AVXを用いた 倍々精度疎行列ベクトル積の高速化

菱沼利彰[†1] 藤井昭宏[†1]
田中輝雄[†1] 長谷川秀彦[†2]

[†1] 工学院大学
[†2] 筑波大学

目次

1. 研究背景･目的
2. 実装，実験環境
3. 実験 -倍々精度ベクトル演算-
4. 実験 -倍々精度疎行列ベクトル積-
5. まとめ

目次

1. 研究背景･目的
2. 実装，実験環境
3. 実験 -倍々精度ベクトル演算-
4. 実験 -倍々精度疎行列ベクトル積-
5. まとめ

研究背景

- 科学技術計算における高精度演算の重要性
  - Krylov部分空間法は丸め誤差が収束に影響
  - 倍々精度演算(≒4倍精度演算)による収束の改善
    - 倍々精度演算には時間がかかる

- CPUの高速化技術：SIMD拡張命令
  - 既存：SSE2 (Pentium4～)(2000年)
  - New：intel AVX (Sandy Bridge~)(2011年)
    - AVXの性能は理論上SSE2の2倍

研究目的

- AVXを用いて行列計算ライブラリを高速化する
- AVXを用いて倍々精度演算を高速化する
  - ベクトル演算 (SWoPP2012)
    - AVX化が性能に与える効果を分析
  - 疎行列ベクトル積(HPCS2013)
    - 疎行列の構造が性能に及ぼす影響を分析
- 倍々精度演算でAVX化の効果を検証する
  - Xeon PhiやHaswellアーキテクチャ
    →SIMD演算の必要性大

目次

1. 研究背景･目的
2. 実装，実験環境
3. 実験 -倍々精度ベクトル演算-
4. 実験 -倍々精度疎行列ベクトル積-
5. まとめ

AVXを用いた実装

- 既存のSSE2コード(Lis)からAVXコードを作成
  - 同時演算数の増加(倍精度2つ→倍精度4つ)
  - 分岐命令の削減
  - 端数処理の増加(1,2,3)

y=ax+y (SSE2 Cコード)

```
for(i=0 ; i<n ; i+=2){
x = load128(vx[i])
y = load128(vy[i])
x = mult128(x,a)
x = add128(x,y)
vy[i] = store128(x)
}
```

y=ax+y (AVX Cコード)

```
for(i=0 ; i<n ; i+=4){
x = load256(vx[i])
y = load256(vy[i])
x = mult256(x,a)
x = add256(x,y)
vy[i] = store256(x)
}
```

反復法ライブラリLis, http://www.ssisc.org/lis/

y = a * x + yのアセンブリコード

2オペランド命令 (SSE2)

```
move(x , temp)    //temp ← x
mult(temp, a)     //temp = a * temp
add(y , temp)     //y = y + temp
```

3オペランド命令 (AVX)

```
mult(a , x , temp)      //temp = a * x
add(temp , y , y)       //y=temp + y
```

$y_{DD} = a_{DD}x_{DD} + y_{DD}$の命令数の内訳

	SSE2	AVX	AVX-SSE2
Load	2	2	0
Store	1	1	0
add + sub	26	26	0
mult	9	9	0
move	13	3	-10
合計	**51**	**41**	**-10**

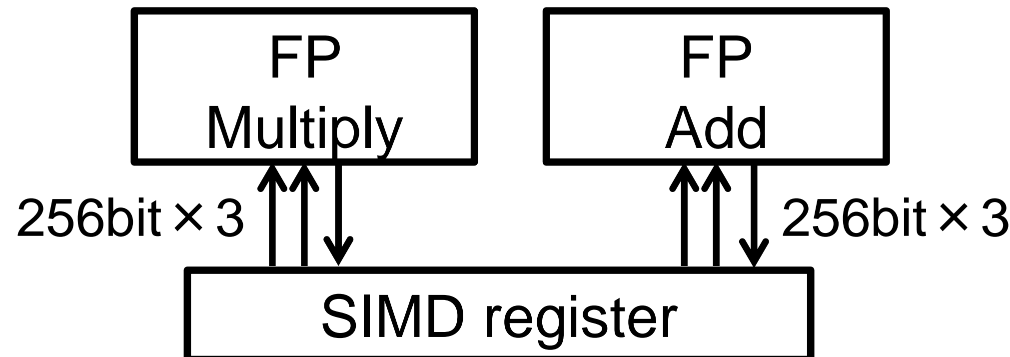
命令数減

実験環境

- CPU : intel Core i7 2600K 4コア 3.4GHz 16GB
  - L3キャッシュ : 8MB
  - メモリ帯域　：21.2GB/s (10.6×2)
- OS : Fedora16
- コンパイラ : intel C/C++ Compiler 12.0.3
  - コンパイルオプション
    - AVXコード　： –O3 **–xAVX** –openmp –fp-model precise
    - SSE2 コード： –O3 **–xSSE2** –openmp –fp-model precise

intel Core i7 2600Kの構成(1コア)

- 演算器



- 理論値
  - AVX:

    3.4G×4(SIMD)×2(積和同時演算) = **27.2GFLOPS / core**
  - SSE2:

    3.4G×2(SIMD)×2(積和同時演算) = **13.6GFLOPS / core**
  - Scalar

    3.4G×2(積和同時演算) = **6.8GFLOPS / core**

目次

1. 研究背景･目的
2. 実装，実験環境
3. 実験 -倍々精度ベクトル演算-
4. 実験 -倍々精度疎行列ベクトル積-
5. まとめ

対象とするベクトル演算

Kogakuin University

名称	演算	Load	Store	倍精度の演算量
axpy	$\boldsymbol{y} = \alpha\boldsymbol{x} + \boldsymbol{y}$	2	1	35
axpyz	$\boldsymbol{z} = \alpha\boldsymbol{x} + \boldsymbol{y}$	2	1	35
dot	$val = \boldsymbol{x} \cdot \boldsymbol{y}$	2	0	35
nrm2	$val = \lVert\boldsymbol{x}\rVert_2$	1	0	31
scale	$\boldsymbol{x} = \alpha\boldsymbol{x}$	1	1	24
xpay	$\boldsymbol{y} = \boldsymbol{x} + \alpha\boldsymbol{y}$	2	1	35

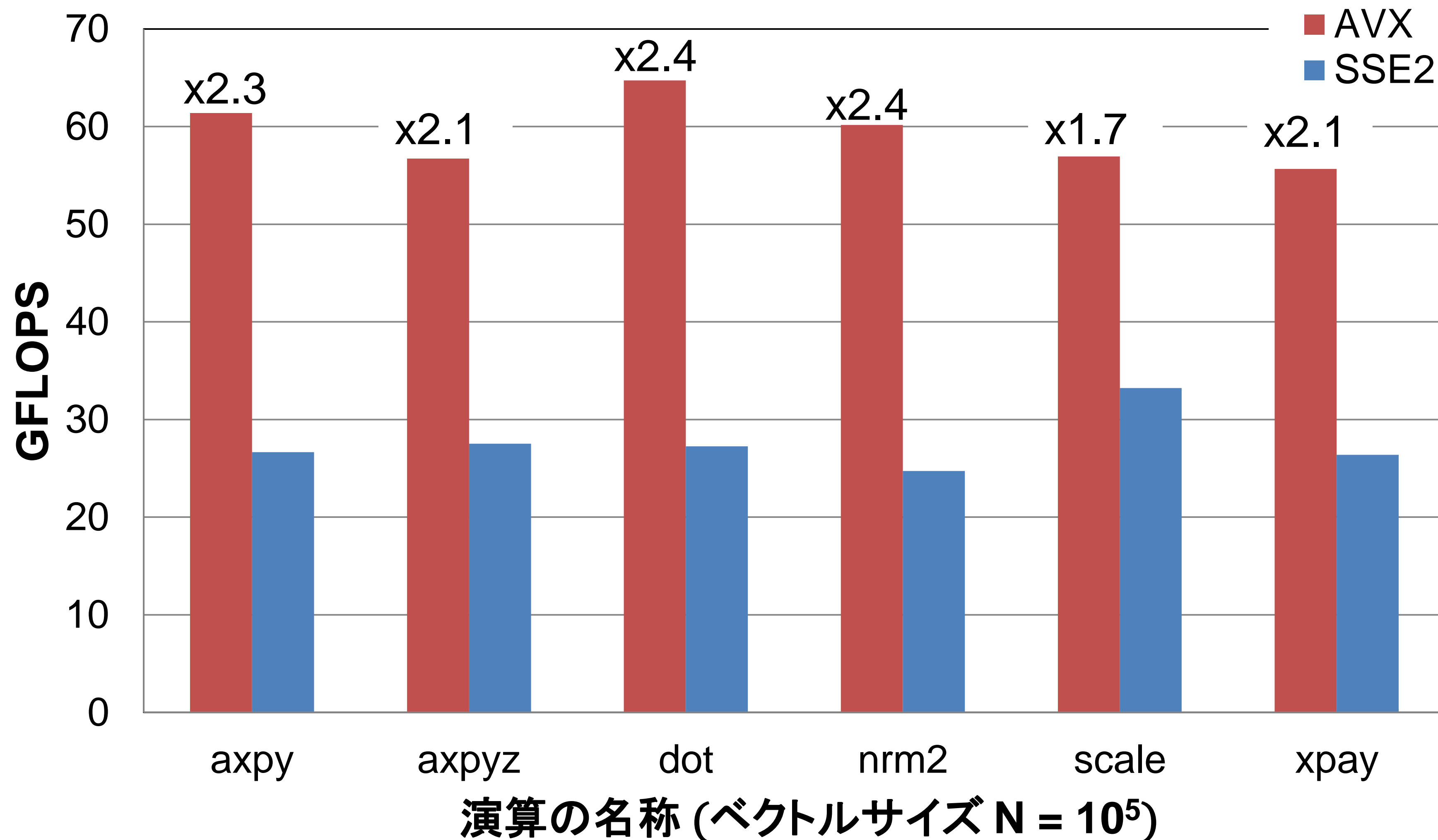
$\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{z}$ ：倍々精度のベクトル

α, val ：倍々精度のスカラー値

実験内容

- 実験1：ベクトル演算の性能
  - ベクトルがキャッシュにおさまる場合(4スレッド)
- 実験2：axpyの分析
  - ベクトルサイズNを変化させた性能(N=10^3から8.0×10^5)
  - マルチスレッドの性能向上比(1～8)
- OpenMPのスケジューリング方式はstaticを用いた

実験1 キャッシュに収まる場合の性能(4スレッド)



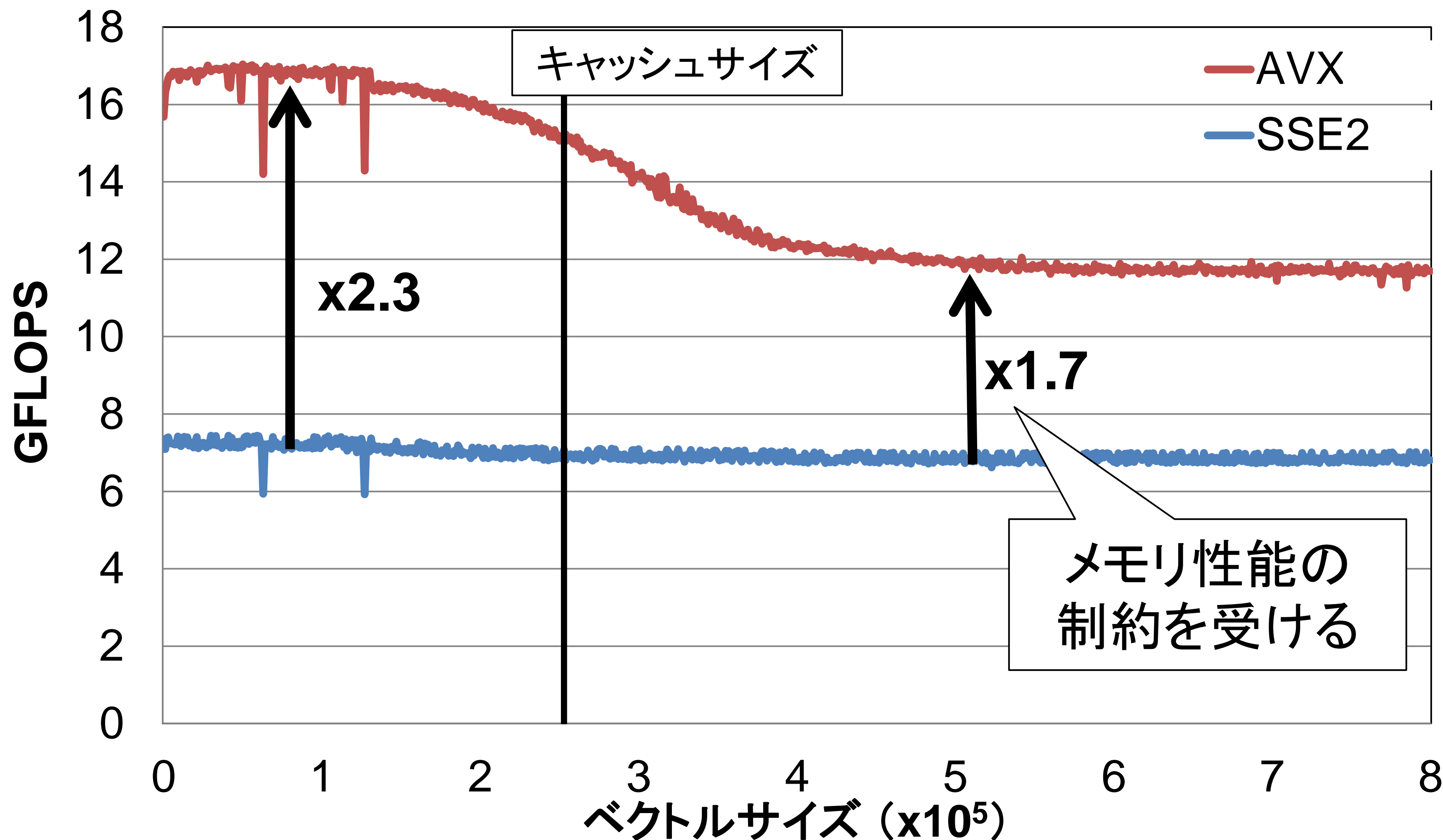
キャッシュに収まる場合の性能

- SSE2と比べ1.7~2.4倍の向上
  - scaleはSSE2の性能が良く向上比が小さい(x1.7)
  - AVXはピーク性能の51~60%，SSE2は45~60%
- move命令の削減数が多いものは向上比が高い

演算名(演算量)	move命令削減数	性能：AVX/SSE2
axpy (35)	10	2.3
axpyz (35)	10	2.1
dot (35)	10	2.4
nrm2 (31)	13	2.4
scale (24)	2	1.7
xpay (35)	10	2.1

実験2 メモリアクセスの影響 (axpy,1スレッド)

Kogakuin University



ピーク性能との比較

- AVXはピーク性能の63%（キャッシュ内, $N=10^5$）
- SSE2はピーク性能の52%
- axpy演算のカーネル演算の内訳

演算	Load	Store	add+sub	mult
演算回数	6	2	26	9

  - 加減算命令と乗算命令に偏りがある(26:9)
    - 理論性能が出ることはない

- 加減算と乗算のバランスを考慮した理論値
  - AVX ：27.2 → 18.3GFLOPS/core (67%)
  - SSE2：13.6 → 9.2GFLOPS/core (67%)

補正ピーク性能との比較

Kogakuin University

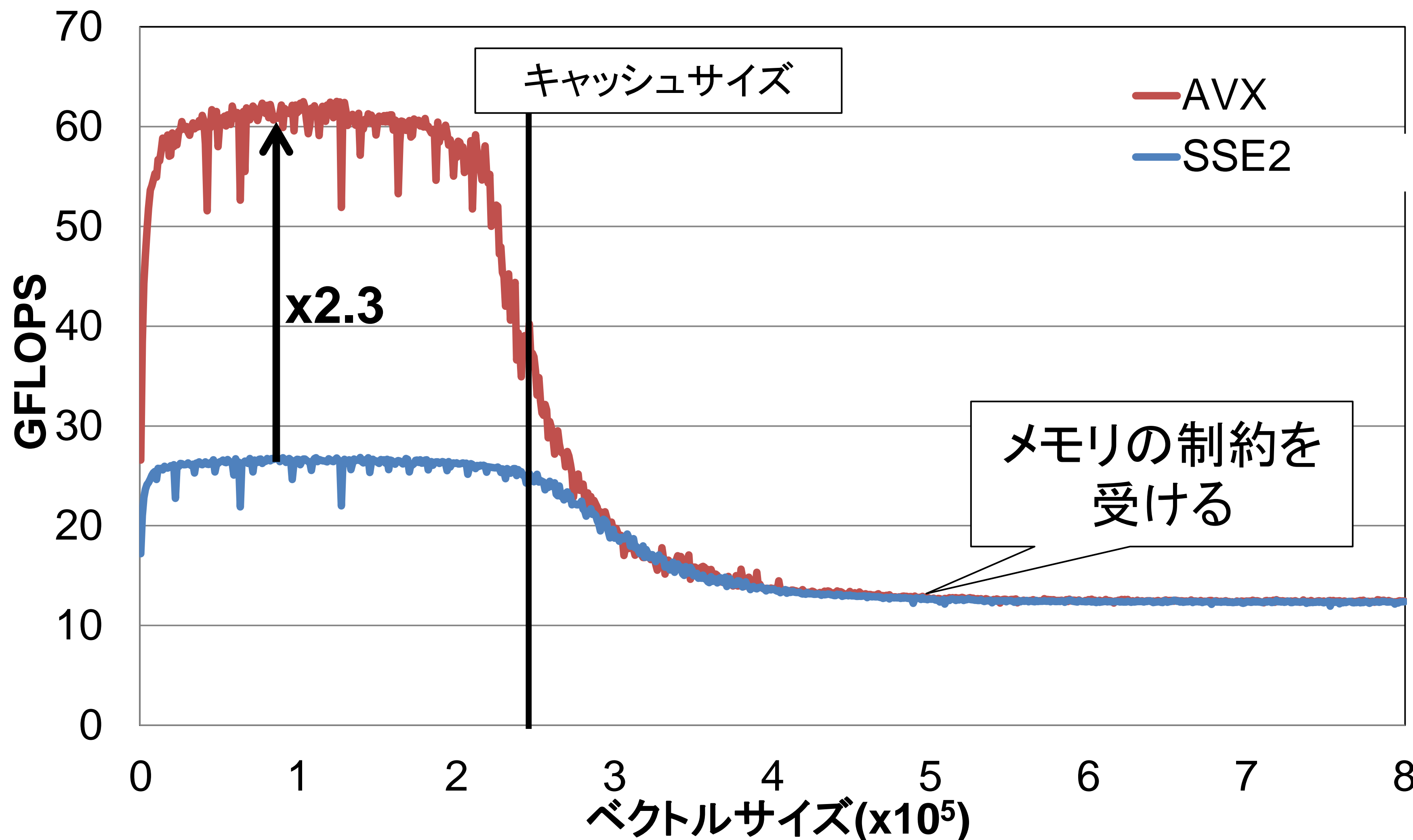
- AVX, 1スレッド, キャッシュに収まる場合

名称	AVXの性能	対ピーク性能比	対補正ピーク性能比	補正ピーク性能
axpy	16.8GFLOPS	62%	92%	18.3GFLOPS
axpyz	16.9GFLOPS	62%	95%	18.3GFLOPS
dot	18.0GFLOPS	66%	98%	18.3GFLOPS
nrm2	17.2GFLOPS	63%	94%	17.6GFLOPS
scale	17.5GFLOPS	64%	96%	21.8GFLOPS
xpay	16.5GFLOPS	61%	90%	18.3GFLOPS

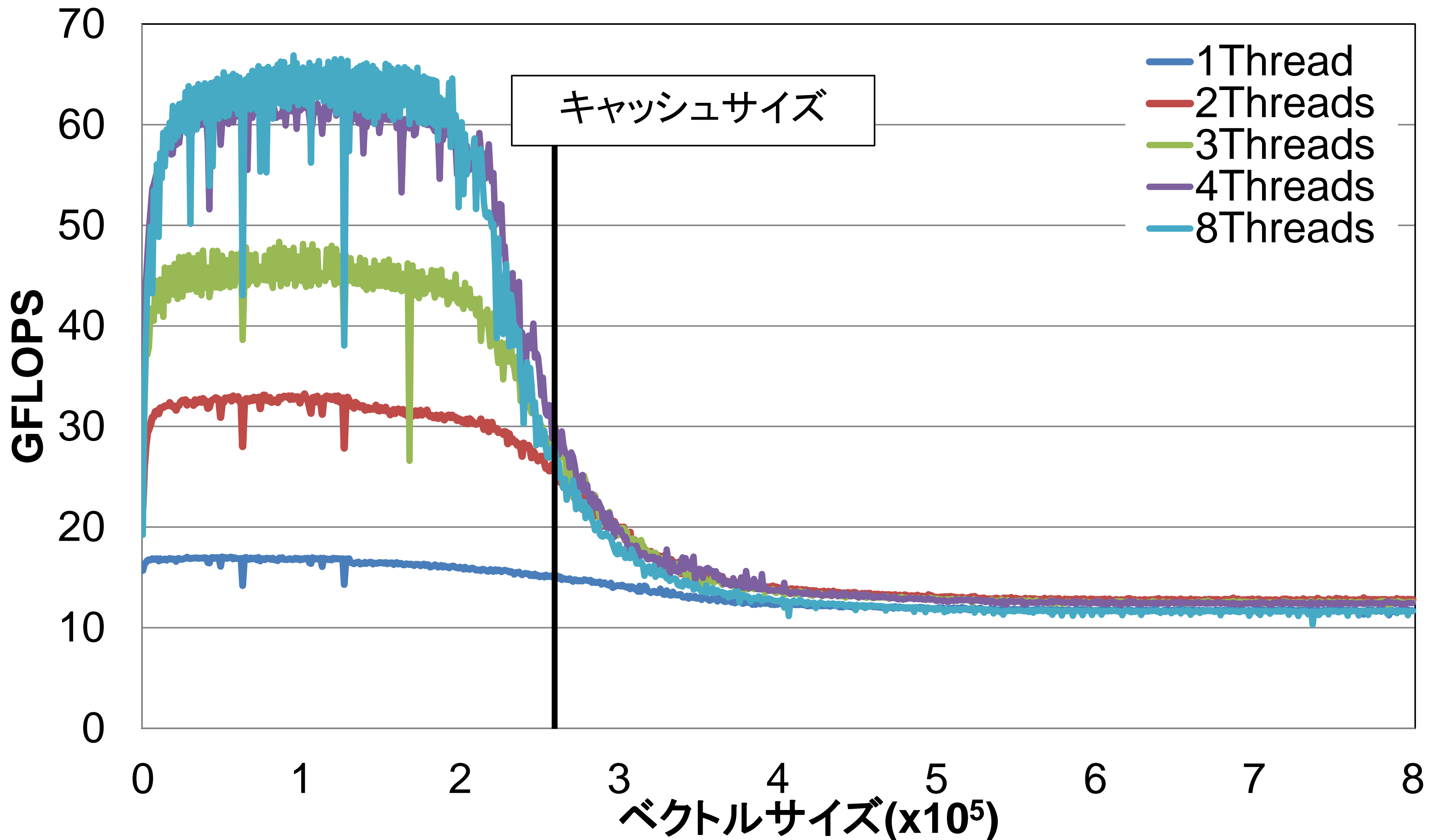
  - AVXの性能は16.5GFLOPSから18.2GLOPS
  - 理論ピーク性能(27.2GFLOPS)の61%から66%
  - 演算バランスを考慮した補正ピーク性能の90%から98%

実験2 メモリアクセスの影響 (axpy,4スレッド)

Kogakuin University



AVXの性能 (axpy,1~8スレッド)



キャッシュに収まる場合の結果 (実験2)

Kogakuin University

()内は対ピーク性能

- 1スレッドにおいて
  - AVXは16.8GFLOPS(62%)，SSE2は7.4GFLOPS(54%)
  - AVXはSSE2の2.3倍の性能
- 4スレッドにおいて
  - AVXは61.2GFLOPS(56%)，SSE2は27.4GFLOPS(51%)
  - AVXはSSE2の2.3倍の性能
  - 1スレッドと比較したAVX，SSE2の性能は3.7倍
- 加減算，乗算のバランスが悪くマシン理論値は出ない
  - 演算バランスを考慮するとピーク性能は27.2→18.3GFLOPS
  - 加減算，乗算のバランスを考慮すると90%以上

キャッシュに収まらない場合の結果 (実験2)

Kogakuin University

()内は対ピーク性能

- 1スレッドにおいて
  - AVXは11.8GFLOPS(43%), SSE2は7.4GFLOPS(54%)
  - AVXはSSE2の1.7倍の性能
- 4スレッドにおいて
  - メモリ性能を上限とした性能に制約される
  - AVX, SSE2ともに性能は13GFLOPS(AVXで12%)
  - マルチスレッドにおいてAVXとSSE2は同等の性能
- メモリ性能の制約を受け性能は約13GFLOPSになる

目次

1. 研究背景･目的
2. 実装，実験環境
3. 実験 -倍々精度ベクトル演算-
4. 実験 -倍々精度疎行列ベクトル積-
5. まとめ

倍精度疎行列A_Dと倍々精度ベクトルx_{DD}の積：Ax

Kogakuin University

- 倍精度疎行列A_DはCRS形式で格納
  - 実問題では，入力は倍精度
  - 倍々精度演算はメモリネックとなる
- 演算の内訳
  - 加減算25回，乗算8回から成る(演算量33)
- 加減算と乗算のバランスを考慮した理論値
  - AVX　：27.2 → 18GFLOPS/core (66%)
  - SSE2　：13.6 → 9GFLOPS/core (66%)

$y_{DD}=A_D * x_{DD}$

Kogakuin University

```
for(i=0 ; i<N ; i++)
        for(j=A_D_row_ptr[ i ] ; j < A_D_row_ptr[ i+1 ] ; j++)
                        y_DD[ i ] = y_DD [ i ] + A_D_value[ j ] * x_DD [ A_D_index[ j ] ]
```

- AVXではjのループを4つずつ同時演算
- 端数(1,2,3)の処理が必要
  - パディング
    - 生成時
    - 実行時
  - SSE2+Scalar
  - Scalar

端数処理方式

Kogakuin University

- 生成時パディング
- 実行時パディング
- SSE2+Scalar
  - AVXとSSE2のレジスタは論理的には別
  - AVXとSSE2のレジスタは物理的には共通
  - 命令の切替時，レジスタ内容がメモリに退避される
  - AVX，SSE2命令を同一コードで使うと性能が低下
- Scalar

端数処理の比較(CRS形式,N=10^5)

Kogakuin University

	帯幅63の性能（実行時間）	帯幅1023の性能（実行時間）
生成時パディング	44.3GFLOPS（47ミリ秒）	47.4GFLOPS（71ミリ秒）
実行時パディング	42.2GFLOPS（49ミリ秒）	47.4GFLOPS（71ミリ秒）
SSE2+Scalar	39.0GFLOPS（53ミリ秒）	41.1GFLOPS（81ミリ秒）
Scalar	41.1GFLOPS（48ミリ秒）	47.1GFLOPS（71ミリ秒）

- CRS生成時パディングの性能が最も高い
- SSE2+Scalarは性能が低い
  - AVXとSSE2の切り替えコストが大きい
- 実行時パディングとScalarの性能差は小さい

実験内容

実験1 :メモリアクセスの影響

実験2 :AVX化の効果

実験3 :非零要素の数による性能の影響

- 端数の数による影響

- OpenMPのスケジューリング方式はguidedを用いた

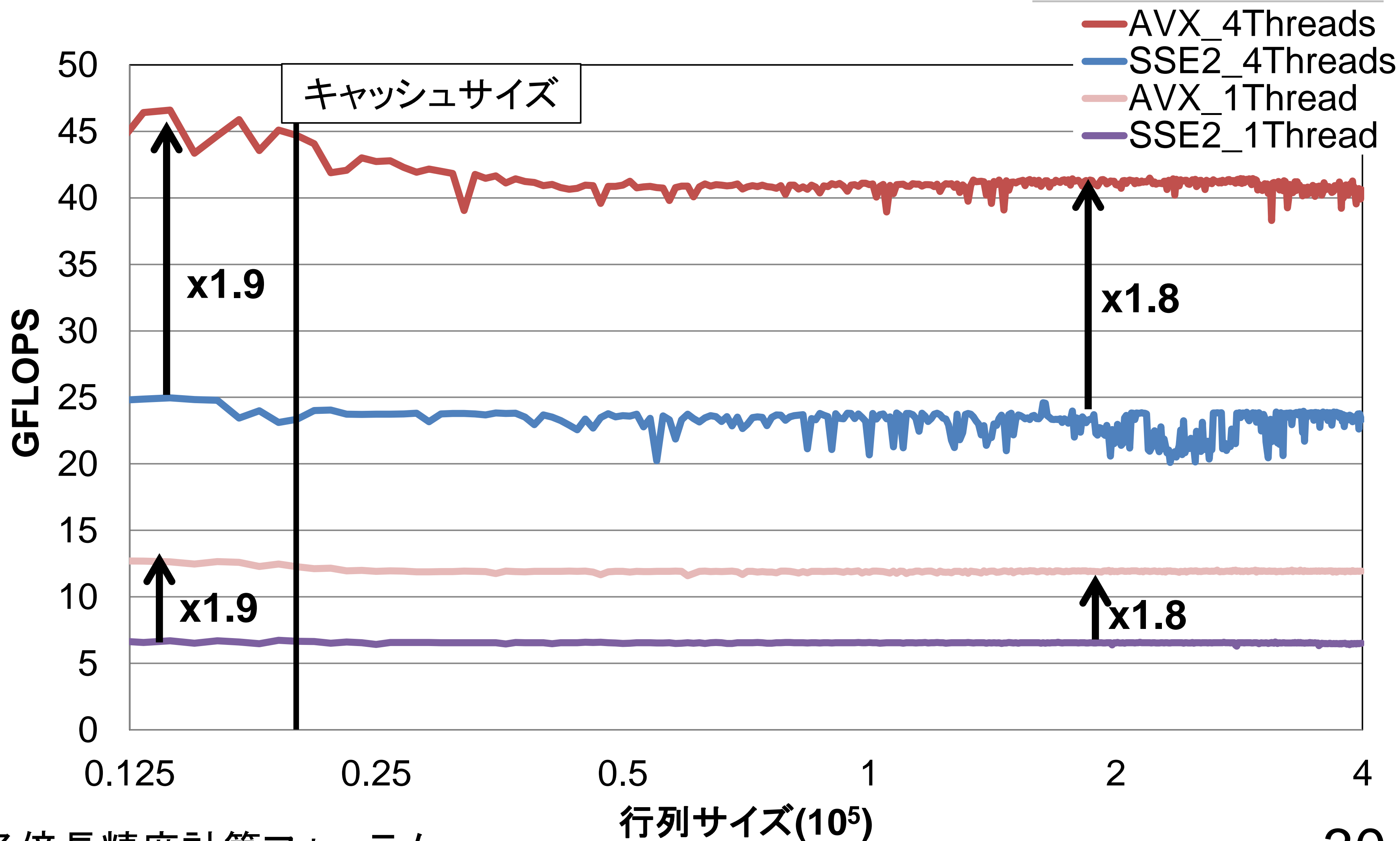
実験に用いる疎行列

B. テスト用帯行列
- if($0 \le j - i \le m$) ai j＝value
- else ai j = 0

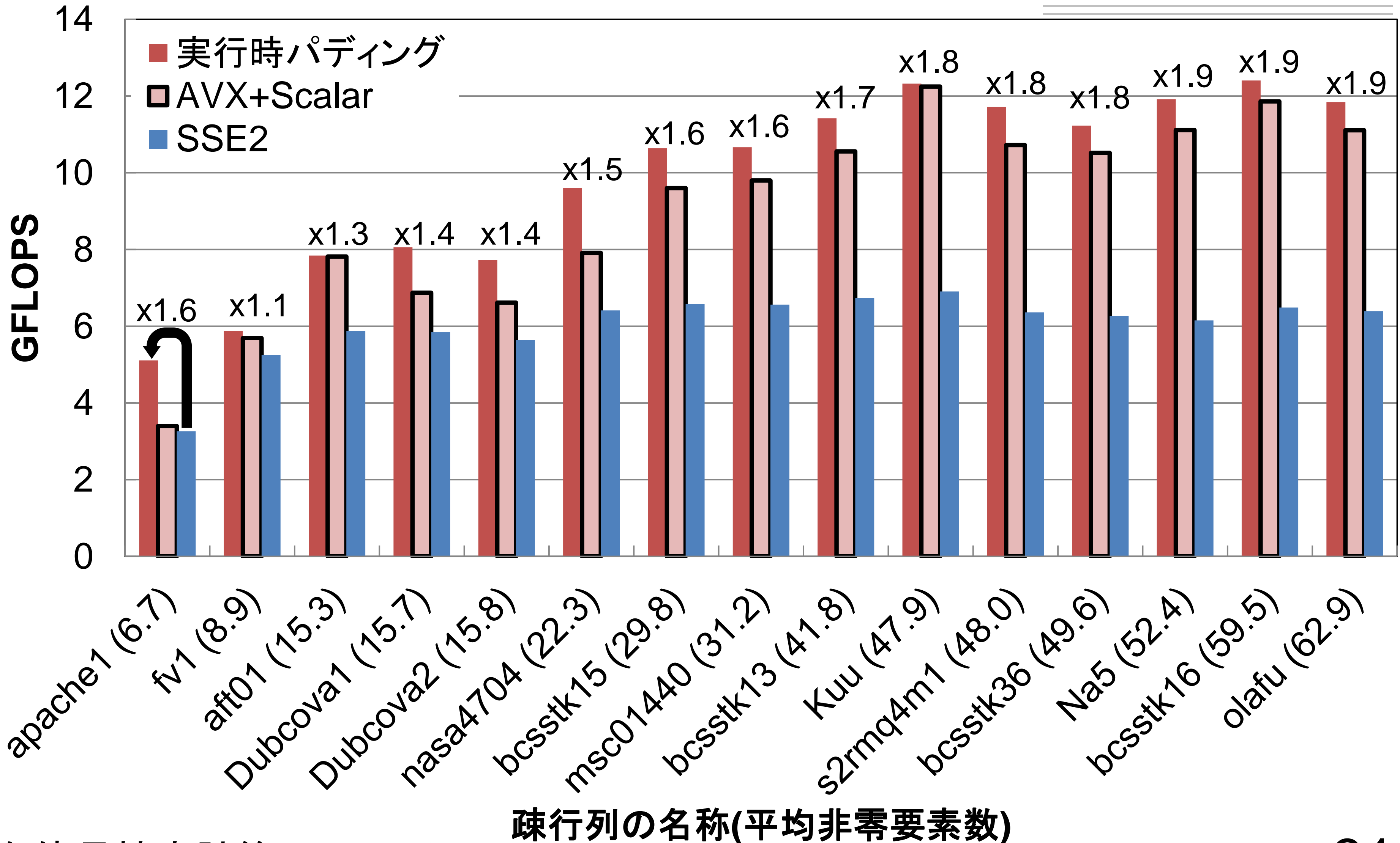
を満たす疎行列

F. The Univ of Florida Matrix Collection (フロリダコレクション)の疎行列15種

メモリアクセスの影響(テスト用帯行列，帯幅32,)

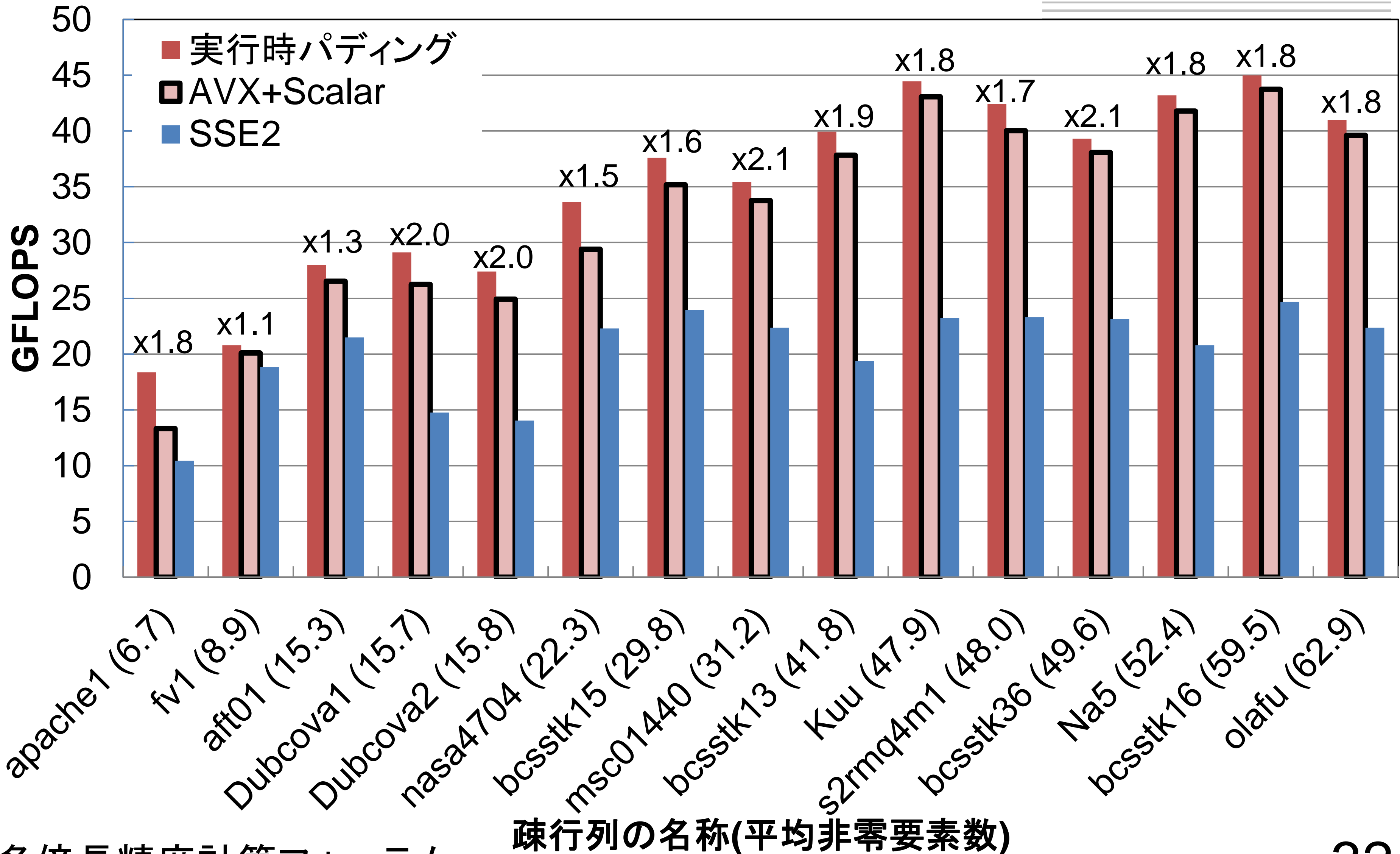


不規則な構造を持つ疎行列の性能(1スレッド)



不規則な構造を持つ疎行列の性能(4スレッド)

Kogakuin University

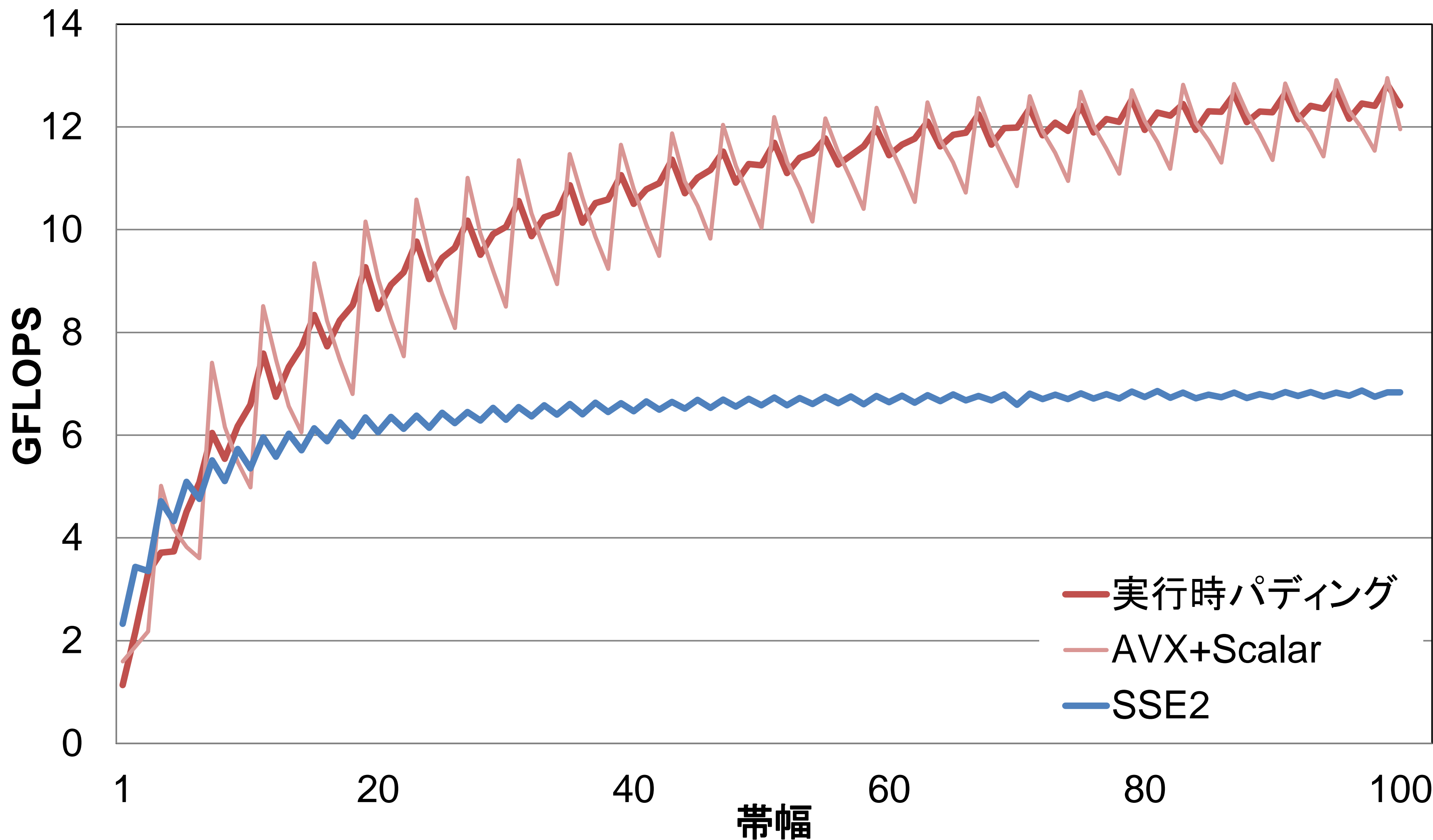


不規則な構造を持つ疎行列の性能

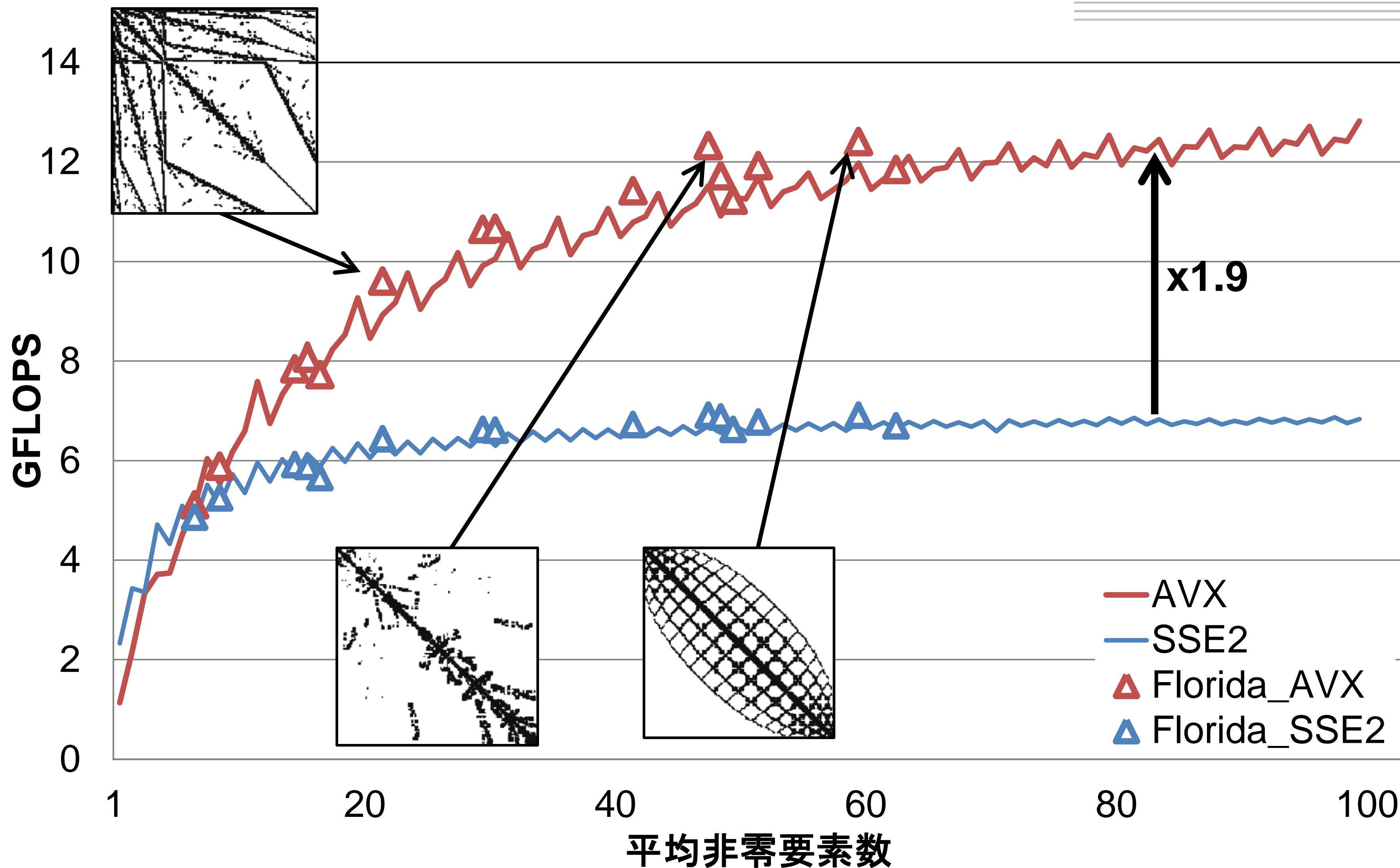
	実行時パディングの性能 (対ピーク)	Scalarの性能(対ピーク)
F，1スレッド	5.1 ~ 12.4GFLOPS (19% ~ 46%)	3.4 ~ 12.2GFLOPS(12% ~ 45%)
F，4スレッド	18.4 ~ 45.0GFLOPS (17% ~ 41%)	13.3 ~ 43.7GFLOPS(12% ~ 40%)

- 端数処理は実行時パディングが有効
  - Scalarとの比は4スレッドで1.03倍から1.37倍
- 1スレッドと4スレッドの性能比はAVXで3.3倍から3.7倍
- 平均非零要素数が多いものは性能が高い

端数の影響(テスト用帯行列，サイズ10^5)



帯幅と平均非零要素数の関係(実行時パディング)



非零要素数による影響

- Scalarは端数の数による影響が大きい
  - 実行時パディングは11.4GFLOPS(帯幅63)から12.1GFLOPS(帯幅64)
  - Scalarは10.3GFLOPS(帯幅63)から12.3GFLOPS(帯幅64)
- 帯幅が広いほど性能が高い
  - 帯幅64のとき
    - AVXは12.1GFLOPS(44%)，SSE2は6.8GFLOPS(50%)
- フロリダコレクションの性能は平均非零要素数に関係
  - 対応する帯行列の1.07倍から0.97倍

()内は対ピーク性能

AxとA^Txの比較

- $y_{DD}=A^T{}_D * x_{DD}$のコード

Ax

```
for(i=0 ; i<N ; i++)
        for(j=A_D_row_ptr[ i ] ; j < A_D_row_ptr[ i+1 ] ; j++)
                y_DD[ i ] = y_DD [ i ] + A_D_value[ j ] * x_DD [ A_D_index[ j ] ]
```

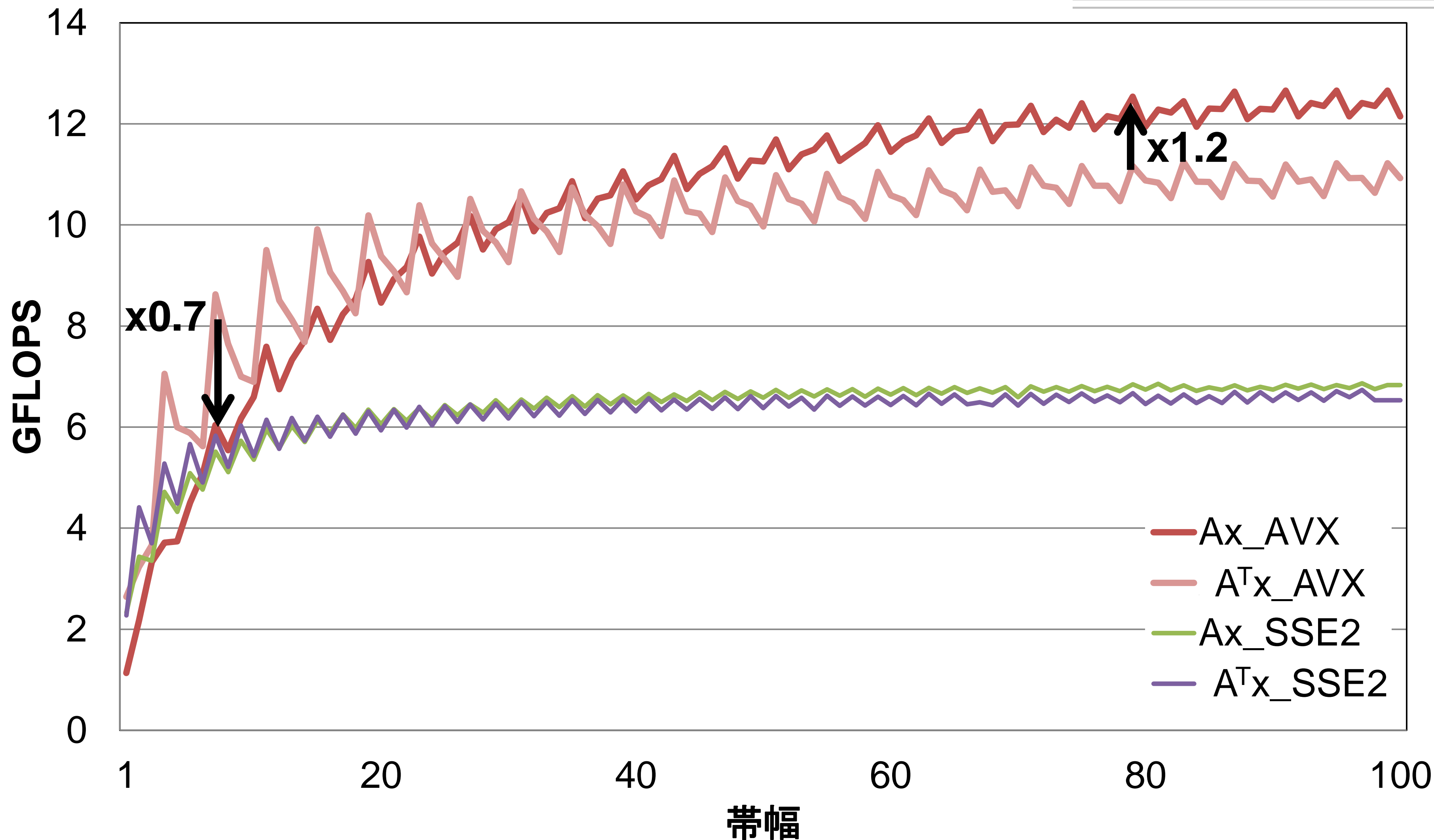
A^Tx

```
for(i=0 ; i<N ; i++)
        for(j=A_D_row_ptr[ i ] ; j<A_D_row_ptr[ i+1 ] ; j++)
                y_DD[ A_D_index[ j ] ] = y_DD[ A_D_index[ j ] ] + A_D_value[ j ] * x_DD [ i ]
```

- AxとA^Txの違いはx_{DD}とy_{DD}へのアクセス
  - Axはx_{DD}へのアクセスがA_{index}に従う
  - A^Txはy_{DD}へのアクセスがA_{index}に従う

A^TxとAxの性能(実行時パディング,1スレッド,N=10^5)

Kogakuin University



目次

1. 研究背景･目的
2. 実装，実験環境
3. 実験 -倍々精度ベクトル演算-
4. 実験 -倍々精度疎行列ベクトル積-
5. まとめ

まとめ (ベクトル演算)

- 4スレッドにおいて
  - キャッシュに収まるとき
    - AVXは61.2GFLOPS(ピーク性能の56%), SSE2は27.4GFLOPS(ピーク性能の51%)
    - AVXとSSE2の性能比は2.3倍
      - move命令の削減効果
  - キャッシュに収まらないとき
    - メモリ性能の制約を受け性能は約13GFLOPSに低下

まとめ (端数処理)

- Scalar
  - 10.3GFLOPSから12.3GFLOPS(帯幅61~64, 1スレッド)
- 実行時パディング
  - 11.4GFLOPSから12.1GFLOPS(帯幅61~64, 1スレッド)
  - 実行時パディングは端数の数による影響を受けにくい
- 実行時パディングは端数計算が多い問題では有効
  - フロリダコレクション(4スレッド)において
    - Scalarの1.03倍から1.37倍(5.1GFLOPS~12.4GFLOPS)

まとめ (倍精度疎行列と倍々精度ベクトルの積)

Kogakuin University

- A_Dを倍精度化：性能はメモリネックになりにくい
  - キャッシュに収まる場合と収まらない場合の性能比は0.9倍
- 性能は平均非零要素数に関係する
  - 対応する帯幅の帯行列と比べ1.07倍から0.97倍
- $A^T x$とAxの性能差は小さい
  - 平均非零要素数が少ないとき，Axは$A^T x$の性能の0.7倍
  - 平均非零要素数が多いとき，Axは$A^T x$の性能の1.2倍

今後の課題

- 倍々精度演算の演算バランスの改善
  - 乗算と比べ加減算が多く，演算器が並列に動かない
  - 加減算を他の演算で置き換える
- 端数処理手法の切り替え
  - サイズ，繰り返し回数から最適な端数処理手法を切り替え

参考文献

1. Bailey, D ,H.: High-Precision Floating-Point Arithmetic in Scientific Computation, Computing in Science and Engineering, pp. 54–61 (2005).
2. 反復解法ライブラリLis, http://www.ssisc.org/lis/
3. The University of Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices/
4. Barrett, R., et al.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, SIAM pp. 57–65 (1994)