Many Integrated Core architecture における倍々精度疎行列ベクトル積

佐々木信一[†, a)]　菱沼利彰[†, b)]　藤井昭宏[†, c)]　田中輝雄[†, d)]

MIC（Many Integrated Core Architecture）のようなメモリ性能に対して演算器性能が十分に高いアーキテクチャではその演算能力を高精度演算に割くことが可能である．高精度演算は演算時間が長いため，倍々精度と呼ばれる倍精度変数 2 つを組み合わせて 1 つの四倍精度変数の値を保持する高速な演算手法がある．本研究では，MIC 向けに SIMD を用いて倍精度疎行列と倍々精度ベクトルの積を実装し，疎行列の格納形式の違いから性能決定の要因を検証した．
疎行列の格納形式である CRS 形式と BCRS 形式を対比させながらメモリアクセスや計算量，列あたりの平均ブロックヒット数，並列化時の AFFINITY と SCHEDULE について検証し，条件の良い問題では BCRS 形式は CRS 形式に対して 17 倍の性能を示した．特にブロックヒット数が性能を決める大きな要因であり，ブロックヒット数が 1.9 以上の時は BCRS 形式，1.9 未満の時は CRS 形式を用いることで多くの疎行列にて高い性能を示すことを明らかにした．

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，発散する．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[1]．高精度演算は演算時間が多くかかるため，これを解決する方法として 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手法がある[2]．倍精度演算の SIMD　(single instruction multiple data) 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[3]．

反復解法の核となる演算はベクトル演算や疎行列ベクトル積 (SpMV：Sparse matrix and vector product)である．それらの倍精度演算での性能はメモリ性能に制約を受ける．菱沼らは CPU 上で AVX (Intel advanced vector extensions) による倍々精度演算の実装と効果を検証し，倍精度疎行列と倍々精度ベクトルの積がメモリネックにならないことを示した．また，SIMD 命令を用いることにより発生した端数処理や浮動小数点レジスタ内での水平加算が性能の劣化要因の一つであると論じている[4]．

2012 年に Intel 社から発表された MIC (Many Integrated Core architecture)は浮動小数点レジスタが 512bit で 50 個以上のコアをもち，倍精度での理論性能は 1TFlops を超える高並列アーキテクチャである[5]. MIC を使う上で高い性能を得るためには命令の SIMD 化が必要不可欠である．

我々は MIC の浮動小数点レジスタ幅に着目して, 端数処理や浮動小数点レジスタ内での水平加算が発生しない BCRS (Block Compressed Row Storage) 形式と呼ばれる疎行列の格納形式が有効であると考えた．

我々は MIC 上で SIMD 命令を用いて疎行列ベクトル積を倍々精度で解く演算を実装し，疎行列の格納形式を比較し性能の決定要因について分析を行った．

† 工学院大学 Kogakuin University (Japan)
a) em14009@ns.kogakuin.ac.jp
b) em13015@ns.kogakuin.ac.jp
c) fuji@cc.kogakuin.ac.jp
d) teru@cc.kogakuin.ac.jp

本論文では，はじめに倍々精度演算の概要，次に実験環境である MIC について, 最後に格納形式別の倍々精度演算による疎行列ベクトル積の SIMD 命令を用いた実装と数値実験を行い，倍々精度演算を用いた疎行列ベクトル積における BCRS 形式の有用性について述べる．

2.　倍々精度演算

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[6]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．

“Double-Double”精度のアルゴリズムでは Knuth[7]が示した倍精度加算(DD_ADD)のアルゴリズムと Dekker[8]が示した倍精度乗算(DD_MULT)のアルゴリズムが用いられており，倍精度の四則演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[4]．

倍々精度加算のアルゴリズムを図 2 に，乗算のアルゴリズムを図 3 に示す. MIC では FMA (Fused Multiply-Add) 演算を用いることで積和演算の中間結果を無限精度で保持できるため, 丸め誤差のない乗算の結果を加算に利用できる．このため, 倍々精度加算は倍精度加算命令 11 回で構成でき演算量は 11flops，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 回で構成されており，FMA 演算は 1 命令で 2flops のため演算量は 10flops となる．

ゆえに，疎行列ベクトル積の核となる倍々精度積和演算は乗算と加算 2 つを合わせた 21flops となる．

ⓒ 2014 Information Processing Society of Japan

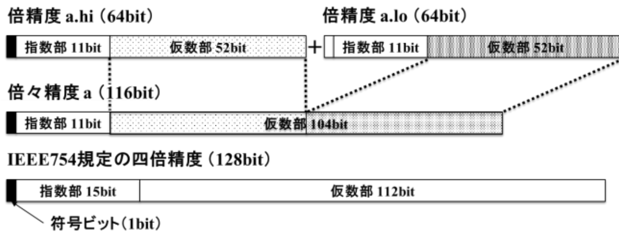

図 1　倍々精度のビット数

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

|x|≧|y|が仮定できる場合	|x|≧|y|が仮定できない場合

FAST_TWO_SUM(x, y, s, e)
{
s = x + y
e = y − (s − x)
}
	TWO_SUM(x, y, s, e)
	{
	s = x + y
	v = s − x
	e = (x − (s − v)) + (y − v)
	}

図 2　倍々精度加算のアルゴリズム

```
DD_MUL(a, b, c)
{
    TWO_PROD(b.hi, c.hi, pl, p2)
    p2 = fmadd(b.hi, c.lo, p2)
    p2 = fmadd(b.lo, c.hi, p2)
    FAST_TWO_SUM(p1, p2, a.hi, b.lo)
}
```

TWO_PROD(x, y, p, e)	fmadd(x, y, z)	fmsub(x, y, z)

{
p = x * y
e = fmsub(x, y, p)
}
	{
	return((x * y) + z)
	}
		{
		return((x * y) − z)
		}

図 3　倍々精度乗算のアルゴリズム

3. MIC 上での倍々精度疎行列ベクトル積

3.1 MIC 環境でのプログラミング

MIC を用いてプログラムを高速化するためには SIMD 並列化が必要である．MIC は浮動小数点レジスタが 512bit であり，SIMD 並列化を行なうと 1 命令で倍精度データ 8 つを同時に処理できる．ただし，従来の CPU 向けに用意された SSE2 (Streaming SIMD extensions) や AVX といった SIMD 拡張命令セットは使用できず，IMCI (Intel Initial Many Core Instructions)と呼ばれる MIC 専用の SIMD 拡張命令セット[9]を用いる必要がある．MIC 上での倍々精度疎行列ベクトル積の実装において，ロードとストアに用いた IMCI の関数を表 1 に示す．

メモリからレジスタへの倍精度データのロードの命令は主に 3 種類ある．LOAD 命令はレジスタに格納したいデータがメモリ上に連続で存在するときに使用でき，ランダムアクセスは最大 1 回である．対して SET 命令はメモリ上のデータ配置が非連続な場合でも使用できるが，ランダムアクセスが最大 8 回発生する．BROADCAST 命令は浮動小数点レジスタに同じデータを 8 つ格納したい場合に使用，ランダムアクセスは最大 1 回である．

表 1　IMCI のロードとストア命令

命令	説明
_mm512_load_pd **(LOAD)**	指定したメモリアドレスから連続する 8 つの倍精度データを浮動小数点レジスタへ格納
_mm512_set_pd **(SET)**	8 箇所のメモリアドレスから 各倍精度データを浮動小数点レジスタへ格納
_mm512_broadcast_pd **(BROADCAST)**	指定したメモリアドレスの倍精度データ 1 つを 浮動小数点レジスタへ 8 つに複製して格納 ※本研究では組み込み関数_mm512_extload_pd()の引数に_MM_BROADCAST_1X8 を与えたものを_mm512_broadcast_pd と呼称
_mm512_store_pd **(STORE)**	浮動小数点レジスタから指定のメモリアドレスへ倍精度データを 8 つ格納

1 命令当たり最大 8 回のランダムアクセスの発生を回避するために MIC 向けの倍々精度演算の実装において，連続する四倍精度データを保持する場合には，構造体などは使用せずに，値を保持するための上位倍精度データと下位部倍精度データをそれぞれ連続に確保することで SIMD 並列化時に高速にレジスタへ読み込む．

3.2 CRS 形式 SpMV

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[10]がある．図 4 は CRS 形式のデータ構造を示したものであり，疎行列 **A** の非零要素数を nnz 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．

(1) value：非零要素の値を納める倍精度配列
(2) index：非零要素の列番号を納める整数配列
(3) 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 N＋1 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，$\boldsymbol{x}$ 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．

IMCI では SET 命令以外にメモリ上の非連続な値を読み込む集約関数_mm512_ i32logather_pd()がある．これは，レジスタに非連続な配列の要素を読み込む際に，読み込みたい配列の添字を格納した別のレジスタを利用することで読み込みを行っている．CRS 形式では **A**.index に添字が連続に保持されているため，LOAD 命令で添字をレジスタに格納でき，SET 命令による実装で発生していた間接参照の影響を緩和できると考えた．

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

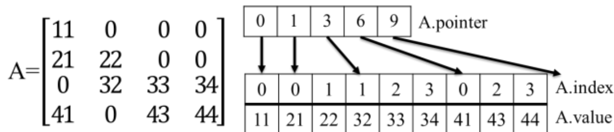

図 4　CRS 形式のデータ構造

```
for(i=0;i<N;++i){
   _mm512d yhv = _mm512_setzero_pd();//レジスタの 8 要素を 0 で初期化
   _mm512d yhl = _mm512_setzero_pd();
   for(j=A->ptr[i];j<A->pointer[i+1];j+=8);{
        _mm512d xhv = _mm512_set_pd(&xh[A->index[j]], &xh[A->index[j+1]],
                   &xh[A->index[j+2]], &xh[A->index[j+3]], &xh[A->index[j+4]],
                   &xh[A->index[j+5]], &xh[A->index[j+6]], &xh[A->index[j+7]]);
        _mm512d xlv = _mm512_set_pd(&xl[A->index[j]] , &xl[A->index[j+1]],
                    &xl[A->index[j+2]], &xl[A->index[j+3]], &xl[A->index[j+4]],
                    &xl[A->index[j+5]], &xl[A->index[j+6]], &xl[A->index[j+7]]);
        _mm512d av = _mm512_load_pd(&A->value[j]);
        DD_FMA(xhv, xlv, av, yhv, ylv);
    }
    fraction_processing();
    reduction();
}
```

図 5　IMCI を用いた CRS 形式の DD-SpMV

り，倍精度疎行列 A_D と倍々精度ベクトル x_{DD} の積 $y_{DD}=A_D x_{DD}$ を DD-SpMV とした．このとき，倍々精度換算および乗算の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，演算量が 19flops になる．

IMCI を用いた CRS 形式の DD-SpMV のコードを図 5 に示す．IMCI を用いて SIMD 並列化すると，行方向に対して 8 つずつデータを処理するため行あたりの非零要素数が 8 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 命令を用いて浮動小数点レジスタに要素数が 8 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 y へ足し込む必要があり，DD_ADD()をトーナメント式に用いた関数を reduction()と定義する．

fraction_processing()では毎回端数の個数を確認しているため条件分岐が発生し, reduction()では DD_ADD()を 7 回行っているため 77flops の演算が各行で発生する．

3.3 BCRS 形式 SpMV

BCRS 形式は，行列が r×c の小行列（ブロック）の集合として扱い, 非零要素を保持するブロックのみを格納する．ブロックの数を blk(the number of blocks)とすると，BCRS 形式の疎行列は以下の 3 本の配列で構成される．

(1) bvalue：ブロックの内の値を納める倍精度配列
　長さは blk×r×c

(2) bindex : 各ブロックの先頭要素の番号を納める整数配列
　長さは nnz / c

(3) bpointer：各ブロック行の開始位置を納める整数配列
　長さは (N + 1) / r

```
for(i=0;i<N;++i){
     _mm512d yhv = _mm512_setzero_pd();
     _mm512d ylv = _mm512_setzero_pd();
     for(j=A->bpointer[i];j<A->bpointer[i+1]-7;j+=8){
          _mm512d xhv = _mm512_load_pd(&xh[A->bindex[j]]);
          _mm512d xlv = _mm512_load_pd(&xl[A->bindex[j]]);
          _mm512d av = _mm512_load_pd(&A->bvalue[j]);
          DD_FMA(xhv, xlv, av, yhv, ylv);
     }
     reduction();
}
```

図 6　IMCI を用いた BCRS_1x8 形式の DD-SpMV

```
for(i=0;i<N-7;i+=8){
     yhv = _mm512_setzero_pd();
     ylv = _mm512_setzero_pd();
     for(j=A->bpointer[i];j<A->bpointer[i+1];++j){
          xhv = _mm512_broadcast_pd(&xh[A->bindex[j]*8]);
          xlv = _mm512_broadcast_pd(&xl[A->bindex[j]*8]);
          av = _mm512_load_pd(&A->bvalue[j]);
          DD_FMA(xhv, xlv, av, yhv, ylv);
     }
     _mm512_store_pd(&yh[i],yhv);
     _mm512_store_pd(&yl[i],ylv);
}
```

図 7　IMCI を用いた BCRS_8x1 形式の DD-SpMV

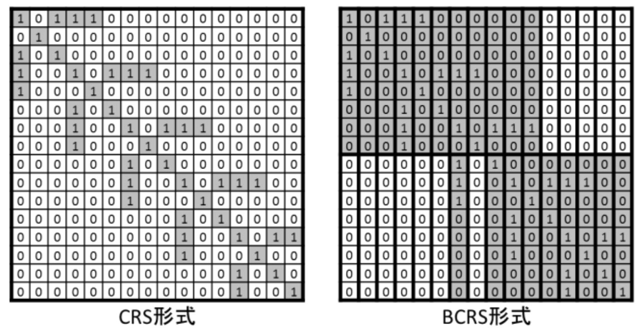

図 8　CRS と BCRS の演算量の違い

BCRS 形式ではブロックサイズによっては端数処理 fraction_processing()と reduction()をなくすことができる．

ブロックサイズが r=1,c=8 の BCRS_1x8 による DD-SpMV のコードを図 6 に示す．疎行列の行方向に倍精度 8 個のブロックを作ることで端数処理は発生せず，x に関してレジスタへのデータの読み込みも連続であるため LOAD 命令が利用でき，内側ループが 8 段飛ばしである．ただし，疎行列の各行でレジスタ内の総和を y へ足し込む必要がある．

ブロックサイズが r=8,c=1 の BCRS_8x1 による DD-SpMV のコードを図 7 に示す．こちらは疎行列の列方向に倍精度 8 個のブロックを作ることで端数処理は発生させず，レジスタへの x の読み込みを BROADCAST 命令で行なう．外側ループが 8 段飛ばしとなっているため，STORE 命令が CRS 形式や BCRS_1x8 の 1/8 となり，BCRS_1x8 のように各行での y への総和演算も発生しない．

BCRS 形式では r×c が 8 の倍数の時に端数処理が発生しなくなる．さらに，r と c のどちらかが 8 の倍数であれば，

x のロードに SET 命令を使用せずランダムアクセスを削減できる．上記以外の実装方法は BCRS_1x8 と BCRS_8x1 を基準にループアンローリングによって調整することになる．

図 8 は CRS 形式と BCRS_8x1 の演算量の違いを示す例であり，彩色した部分が演算に用いる疎行列の要素である．BCRS 形式は 0 も演算に含むため CRS 形式に比べ演算量が増加する．最も演算量が増えるケースはすべてのブロックに非零要素が 1 つしかないときで CRS 形式の r×c 倍となる．BCRS_1x8 は reduction()が発生するため，BCRS_8x1 に比べ演算量が増加，STORE 命令回数も 8 倍であり，BCRS_8x1 に対して性能が低いと予想し，BCRS_8x1 に対して実装・評価を行った．

4. 数値実験

4.1 実験環境

実験に用いた環境を表 2 に示す. MIC には Intel Xeon Phi 5110P を用いた．

Intel Xeon Phi 5110P は，動作周波数 1.053GHz，60 コア 240 スレッド，1 コアにつき 1 つ FMA 演算器をもち (独立な加算器と乗算器は持たない), 浮動小数点レジスタの幅は 512bit (倍精度データ 8 つ分) であるため, IMCI を用いたときの倍精度演算の理論ピーク性能は 1010.88GFlops である．

メモリバンド幅は 320GB/s であるが，本実験では ECC を有効にしているためこれよりも低い値となる．条件により値が変動するが，最低でも約 140GB/s のメモリバンド幅[11]となるため, 本論文でメモリの評価をする際は 140GB/s を用いる．

コンパイルはホスト・プラットフォーム上で行う．コンパイラには intel C/C++ Compiler を用い，コンパイルオプションには，並列化および時間計測のために“-openmp”，最適化レベルを“-O3”に指定，精度に影響しない最適化を行うために“-fp-model-precise”，倍々精度演算では FMA 演算を用いるため“-fma”，コンパイラの自動ベクトル化を抑制するために“-no-vec”を用いた．

MIC のプログラミングモデルのうち，本研究ではノードの演算性能を評価するため MIC 上で直接プログラムを実行する Native Model で行った(オプション：-mmic).

MIC は演算実行時の環境変数により性能が大きく変動するため，本論文において断りがない場合は並列化に関する環境変数は“KMP_PLACE_THREADS = 60C,4T”，“KMP_AFFINITY = compact”，“OMP_SCHEDULE = static”に統一する．また，MIC はスリープから立ち上げた直後のスレッドで演算を行なうと性能が十分に出ないため演算時はスレッドをスリープさせないために“KMP_BLOCKTIME = infinite”とする．

分析には行列の構造が単調で評価が行いやすい帯行列と実問題より作られた The Univ. of Florida Sparse Matrix

表 2　実行環境

MIC	Intel Xeon Phi 5110P
コア数 クロック周波数 ピーク性能 (倍精度)	60 1.053 [GHz] 1010.88 [GFlops] (1.053[GHz] × 2 × 8(SIMD) × 60[コア])
補正ピーク性能 (倍々精度)	600.21 [GFlops] (19 / (2 × 16) × 1010.88[GFlops])
総 L1 キャッシュ 総 L2 キャッシュ	2 [MB]　(32[KB] × 60[コア]) 30 [MB]　(512[KB] × 60[コア])
メモリ(GDDR5) メモリバンド幅	8[GB] 320 [GB/s]　(ECC 無効時) 140 [GB/s]　(ECC 有効時)
μOS Version MPSS Version	2.6.38.8-g32944d0 2.1.4982-15
プログラミングモデル	Native Model (直接実行)
コンパイラ オプション	Intel C/C++ Compiler ver.13.1.0 -mmic -fma -no-vec -openmp -ipo -O3
環境変数	KMP_BLOCKTIME = infinite KMP_PLACE_THREADS = 60C,4T KMP_AFFINITY = compact OMP_SCHEDULE = static

Collection [12] (フロリダコレクション) のうち nnz が 10^4 から 10^7 の 53 種類の実数かつ正方対称の疎行列を用いた．このとき，N, nnz がすべて異なるものを選んだ．

帯行列 **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，DD_FMA が 19flops であるため，性能 [Flops] ＝19×nnz / time と定義する．また，ここで菱沼らが提案する Corrected peak performance (補正ピーク性能)[4]という考えを適用すると Xeon Phi 5100P における DD_FMA は内部的には倍精度の命令 16 回に対して 19flops の演算をしているため，補正ピーク性能は 19 / (2 × 16) × 1010.88[GFlops] = 600.21[GFlops]となる．

4.2 メモリアクセスの影響

BCRS 形式の DD-SpMV のメモリアクセスの影響を確認するために, 帯幅を 64 に固定して行列サイズ N を 10^3 から 4.0×10^5 まで変化させた帯行列を用いて計測を行った．

BCRS 形式の DD-SpMV の比較対象として，CRS 形式の DD-SpMV と ***x*** のロードに集約関数_mm512_i32logather_pd ()を用いた CRS 形式の DD-SpMV (CRS_gather)を計測した．

MIC がメモリからロード・ストアするデータ量は倍々精度ベクトル(16 バイト) ***x***, ***y***，倍精度の **A**.value，4 バイト整数型の **A**,index, **A**.pointer である．今回用いた帯行列は CRS 形式と BCRS 形式とで計算量がほぼ同じ（BCRS 形式の計算量/CRS 形式の計算量＝1.1）である．メモリバンド幅が 140GB/s であるため，メモリ性能に制約を受けた場合の理

論ピーク性能は 243.4GFlops である．

CRS 形式，CRS_gather，BCRS 形式の DD-SpMV の実行性能を図 9 に示す．このとき，行列サイズ N=3.6×10^4 まではキャッシュ容量に収まる．

データサイズがキャッシュ容量に収まる場合（N=3×10^4）の性能を表 3 に，キャッシュ容量に収まらない場合（N=4.0×10^5）の性能を表 4 に示す．

キャッシュ容量を基準に同じ実装手法を比較した場合，大きな性能の変化はなかった．一方，実装手法の違いによる性能差について CRS 形式を基準に比較すると，

- CRS_gather は SET 命令による CRS 形式の約 1.23 倍
- BCRS 形式では約 1.94 倍

となった．gather 命令を使うことで性能は CRS の 1.2 倍となり，BCRS 形式を用いた場合は gather 命令を用いた場合の 1.6 倍，CRS 形式に対してであれば 1.9 倍の性能となり高い効果が得られた．また，CPU 同様 MIC においても倍精度疎行列と倍々精度ベクトルの積がメモリネックにならないことを示した．

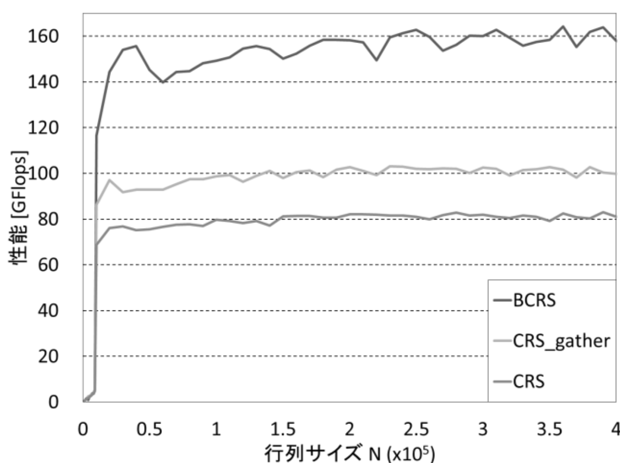

図 9　データサイズの増加に伴う性能変化

表 3 データがキャッシュ容量に収まる場合の性能

	性能 [GFlops]	補正ピーク性能比[%]	メモリ性能比[%]
CRS	76.9	12.8	31.6
CRS_gather	91.7	15.3	37.7
BCRS	154.4	25.7	63.4

表 4　データがキャッシュ容量に収まらない場合の性能

	性能 [GFlops]	補正ピーク性能比[%]	メモリ性能比[%]
CRS	81.0	13.5	33.3
集約 CRS	99.8	16.6	41.0
BCRS	157.8	26.3	64.8

4.3 計算量の違いによる性能

BCRS 形式を DD-SpMV に用いた場合, 0 を含めて計算を行うため疎行列の構造によっては計算量が最大 r×c 倍になる．表 5 は CRS 形式と BCRS 形式の計算量の差が 1.1 倍になるテスト用疎行列と 8 倍になるテスト用疎行列（各非零要素の間に 8 つの 0 をいれた疎行列）の性能を比較したものである．このとき行列サイズ N＝10^5 である．

計算量がほぼ同じ（1.1 倍）場合には BCRS 形式を用いると，CRS 形式を用いた場合の性能を上回り，約 1.9 倍の性能を示した．一方，CRS 形式に対して BCRS 形式の計算量が 8 倍となる場合には性能は BCRS 形式の性能は CRS 形式よりも低く約 0.25 倍の性能を示した．

メモリアクセスを性能に考慮しない場合，計算量が 8 倍になれば性能は 1/8 倍になる．この実験結果からは計算量が 8 倍になる最悪のケースでも性能の劣化は 1/4 程度までに抑えられている．MIC を扱う上で行列の増加によって計算量が増加することになっても端数処理と総和計算がなくメモリアクセスを改善する BCRS 形式の方が性能向上につながると考えられる．

表 5　BCRS 化による計算量増加の最良と最悪のケース

計算量増加率	CRS [GFlops]	BCRS [GFlops]	BCRS/CRS
1.1 倍	81.0	157.8	1.94
8.0 倍	79.1	19.9	0.25

4.4 フロリダコレクションへの適用

フロリダコレクションの問題における CRS 形式と DD-SpMV の性能を図 10 に示す．このとき，横軸は疎行列の列あたりの平均ブロックヒット数 nnz/blk とする．BCRS 形式と CRS 形式の性能を比較すると，BCRS 形式は CRS 形式の性能に対して 0.4~17 倍の性能であった．BCRS 形式の性能が CRS 形式を上回ったのは 53 個の問題中 36 個の問

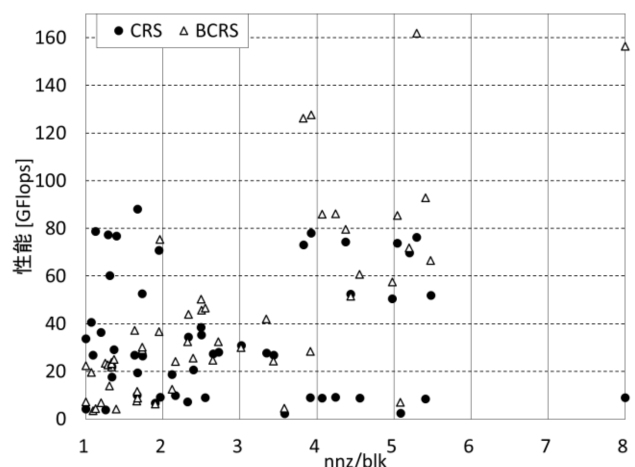

図 10　フロリダコレクションでの CRS と BCRS の比較

題であり，BCRS 形式の性能が必ずしも高い性能を示すわけではなかったため，問題に応じて格納形式を切り替える必要がある．

BCRS 形式の性能は列あたりの平均ブロックヒット数 nnz/blk に依存しており，この値が高いほど BCRS 形式での性能が高い傾向にあった．逆に，この値が低いと CRS 形式の性能を下回る傾向にある．CRS 形式と BCRS 形式とで性能が逆転する点が nnz/blk=1.9 付近であるため，2 種の格納形式を問題に応じて使い分ける際に nnz/blk を用いることで簡易的に自動最適化が出来ると考えられる．

4.5 データの割り当て方による性能差

MIC で並列計算を行なう際に，コアへのスレッドの割り当て方に関する AFFINITY と，スレッドのデータの分割に関する OpenMP のスケージューリング方式 (SCHEDULE) も性能に影響する．演算時に MIC のすべてのコアとスレッドを使用する場合は compact と balanced が同一になるため，本実験では AFFINITY は scatter と compact を，SCHEDULE は dynamic, static, guided を使用し，計 6 種類の組み合わせと比較対象として CRS 形式（compact, static）の DD-SpMV を計測した．それらの性能を図 11 に示す．

AFFINITY はすべての問題において compact が高い性能を示した．SCHEDULE については 53 個の問題中 45 個で static が高い性能を示した．残りの 16 個は dynamic が高い性能を示し，guided が最も高い性能を示すケースはなかった．

nnz/blk≧1.9 のときに static の性能が低い傾向にあり，それらに対しては dynamic を使用することで性能の改善が見られる．その際に dynamic の BCRS 形式と CRS 形式を比較すると CRS 形式が高い性能を示している．

以上より，問題に応じて適切な格納形式を選ぶ際に BCRS 形式についての compact, static の組合せだけを考慮すれば良く，その性能が低いときは CRS 形式を用いればよい．

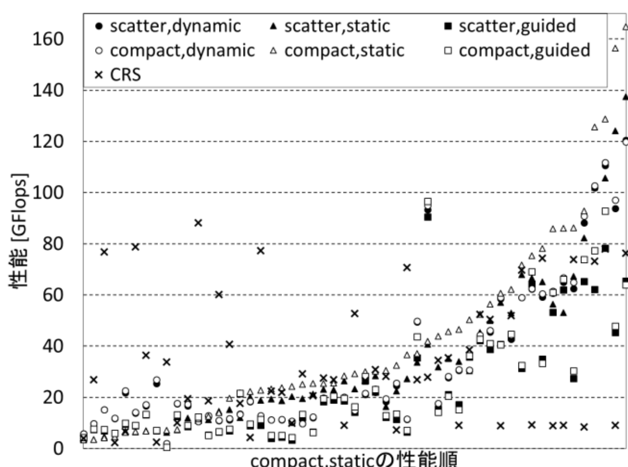

図 11 AFFINITY と SCHEDULE の組合せによる性能比較

5. まとめ

本研究では MIC 上で SIMD 命令を用いて倍精度疎行列と倍々精度ベクトルの積 DD-SpMV($\boldsymbol{y}_{DD}=\mathbf{A}_D\boldsymbol{x}_{DD}$) を CRS 形式と BCRS 形式の 2 種類の疎行列の格納形式で実装，性能の決定要因について分析を行った．

CRS 形式の DD-SpMV では $\boldsymbol{x}$ に関してメモリアクセスが悪く，SIMD 化の弊害として各行で端数処理と総和計算も発生する．BCRS 形式ではメモリアクセスを改善し，CRS 形式で問題であった端数処理と総和計算をなくすことができる．ただし，計算量は CRS 形式に比べ多くなりやすい．

フロリダコレクションを用いて評価すると今回取り扱った 53 個の問題のうち 36 個では BCRS 形式は CRS 形式よりも高い性能であり，最も高いもので 17 倍の性能を示した．BCRS 形式での性能は列あたりの平均ブロックヒット数 nnz/blk に依存している．nnz/blk≧1.9，すなわち BCRS 形式の計算量が CRS 形式の 4 倍以下のとき BCRS 形式の性能が CRS 形式の性能を上回る傾向にある．

BCRS 形式において，異なる AFFINITY と SCHEDULE の組合せについて性能を確認したところ，同一コア内にスレッドが近接してキャッシュヒットしやすくなるため，すべての問題において AFFINITY が compact のとき高い性能を示した．SCHEDULE については 53 個の問題中 39 個で static が高い性能を示し，残りの 12 個は dynamic が高い性能を示した．static の性能が低く，dynamic を使用することで性能の改善されるのはスレッド毎の処理量が不均等な問題である．スレッド毎の処理量が不均等な問題というのは疎行列の非零要素が分散して配置されているため，ブロックヒット数も低くなりやすい．このため，static の性能より dynamic の性能のほうが高い問題の多くは nnz/blk＜1.9 の傾向にあり，dynamic の BCRS 形式を用いても CRS 形式の方が高い性能を示している．

以上より MIC 上で DD-SpMV を解く際は過半数の問題において CRS 形式よりも BCRS 形式の方が高い性能を示した．すべての問題において最適な性能を出すためには使い分けが必要であり，その指標として列あたりの平均ブロックヒット数が 1.9 以上のときに BCRS 形式，1.9 未満のときに CRS 形式とすることで最適な性能が求められる傾向にある．この際に，BCRS 形式の並列化に関わる環境変数の AFFINITY と SCHEDULE は compact と static の組合せだけを考慮すれば良い．

今後の課題として，BCRS 形式に変換するコストよりも低いコストで算出できる使い分けのための指標の見つけることや他の疎行列の格納形式を用いての分析がある．

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

ⓒ 2014 Information Processing Society of Japan

Linear Algebra, 2003.
[2] Bailey, D, H.: A fortran-90 double-double library. http://crd-legacy.lbl.gov/~dhbailey/mpdist/.
[3] 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.
[4] T. Hishinuma, A. Fujii, T. Tanaka, and H. Hasegawa.: AVX acceleration of DD arithmetic between a sparse matrix and vector, Lecture Notes in Computer Science 8384, pp. 622-631, Springer, 2014.
[5] Intel® Xeon Phi™ Coprocessor, http://software.intel.com/en-us/mic-developer.
[6] Bailey, D, H.: High-Precision Floating-Point Arithmetic in Scientific Computation, computing in Science and Engineering, pp. 54–61, 2005.
[7] Knuth, D, E.: The Art of Computer Programming: Seminumerical Algorithms, Vol. 2, Addison-Wesley, 1969.
[8] Dekker,T.: A floating-point technique for extending the available precision, Numerische Mathematik, Vol. 18, pp. 224–242, 1971.
[9] Parallel Programming and Optimization with Intel® Xeon Phi™ Coprocessors, 1st Edition, COLFAX, pp. 91, 2013.
[10] Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, SIAM pp.57-65, 1994.
[11] The University of Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices/.
[12] Intel Xeon Phi Coprocessor High Performance Programming, 1st Edition, MORGAN KAUFMANN, pp.376, 2013.

ⓒ 2014 Information Processing Society of Japan