倍精度BCRS形式疎行列と倍々精度ベクトル積のAVX2による高速計算

菱沼 利彰 (工学院大学)
田中 輝雄 (工学院大学)
長谷川 秀彦 (筑波大学)

目次

1. 研究背景･目的
2. 倍々精度演算
3. AVX2の効果
4. BCRSの効果
5. まとめ

背景

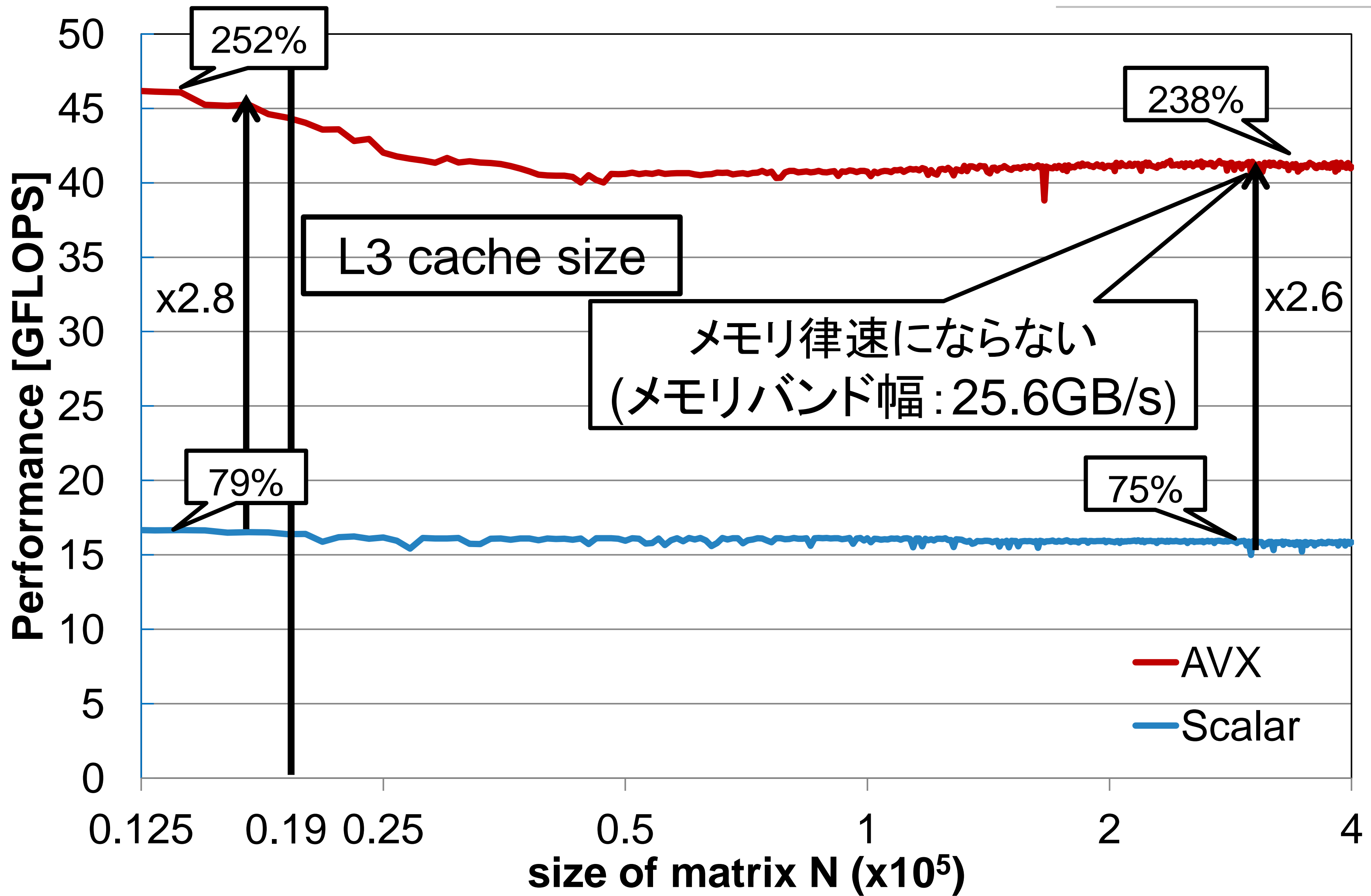
- 倍々精度(DD)演算をAVXを用いて高速化している†
- 倍精度疎行列と倍々精度ベクトルの積：DD-SpMV
  - DD-SpMVは演算器ネック
  - AVXを用いたCRS形式による実装では端数処理などが必要

さらなる性能向上のために

1. Fused-Multiply-and-Add(FMA)命令の使用
   - DD乗算アルゴリズムの演算量を減らせる
2. 疎行列の格納形式BCRS形式
   - ブロックサイズAVXに合わせれば端数処理を無くせる
   - ランダムアクセスを削減できる
   - しかし，データ量，演算量が増える

†菱沼，他，AVXを用いた倍々精度疎行列ベクトル積の高速化，第3回多倍長フォーラム

AVX化の効果(帯行列，帯幅32)



目的

1. AVX2を用いたDD-SpMVの高速化
   - 4つの倍精度演算にFMA命令を同時実行できる
   - FMA命令を用いればDD乗算のアルゴリズムの命令数を削減可能

2. BCRS形式 DD-SpMVの評価
   - AVX2向けの最適なブロックサイズの評価
   - 端数処理などの削減効果
   - 演算量増加による効果

目次

1. 研究背景･目的
2. 倍々精度演算
3. AVX2の効果
4. BCRSの効果
5. まとめ

DD-SpMVに必要な演算カーネル

1. DD += D × DD (DD_ADD_MULT)
2. DD = DD + DD (DD_ADD)

- DD演算は倍精度演算の組み合わせから成る
- FMA命令は，DD乗算アルゴリズムの計算量を削減可能
  - FMA命令は積の結果を誤差なく加算に利用可能

DD_ADD_MULTの命令数

	add命令	mult命令	fma命令	合計
FMAなし	25	8	0	33 命令
FMAあり	14	1	2	17 命令

DD_ADDの命令数

	add命令	mult命令	fma命令	合計
FMAあり，なし	11	0	0	11 命令

目次

1. 研究背景・目的
2. 倍々精度演算
3. AVX2の効果
4. BCRSの効果
5. まとめ

DD-SpMVに用いるAVX2のロード・ストア命令

- set命令：av = _mm256_set_pd(a, b, c, d)
  - 結果：レジスタ av = {a, b, c, d}　非連続データのロード
- load命令：av = _mm256_set_pd(x)
  - 結果：レジスタ av = {x[1], x[2], x[3], x[4]}　連続データのロード
- broadcast命令：_mm256_broadcast_pd(a)
  - 結果：レジスタ av = {a, a, a, a}　1データをブロードキャスト
- store命令：構文：_mm256_store_pd(a, av)
  - 結果：avのデータをアドレスaから連続にストア

avはAVXレジスタ
a,b,c,dは倍精度の変数
xは倍精度の配列

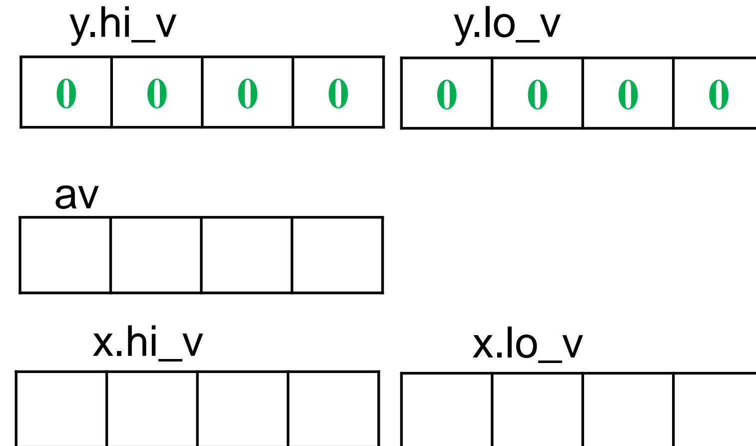
CRS形式の生成 (1-4行目)

y_hi	y_lo		A									x.hi	x.lo
0	0		1	0	4	3	0	6	0	5		1	0
0	0		0	2	0	0	0	0	0	0		2	0
0	0		3	0	5	0	1	4	0	0		3	0
0	0		3	0	0	6	0	0	0	0		4	0
0	0	=	0	0	4	0	2	0	0	0	×	5	0
0	0		2	0	3	0	0	1	0	0		6	0
0	0		0	0	0	0	0	0	4	0		7	0
0	0		0	0	0	0	0	0	0	5		8	0

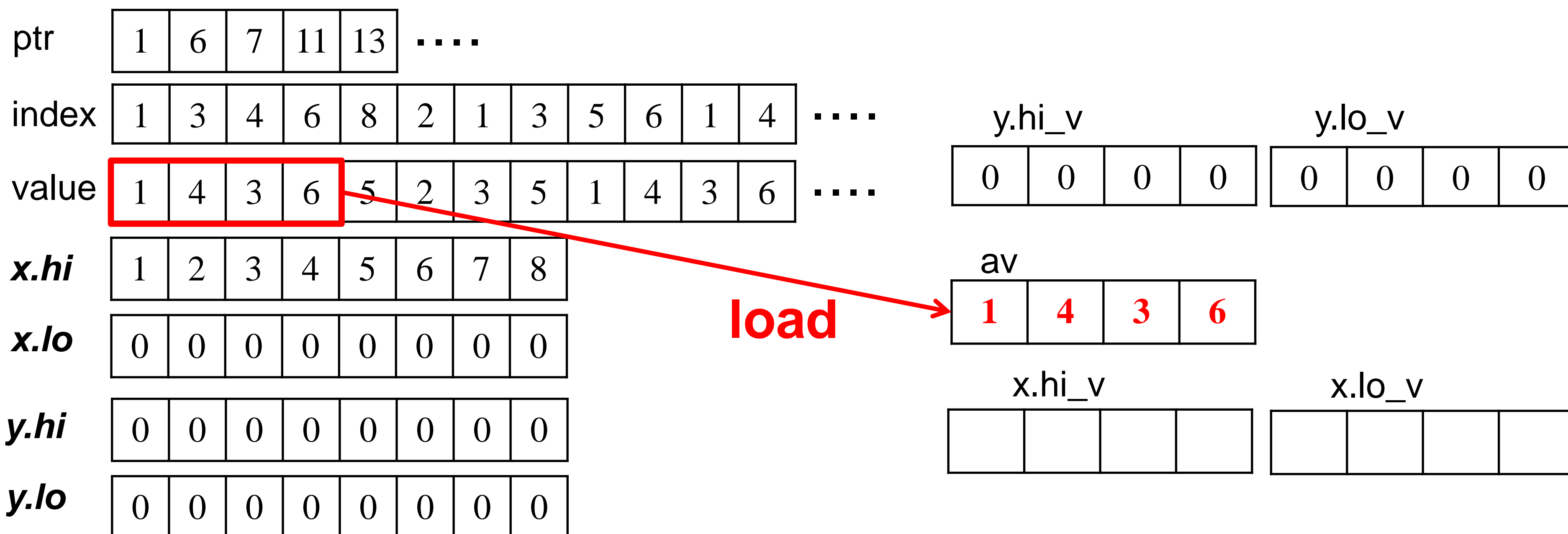
type	name													
int	ptr	1	6	7	11	13	·······							
int	index	1	3	4	6	8	2	1	3	5	6	1	4	·······
double	value	1	4	3	6	5	2	3	5	1	4	3	6	·······
double	*x.hi*	1	2	3	4	5	6	7	8					
double	*x.lo*	0	0	0	0	0	0	0	0					
double	*y.hi*	0	0	0	0	0	0	0	0					
double	*y.lo*	0	0	0	0	0	0	0	0					

yの初期化 (i=1)

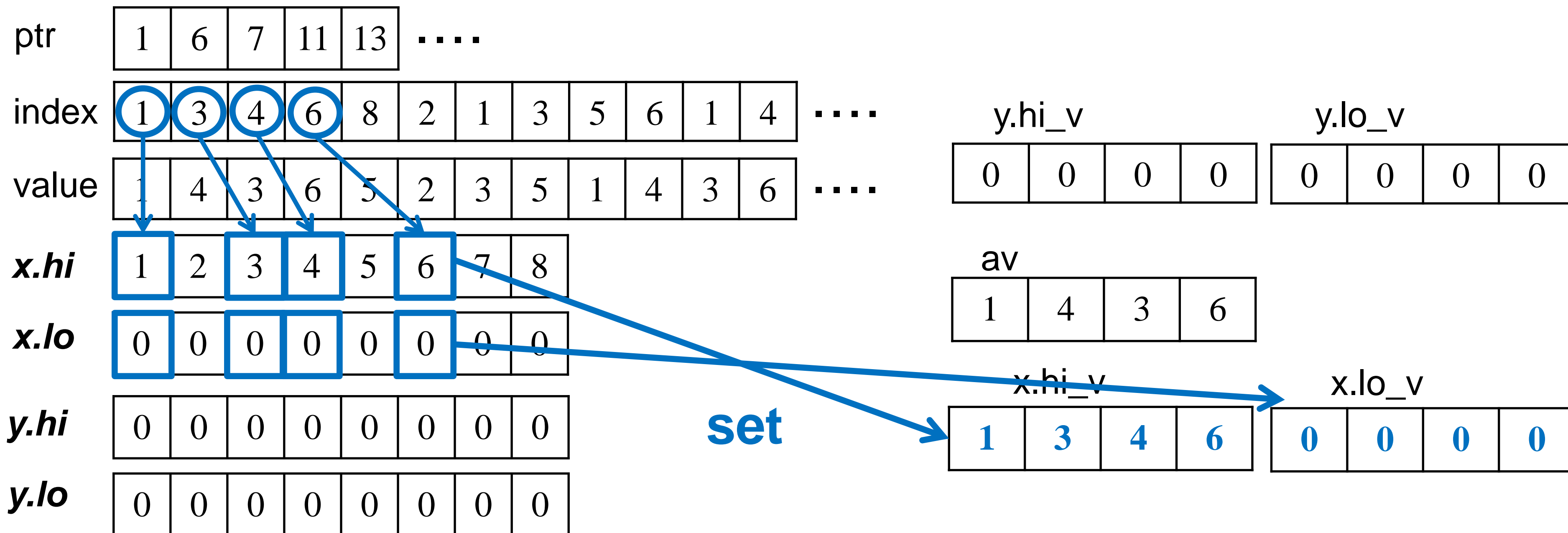
ptr	1	6	7	11	13	····							
index	1	3	4	6	8	2	1	3	5	6	1	4	····
value	1	4	3	6	5	2	3	5	1	4	3	6	····
x.hi	1	2	3	4	5	6	7	8					
x.lo	0	0	0	0	0	0	0	0					
y.hi	0	0	0	0	0	0	0	0					
y.lo	0	0	0	0	0	0	0	0					



Aのロード(j=1-4)



xのロード(j=1-4)



倍々精度積和演算

ptr	1	6	7	11	13	….							
index	1	3	4	6	8	2	1	3	5	6	1	4	….
value	1	4	3	6	5	2	3	5	1	4	3	6	….
x.hi	1	2	3	4	5	6	7	8					
x.lo	0	0	0	0	0	0	0	0					
y.hi	0	0	0	0	0	0	0	0					
y.lo	0	0	0	0	0	0	0	0					

y.hi_v

1	12	12	36

y.lo_v

0	0	0	0

+=

av

1	4	3	6

DD_ADD_MULT

×

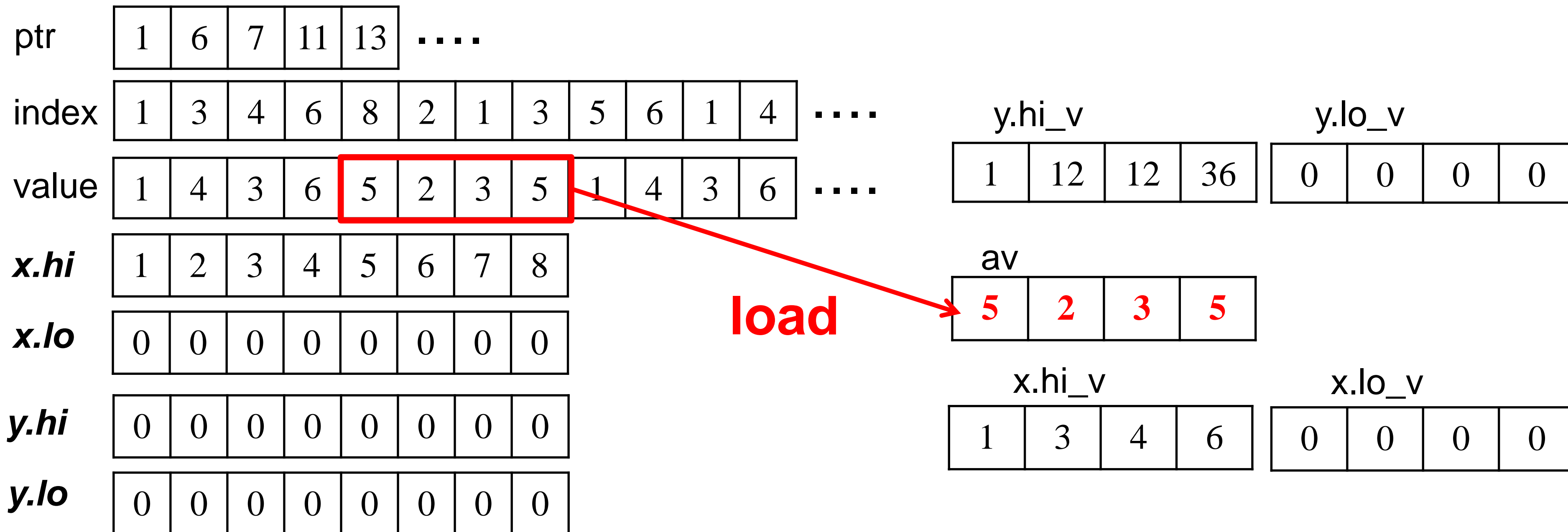
x.hi_v

1	3	4	6

x.lo_v

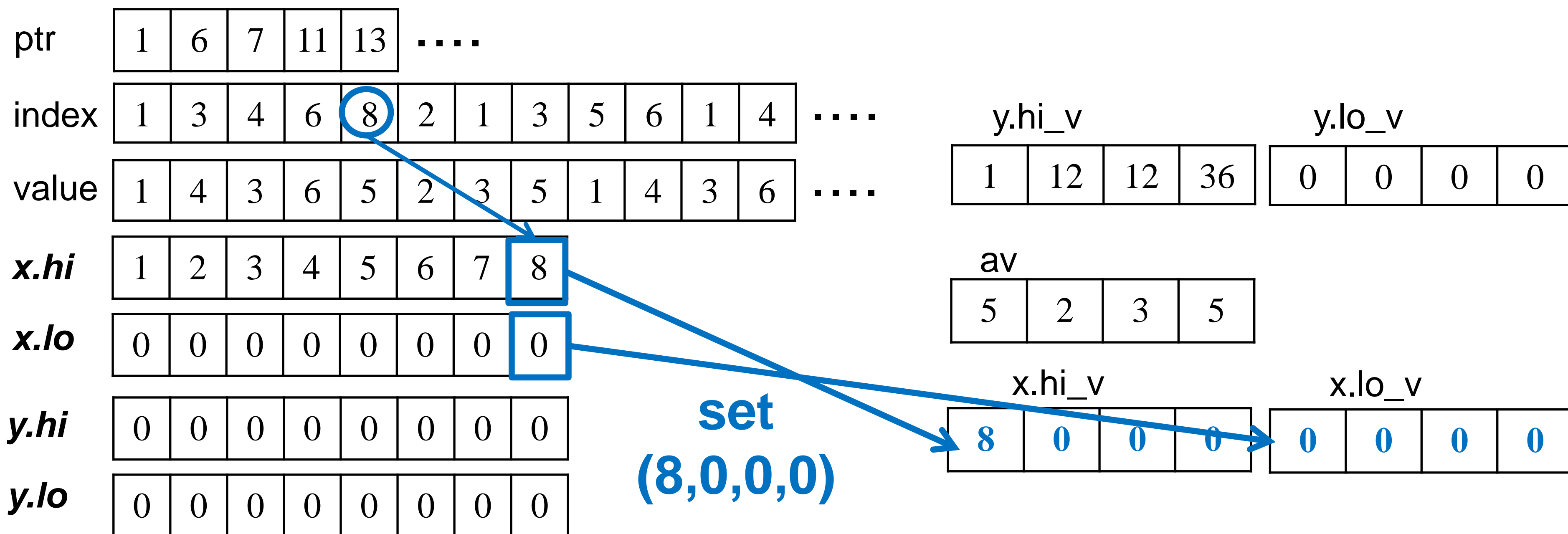
0	0	0	0

Aのロード(j=5,端数処理)



xのロード(j=5,端数処理)

Kogakuin University



倍々精度積和演算(端数処理)

ptr	1	6	7	11	13	····							
index	1	3	4	6	8	2	1	3	5	6	1	4	····
value	1	4	3	6	5	2	3	5	1	4	3	6	····
x.hi	1	2	3	4	5	6	7	8					
x.lo	0	0	0	0	0	0	0	0					
y.hi	0	0	0	0	0	0	0	0					
y.lo	0	0	0	0	0	0	0	0					

y.hi_v

41	12	12	36

y.lo_v

0	0	0	0

+=

av

5	2	3	5

DD_ADD_MULT

×

x.hi_v

8	0	0	0

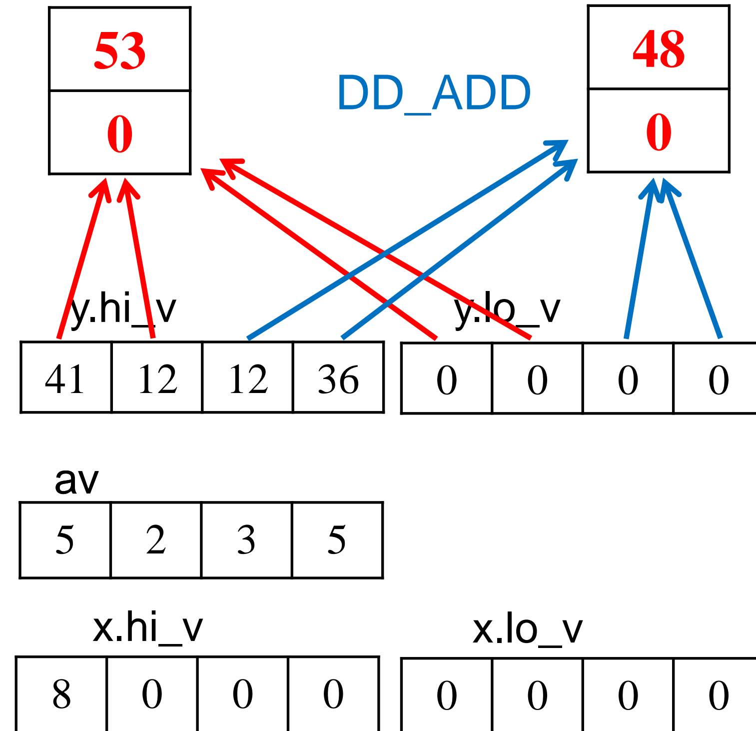
x.lo_v

0	0	0	0

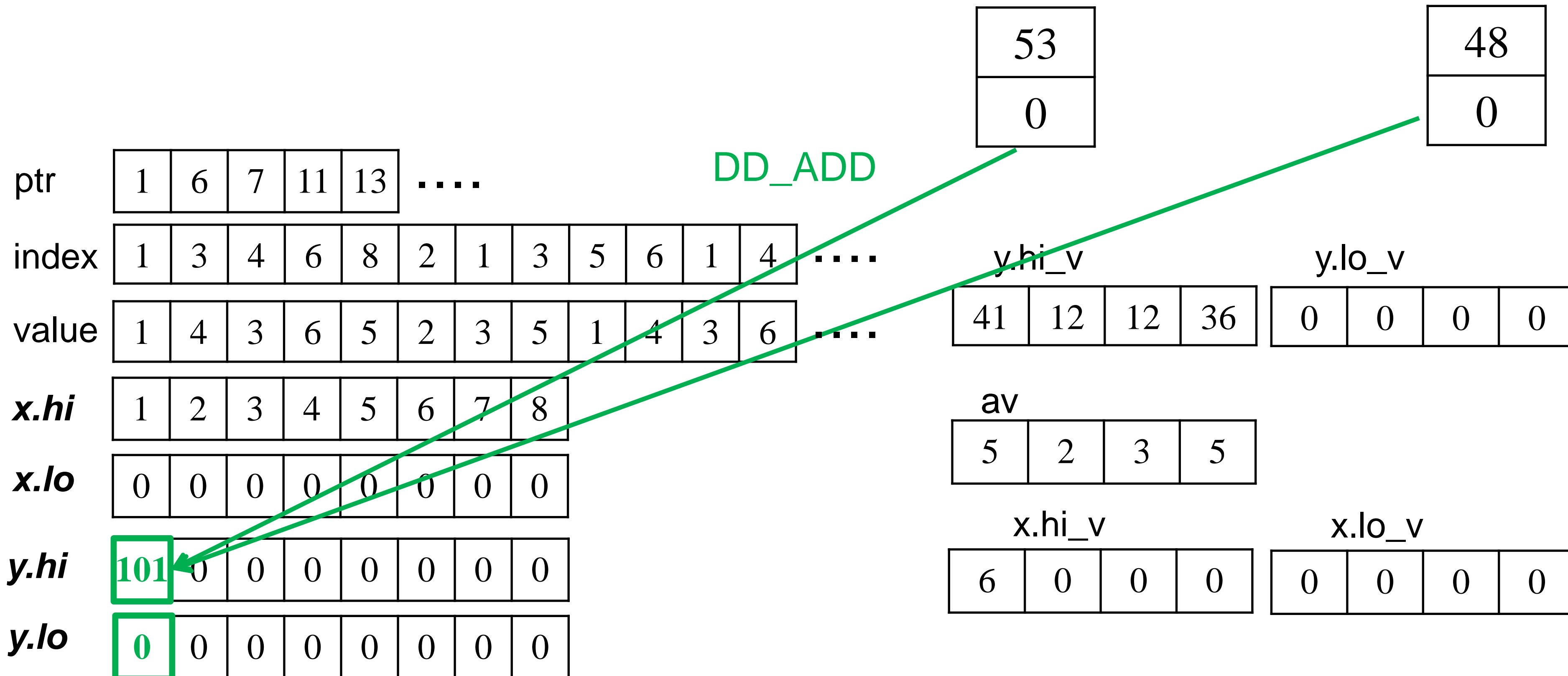
yへの足し込み(リダクション) (i=1)

Kogakuin University

ptr	1	6	7	11	13	….							
index	1	3	4	6	8	2	1	3	5	6	1	4	….
value	1	4	3	6	5	2	3	5	1	4	3	6	….
x.hi	1	2	3	4	5	6	7	8					
x.lo	0	0	0	0	0	0	0	0					
y.hi	0	0	0	0	0	0	0	0					
y.lo	0	0	0	0	0	0	0	0					



yへの足し込み(リダクション) (i=1)



AVXを用いたCRS形式のDD-SpMV (j loop)


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

- set命令(ランダムアクセス)が必要

AVXを用いたCRS形式のDD-SpMV (端数処理)


```
if (A->ptr[i+1] - j == 3){ // 端数3のとき
    av = _mm256_load_pd(&A->value[j]);
    xv = _mm256_set_pd(x[A->index[j]], 0, 0, 0);
    DD_ADD_MULT (av, xv, yv);
}
```

- 各行で最大1回発生
- set命令(ランダムアクセス)が必要
- 同様に端数2, 1についても行う

AVXを用いたCRS形式のDD-SpMV (リダクション)


```
double tmp[4] = {0,0,0,0};
double tmp1= 0, tmp2=0;

_mm256_store_pd(tmp, yv);
tmp1   = DD_ADD(tmp[1], tmp[2]);
tmp2   = DD_ADD(tmp[3], tmp[4]);
y[i]   = DD_ADD(tmp1, tmp2);
```

- 各行で必ず1回発生
- DD_ADD3回(11命令×3)から成る

実験環境

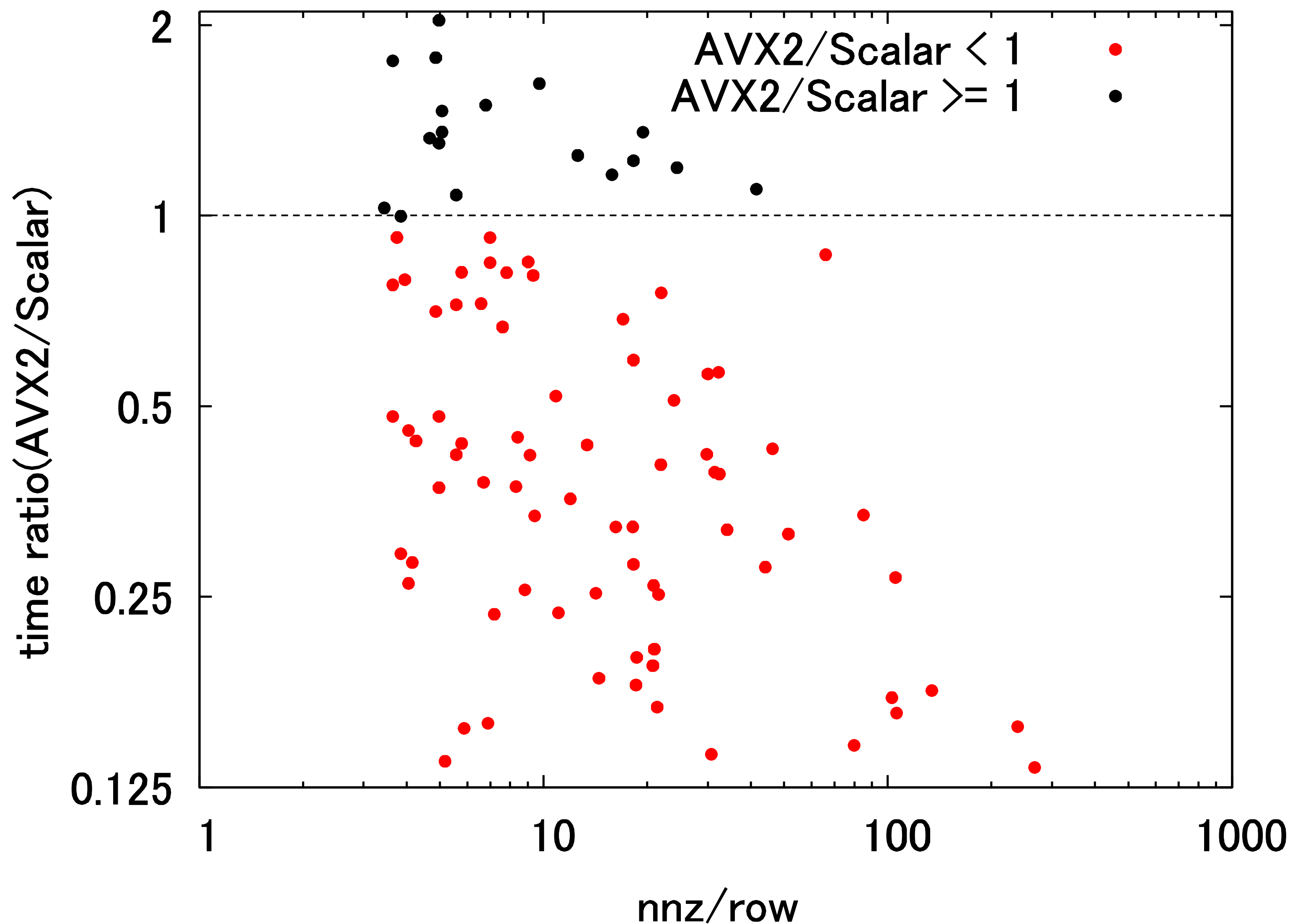
- CPU : intel Core i7 4770K 4core 3.4GHz （AVX2)
  - L3 キャッシュ : 8MB
- メモリ : 16GB (8GB×2 デュアルチャネル)
  - メモリバンド幅 ： 12.8 [GB/s]×2 = 25.6 GB/s
- OS : CentOS 6.4
- コンパイラ : intel C/C++ Compiler 13.0.1
  - Scalar : –O3 **–no-vec** –openmp –fp-model precise (FMAなし)
  - FMAなし : –O3 **–xAVX** –openmp –fp-model precise
  - FMAあり : –O3 **–xCORE-AVX2** –openmp –fp-model precise
- OpenMP スケジューリング方式 : guided (4threads)

実験に用いる疎行列

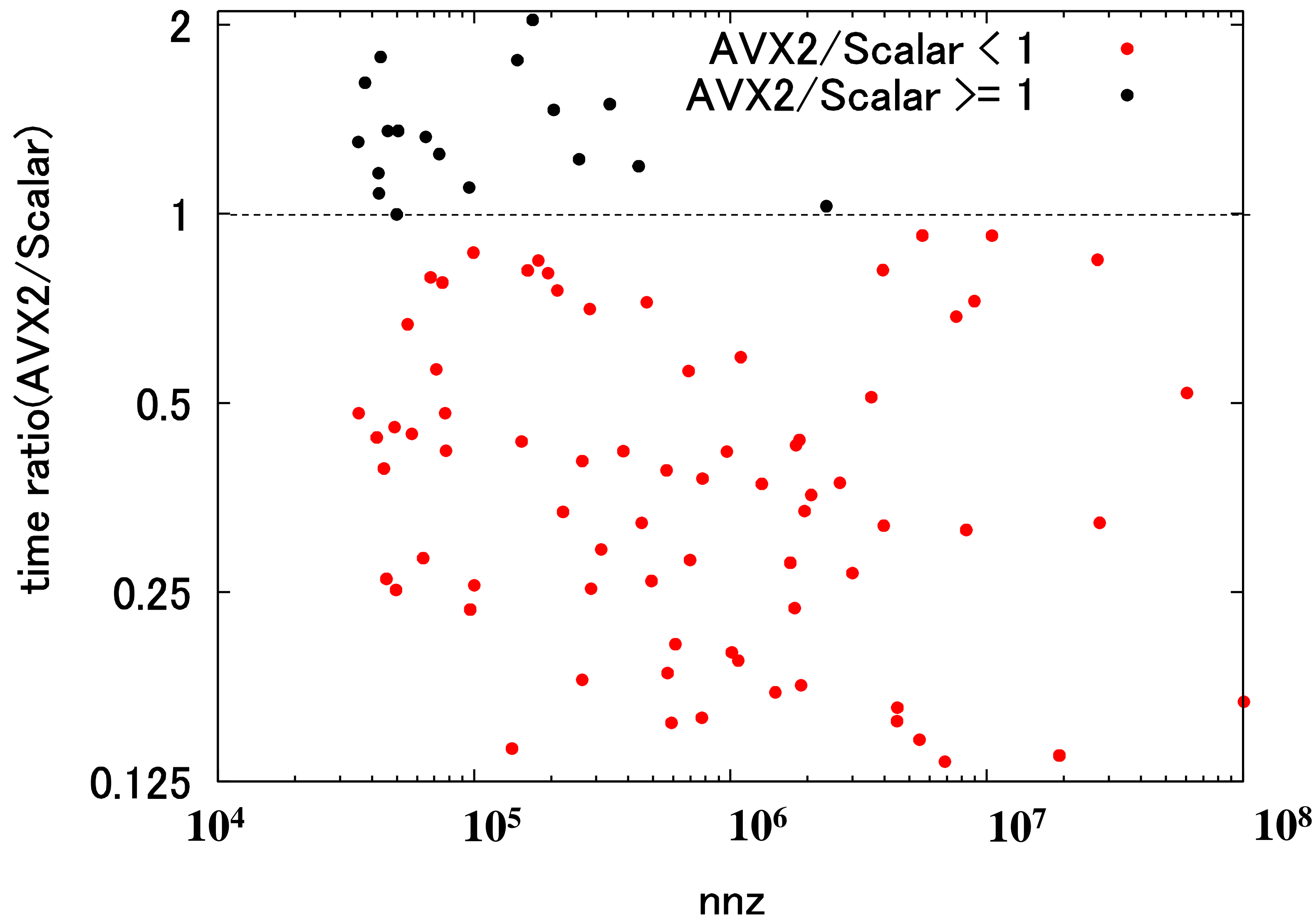
- The Univ of Florida Sparse Matrix Collectionから
  - 行列サイズ1000以上
  - 非対称

を満たす100種の疎行列を選んだ

AVX2の効果 (AVX2とScalarの時間の比)

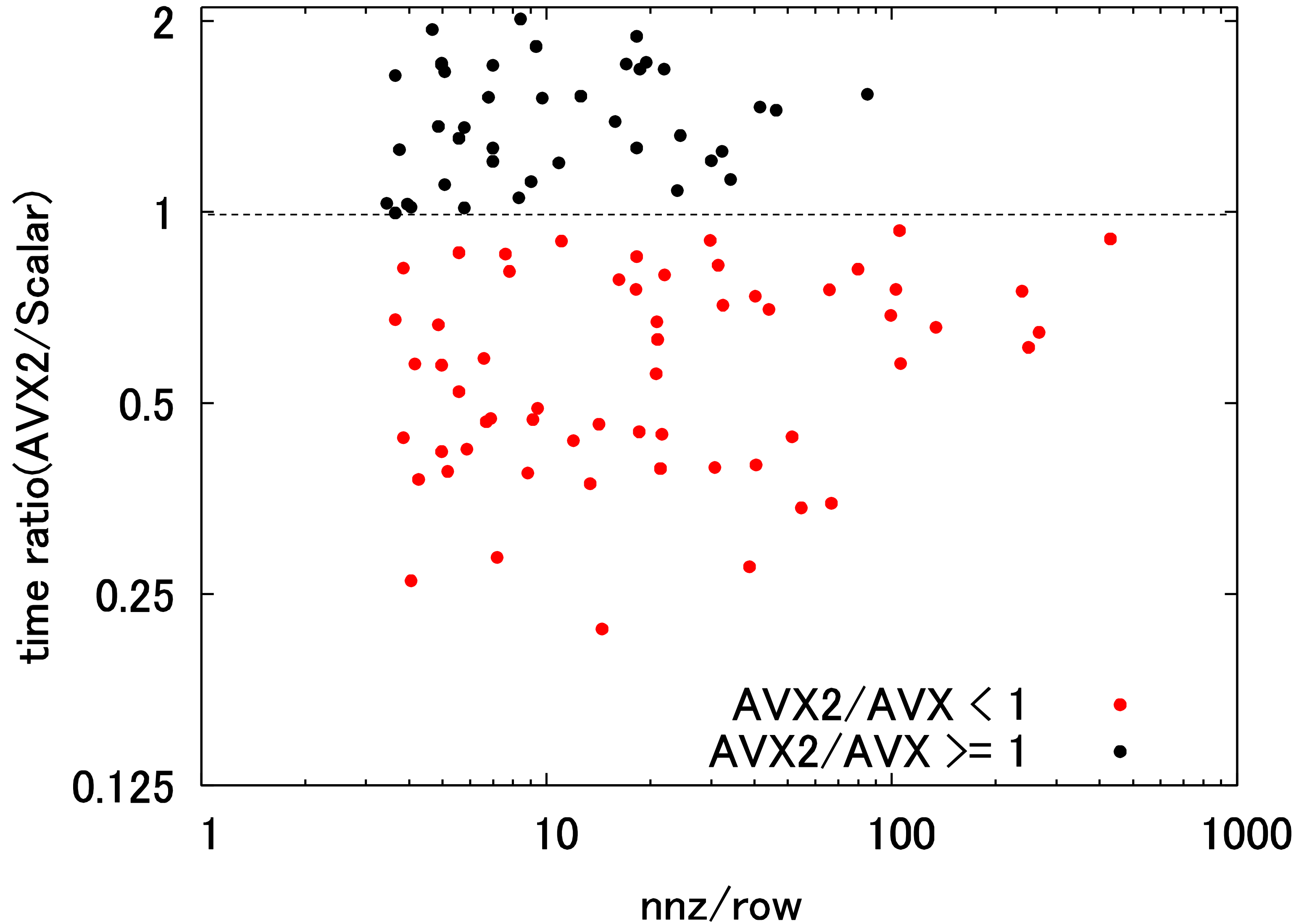


AVX2の効果　サイズの影響



AVXとAVX2の比較

Kogakuin University



AVX2の効果

- AVX2はScalarと比べ
  - 計測時間の比は**0.1-2.1倍**
  - **81/100**問題がScalarより高速
- 大きい問題においてAVX2はScalarより高速
- nnz/rowが小さいときAVX2化の効果が小さい
  - 各行でリダクション，端数処理を必要とするため
- AVX2はAVXと比べ，
  - 実行時間の比は0.2-2.1倍，55/100問題で効果がある

AVX, AVX2, Scalarの組み合わせ

	最も性能が高い問題数	100問の合計時間 [秒]
Scalarのみ	14	2.57 (3.9)
AVXのみ	38	1.07 (1.6)
AVX2のみ	49	0.73 (1.1)
最適な組み合わせ	100	0.66 (1)

()内は比率

- AVX2が最も早いのは49/100問題
- AVX2は，最適な組み合わせと比べ1.1倍
  - AVX2は大きい問題において有効
  - 使い分けの必要性は低い
- AVX2によるDD-SpMVは有効

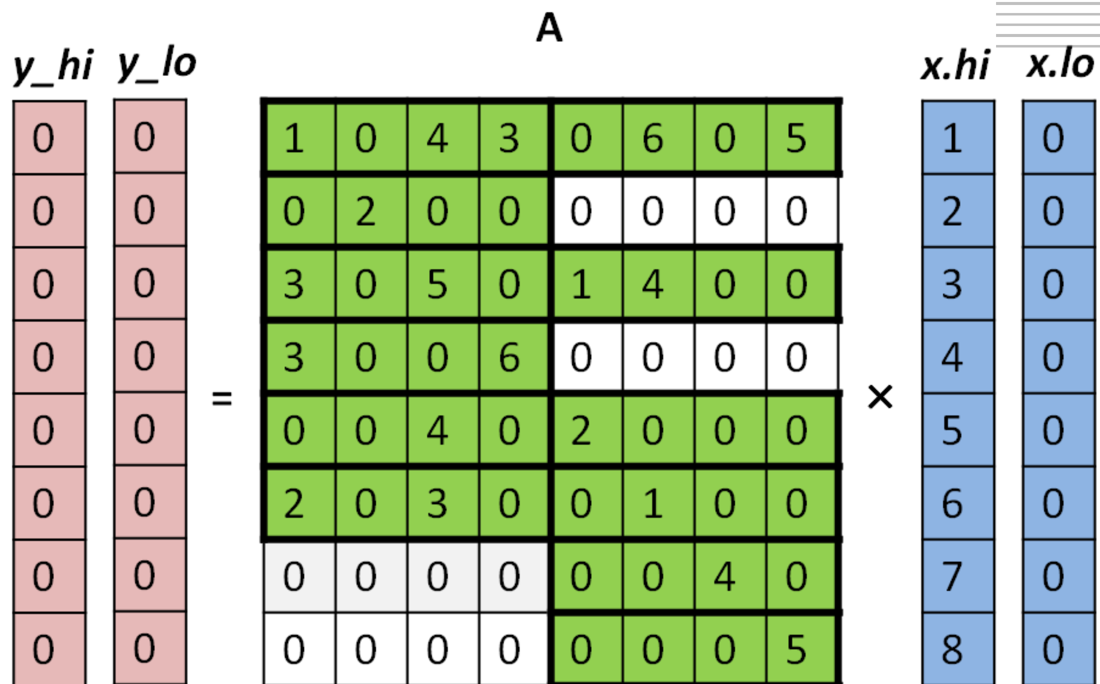
目次

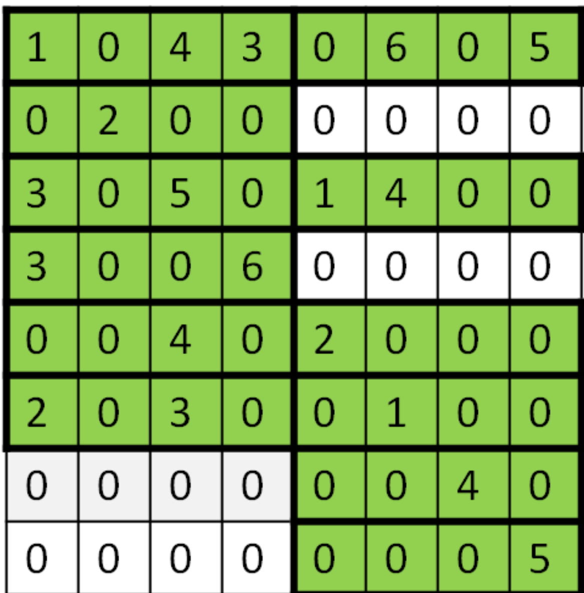
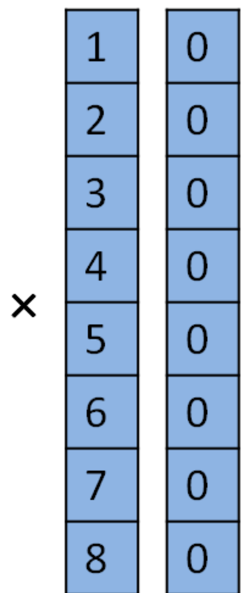
1. 研究背景・目的
2. 倍々精度演算
3. AVX2の効果
4. BCRSの効果
5. まとめ

BCRS形式DD-SpMV

- Block Compressed Row Storage形式
  - 疎行列を長さr×cの小行列にブロック化
    - 小行列は0を含む密行列
- メリット
  - r, cをAVXのベクトル長に合わせれば端数処理がなくせる
  - ベクトル***x***に対するset命令をなくせる
- デメリット
  - 演算数が増加

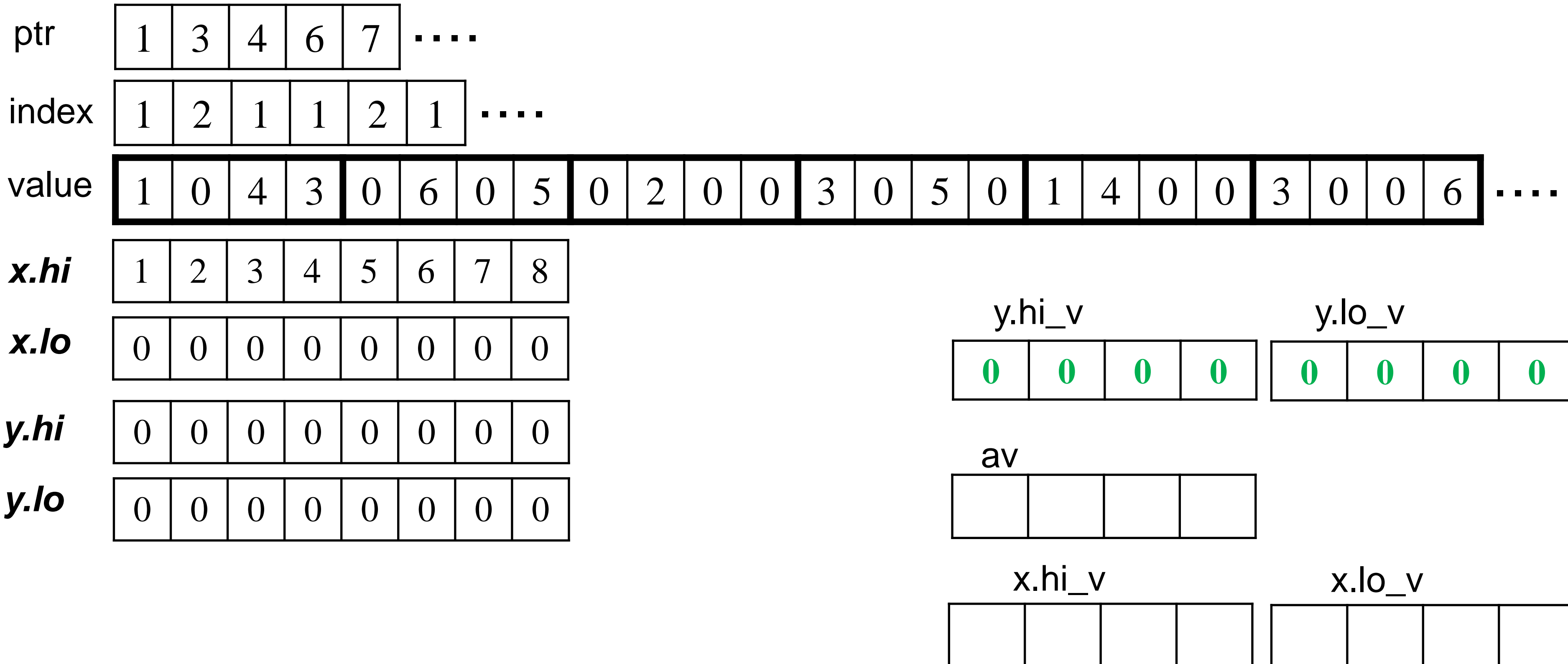
BCRS1x4形式の生成(1-4行目のみ)



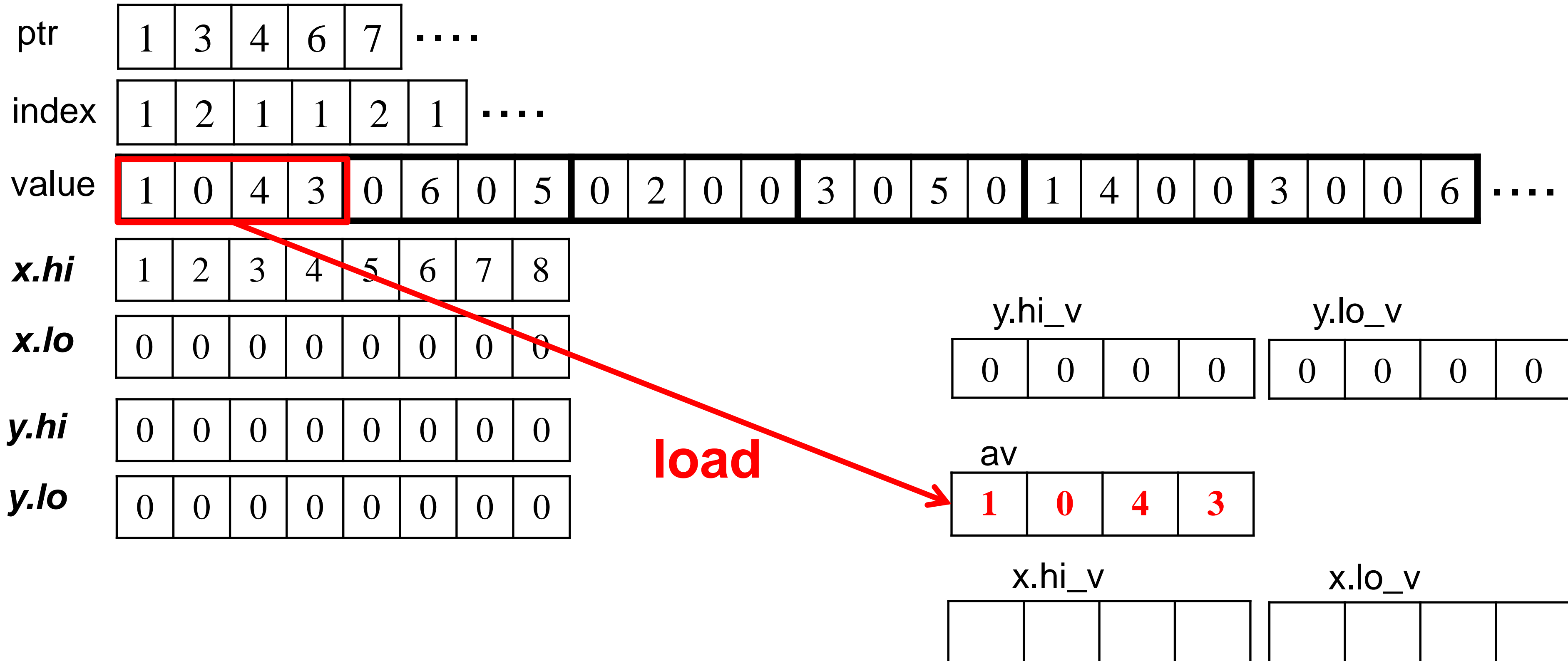



type	name	values
int	ptr	1 3 4 6 7 ....
int	index	1 2 1 1 2 1 ....
double	value	1 0 4 3 0 6 0 5 0 2 0 0 3 0 5 0 1 4 0 0 3 0 0 6 ....
double	*x.hi*	1 2 3 4 5 6 7 8
double	*x.lo*	0 0 0 0 0 0 0 0
double	*y.hi*	0 0 0 0 0 0 0 0
double	*y.lo*	0 0 0 0 0 0 0 0

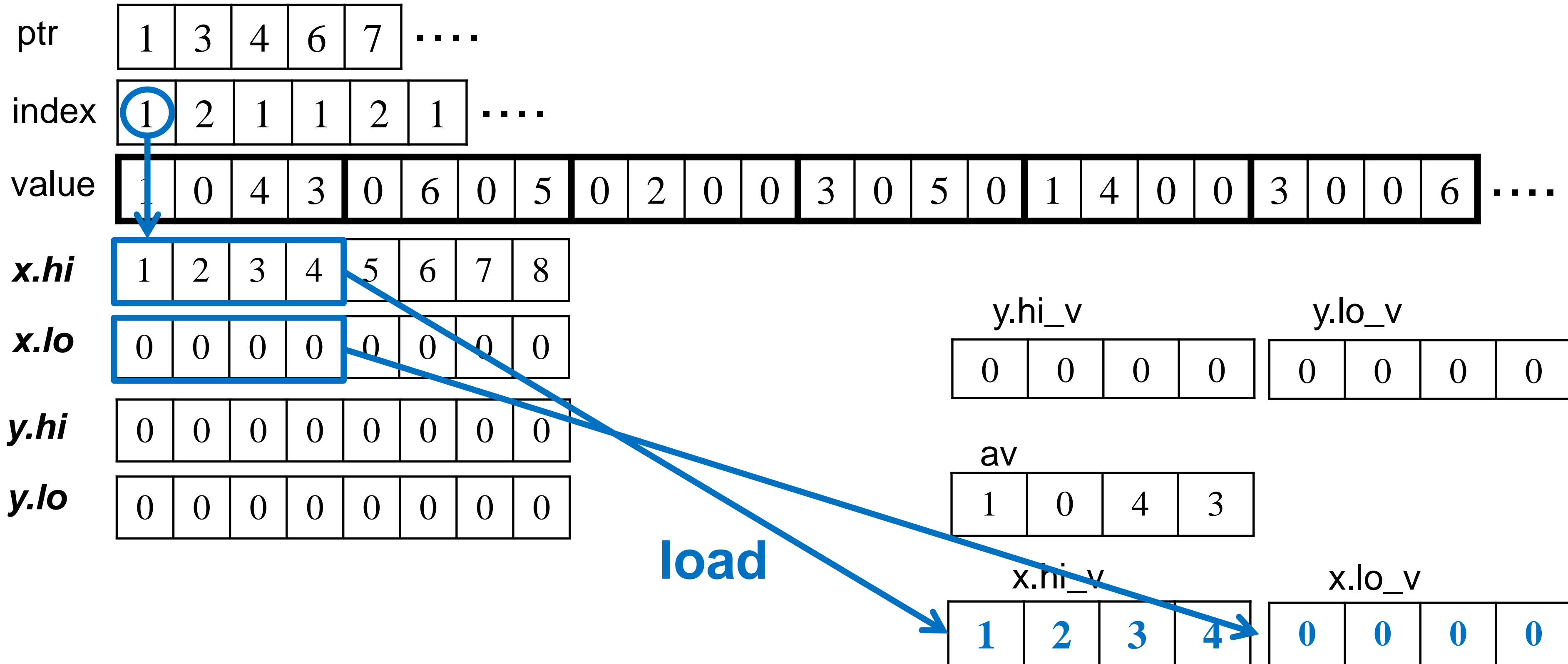
yの初期化(i=1)



Aのロード(ブロック列=1)



xのロード(ブロック列=1)



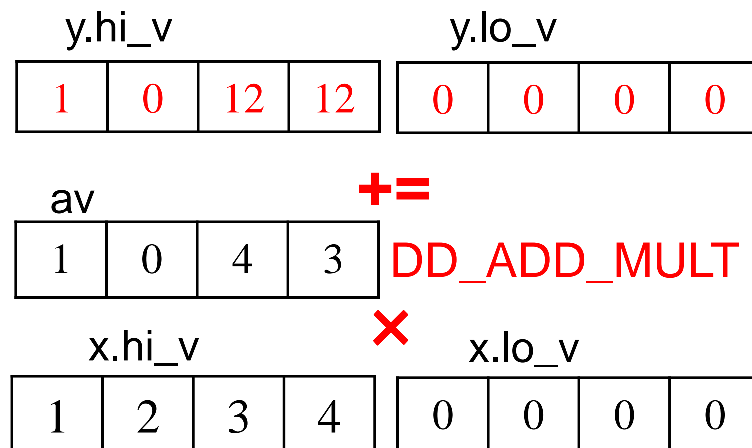
倍々精度積和演算(ブロック列=1)

ptr	1	3	4	6	7	····

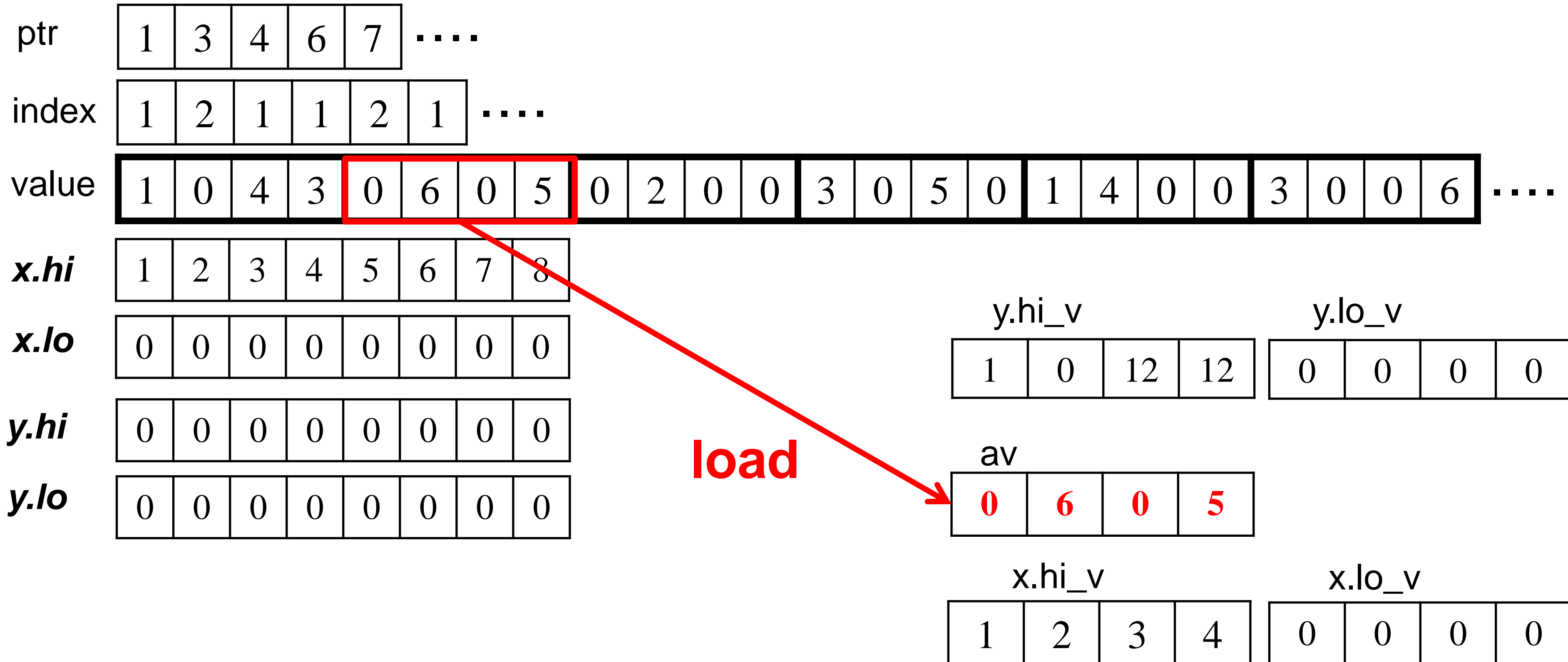
index	1	2	1	1	2	1	····

value	1	0	4	3	0	6	0	5	0	2	0	0	3	0	5	0	1	4	0	0	3	0	0	6	····

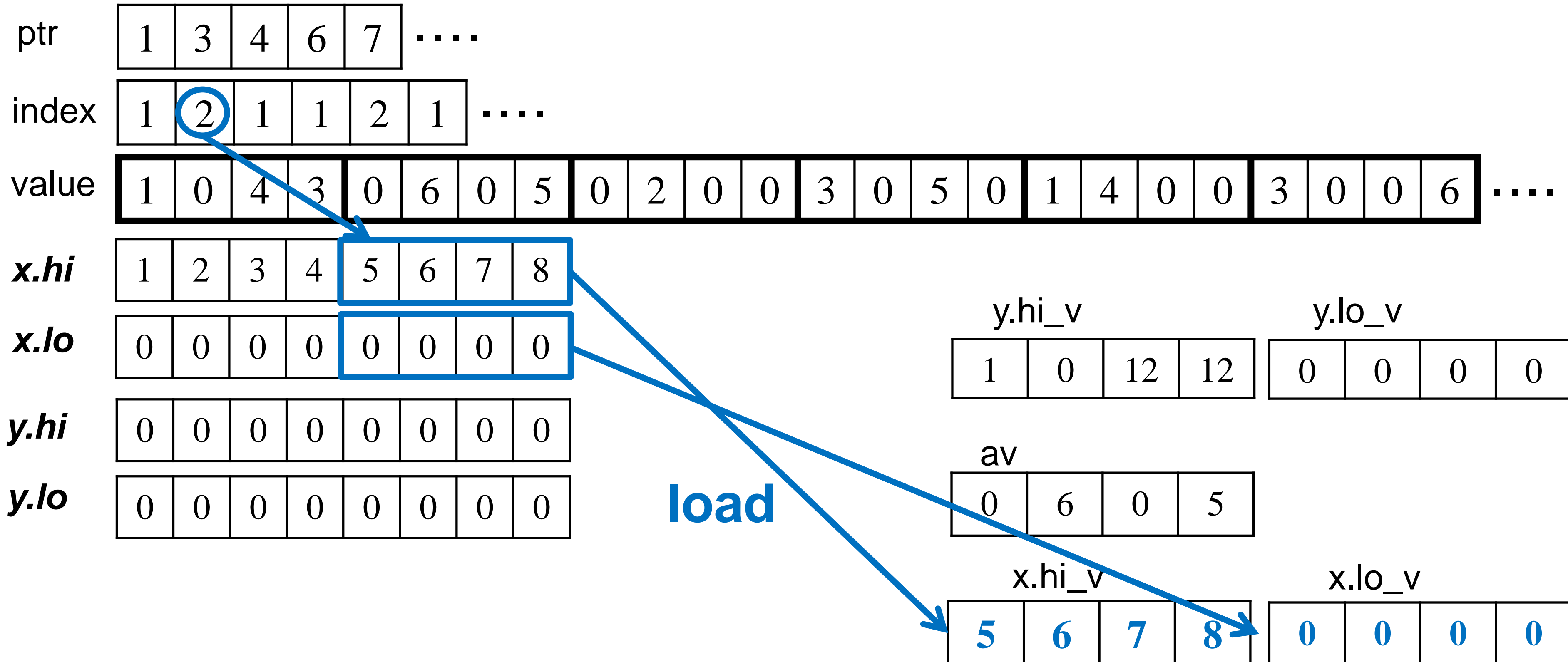
x.hi	1	2	3	4	5	6	7	8
x.lo	0	0	0	0	0	0	0	0
y.hi	0	0	0	0	0	0	0	0
y.lo	0	0	0	0	0	0	0	0



Aのロード(ブロック列=2)



xのロード(ブロック列=2)



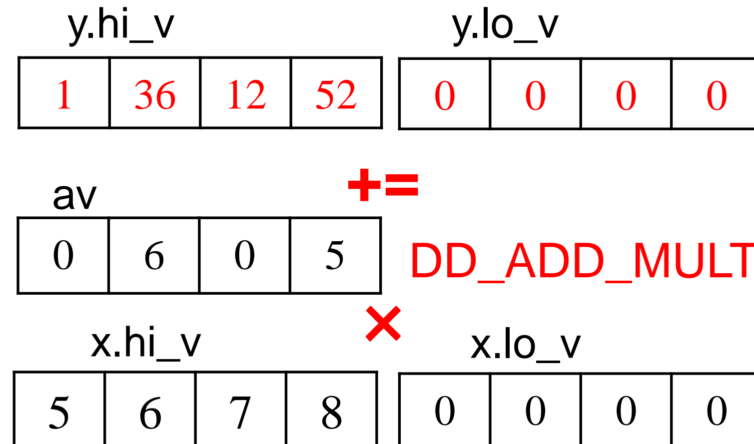
倍々精度積和演算(ブロック列=2)

ptr	1	3	4	6	7	….

index	1	2	1	1	2	1	….

value	1	0	4	3	0	6	0	5	0	2	0	0	3	0	5	0	1	4	0	0	3	0	0	6	….

x.hi	1	2	3	4	5	6	7	8
x.lo	0	0	0	0	0	0	0	0
y.hi	0	0	0	0	0	0	0	0
y.lo	0	0	0	0	0	0	0	0



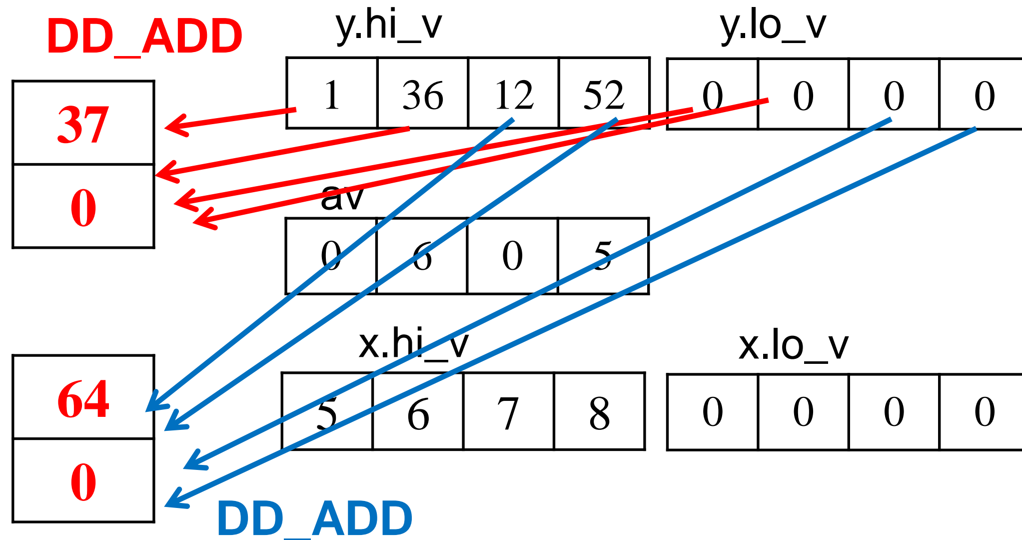
yのリダクション(i=1)

ptr	1	3	4	6	7	….

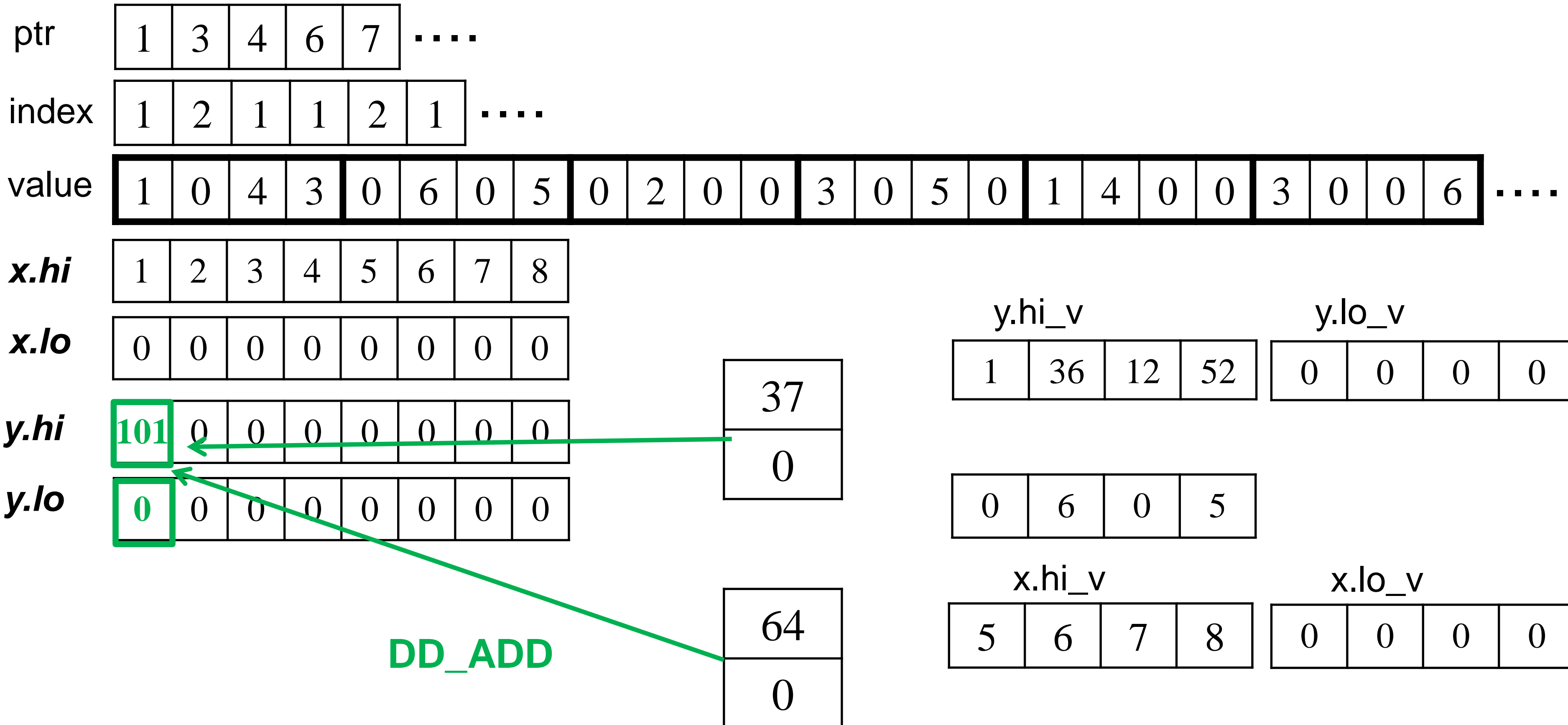
index	1	2	1	1	2	1	….

value	1	0	4	3	0	6	0	5	0	2	0	0	3	0	5	0	1	4	0	0	3	0	0	6	….

x.hi	1	2	3	4	5	6	7	8
x.lo	0	0	0	0	0	0	0	0
y.hi	0	0	0	0	0	0	0	0
y.lo	0	0	0	0	0	0	0	0



yのリダクション(i=1)



AVXを用いたBCRS1x4 DD-SpMV


```
for(i=0;i<N;i++){
        yv = _mm256_zero_sd(&y[i]);
        for(j=A->ptr[i] ; j<A->ptr[i+1] – (4-1) ; j+=4){
                av = _mm256_load_pd(&A->value[j*4]);
                xv = _mm256_load_pd(&x[A->index[j]]);
                DD_ADD_MULT (av, xv, yv);
        }
        reduction();
}
```

- set命令(ランダムアクセスが発生)なし
- 端数処理なし

BCRS4x1形式の生成(1-4行目のみ)

y_hi	*y_lo*		A									*x.hi*	*x.lo*
0	0		1	0	4	3	0	6	0	5		1	0
0	0		0	2	0	0	0	0	0	0		2	0
0	0		3	0	5	0	1	4	0	0		3	0
0	0		3	0	0	6	0	0	0	0		4	0
0	0	=	0	0	4	0	2	0	0	0	×	5	0
0	0		2	0	3	0	0	1	0	0		6	0
0	0		0	0	0	0	0	0	4	0		7	0
0	0		0	0	0	0	0	0	0	5		8	0

type	name	values
int	ptr	1 8 ····
int	index	1 2 3 4 5 6 8 ·····
double	value	1 0 3 3 | 0 2 0 0 | 4 0 5 0 | 3 0 0 6 | 0 0 1 0 | 6 0 4 0 | 5 0 0 0 ····
double	***x.hi***	1 2 3 4 5 6 7 8
double	***x.lo***	0 0 0 0 0 0 0 0
double	***y.hi***	0 0 0 0 0 0 0 0
double	***y.lo***	0 0 0 0 0 0 0 0

yの初期化 (ブロック行=1)

Kogakuin University

ptr

1	8	14	….

index

1	2	3	4	5	6	8	….

value

1	0	3	3	0	2	0	0	4	0	5	0	3	0	0	6	0	0	1	0	6	0	4	0	5	0	0	0

x.hi

1	2	3	4	5	6	7	8

x.lo

0	0	0	0	0	0	0	0

y.hi

0	0	0	0	0	0	0	0

y.lo

0	0	0	0	0	0	0	0

y.hi_v

0	0	0	0

y.lo_v

0	0	0	0

av

x.hi_v

x.lo_v

Aのロード(j=1)

ptr

1	8	14	….

index

1	2	3	4	5	6	8	….

value

1	0	3	3	0	2	0	0	4	0	5	0	3	0	0	6	0	0	1	0	6	0	4	0	5	0	0	0

x.hi

1	2	3	4	5	6	7	8

x.lo

0	0	0	0	0	0	0	0

y.hi

0	0	0	0	0	0	0	0

y.lo

0	0	0	0	0	0	0	0

load

y.hi_v

0	0	0	0

y.lo_v

0	0	0	0

av

1	0	3	3

x.hi_v

x.lo_v

xのロード(j=1)

ptr

1	8	14	・・・・

index

1	2	3	4	5	6	8	・・・・

value

1	0	3	3	0	2	0	0	4	0	5	0	3	0	0	6	0	0	1	0	6	0	4	0	5	0	0	0

x.hi

1	2	3	4	5	6	7	8

x.lo

0	0	0	0	0	0	0	0

y.hi

0	0	0	0	0	0	0	0

y.lo

0	0	0	0	0	0	0	0

y.hi_v

0	0	0	0

y.lo_v

0	0	0	0

av

1	0	3	3

broadcast

x.hi_v

1	**1**	**1**	**1**

x.lo_v

0	**0**	**0**	**0**

倍々精度積和演算(j=1)

ptr

1	8	14	・・・・

index

1	2	3	4	5	6	8	・・・・

value

1	0	3	3	0	2	0	0	4	0	5	0	3	0	0	6	0	0	1	0	6	0	4	0	5	0	0	0

x.hi

1	2	3	4	5	6	7	8

x.lo

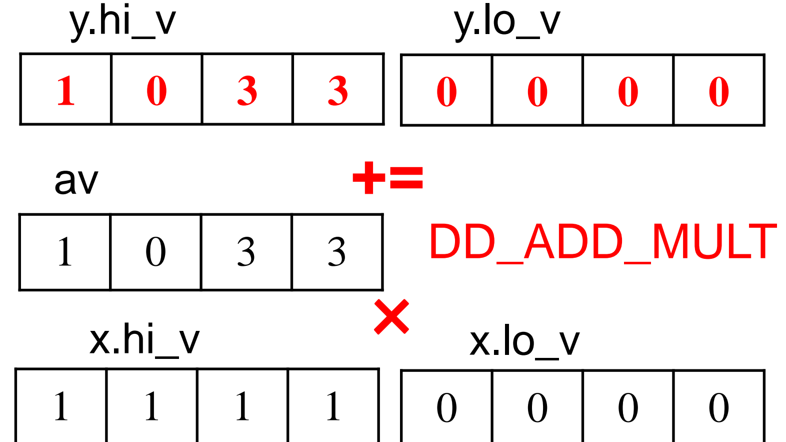
0	0	0	0	0	0	0	0

y.hi

0	0	0	0	0	0	0	0

y.lo

0	0	0	0	0	0	0	0



A,xのロード，倍々精度積和演算 (j=2)

ptr

1	8	14	・・・・

index

1	2	3	4	5	6	8	・・・・

value

1	0	3	3	0	2	0	0	4	0	5	0	3	0	0	6	0	0	1	0	6	0	4	0	5	0	0	0

x.hi

1	2	3	4	5	6	7	8

x.lo

0	0	0	0	0	0	0	0

y.hi

0	0	0	0	0	0	0	0

y.lo

0	0	0	0	0	0	0	0

y.hi_v

1	**4**	**3**	**3**

y.lo_v

0	**0**	**0**	**0**

+=

DD_ADD_MULT

av

0	2	0	0

×

x.hi_v

2	2	2	2

x.lo_v

0	0	0	0

A,xのロード，倍々精度積和演算 (j=3)

ptr

1	8	14	・・・・

index

1	2	3	4	5	6	8	・・・・

value

1	0	3	3	0	2	0	0	4	0	5	0	3	0	0	6	0	0	1	0	6	0	4	0	5	0	0	0

x.hi

1	2	3	4	5	6	7	8

x.lo

0	0	0	0	0	0	0	0

y.hi

0	0	0	0	0	0	0	0

y.lo

0	0	0	0	0	0	0	0

y.hi_v

13	**4**	**18**	**3**

y.lo_v

0	**0**	**0**	**0**

+= DD_ADD_MULT

av

4	0	5	0

×

x.hi_v

3	3	3	3

x.lo_v

0	0	0	0

A,xのロード，倍々精度積和演算 (j=4)

ptr

1	8	14	･･･

index

1	2	3	4	5	6	8	･･･

value

1	0	3	3	0	2	0	0	4	0	5	0	3	0	0	6	0	0	1	0	6	0	4	0	5	0	0	0

x.hi

1	2	3	4	5	6	7	8

x.lo

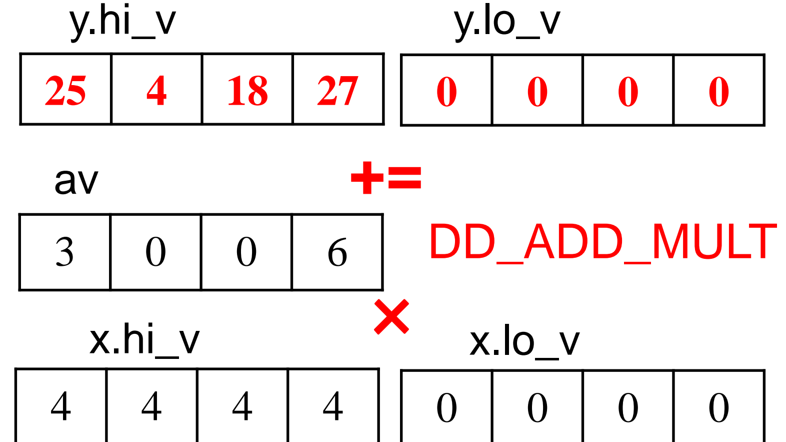
0	0	0	0	0	0	0	0

y.hi

0	0	0	0	0	0	0	0

y.lo

0	0	0	0	0	0	0	0



A,xのロード，倍々精度積和演算 (j=5)

ptr

1	8	14	····

index

1	2	3	4	5	6	8	····

value

1	0	3	3	0	2	0	0	4	0	5	0	3	0	0	6	0	0	1	0	6	0	4	0	5	0	0	0

x.hi

1	2	3	4	5	6	7	8

x.lo

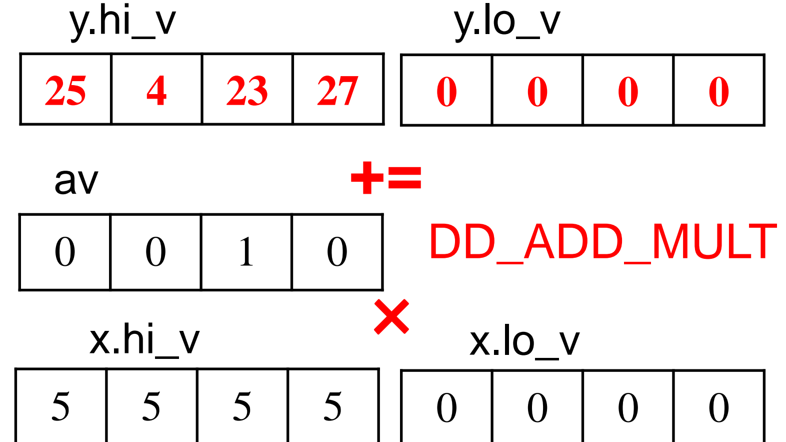
0	0	0	0	0	0	0	0

y.hi

0	0	0	0	0	0	0	0

y.lo

0	0	0	0	0	0	0	0



A,xのロード，倍々精度積和演算 (j=6)

ptr

1	8	14	・・・・

index

1	2	3	4	5	6	8	・・・・

value

1	0	3	3	0	2	0	0	4	0	5	0	3	0	0	6	0	0	1	0	6	0	4	0	5	0	0	0

x.hi

1	2	3	4	5	6	7	8

x.lo

0	0	0	0	0	0	0	0

y.hi

0	0	0	0	0	0	0	0

y.lo

0	0	0	0	0	0	0	0

y.hi_v

61	**4**	**47**	**27**

y.lo_v

0	**0**	**0**	**0**

+= DD_ADD_MULT

av

6	0	4	0

×

x.hi_v

6	6	6	6

x.lo_v

0	0	0	0

A,xのロード，倍々精度積和演算 (j=7)

ptr

1	8	14	････

index

1	2	3	4	5	6	8	････

value

1	0	3	3	0	2	0	0	4	0	5	0	3	0	0	6	0	0	1	0	6	0	4	0	5	0	0	0

x.hi

1	2	3	4	5	6	7	8

x.lo

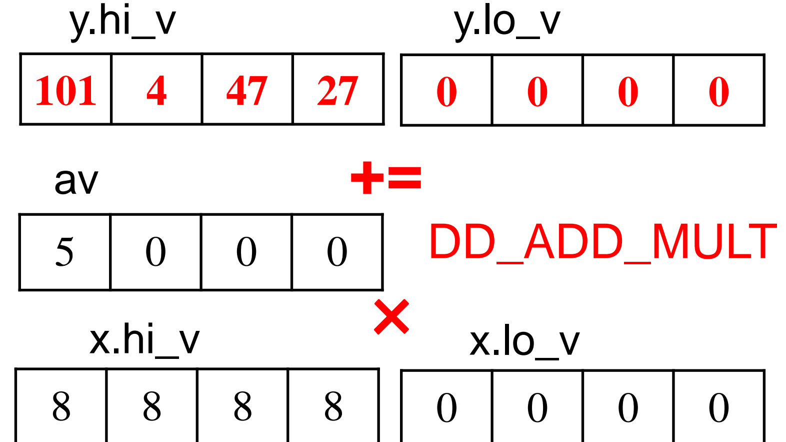
0	0	0	0	0	0	0	0

y.hi

0	0	0	0	0	0	0	0

y.lo

0	0	0	0	0	0	0	0



yのストア(ブロック行=1)

ptr

1	8	14	….

index

1	2	3	4	5	6	8	….

value

1	0	3	3	0	2	0	0	4	0	5	0	3	0	0	6	0	0	1	0	6	0	4	0	5	0	0	0

x.hi

1	2	3	4	5	6	7	8

x.lo

0	0	0	0	0	0	0	0

y.hi

101	4	47	27	0	0	0	0

y.lo

0	0	0	0	0	0	0	0

y.hi_v

101	4	47	27

y.lo_v

0	0	0	0

av

5	0	0	0

store

x.hi_v

8	8	8	8

x.lo_v

0	0	0	0

AVXを用いたBCRS4x1 DD-SpMV


```
for(i=0 ; i<N ; i+=4){
    yv = _mm256_zero_pd(&y[i*4]);
    for(j=A->ptr[i] ; j<A->ptr[i+1] ; j++){
        av = _mm256_load_pd(&A->value[j*4]);
        xv = _mm256_broadcast_sd(&x[A->index[j]*4]);
        DD_ADD_MULT(av, xv, yv);
    }
    _mm256_store_pd(&y[i*4], yv);
}
```

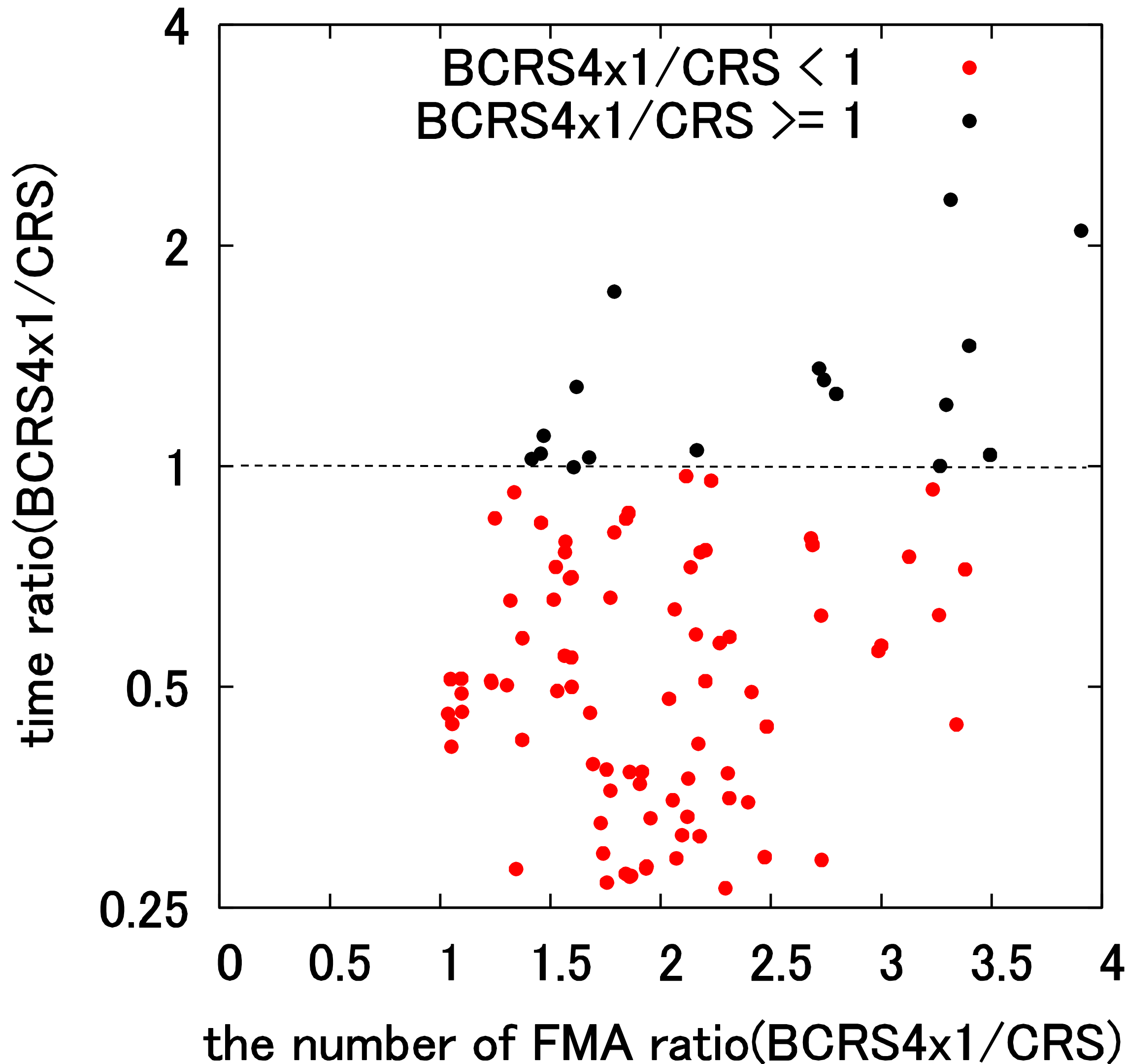
- set命令(ランダムアクセスが発生)なし
- 端数処理なし
- reductionなし

考えられる実装

	xのロード	リダクション	端数処理	yのストア	演算量の増加
CRS	set	あり	あり	各行	x1
1x4	load	あり	なし	各行	最大4倍
× 2x2	set	あり	なし	2行おき(set命令	最大4倍
◎ 4x1	broadcast	なし	なし	4行おき	最大4倍

- 1x4, 2x2, 4x1は端数処理が不要 (4の倍数)
- 2x2は最も性能が悪いと予測 (実装しない)
  - set命令，リダクションが必要，yのストアにset命令が必要
- AVXにおいて4x1が最適であると予測
  - リダクションが不要, yのストアが4行おきにしか発生しない
- 4x1のアンローリング : (4x1)x4の実装も行った

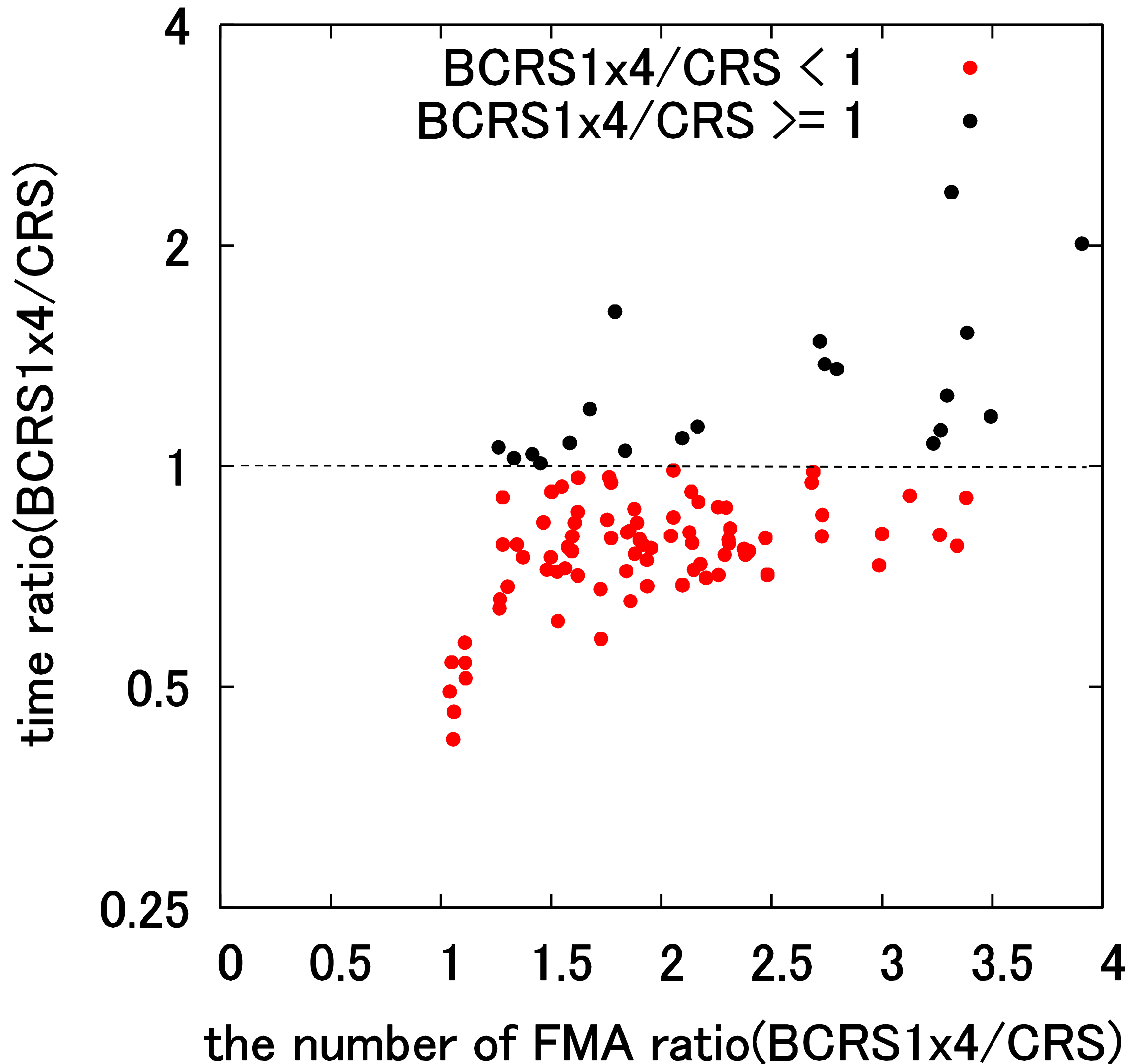
BCRS4x1の性能



最大	2.33
最小	0.27
平均	0.67
1以下	83/100
演算量の増加	最大3.9倍

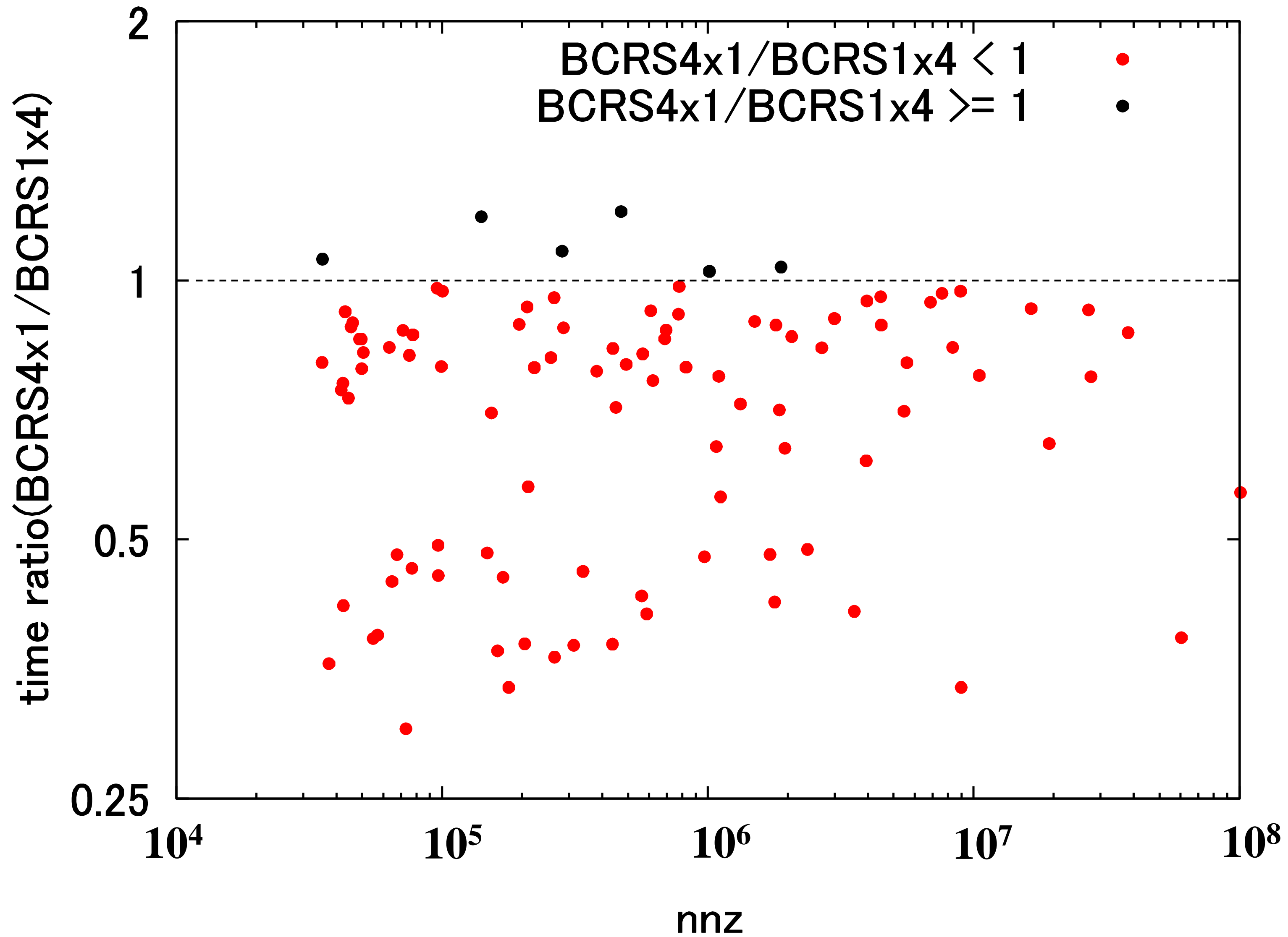
BCRS1x4の性能

Kogakuin University

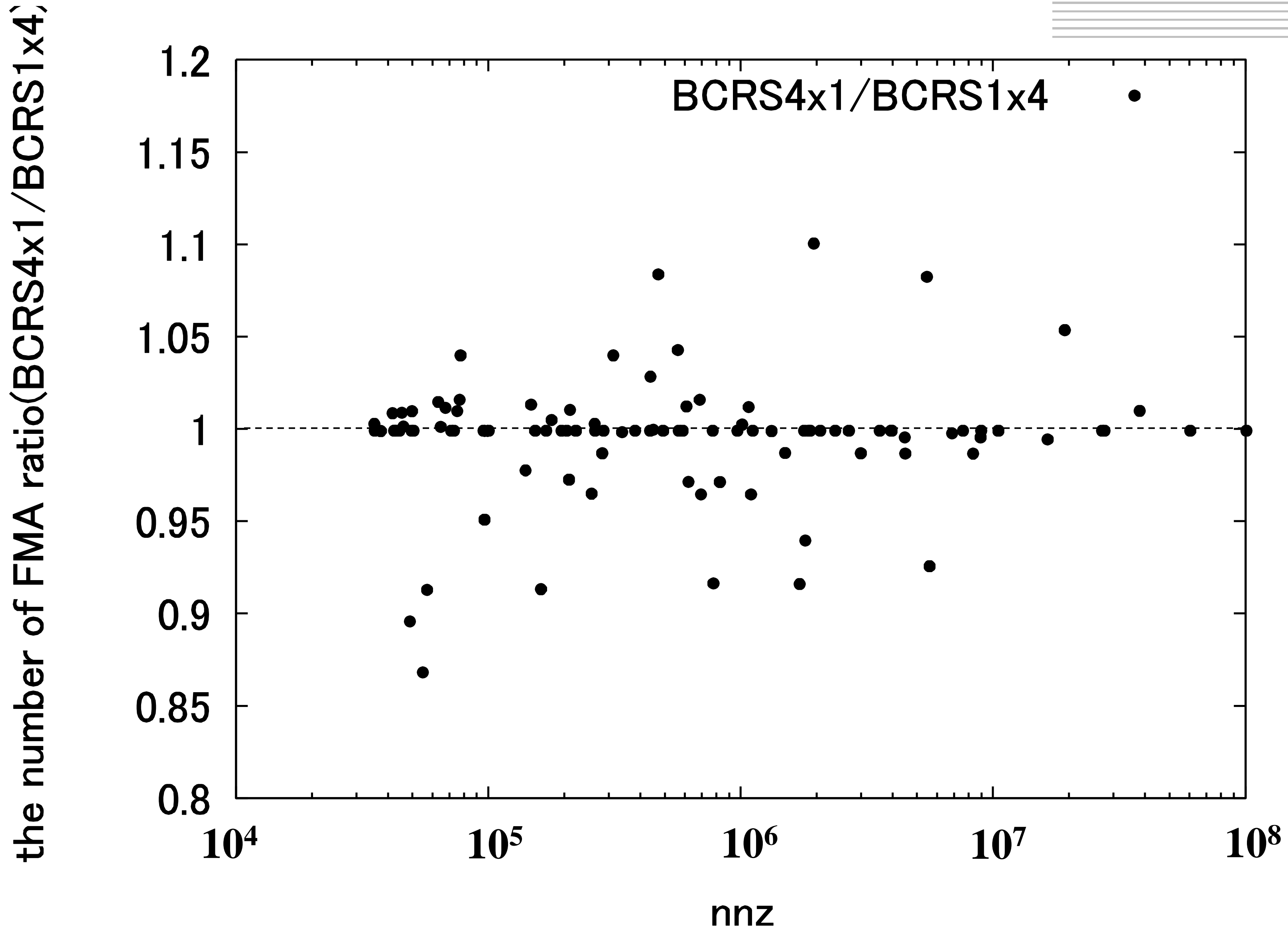


最大	2.38
最小	0.43
平均	0.88
1以下	80/100
演算量の増加	最大3.9倍

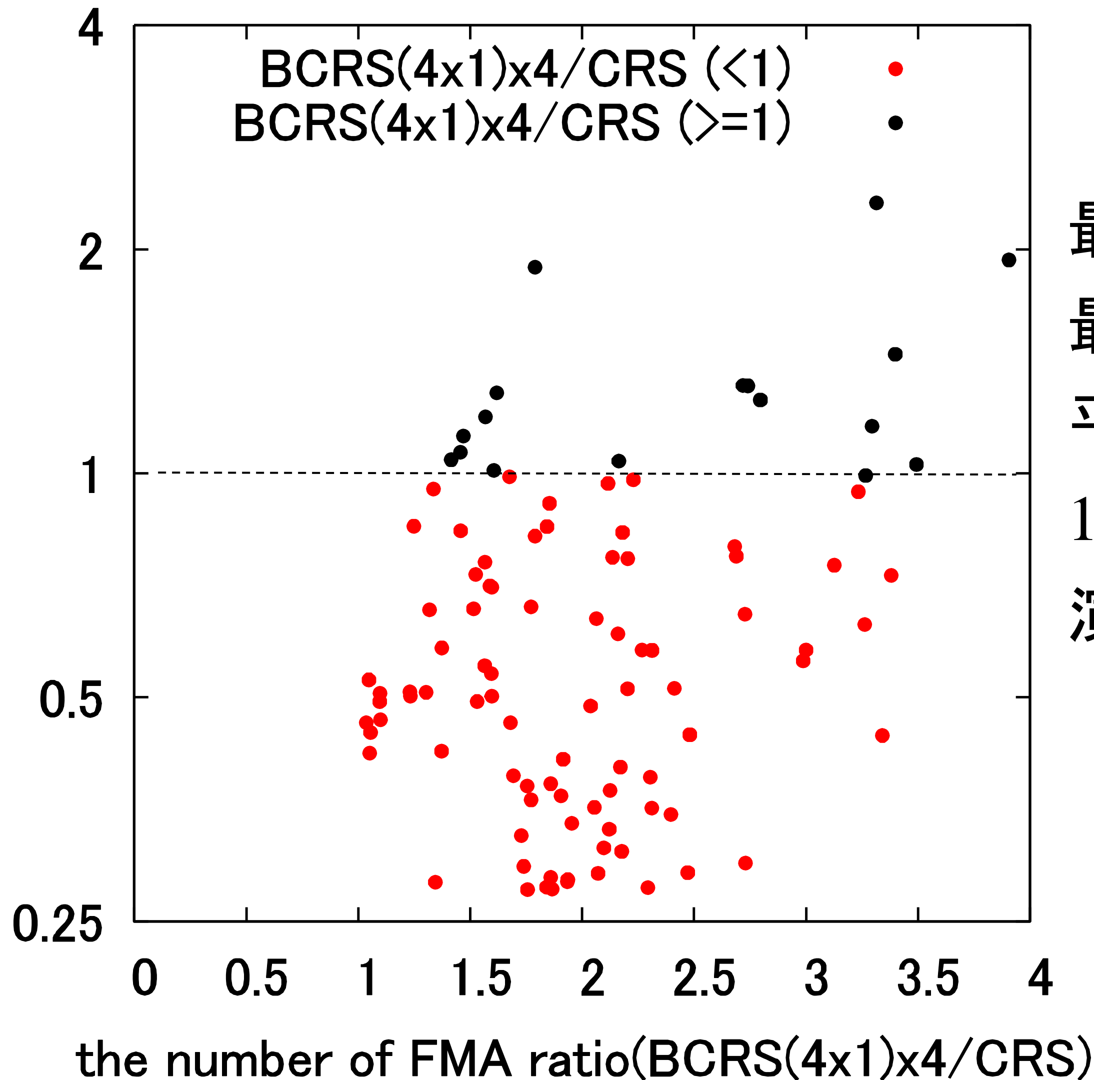
BCRS4x1と1x4の比較



増加した演算量 (BCRS4x1, 1x4)



BCRS(4x1)x4の性能



最大	2.33
最小	0.28
平均	0.68
1以下	83/100
演算量の増加	最大3.9倍

BCRSの効果

	最も性能が高い問題数	100問の合計時間［秒］
CRS	14	0.73 (1.37)
1x4	2	0.88 (1.33)
4x1	70	0.54 (1.01)
(4x1)x4	14	0.55 (1.03)
最適な組み合わせ	100	0.53 (1)

()内は比率

- 4x1は効果的，最適な組み合わせとの比は1.01
- 1x4の効果は小さい
- アンロールの効果はない
  - (4x1)x4は4x1との差が小さく，最適な組み合わせとの比は1.03
- 問題ごとの4x1と1x4の増加した演算量の差は小さい

目次

1. 研究背景･目的
2. 倍々精度演算
3. AVX2の効果
4. BCRSの効果
5. まとめ

まとめ

- AVX2を用いてDD-SpMVを高速化した
  - Scalar, AVX, AVX2の最適な組み合わせと比べ1.1倍
  - AVX2は大きい問題の方が効果が高い
  - AVXと比べて小さい問題では遅いケースがあった

- BCRS形式を用いてDD-SpMVの高速化した
  - 4x1が最も効果的. 各形式の最適な組み合わせとの比は1.01
  - 問題ごとの4x1と1x4の増加した演算量の差は小さい
  - 1x4やアンロールの効果はない

今後の課題

- 小さい問題でのAVX2の性能向上
- 様々なアンロールサイズの検討
  - 8x1, (4x1)xN

- さらなる高データ並列環境，Xeon Phiへの適用
  - Xeon Phiはベクトル長が8
    → 8x1が最良と予想できるが，演算量も最大8倍
  - レジスタを32本もつため，アンロールの効果が異なると予想

参考文献

- Hasegawa, H.: Utilizing the Quadruple-Precision oating-Point Arithmetic Operation for the Krylov Subspace Methods, The 8th SIAM Conference on Applied Linear Algebra (2003).
- 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，第3回多倍長精度計算フォーラム
- Hitoshi Kotakemori, et.al. Prefomance Evaluation of Parallel Sprase Matrix-Vector Products on SGIAltix3700, Lecture Notes in Computer Science 4315, pp. 153-163, Springer, 2008 at IWOMP 2005.
- E. Im, K. Yelick, and R. Vuduc.: SPARSITY: Optimization Framework for Sparse Matrix Kernels, International Journal of High Performance Computing Applications February 2004 18, pp. 135-158 (2004)