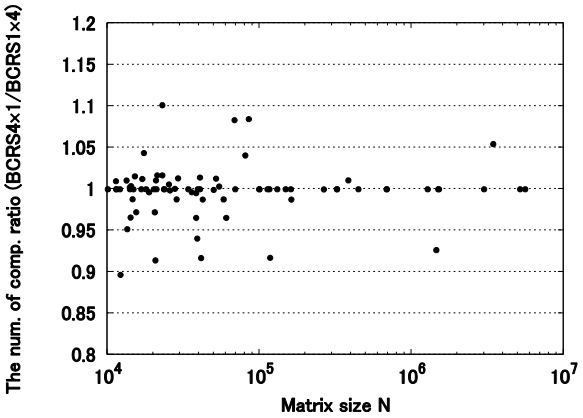

図 12　BCRS1×4 と BCRS4×1 の演算量の比
Fig. 12　The number of computation ratio of BCRS4×1 compared to BCRS1×4.

表 4　各形式の合計時間 [ms]．カッコ内は相対性能
Table 4　Total elapsed time of each storage formats [ms] (relative performance).

	Total elapsed time	The number of the best matrices
CRS	730 (1.37)	14
BCRS1×4	880 (1.33)	4
BCRS4×1	540 (1.01)	82
The best combination	530 (1)	100

我々は，これらの結果を通して以下の結論を得た．

(1) 問題ごとの BCRS1×4 と BCRS4×1 の演算量の差は小さく，BCRS1×4 は各行でベクトル $\boldsymbol{y}$ への足し込みを必要とするため，BCRS4×1 と比べ効果が小さい．
(2) 最も効果的なブロックサイズは BCRS4×1 であり，格納形式を最適に組み合わせた場合との比は 1.01 倍と小さい．
(3) メモリアクセスの改善効果は問題サイズが大きいほど有効であり，10^5 以上の問題において BCRS4×1 は CRS 形式よりも高速である．

4.5　ブロック・アンローリング

本節では，BCRS4×1 をアンローリングした，BCRS(4×1)×2 と，BCRS(4×1)×4 の効果について検証する．

表 5 に，BCRS4×1，(4×1)×2，(4×1)×4 のフロリダコレクション 100 問題における合計時間と，最も性能が良かった個数を示す．

結果から，ブロック・アンローリングを行っても合計時

表 5　ブロック・アンローリングの効果 [ms]
Table 5　The effect of block unrolling [ms].

	Total elapsed time	The number of the best matrices
BCRS4×1	540	86
BCRS(4×1)×2	540	9
BCRS(4×1)×4	550	5

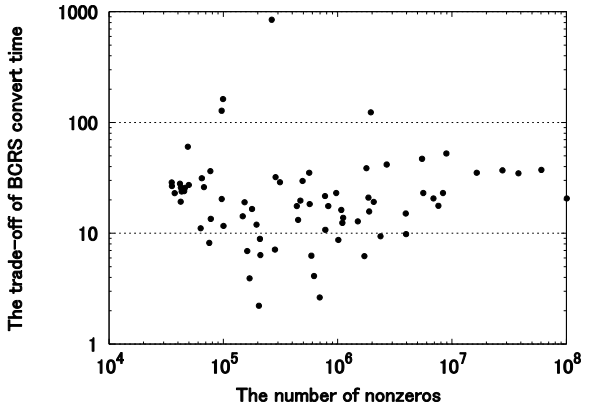

図 13　BCRS4×1 の変換時間と計算時間短縮のトレードオフ
Fig. 13　Trade-off of convert from CRS to BCRS and computing time.

間にはほとんど変化はなく，BCRS4×1 において最も性能が高かった個数は 86 問題で，ブロック・アンローリングは効果がない．

4.6　BCRS の生成コスト

CRS から BCRS に変換する時間と計算時間の短縮によるトレードオフについて述べる．反復解法において，疎行列は使いまわすことが考えられるため，何試行行えば，CRS から BCRS にしたことによる計算時間の短縮時間が，CRS 形式を BCRS 形式に変換する時間を上回るかのトレードオフの指標として，

- BCRS の変換時間 / (CRS の計算時間 − BCRS の計算時間)

を用いて評価を行った．

図 13 に BCRS4×1 において CRS 形式より高速だったフロリダコレクションの 83 問題における BCRS4×1 の変換時間とのトレードオフを示す．結果から，BCRS 形式の変換時間は，多くの問題で問題サイズにかかわらず 100 回程度の反復で生成コストを回収でき，BCRS4×1 による計算時間の短縮効果とくらべ，生成コストは小さい．

4.7　BCRS4×1 を用いた倍々精度 BiCGStab 法

BCRS4×1 の反復解法への効果を確認する．解法は BiCGStab 法，収束条件は，1.0×10^{-8} とし，CRS，BCRS 形式の DD-SpMV と，倍精度 CRS 形式の SpMV の比較を行った．なお，計測時間に BCRS の変換時間は含めていない．対象問題は，BCRS4×1 の効果があった 83 問の中から，キャッシュに収まらないサイズを持ち，倍々精度によって収束が改善される以下の 3 問を選んだ．

表 6　BiCGStab 法の収束時間．カッコ内は反復回数
Table 6　Execution time (iter.) of BiCGStab.

	Double, CRS	DD, CRS	DD, BCRS4×1
epb3	0.68 (3,904)	2.71 (2,712)	1.82 (2,708)
ex11	1.48 (1,550)	1.97 (1,312)	1.01 (1,313)
Raj1	No conv.	21.43 (8,102)	15.76 (8,102)

(1) N = 84,617，nnz = 463,625，BCRS4×1 にすることで CRS 形式と比べ演算量が 1.6 倍，DD-SpMV の計算時間が 0.66 倍になる “epb3”
(2) N = 16,614，nnz = 1,096,948，BCRS4×1 にすることで CRS 形式と比べ演算量が 1.4 倍，DD-SpMV の計算時間が 0.43 倍になる “ex11”
(3) N = 263,743，nnz = 1,302,464，BCRS4×1 にすることで CRS 形式と比べ演算量が 2.2 倍，DD-SpMV の計算時間が 0.69 倍になる “Raj1”

表 6 に，BiCGStab 法の収束までの時間と反復回数を示す．この実験から，我々は以下のような結果を得た．

- “epb3” は，倍々精度 BCRS 形式を用いることで倍々精度 CRS 形式と比べ収束時間が約 67%，倍精度 CRS 形式と比べ収束回数が約 70%になった．
- “ex11” は，倍々精度 BCRS 形式を用いることで倍精度 CRS 形式と比べ収束時間が約 68%，収束回数が約 85%，倍々精度 CRS 形式と比べ収束時間が約 51%になった．
- “Raj1” は，倍々精度にすることで問題が求解できるようになり，BCRS 形式を用いることで CRS 形式と比べ収束時間が約 74%になった．

5.　まとめ

本研究では，BCRS 形式の倍精度疎行列 $\boldsymbol{A}$ と倍々精度ベクトル $\boldsymbol{x}$ の積 $\boldsymbol{y} = \boldsymbol{Ax}$; DD-SpMV を AVX2 を用いて高速化を行った．AVX2 を用いた CRS 形式における DD-SpMV は各行で端数処理や，レジスタ内の要素を $\boldsymbol{y}$ に足し込む必要がある．BCRS 形式は，これらの計算をなくし，メモリアクセスを改善することができるが，計算量が増えるという問題点がある．

実験の結果，BCRS1×4 や，ブロック・アンローリングの効果は小さく，最も最適なブロックサイズは端数処理や $\boldsymbol{y}$ への足し込みをなくせる BCRS4×1 であると結論づけた．

実験の結果，我々は端数処理や $\boldsymbol{y}$ への足し込みにかかる時間は問題サイズ 10^5，1 行あたりの非零要素数 32 の問題において全体の約 19%と推定できた，

メモリアクセスの改善効果は行列サイズに依存し，行列サイズが大きな問題では，メモリアクセスの改善効果が高

いことが分かり，今回扱った 100 問題において，行列サイズが 10^5 以上の問題はすべて BCRS4×1 が CRS より高速であるという結果が得られた．

また，CRS 形式を BCRS 形式に変換する時間は，BCRS 形式によって得られる高速化効果により多くの場合約 100 反復で回収でき，実際の反復解法においても有効であると考えられる．

倍々精度 BCRS 形式を用いた BiCGStab 法は，倍々精度 CRS 形式と比べ収束回数の影響を受けずに約 51%の時間で求解でき，倍精度 CRS 形式で行った場合と比べ収束時間を 68%，収束回数を約 85%にできた．これにより，AVX2 を用いた倍精度 BCRS 形式疎行列と倍々精度ベクトルの積は実際の反復解法で有効であることが分かった．今後の課題として，分散メモリ環境上で BCRS 形式を用いて実際の反復解法の高速化を行う．

参考文献

[1] Hasegawa, H.: Utilizing the Quadruple-Precision floating-Point Arithmetic Operation for the Krylov Subspace Methods, *The 8th SIAM Conference on Applied Linear Algebra* (2003).

[2] Bailey, D.H.: High-Precision Floating-Point Arithmetic in Scientific Computation, *Computing in Science and Engineering*, pp.54–61 (2005).

[3] Li, X. et al.: Design, implementation and testing of extended and mixed precision BLAS, *ACM Trans. Math. Software*, Vol.28, No.2, pp.152–205 (2002).

[4] Intel: Intrinsics Guide, available from ⟨http://software.intel.com/en-us/articles/intel-intrinsics-guide⟩.

[5] Barrett, R. et al.: *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, pp.57–65 (1994).

[6] 菱沼利彰，藤井昭宏，田中輝雄，長谷川秀彦：AVX を用いた倍々精度疎行列ベクトル積の高速化，2013 年ハイパフォーマンスコンピューティングと計算科学シンポジウム，pp.23–31 (2013).

[7] Hishinuma, T., Fujii, A., Tanaka, T. and Hasegawa, H.: AVX acceleration of DD arithmetic between a sparse matrix and vector, Lecture Notes in Computer Science 8384, pp.622–631, Springer, 2014 at *the 10th International Conference on Parallel Processing and Applied Mathematics* (*PPAM 2013*), Part 1, Warsaw, Poland (2013).

[8] Kotakemori, H., Hasegawa, H., Kajiyama, T., Nukada, A., Suda, R. and Nishida, A.: Performance Evaluation of Parallel Sparse Matrix-Vector Products on SGI Altix3700, *Proc. 1st International Workshop on OpenMP*, Lecture Notes in Computer Science 4315, pp.153–163 (2008).

[9] Dekker, T.: A floating-point technique for extending the available precision, *Numerische Mathematik*, Vol.18, pp.224–242 (1971).

[10] Knuth, D.E.: *The Art of Computer Programming: Seminumerical Algorithms*, Vol.2, Addison-Wesley (1969).

[11] 小武守恒，藤井昭宏，長谷川秀彦，西田　晃：反復法ライブラリ向け 4 倍精度演算の実装と SSE2 を用いた高速化，情報処理学会論文誌コンピューティングシステム，Vol.1, No.1, pp.73–84 (2008).

[12] Saad, Y.: SPARSKIT: A basic tool-kit for sparse matrix computations, version2 (1994), available from ⟨http://www-users.cs.umn.edu/~saad/software/SPARSKIT/index.html⟩.

[13] Im, E., Yelick, K. and Vuduc, R.: SPARSITY: Optimization Framework for Sparse Matrix Kernels, *International Journal of High Performance Computing Applications*, Vol.18, No.1, pp.135–158 (2004).

[14] The University of Florida Sparse Matrix Collection, available from ⟨http://www.cise.ufl.edu/research/sparse/matrices/⟩.

菱沼 利彰　（学生会員）

1990 年生．2013 年工学院大学情報学部情報デザイン学科卒業．同大学大学院修士課程在学中．コンピュータアーキテクチャ，高精度演算に興味を持つ．

藤井 昭宏　（正会員）

1975 年生．1999 年東京大学理学部情報科学科卒業．2004 年同大学大学院情報理工系研究科コンピュータ科学専攻博士課程修了．博士（情報理工）．工学院大学情報学部講師．大規模線形問題に対するマルチレベルな解法に興味を持つ．電子情報通信学会，IEEE-CS 各会員．

田中 輝雄　（正会員）

1958 年生．1983 年電気通信大学大学院情報数理工学研究科修士課程修了．2007 年同大学院情報システム学研究科博士課程修了．博士（工学）．1983 年（株）日立製作所中央研究所入所．2011 年工学院大学情報学部教授．専門は，大規模数値計算アルゴリズム．日本応用数理学会，IEEE-CS 各会員．

長谷川 秀彦　（正会員）

1958 年生．1983 年筑波大学大学院博士課程社会工学研究科中退．筑波大学図書館情報メディア系教授．博士（工学）．数値線形代数全般に興味を持つ．日本応用数理学会，SIAM，ACM 各会員．