SSE2を用いた反復解法ライブラリ Lis 4倍精度版の高速化

小武守 恒（JST・東京大学）
藤井 昭宏（工学院大学）
長谷川 秀彦（筑波大学）
西田 晃（中央大学・JST）

発表の流れ

- はじめに
- 4倍精度演算について
- Lisへの実装
- SSE2による高速化
- 性能評価
  - スピード
  - 収束
- まとめ

はじめに

- クリロフ部分空間法たとえばCG法は，理論的には高々n回（nは係数行列の次元数）の反復で収束
- 丸め誤差の影響
  - 収束までに多くの反復が必要
  - 停滞
- 収束の改善には高精度演算，例えば4倍精度演算が有効であるが計算コスト大

- 反復解法ライブラリLisへ4倍精度演算を実装
- 4倍精度演算にSSE2を活用して高速化

double-double 精度演算

- FORTRANは4倍精度浮動小数をサポート

IEEE754準拠の4倍精度の表現形式

	指数部 15ビット	仮数部 112ビット

- Bailey
  - 倍精度浮動小数を2個用いて4倍精度を実現
  - double-double精度a = a.hi + a.lo, |a.hi|>|a.lo|

double-double精度の表現形式

	指数部 11ビット	仮数部 52ビット		指数部 11ビット	仮数部 52ビット

基本演算（表記）

- fl(a + b) :a + bの倍精度浮動小数加算
- err(a + b): a + b = fl(a + b) + err(a + b)
- 乗算a × bについても同様

4倍精度四則演算のための倍精度基本演算

- s=fl(a+b), e=err(a+b)を計算．|a|>=|b|

```
FAST_TWO_SUM(a,b,s,e) {
   s = a + b
   e = b - (s - a)
}
```

- s=fl(a+b), e=err(a+b)を計算．

```
TWO_SUM(a,b,s,e) {
   s = a + b
   v = s - a
   e = (a - (s - v)) + (b - v)
}
```

- aをa=h+lに分割(仮数部を26bitずつに)

```
SPLIT(a,h,l) {
   t = 134217729.0 * a
   h = t - (t - a)
   l = a - h
}
```

- p=fl(a×b),e=err(a×b)を計算

```
TWO_PROD(a,b) {
   p = a * b
   SPLIT(a,ah,al)
   SPLIT(b,bh,bl)
   e = ((ah*bh-p)+ah*bl+al*bh)+al*bl
}
```

4倍精度加算・乗算

- 加算a = b + c.
  a=(a.hi,a.lo),
  b=(b.hi,b.lo),
  c=(c.hi,c.lo)

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

- 乗算a = b × c

```
MUL(a,b,c) {
  TWO_PROD(b.hi,c.hi,p1,p2)
  p2 = p2 + (b.hi * c.lo)
  p2 = p2 + (b.lo * c.hi)
  FAST_TWO_SUM(p1,p2,a.hi,a.lo)
}
```

Lisへの実装

- Lis
  - 反復解法ライブラリ
  - さまざまな反復法、前処理が組合わせて使える
- 反復解法を4倍精度演算に置き換え
  - 行列ベクトル積(matvec)
  - ベクトルの内積(dot)
  - ベクトルおよびその実数倍の加減(axpy)

Lisへの実装の条件

- ユーザーインタフェイスに影響を与えることなく4倍精度演算が利用できるよう以下のようにする
  - 係数行列A, 右辺ベクトル b は倍精度
  - 解ベクトル x の入出力は倍精度,
    反復解法中では4倍精度
  - 反復解法中のベクトル,スカラーは4倍精度
- 混合精度演算関数が必要

matvec, dot, axpy

- 主な処理は積和演算

1. FMA(Floating Multiply-Add)
   - 4倍精度の積和演算関数
   - dotとaxpyで利用
2. FMAD
   - 混合精度(4倍精度と倍精度)の積和演算関数
   - matvecで利用

FMA・FMADの実装

- FMA a = a + b × c

```
FMA(a,b,c) {
  TWO_PROD(b.hi,c.hi,p1,p2)
  p2 = p2 + (b.hi * c.lo)
  p2 = p2 + (b.lo * c.hi)
  FAST_TWO_SUM(p1,p2,p1,p2)
  TWO_SUM(a.hi,p1,sh,eh)
  TWO_SUM(a.lo,p2,sl,el)
  eh = eh + sl
  FAST_TWO_SUM(sh,eh,sh,eh)
  eh = eh + el

FAST_TWO_SUM(sh,eh,a.hi,a.lo) }
```

- FMAD a = a + b × c

```
FMAD(a,b,c) {
  TWO_PROD(b,c.hi,p1,p2)
  p2 = p2 + (b * c.lo)

  FAST_TWO_SUM(p1,p2,p1,p2)
  TWO_SUM(a.hi,p1,sh,eh)
  TWO_SUM(a.lo,p2,sl,el)
  eh = eh + sl
  FAST_TWO_SUM(sh,eh,sh,eh)
  eh = eh + el

FAST_TWO_SUM(sh,eh,a.hi,a.lo) }
```

4倍精度 vs 倍精度

- ポアソン方程式を5点中心差分で離散化した行列(次数1,000,000)
- 行列格納形式はCRS
- 前処理なしBiCG法を50回反復

	実行時間(比)
FORTRAN4倍精度	23.61
matvecでFMAを利用	8.26
matvecでFMADを利用	8.17
倍精度	1.00

SSE2（Streaming SIMD Extension）

- SSE2はIntel Pentium4に搭載されたx87命令に代わる高速化命令
- 128bitのデータに対してSIMD処理
  - 64bit倍精度浮動小数なら同時に2つの演算
- SSE2の組込み関数を利用

SSE2化の2つのアプローチ

- A: 四則演算関数のSSE2化
  - 演算の依存関係が存在
  - 関数FMA_SSE2, FMAD_SSE2を用意
- B: dot, axpy, matvecを含めたSSE2化
  - 2段のループアンローリングを行うとFMAが2回
  - 2個同時に積和演算をする関数FMA2_SSE2, FMAD2_SSE2を用意

A:四則演算関数のSSE2化

- 依存関係のため約50%程度しかSSE2のpd命令で処理できない

FMA_SSE2のADD：pdで処理できる部分

```
xb = _mm_loadu_pd((double*)b);
xc = _mm_loadu_pd((double*)c);
xs = _mm_add_pd(xb,xc);
xe = _mm_sub_pd(xs,xb);
xt = _mm_sub_pd(xs,xe);
xc = _mm_sub_pd(xc,xe);
xb = _mm_sub_pd(xb,xt);
xb = _mm_add_pd(xb,xc);
xe = _mm_unpackhi_pd(xb,xb);
xc = _mm_unpackhi_pd(xs,xs);
```

FMA_SSE2のADD ：pdで処理できない部分

```
xt = xs;
xb = _mm_add_sd(xb,xc);
xt = _mm_add_sd(xt,xb);
xs = _mm_sub_sd(xt,xs);
xb = _mm_sub_sd(xb,xs);
xb = _mm_add_sd(xb,xe);
xc = _mm_add_sd(xt,xb);
_mm_store_sd(&a->hi,xc);
xc = _mm_sub_sd(xc,xt);
xb = _mm_sub_sd(xb,xc);
_mm_store_sd(&a->lo,xb);
```

B:dot,axpy,matvecを含めたSSE2化

- 100%SSE2のpd命令で処理できる

FMA2_SSE2のADDの部分

```
bh = _mm_loadu_pd(&y[i]);
bl = _mm_loadu_pd(&yl[i]);
sh = _mm_add_pd(bh,ch);
th = _mm_sub_pd(sh,bh);
t0 = _mm_sub_pd(sh,th);
ch = _mm_sub_pd(ch,th);
bh = _mm_sub_pd(bh,t0);
bh = _mm_add_pd(bh,ch);
sl = _mm_add_pd(bl,p2);
th = _mm_sub_pd(sl,bl);
t0 = _mm_sub_pd(sl,th);
p2 = _mm_sub_pd(p2,th);
```

```
bl = _mm_sub_pd(bl,t0);
bl = _mm_add_pd(bl,p2);
bh = _mm_add_pd(bh,sl);
th = sh;
th = _mm_add_pd(th,bh);
sh = _mm_sub_pd(th,sh);
bh = _mm_sub_pd(bh,sh);
bh = _mm_add_pd(bh,bl);
sh = _mm_add_pd(th,bh);
_mm_storeu_pd(&y[i],sh);
sh = _mm_sub_pd(sh,th);
bh = _mm_sub_pd(bh,sh);
_mm_storeu_pd(&yl[i],bh);
```

性能評価

- 疎行列反復解法において
- 前処理なしのBiCG法を用いて比較
  - Lisの倍精度
  - FORTRANで作成した4倍精度
  - Lisの4倍精度(FMAを利用)
  - Lisの4倍精度(FMA_SSE2を利用　A)
  - Lisの4倍精度(FMA2_SSE2を利用　B)

テスト行列

- 行列A1
  - ポアソン方程式を5点中心差分で離散化
- 行列A2
  - Toeplitz行列

$$\begin{pmatrix} 2 & 1 & & & & \\ 0 & 2 & 1 & & & \\ \gamma & 0 & 2 & 1 & & \\ & \ddots & \ddots & \ddots & \ddots & \\ & & \gamma & 0 & 2 & 1 \\ & & & \gamma & 0 & 2 \end{pmatrix}$$

- 右辺ベクトルbはすべてを1
- 初期ベクトルx_0はすべてを0
- 収束判定基準は相対残差ノルム10^{-12}

計算環境

CPU	Xeon	Opteron
Clock	2.8GHz	2.0GHz
L1D Cache	8KB	64KB
L2 Cache	512KB	1MB
Memory	1GB	1GB
OS	Linux 2.4.20smp	Linux 2.6.4smp
Compiler	Intel C++ 9.0 Intel Fortran 9.0	

- 最適化オプションは-O3
- FMAが記述されているCファイルは浮動小数の最適化を抑制(-mp)

スピードテスト

- テスト行列A1で50回反復した結果

Xeon

次数	倍精度	FORTRAN	FMA	FMA_SSE2	FMA2_SSE2
100	0.00034	0.02141	0.00646	0.00492	0.00312
1000	0.00279	0.20786	0.06356	0.04818	0.03032
10000	0.05717	2.01729	0.66783	0.52825	0.33641
100000	0.82734	20.09851	6.81664	5.29807	3.46160
1000000	8.44022	199.23818	68.93871	53.41777	34.91778
算術平均	1.86557	44.31665	15.29864	11.85944	7.74985

Opteron

次数	倍精度	FORTRAN	FMA	FMA_SSE2	FMA2_SSE2
100	0.00037	0.00826	0.00669	0.00457	0.00316
1000	0.00357	0.08037	0.06687	0.04561	0.03171
10000	0.04402	0.78882	0.68794	0.47166	0.33072
100000	0.67599	7.95484	7.03384	4.84892	3.47609
1000000	7.19525	79.85377	70.19153	48.42959	34.90340
算術平均	1.58384	17.73721	15.59738	10.76007	7.74901

スピードテストのまとめ

- FMA_SSE2の性能

比較対象	Xeon	Opteron
FMA	1.29	1.46

- FMA2_SSE2の性能

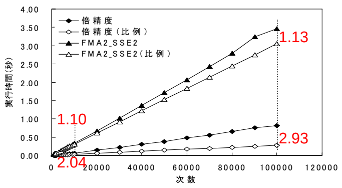
比較対象	Xeon	Opteron
FMA	2.02	2.07
FORTRAN	6.24	2.42
倍精度	0.17	0.15

スピードテスト (N=10^6)

実行時間(秒)

	Xeon	Opteron
FORTRAN	199.2 (24)	79.8 (11)
FMA	68.9 (8.2)	70.1 (9.9)
FMA_SSE2	53.4 (6.4)	48.4 (6.8)
FMA2_SSE2	34.9 (4.2)	34.9 (4.9)
倍精度	8.4 (1)	7.1 (1)

スピードテストのまとめ(続)



- 倍精度演算は2,000から徐々に大きく増加
  - 1,000でL1D 8KB
  - 100,000でL2 512KBから外れる
- 4倍精度演算は次数にほぼ比例して増加
  - 4倍精度演算はデータのロードとストアよりも演算の割合が大きい

収束テスト

Xeon上でテスト行列A2(次数100,000)の結果
(実行時間(秒), 反復回数, 残差ノルム)

	倍精度			FORTRAN			FMA2_SSE2		
γ	実行時間	反復回数	残差	実行時間	反復回数	残差	実行時間	反復回数	残差
1	0.78266	58	1.84E-10	18.60900	58	1.84E-10	3.78737	58	1.84E-10
1.1	0.94361	70	2.23E-10	22.70676	70	2.23E-10	4.58301	70	2.23E-10
1.2	1.17442	86	3.03E-10	27.97903	86	3.03E-10	5.61647	86	3.03E-10
1.3				36.25341	113	2.47E-10	7.40245	113	2.47E-10
1.4				70.18287	155	2.85E-10	10.11311	155	2.85E-10

- γ = 1.2までは両方ともに収束している
- γ = 1.3以上では倍精度は停滞. 4倍精度は収束
- 完全な4倍精度とFMA2_SSE2は同等の収束性

新　演算精度を変えたリスタート

- 倍精度で解いた解（あるいは途中の値）を初期値として4倍精度で解く

```
for(k=0;k<最大反復回数;k++) {
  倍精度演算の反復解法
  if( nrm2<restart_tol ) break;
}
for(k=k+1;k<最大反復回数;k++) {
  4倍精度演算の反復解法
}
```

  – nrm2は相対残差ノルム
  – restart_tolはリスタートするための判定基準

精度を変えたリスタートの結果

- Xeon上でテスト行列A2（次数100,000）
- γ=1.3，restart_tolを10^{-6}とした場合

	実行時間(秒)	反復回数
4倍精度	7.40	113
リスタート	4.43 (倍:0.45，4倍:3.98)	104 （倍:35，4倍:69）

- 4倍精度で解いた場合と比べて1.67倍速い
- 倍精度で解いた場合と比べて∞倍

まとめ

- SSE2を用いた4倍精度演算を実装
  – Lisへdouble-double精度演算を実装
    - 係数行列A，右辺ベクトル b は倍精度
    - 解ベクトル x の入出力は倍精度，反復解法中では4倍精度
    - 反復解法中のベクトル,スカラーは4倍精度
  – 四則演算関数のSSE2化
  – dot, axpy, matvecを含めたSSE2化

まとめ（続き）

- スピードテストより
  – 完全な4倍精度と比較して平均4.33倍
  – 2段のアンローリングにより約2倍の高速化
  – 倍精度の約4.5倍程度
- 収束テストより
  – 完全な4倍精度とほぼ同等の収束特性
- 収束の改善を図る新たな手法が可能となった
  – 倍精度と4倍精度の混合精度の反復解法

まとめ（続き）

- 高精度演算の展望
  – ILU前処理などの逐次的な前処理を利用しなくとも少ない反復回数で収束する可能性
  – データアクセスよりも演算の割合が大きく並列化には適している
- 並列化によって倍精度との性能差は縮まりより現実的になる

今後の課題

- 4倍精度をうまく活用して収束性を向上，安定して解けるロバストなライブラリの開発