Vol.2012-HPC-135 No.16
2012/8/2

反復法ライブラリ向け倍々精度演算の AVX を用いた高速化

菱沼利彰[†1] 浅川圭介[†2] 藤井昭宏[†1]
田中輝雄[†1] 長谷川秀彦[†3]

計算性能の向上に伴い，高精度で計算を行うことが多くの場面で必要になってきている．4 倍精度で効率良く計算する手法の中に，倍精度変数を 2 つ用いて 1 つの変数の値を保持する倍々精度演算がある．反復解法ライブリ Lis ではこの倍々精度演算が SSE2 を用いて実装されている．本研究ではその内部使われているベクトル演算を AVX 命令を用いてベクトル長を伸ばし、高速化を行った．その結果，ベクトル演算のデータが L3 キャッシュに収まる場合には，SSE2 版と比較して最大 1.4～2.3 倍の高速化が実現できた．

Acceleration of Double-Double Precision Operation for Iterative Solver Library using AVX

Toshiaki Hishinuma,[†1] Keisuke Asakawa[†2],Akihiro Fujii,[†1]
Teruo Tanaka[†1] and Hidehiko Hasegawa[†3]

As computing performance increases generation after generation, high precision calculation comes to be needed in many situations. One of the efficient methods to calculate in quadruple precision is to use double-double precision routines which use two double precision variables for one quadruple precision variable. The iterative solver library Lis has vectorized double-double precision routines with SSE2. In order to accelerate these routines, this paper implemented double-double precision vector operation of Lis by using AVX instructions instead of SSE2. Our vector operation routines with AVX achieved up to 2.3 times speed up from the same routines with SSE2, when vector data is included in L3 cache.

1. はじめに

計算性能の向上に伴い，高精度で計算を行うことが多くの場面で必要になってきている．4 倍精度で効率良く計算する手法の中に，倍精度変数を 2 つ用いて 1 つの変数の値を保持する倍々精度演算がある．

倍々精度の演算には倍精の演算と比較してかなりの計算時間がかかる．現在，反復法ライブラリ Lis[1][2] では，Intel の SingleInstruction Multiple Data (SIMD)拡張命令である Streaming SIMD Extensions 2 (SSE2)を用いて倍々精度演算の高速化が実装されている．

一方，ハードウェアの進化により，Sandy Bridge マイクロアーキテクチャに Intel Advanced Vector Extensions (AVX)と呼ばれる SSE2 に代わる拡張命令が新たに導入された．

本研究では，このAVX命令をLis内で利用されるSIMD拡張命令に適用し高速化を図り，特性分析を行うこととした．

2. 倍々精度演算

図 1 に，Lis での倍々精度のデータ構造を示す．

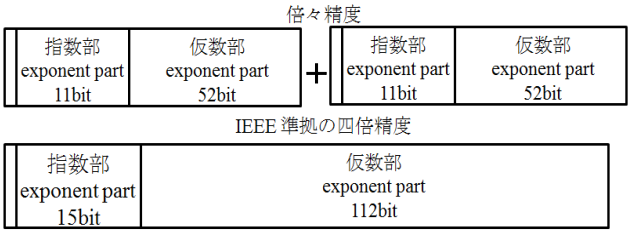

図 1 倍々精度のビット数

Fig.1 bit number of Double-Double precision

Lis では，四倍精度を実装するために，Bailey が提案した倍精度浮動小数点数を用いた"Double-Double"精度のアルゴリズム[3]を SSE2 を用いて倍々精度で実装している．

Bailey の Double-Double 精度のアルゴリズムにおいて，Double-Double 精度浮動小数 a を

a = a.hi + a.lo , 1/2 ulp(a.hi) ≥ |a.lo|（上位 a.hi と下位 a.lo は倍精度）とし，四倍精度演算を倍精度の四則演算の組合せで実現する．これは Dekker[6]と Knuth[7]アルゴリズムに基づいている．

倍精度の仮数部は 52bit であるため，実装される倍々精度の仮数部は 104bit となる．これは IEEE 準拠の四倍精度の仮数部 112bit に比べて 8bit 少ない．しかし，精度としてはほぼ同様である上，四倍精度演算に比べて倍々精度演算は計算が複雑でなく，整数演算による四倍精度演算のエミュレートより高速な演算を行うこと可能である．

倍精度 2 つを用いての実装を行う場合，SIMD 命令の使用を行うことが可能なため，Lis では四倍精度を倍々精度

[†1] 工学院大学情報学部
Faculty of Informatics, Kogakuin University
[†2] インターフェイス株式会社
Interface Co.,Ltd.
[†3] 筑波大学図書館情報メディア系
Faculty of Library, Information and Media Science University of Tsukuba

を用いて実装している．

3. 対象となる演算

3.1 AVX による実装と高速化

本研究では反復法ライブラリをベースとしたため，**表 1** に示す演算のみを実装した．ただしこの実装は一般的な用途にも適用可能なものである．

表 1 演算の一覧

Table 1 list of calculation

演算の名称 Name of calculation	演算 calculation	ロード，ストア Load, Store
axpy	$y = \alpha x + y$	2，1
axpyz	$z = \alpha x + y$	2，1
xpay	$y = x + \alpha y$	2，1
scale	$x = \alpha x$	1，1
dot	$val = x \cdot y$	2，0
nrm2	$val = \|x\|_2$	1，0

ここで，α 及び val は倍々精度のスカラー値，$\boldsymbol{x}$，$\boldsymbol{y}$及び $\boldsymbol{z}$ は倍々精度のベクトルである．

3.2 倍々精度加算

Lis では，Dekker と Knuth のアルゴリズムに基づいて，**図 2** の方法で丸め誤差のない倍精度の加算を実装している．

```
(Ⅰ) |x| ≥ |y| が仮定できる場合：
  FAST_TWO_SUM(x,y,s,e)
  {
    s = x + y
    e = y – (s – x)
  }

(Ⅱ) |x| ≥ |y| が仮定できない場合：
  TWO_SUM(x,y,s,e)
  {
    s = x + y
    v = s – x
    e = (x – (s – v)) + (y – v)
  }
```

図 2 丸め誤差のない倍精度加算

Fig.2 Rounding error free addition

これら(Ⅰ),(Ⅱ)を用いることで，倍々精度演算 $a = b + c$ を計算することができる．倍々精度の加算は，ある倍々精度浮動小数点数 x の上位 64bit を *x.hi*，下位 64bit を *x.lo* とし，fl*(x + y)* を $x + y$ の倍精度加算の結果， err*(x + y)* を，$x + y$ = fl*(x + y)* + err*(x + y)* を満たす $x + y$ の倍精度加算を行ったときの丸め誤差部分であるとすると，まず b と c の上位 *b.hi* と *c.hi* に丸め誤差のない加算を行い：

b.hi + *c.hi* = fl*(b.hi* + *c.hi)* + err*(b.hi* + *c.hi)*

とし，次に，b と c の下位と err*(b.hi*+*c.hi)*の加算：

err*(b.hi* + *c.hi)* = fl*(b.lo* + *c.lo* + err*(b.hi* + *c.hi))*

を行うと， fl*(b.hi* + *c.hi)* + err*(b.hi* + *c.hi)* は倍々精度加算 $b + c$ の近似となる Lis では，高速な倍々精度の演算を目的としているので下位の誤差 err*(eh + b.lo + c.lo)* は無視する．**図 3** に倍々精度加算 $a = b + c$ の方法を示す．

```
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)
}
```

図 3 倍々精度加算

Fig.3 Double–Double precision addition

3.3 倍々精度乗算

加算と同様に $x * y = \mathrm{fl}(x * y) + \mathrm{err}(x + y)$ であるとする．**図 4** に，倍々精度の乗算を行う際のアルゴリズムを示す．SPLIT は，倍精度小数 x を $x = h + l$ に分割する．ここでの h は x の仮数部の上位 26bit であり，l は残りの 26bit である．

```
SPLIT(x,h,l)
{
  t = 134217729.0 * x
  h = t – (t – x)
  l = x – h
}

TWO_PROD(x,y,p,e)
{
  P = x * y
  SPLIT(x,xh,xl)
  SPLIT(y,th,tl)
  e = ((xh * yh – p) + xh * yl + xl * yh) + xl * yl
}
```

図 4 丸め誤差のない倍精度乗算

Fig.4 Rounding error free multiplication

図 2 と**図 4** を用いることで倍々精度乗算 $a = b * c$ を計算できる．$p1$ = fl(*b.hi* * *c.hi*) ，$p2$ = err($b.hi * c.hi$) とするとき，倍々精度乗算には，まず *b.hi* と *c.hi* に丸め誤差のない乗算を行い， *b.hi* * *c.hi* = fl*(b.hi* * *c.hi)* + err*(b.hi* * *c.hi)* を求め，

次に $b.hi$ と $c.lo$ の乗算結果と， $b.lo$ と $c.lo$ の乗算結果と， $p2$ の加算：

$$p2 = \mathrm{fl}(p2 + \mathrm{fl}(b.hi * c.lo) + \mathrm{fl}(b.lo * c.hi))$$

を行うと， $p1 + p2$ は倍々精度乗算 $b * c$ の近似となる．**図 5** に倍々精度乗算 $a = b * c$ の方法を示す．

```
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,b.lo)
}
```

図 5 倍々精度乗算

Fig.5 Double–Double precision multiplication

3.4 対象の AVX による実装と高速化

AVX は SSE2 に替わる SIMD 拡張命令であり，SSE2 が 128bit のデータに対して SIMD 演算を行うことが出来るが，その拡張である AVX は 256bit のデータに対して SIMD 演算を行うことができる．

SIMD 命令は 1 命令で複数のデータを処理するような命令であるが，SSE はその数が倍精度にして 2 で，AVX は 4 である．厳密には AVX は同時処理数を今後増やせるようになっており，またアセンブリレベル，マシン語レベルでの違いもある．しかし主要な C コンパイラで利用できる SSE，AVX 用の組込関数では多くの場合同時処理数以外を意識する必要は無い．

倍精度演算を行う場合，SSE2 と AVX の主な違いは一命令で倍精度浮動小数点数を同時に 2 つ処理できるか 4 つ処理できるかである．少なくとも C 言語上では，アラインメントを意識する必要はあるものの，それ以上の違いはほとんど無い．ただし，SSE2 は 128*1bit レジスタであるのに対し，128*2bit レジスタとして実装されているため，水平演算に関して 128bit 境界を越えての演算を行うことが出来ないため，dot や nrm2 の最終結果を出す部分でプログラムの実装方法が異なる．

ベースとした Lis ライブラリでは既に SSE2 が用いられているので主な実装作業は SSE2 から AVX への置き換えとなる．これは基本的に同時処理数を変更すればよいが具体的には SIMD 命令に対応する組込関数の名前，ループ内でのインデックス計算，端数処理を変更し，配列のアラインメントを合わせる必要がある．

その概要を**図 6** の擬似コードで示す．

SSE		AVX
x = load128(vx[i])		x = load256(vx[i])
y = load128(vy[i])		y = load256(vy[i])
x = mul128(x,a)	→	x = mul256(x,a)
x = add128(x,y)		x = add256(x,y)
store128(vx[i],x)		store256(vx[i],x)
i += 2		i += 4

図 6 擬似コードによる SSE2 から AVX への置換の概要

Fig.6 pseudo Code of SSE2 and AVX

図 6 は，axpy 演算を行う疑似コードである．各演算を行う際に，128bit の SSE2 レジスタを利用した倍精度 2 つの SIMD 演算を，256bit の AVX レジスタを利用した倍精度 4 つでの SIMD 演算を行うものに変更している．

実際には C 言語で AVX 命令で用意されている命令が 1 対 1 で対応する組込関数を用いて記述した．

4. 数値実験

ここでは，倍々精度のベクトルに対する axpy 演算，dot 演算について，データサイズやスレッド数による性能評価を行った後，その他のベクトル演算に対しても SSE2 と AVX による高速化率に対して比較を行う．

4.1 環境と条件

以下の環境で実験を行った．

CPU : Intel Core i7 2600K
- -Intel Sandy Bridge マイクロアーキテクチャ
- -4 コア
- -L3 キャッシュ 8MB
- -動作周波数 3.4GHz

コンパイラ : Intel C/C++ Compiler 12.0.3
- -オプション–O3–xAVX–openmp–fp-model precise

メモリ : DDR3-1333 Dual Channel 16GB

OS : Fedora16

比較を行うため AVX 版及び SSE2 版に加えて SIMD 命令を用いないスカラー版を用意した．ただし，SSE2 版は 128bit の AVX 命令を，スカラー版は AVX のスカラー命令を用いた．実際には AVX と SSE2，スカラーでは同時処理数以外にも若干の違いはあるが, SSE2 版はソースコードの変更を必要とせず容易に AVX 命令を利用することができる．また，スカラー版は自動ベクトル化を抑制するために”-no-vec”を付加した．

また，今後のデータで性能を便宜上 FLOPS で表すが，これは単精度演算を対象にしたものでなく，倍々精度演算の性能である．

4.2 axpy 演算の性能分析

まず**図 7** に示すのはベクトル演算 axpy を 1 スレッドで実

行した際に N を 1 から 400000 まで変化させたときの結果である．

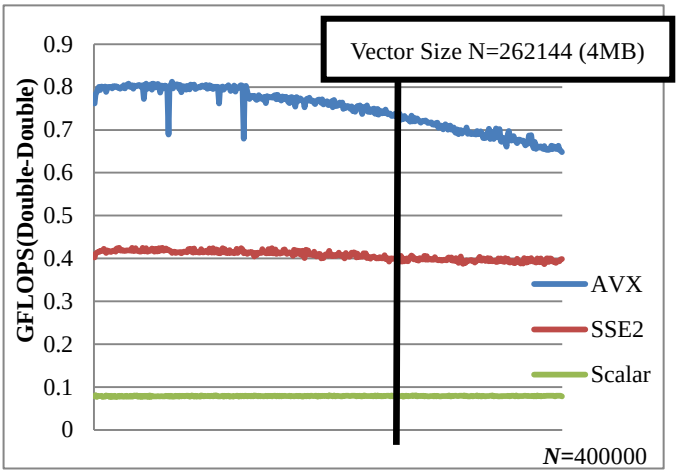

図 7 axpy(1 スレッド)
Fig.7 axpy (1 Thread)

1 スレッドでの実験の結果，SSE2 が最大 0.4GFLOPS の性能を出しているのに対し，AVX は最大 0.8GFLOPS と，約 2 倍の性能を引き出すことができた．SSE2 は，スカラーの結果と比較して約 4 倍以上の性能が出ている．

理論上ではスカラーは SSE2 の半分の性能が出るはずだが，このような結果となってしまった．倍々精度の演算アルゴリズムは SSE2 とスカラーで同様だが，SSE2 と違いスカラーはキャッシュのロードストアが複数の要素でまとめて行われることがないため，このような結果になったと考えられる．

AVX はベクトルサイズの増加に従って性能が徐々に減少し，N が 400000 のとき，0.65GFLOPS 程に減少してしまった．減少の下限に対して調べるため，ベクトルサイズ N を 1200000 まで増やしてまで実験を行ったところ，L3 キャッシュのサイズを超えたところから徐々に減少し始め，最終的には 0.6GFLOPS 程度になりサイズによらず一定の性能を出すようになり，SSE2 はほぼ 0.4GFLOPS で変わらず，1 スレッドの場合は計測したサイズでは，AVX と SSE2 の性能差が埋まることはなかった．

実際のアプリケーションなどで利用する際には，キャッシュ容量に収まるよう問題を分割するように調整することで，計算機の性能を十分に引き出せるのではないかと考えられる．

次に，axpy 演算を 4 スレッドでベクトルサイズ N を 400000 まで増加させたときの結果を**図 8** に示す．

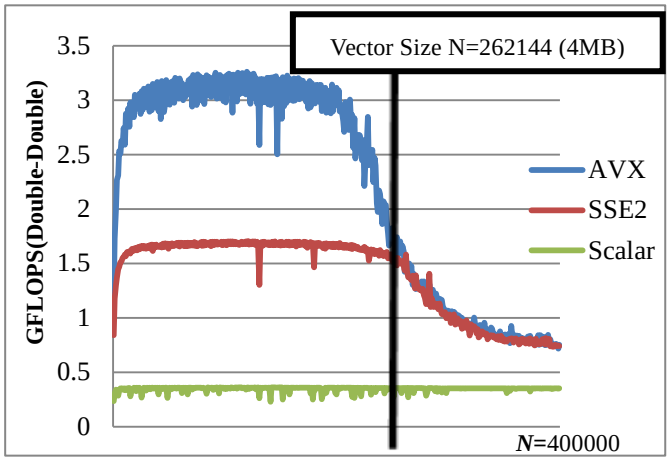

図 8 axpy(4 スレッド)
Fig.8 axpy (4 Threads)

実験の結果，SSE2 は最大 1.7GFLOPS，AVX は最大 3.3GFLOPS と，約 1.9 倍の性能を引き出すことができた．1 スレッド同様に，スカラーはロードストアが最適化されていないため，SSE と比べて 20%程の性能であった．

4 スレッドでは，AVX においてベクトルサイズが一定の値を超えたところから減少を始め，キャッシュサイズを超えるサイズになると，性能が SSE2 と同様になってしまった. SSE2 も同様にベクトルサイズが増加すると性能が低下するが，キャッシュサイズを超えたところから減少が始まることがわかる．このことから，メモリ性能がボトルネックになっていると考えられる．

また, SSE2 の計算性能の減少は 1 スレッドのときには見られなかったが，マルチスレッドにすることで，メモリへの要求が高まり，メモリ性能の限界がより顕著に現れるようになったと考えられる。

次に，SSE2，AVX のスレッド数を 1 から 8，ベクトルサイズ N を 1 から 400000 まで増加させたときの，SSE2 に対する AVX の向上率を**図 9** に示す．

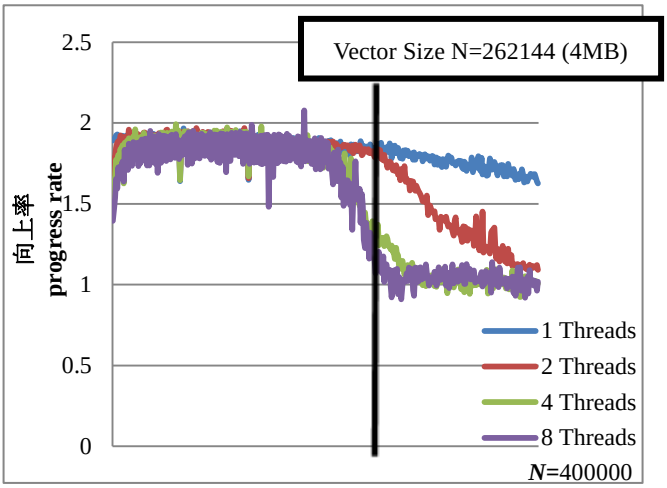

図 9 各スレッドの向上率(axpy)
Fig.9 progress rate of Threads (axpy)

スレッド数やベクトルサイズを増加させても，AVX は SSE2 を下回らず，ベクトルサイズがキャッシュサイズを超えない値においては約 2 倍の性能を引き出せていることがわかる．

4 スレッド，8 スレッドのときの減少傾向は同様だが，2 スレッドのときの性能は，4 スレッド，8 スレッドのときと比べて減少が緩やかであった．しかし，最終的には 1 スレッド時以外は SSE2 と同様の性能になる．

これらのことから，SSE2，AVX 両方の演算において，ベクトルサイズをキャッシュサイズに最適化できるよう，問題を分割することで，計算機の性能を十分に引き出し，並列化の効果を十分に発揮できると考えられる．

4.3 dot 演算の性能分析

次に，dot 演算を 1 スレッドでベクトルサイズ N を 1 から 400000 まで増加させたものときの結果を**図 10** に示す．

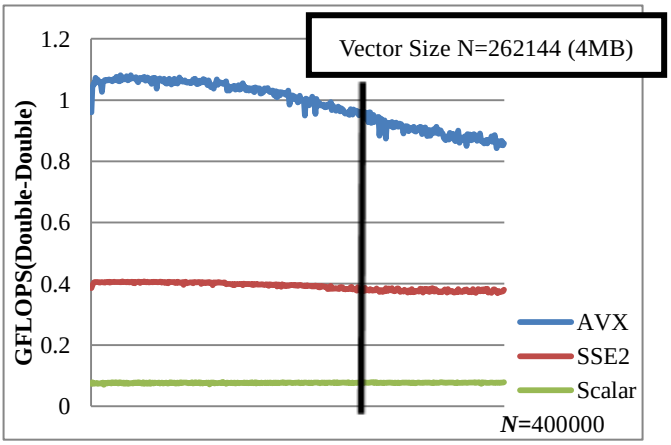

図 10 dot(1 スレッド)
Fig.10 dot (1 Thread)

1 スレッドでの実験の結果，SSE2 が 0.4GFLOPS の性能を出しているのに対し，AVX は 1GFLOPS と約 2.5 倍の性能を引き出すことができた．スカラー演算は約 0.1GFLOPS と axpy 演算でのときと同様に，SSE2 と比較してロードストアの違いから，性能が引き出せていない結果となった．

axpy 演算と dot 演算の大きな違いは，dot 演算は結果を倍々精度のスカラー値に対して累積していくことである．

axpy と比較して演算量に対するメモリアクセス量が減るため，AVX の高速化がより有効になっている．

axpy のときと同様に，AVX に対して N を 1200000 まで増加させて実験を行ったが，0.8GFLOPS ほどで安定した性能で演算を行うようになった．減少の傾向も axpy のときと同様であり，1 スレッドのときは SSE2 と比較して AVX の方がより高速であった．

次に，dot 演算を 4 スレッドでベクトルサイズ N を 1 から 400000 まで増加させたときの結果を**図 11** に示す．

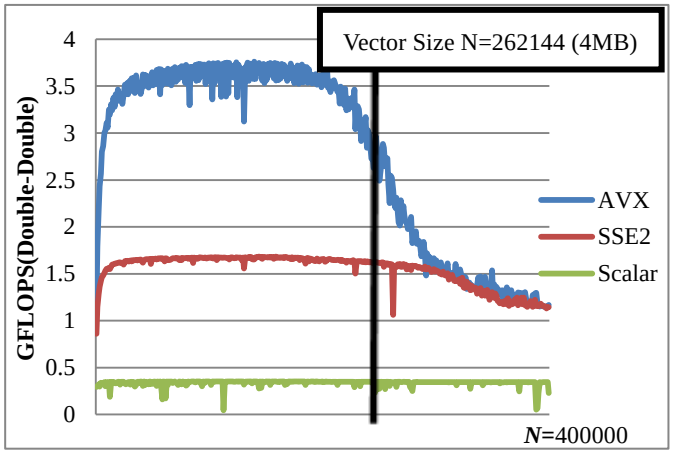

図 11 dot(4 スレッド)
Fig.11 dot (4 Threads)

4 スレッドでの実験の結果，SSE2 が 1.6GFLOPS の性能を出しているのに対し，AVX は 3.7GFLOPS と，2.3 倍の性能を引き出すことができた．1 スレッドのときと同様に，Scalar は十分に性能が引き出されなかった．

4 スレッドでは，axpy 演算の 4 スレッド同様，AVX においてベクトルサイズが一定の値を超えたところから減少を始め，キャッシュサイズを超えるサイズになると，性能が SSE2 と同様程度になった．

次に，SSE2，AVX のスレッド数を 1 から 8 ベクトルサイズ N を 1 から 400000 まで増加させたときの，SSE2 に対する AVX の向上率を**図 12** に示す．

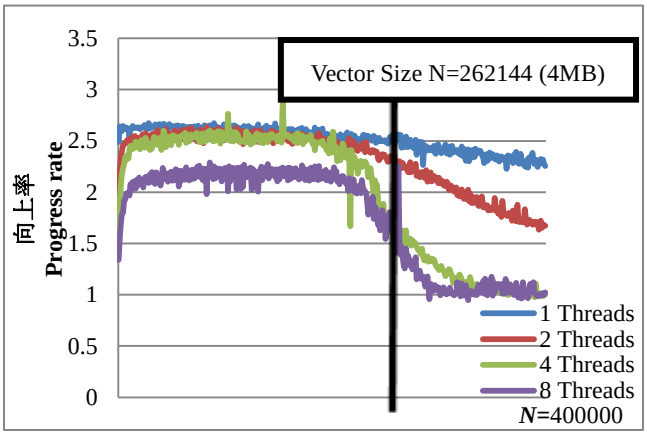

図 12 各スレッドの向上率(dot)
Fig.12 progress rate of Threads (dot)

axpy 同様，スレッド数やベクトルサイズを増加させても，AVX は SSE2 を下回らないことがわかる．性能については，SSE2 と比較して最大 2.6 倍ほど出ているが，上述したとおり理論上では AVX は SSE2 の 2 倍の性能しか出ないはずであるので，内部演算やキャッシュとのアクセスが SSE2 と比べ AVX がより最適化されているのではないかと考えられる．

4.4 その他のベクトル演算の性能分析

次に，axpy，dot も含めた全てのベクトル演算について 4 スレッドで計測した際の性能を**図 13** に示す．ただし，上記の axpy，dot の演算から，キャッシュサイズ以上のベクトルサイズ N の性能に関しては SSE2 と同様の値に収束することが分かったため，今回の計測で用いたベクトルサイズ N を，入出力に用いるデータが L3 キャッシュサイズのほぼ半分になる場合の結果を示した．なお，axpyz などは入出力の数が他の演算より多く Scale などは少ないため，各々の演算によってベクトルサイズ N は異なる．

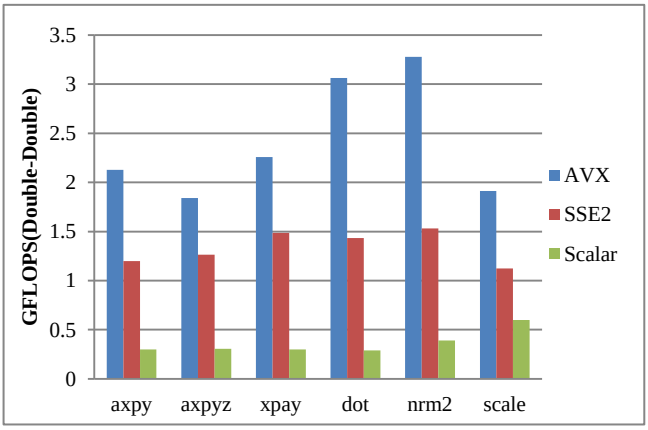

図 13 ベクトル演算の性能
Fig.13 performance of vector calculation

左側 3 つの axpy，axpyz 及び xpay は実質的に同じ処理であるので，性能はほぼ同じものとなる．この中で axpyz が他の 2 つの演算と比べて AVX 性能が落ちているのは，アクセスするベクトルが 1 本増えたためと考えられる．
axpy 系列ではこれら 3 つの結果を平均して性能の向上率が SSE2→AVX では 1.7 倍となった．

dot 及び nrm2 演算に関しては，axpy 系列の演算と比較し，結果の巻き戻しがなくメモリアクセス量が少なくなるため，AVX の高速化がより有効になり，SSE2 と比較してほぼ 2 倍となった．いずれのベクトル演算でもキャッシュに収まる場合は，SSE2 で高速化したものに対し，AVX での高速化が有効であることがわかった．

5. まとめ

本研究では，AVX 命令を反復法ライブラリ Lis 内において利用される SIMD 拡張命令に適用し高速化を図り，特性分析を行った．ベクトル演算において，SSE2 と比較して，問題がキャッシュサイズに収まる範囲内では 4 スレッドでの計測実験において 1.4 ~ 2.3 倍の性能となった．スレッド数を減らした 1 スレッドでの試行においては，キャッシュサイズを超えても SSE2 と比較して AVX による高速化に成功した．

AVX によるベクトル演算では，SSE2 よりもメモリアクセスに対する要求が強くなる．そのため，高い性能を達成するためにはデータをキャッシュに収まる範囲で計算をさせる重要性が確認できた．

参考文献

1) 小武森恒，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(June 2008)
2) 反復解法ライブラリ Lis, http://www.ssisc.org/lis/
3) Bailey, D.H.: A fortran-90 double-double library. http://www.nersc.gov/~dhbailey/mpdist/mpdist.html
4) 小武守恒,藤井昭宏,長谷川秀彦,西田晃：SSE2 を用いた反復解法ライブラリ Lis4 倍精度版の高速化，情報処理学会研究報告，2006-HPC-108, pp.7–12 (2006).
5) 小武守恒，藤井昭宏，長谷川秀彦，西田晃：倍精度と 4 倍精度の混合型反復法の提案，HPCS2007, pp.9–16 (2007).
6) Dekker,T.: A floating-point technique for extending the available precision, Nu-merische Mathematik, Vol.18, pp.224–242 (1971).
7) Knuth,D,E. : The Art of Computer Programming: Seminumerical Algorithms,Vol.2, Addison-Wesley (1969).
8) Bailey, D,H.: High-Precision Floating-Point Arithmetic in Scientific Computation,
Computing in Science and Engineering, pp.54–61 (2005).