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]

計算性能の向上に伴い，高精度による計算が多くの場面で可能となっている．4 倍精度を効率良く実現する手法として，2 つの倍精度変数で 1 つの 4 倍精度変数を表現する倍々精度演算がある．本研究では，疎行列とベクトルの演算に使われる基本演算を AVX 命令を用いて高速化し，性能を決定するパラメタについて分析を行うことにより，倍々精度演算を AVX で行う際の効果を示した．

AVX 命令を用いた場合，同時演算数の増加，3 オペランド化によるレジスタ退避，復元処理の減少などの効果が大きく，SSE2 の性能と比べて，キャッシュに収まる範囲のベクトル間の演算では約 1.7 から 2.3 倍の性能となったが，キャッシュに収まらない場合は，キャッシュアクセス，メモリアクセスが大きなボトルネックになることがわかった．倍精度の疎行列と倍々精度のベクトルの積では，約 1.1 から 1.9 倍の性能となり，メモリアクセスはボトルネックとならず，疎行列の 1 行あたりの非零要素の数が性能に大きな影響を与えていることがわかった．これらの結果から，倍々精度の疎行列ベクトル積の性能を予測する 1 つの指標を導出した．

AVX Acceleration of Sparse Matrix-Vector Multiplication in Double-Double

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

As computing performance is improved generation after generation, high precision computation becomes possible in many situations. One of the efficient methods to perform quadruple precision is to use Double-Double precision which uses two double precision variables for one quadruple precision variable. In this paper, the authors tuned basic operation kernels of sparse matrices and vectors in Double-Double precision using AVX, and analyzed their performance.

The AVX speedup ratio of the Double-Double vector operations is from 1.7 to 2.3 when data stored in the cache. The reason of performance acceleration is number of operations in the same time and elimination of backup and recovery values on registers by three operands instruction, The AVX performance decreases when data not stored in the cache, because of cache hit ratio and memory bandwidth. The AVX speedup ratio of the product of Double precision sparse matrix and Double-Double precision vector is from 1.1 to 1.9. An average number of nonzero elements per row affects to the performance, but a memory bandwidth does not affect to the performance. The authors define one metric to forecast the AVX performance of the product of sparse matrix and vector in Double-Double.

1. はじめに

計算性能の向上に伴い，高精度で計算をすることが多くの場面で可能となってきている．また，CG 法等のクリロフ部分空間法の収束性は丸め誤差に大きく影響されるため，収束の改善を図るには高精度演算が有効である[1].

しかし，Fortran などに実装されている整数演算による 4 倍精度演算の実行には倍精度演算と比較してかなりの計算時間がかかる[2]. 効率良く高精度計算をする手法のひとつに，倍精度変数を 2 つ用いて 1 つの 4 倍精度変数の値を保持し，4 倍精度演算を実行する倍々精度演算という手法がある[3].

反復解法ライブラリ Lis[4] [5]では，Intel の Single Instruction 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，Intel の Sandy Bridge マイクロアーキテクチャに 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．

われわれは，Lisで倍々精度演算に用いられているSSE2命令をAVX命令に置き換えることにより，ベクトル演算にどのような性能の向上が生じるかについて，研究を進めてきた[6].

本論文では，はじめに倍々精度演算の概要とその演算，次にAVXの特徴と今回実験に用いる実験環境，最後に倍々精度を用いたベクトル演算と疎行列ベクトル積に対する数値実験を行い，AVXによる倍々精度演算の実装の効果と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[3]を用い，倍精度変数 2 つを用いて 4 倍精度変数を実装する手法である．これによって生成され

[†1] 工学院大学情報学部
Faculty of Informatics, Kogakuin University
[†2] 筑波大学図書館情報メディア系
Faculty of Library, Information and Media Science, University of Tsukuba

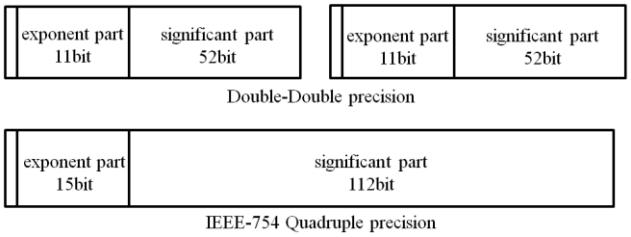

図 1　倍々精度のビット数

Figure 1　Bit pattern of Double-Double precision number

る”Double-Double”精度の浮動小数点変数を倍々精度と呼ぶ.

"Double-Double"精度のアルゴリズムにおいて倍々精度変数 a を a = a.hi + a.lo, ulp(a.hi) ≥ 2 |a.lo|（上位 a.hi と下位 a.lo は倍精度）とする．なお，ulp(*x*)は *x* の仮数部の"unit in the last place"を意味する[7]．図 1 に倍々精度変数と IEEE754 準拠の 4 倍精度変数のデータ構造を示す.

倍精度の仮数部は 52bit であるため，倍々精度の仮数部は 104bit となる．これは IEEE754 準拠の 4 倍精度の仮数部 112bit に比べて 8bit 少なく，指数部も 11bit と 4bit 少ない．しかし，精度としてはほぼ同様で，整数演算による四倍精度演算と比べ高速に実行することが可能である[2].

倍々精度の四則演算は Dekker[8]と Knuth[7]の丸め誤差のない倍精度の加算と乗算アルゴリズムに基づき，倍精度の四則演算の組み合わせで実現する.

丸め誤差のない倍精度加算アルゴリズムは，加減算のみで成り立ち，演算数は 11flops である．丸め誤差のない倍精度乗算アルゴリズムは, 乗算 9 回と加減算 15 回によって構成され，演算数は 24flops である.

このように，倍々精度演算は倍精度演算の組み合わせによって実行されるため，SIMD 命令を用いて複数の命令を同時に発行することにより高速化が期待できる.

3.　AVX の概要と実験環境

3.1　AVX の概要

AVX は SSE2 の後継となる SIMD 拡張命令である．SSE2 は浮動小数演算に使える 128bit の xmm レジスタと呼ばれる SIMD レジスタが 16 本使用できるが，AVX は 256bit の ymm レジスタと呼ばれる SIMD レジスタが 16 本使用できる. そのため, SSE2 は 2 つの倍精度のデータに対して SIMD 演算ができるが，AVX は 4 つの倍精度のデータに対して SIMD 演算ができる．厳密には AVX は今後同時処理数を増やせるような設計になっているが，今回の実験環境では 256bit である.

図 2 に AVX の SIMD レジスタの構造を示す．AVX がサポートされている Sandy Bridge 以降のアーキテクチャにおいて, xmm レジスタと ymm レジスタが 16 本ずつあるわけでなく，xmm レジスタは ymm レジスタの下位 128bit を使用し, ymm レジスタに包括されるような設計のため, SIMD レジスタは 16 本である.

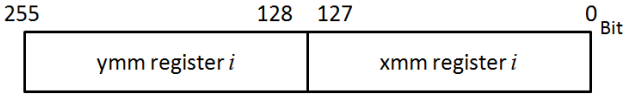

図 2　AVX の SIMD レジスタの構造

Figure 2　Architecture of AVX SIMD register

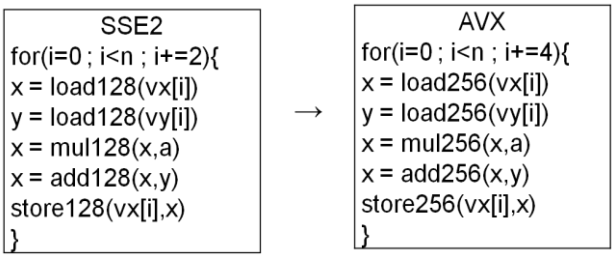

図 3　擬似 C コードによる SSE2 から AVX への置換の概要

Figure 3　Pseudo C code of SSE2 and AVX

また，このような ymm レジスタの設計から，AVX 命令は，同一 ymm レジスタ上の上位と下位の 128bit 境界を越えての水平演算を行うことはできない.

AVX では，VEX プリフィックスというプリフィックス方式が採用され, SSE2 が 2 オペランドの命令しか実行できないのに対し，3 または 4 オペランドの命令が実行できるようになり，レジスタ退避，復元処理の記述を省くことができる[9].

倍々精度演算において，SSE2 と AVX の主な違いは一命令で倍精度変数をいくつ同時に演算できるかである．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, それ以上の違いはない.

そのため，SSE2 向けのコードを AVX 向けのコードに置き換える際には，同時処理数を変更するような改変を施すだけでよい．本研究では，Lis 内の SIMD 命令に対応する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作業を行った．その概要を図 3 に示す.

3.2　実験環境

実験には，表 1 に示す計算環境を用いた．Intel Core i7 2600K は Sandy Bridge マイクロアーキテクチャであり，動作周波数は 3.4GHz，4 コア 8 スレッドの構造で，1 コアあたり乗算器と加算器を 1 つずつ持っているため，AVX を使用し倍精度を 4 つ同時に演算した場合の倍精度演算の理論ピーク性能は，$3.4\mathrm{G} \times 4(\mathrm{core}) \times 4(\mathrm{AVX}) \times 2(\text{演算器}) = 108.8\mathrm{GFLOPS}$，SSE2 を使用した場合はその半分の 54.4GFLOPS となる.

コンパイラは Intel C/C++ Compiler 12.0.3 を用い，最適化を行う"-O3"オプション，並列計算を行うため"-openmp"オ

表 1　実験環境

Table 1　Testing environment

CPU	Intel Core i7 2600K 3.4GHz Intel Sandy Bridge Microarchitecture
Number of core	4
L3 Cache Size	8MB
Memory	DDR3-1333 Dual Channel 16GB
Memory bandwidth	21.2GB/s (10.6×2)
OS	Fedora 16
Compiler	Intel C/C++ Compiler 12.0.3
Compile-options(AVX)	–O3–xAVX–openmp–fp-model precise
Compile-options(SSE2)	–O3–xSSE2–openmp–fp-model precise

プション，最適化により精度に影響を及ぼさないために"fp-model precise"オプションを用い．SSE2 向けのコードには"-xSSE2"オプションを，AVX 向けのコードには"-xAVX"オプションを用いた．

また，AVX 化による計算精度に変化がないか確認するため，全ての演算において C ソースコード上で SSE2 の結果と減算を行い，結果が 0 になることを確認した．

4.　倍々精度ベクトル演算

4.1　対象とするベクトル演算

AVX を用いた倍々精度演算の高速化の効果を確認するため，表 2 の倍々精度ベクトル演算を対象とした実験を行った．ここで，α 及び val は倍々精度のスカラー値，$\boldsymbol{x}$，$\boldsymbol{y}$および $\boldsymbol{z}$ は倍々精度のベクトルである．

また，各演算のロード，ストア数と演算数を表 2 に示す．Complexity は演算カーネル内の演算数である．

表 2 倍々精度ベクトル演算

Table 2 Double-Double precision vector operations

Name of operation	Operation	Load	Store	Complexity
axpy	$\boldsymbol{y} = \alpha\boldsymbol{x} + \boldsymbol{y}$	2	1	35
axpyz	$\boldsymbol{z} = \alpha\boldsymbol{x} + \boldsymbol{y}$	2	1	35
xpay	$\boldsymbol{y} = \boldsymbol{x} + \alpha\boldsymbol{y}$	2	1	35
scale	$\boldsymbol{x} = \alpha\boldsymbol{x}$	1	1	24
dot	$val = \boldsymbol{x} \cdot \boldsymbol{y}$	2	0	35
nrm2	$val = \|\boldsymbol{x}\|_2$	1	0	31

4.2　ベクトル演算の性能

4 スレッド化した AVX 向けのコードと SSE2 向けのコードでベクトルサイズ N を 10^5 で計測した結果を図 4 に示す．なお，計測は最低で 70 回繰り返し，得られた値を回数で除算した値を用いた．

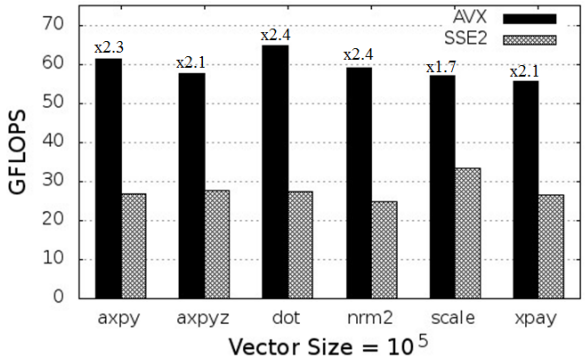

図 4　ベクトル演算の性能

Figure 4　Performance of vector operations

実験の結果，"scale"以外の演算で AVX と SSE2 の比は約 2.1 から 2.4 倍だった．AVX の理論値は SSE2 の 2 倍のはずであるが，今回”scale"以外の演算において AVX は SSE2 の 2 倍以上の倍率がでている．

この原因は，AVX は VEX プリフィックスによって発行された 3 オペランド命令が実行できることが要因の 1 つとして考えられる．SSE2 は 2 オペランド命令しか使用できないため，演算のソースオペランドを書き換えてしまうため，後で使う変数などは temp 変数などを用意して途中結果を保存しなければならない．しかし，AVX では 3 オペランド命令が実行できることにより，レジスタ退避，復元のための temp 変数などに対する"move"命令が削減される．2 オペランドから 3 オペランドになることによってレジスタ退避，復元命令が削減される例を図 5 に示す．

”nrm2”と”scale”以外の演算では，基本ループ内において倍々精度加算 1 回と乗算 1 回によって成り立つ FMA というマクロ関数を用いている[4]．FMA マクロ関数の演算量は倍々精度加算 (11flops)と倍々精度乗算(24flops)のの合計である 35flops である．FMA マクロ関数を SSE2 で実行した場合はレジスタ退避，復元処理の回数が 13 回なのに対し，AVX では 3 回と，基本ループ内の処理が 10 回削減されていた．演算量が 35flops であることを考えると，性能に大きな影響を与えることがわかる．

"scale"の倍率は約 1.7 倍で，他のベクトル演算の倍率に比べて低くなっている．これは SSE2 の"scale"の性能が良いためである．また，"scale"で用いている倍々精度乗算を 1 回行う MUL マクロ関数は，レジスタ退避，復元処理が

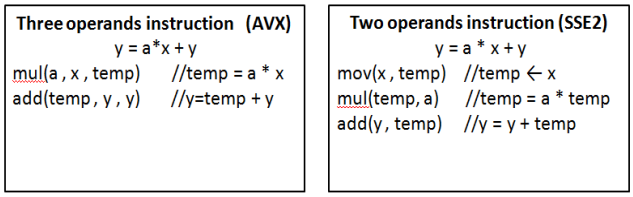

図 5　3 オペランドと 2 オペランドの違い

Figure 5　Difference between three and two operands

SSE2 では 5 回なのに対し，AVX では 3 回であり，FMA マクロ関数を使用している他のベクトル演算と比べて 3 オペランド化の効果があまり見られないことも倍率が低い原因であると考えられる．

なお，SSE2 のソースコードに対し，コンパイルオプション"-xAVX"を使いコンパイルすると，AVX の搭載されているマシン上では VEX プリフィックスにより xmm レジスタを用いた 3 オペランド命令が発行されるため，コンパイルオプション"-xSSE2"を使った場合より性能が向上する．今回の SSE2 の実験項目に対して"-xAVX"コンパイルオプションを付けた場合，各々20 から 40%程度性能が向上し，SSE2 コードに対し"-xAVX"を用いて 3 オペランド化したものは，“-xAVX”を用いた 3 オペランドの AVX に対し約 1.5 倍から 1.9 倍となった．

4.3 “axpy"の性能分析(1 スレッド)

"axpy"を 1 スレッドでベクトルサイズ N を 10^3 から 8.0×10^5 まで 10^3 ずつ増加させたときの結果を図 6 に示す．L3 キャッシュのサイズは 8MB で，長さ 2.6×10^5 の倍々精度ベクトルを 2 本格納できる．

ベクトルサイズが 10^3 から 1.5×10^5 前後までは SSE2 と比べ約 2.3 倍だが，サイズの増加に伴い徐々に性能が低下し最終的に約 1.7 倍となる．

ベクトルがキャッシュに収まるとき，AVX の性能は約 16.8GFLOPS となった．1 コアあたりでの AVX を用いた場合の理論ピーク性能は 27.2GFLOPS なので，理論ピーク性能の 61.8%である．

この原因の 1 つとして，倍々精度の加算と乗算アルゴリズムの演算のバランスが悪いことが考えられる．Core i7 2600K は，乗算器と加算器が 1 つずつ搭載されている．しかし，FMA マクロ関数は，加減算命令 26 回，乗算命令 9 回によって成り立ち，加減算と乗算の回数に偏りがあるため，乗算器と加算器が並列して動作することを前提とした理論ピーク性能がでることはない．乗算と加減算のバランスを考慮すると，FMA マクロ関数の理論ピーク性能は約 18.3GFLOPS となり，前述のベクトルがキャッシュに収まる場合の性能 16.8GFLOPS は理論ピーク性能の 91.8%である．演算以外の繰り返し構文の分岐命令などを考慮すると，ベクトルがキャッシュに収まるとき，"axpy"は演算器の限界まで性能がでていることがわかる．

また，ベクトルがキャッシュに収まらない場合，SSE2 はサイズの増加に対してほとんど変化がないのに対し，AVX はベクトルの増加に従って徐々に性能が落ち，SSE2 の 1.7 倍程度まで低下する．この原因の 1 つとして，AVX は SSE2 と比べ同時演算数の増加によってメモリへの要求が高いため，キャッシュストールが発生していることが考えられる．

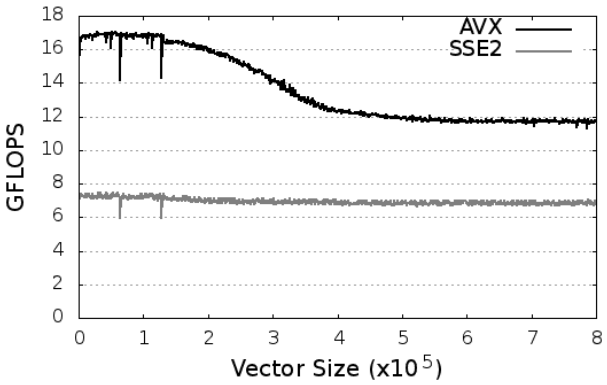

図 6　"axpy"の性能 (1 スレッド)

Figure 6　Performance of "axpy" operation (1 Thread)

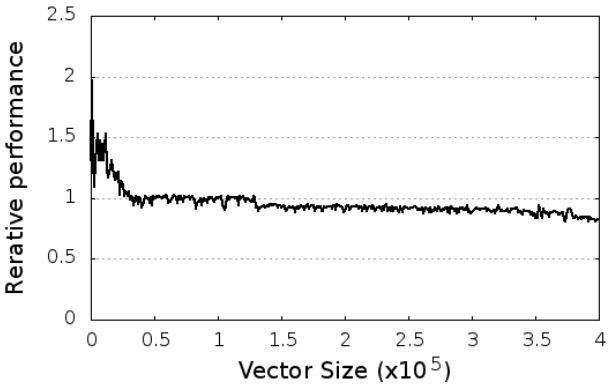

図 7　メモリアクセスの相対性能 (1 スレッド)

Figure 7　Relative performance of memory access (1 Thread)

キャッシュ，メモリ性能を分析するため，"axpy"のロード，ストア命令のみを実行し性能の測定を行った．FMA マクロ関数は load 命令 6 回, store 命令 2 回によって成り立っており，これら 8 回の命令のみを AVX 命令で行い，ベクトルサイズを 10^3 から 4.0×10^5 まで 10^3 ずつ増加させ測定した. なお, コンパイラによる最適化で消去されないよう，計測ループ外で標準出力への出力処理を行った．"axpy"がベクトルサイズ 10^5 のときの性能を基準としたキャッシュ，メモリアクセスの相対性能(相対実行時間の逆数)を図 7 に示す.

実験の結果，ベクトルサイズが小さい間はキャッシュ，メモリアクセスの性能が高いが，ベクトルサイズが約 1.3×10^5 に達すると性能が低下する結果となった．

この結果から，"axpy"の性能が低下する原因の 1 つとしてキャッシュ，またはメモリアクセスがボトルネックになっていることがわかった，データがキャッシュに収まらないとき, データサイズと実測時間から 1 秒あたりに CPU が処理しているデータ量を求めると，AVX は約 12GB/s となる. 今回のマシンのメモリ帯域は理論値最大 21.2GB/s であるため，メモリの帯域限界でなく，性能の低下はキャッシュストールの発生が原因であると考えられる．

4.4 “axpy"の性能分析(4 スレッド)

次に，スレッド化の効果を確認するため， 4 スレッドでベクトルサイズ N を 10^3 から 8.0×10^5 まで 10^3 ずつ増加させたときの"axpy"の結果を図 8 に示す.

実験の結果から，ベクトルがキャッシュに収まるとき，AVX は SSE2 の比は約 2.3 倍となり，1 スレッドのときとほぼ同様の性能向上が見られた．また，4 スレッドにしたことで 1 スレッドと比べ約 3.7 倍の向上となり，スレッド化による効果があることがわかった．しかし，ベクトルがキャッシュサイズに収まらない場合は，サイズの増加に従い AVX, SSE2 どちらにおいても性能が低下し, 最終的に，AVX は SSE2 とほぼ同様の約 12GFLOPS となる.

性能が低下する原因として，キャッシュにデータが収まらなくなったことによるキャッシュストールの発生と，4 スレッド化によるメモリへの要求の増加によるメモリネックが考えられる．1 秒あたりに CPU がメモリに要求し実行しているデータ量を計算すると，AVX においてデータがキャッシュに収まるときは約 55GB/s であった．今回のマシンのメモリ帯域は理論値最大 21.2GB/s であり，キャッシュに収まる範囲においてはキャッシュアクセスが有効なためメモリ性能は制約となりにくい．しかし，データがキャッシュに収まらなくなるとキャッシュミスが発生しやすくなることにより，計算性能はメモリ性能の制約うけ，1 秒あたりに CPU が処理しているデータ量は約 12GB/s となった.

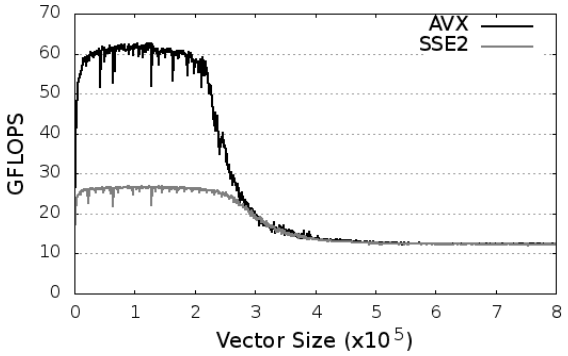

図 8　"axpy"の性能 (4 スレッド)

Figure 8　Performance of "axpy" operation (4 Threads)

4.5 “axpy"の性能分析(マルチスレッド化)

スレッド数を増加させたときのキャッシュストール，メモリネックによる性能の変化を確認するため，"axpy"においてベクトルサイズ N を 10^3 から 8.0×10^5 まで 10^3 ずつ増加に示す.

すべてのスレッド数において，ベクトルがキャッシュに収まる場合 SSE2 の 2 倍以上の性能がでているが，キャッシュに収まらない場合，1 スレッド以外は最終的に SSE2 と同性能になった．原因は，同時演算数の増加やマルチスレッド化によりキャッシュ, メモリへのデータ要求が増え，

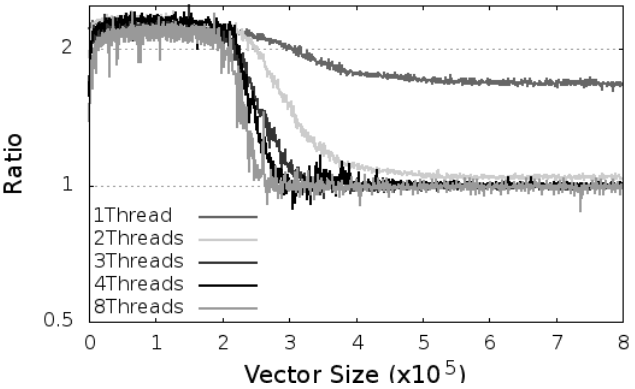

図 9　"axpy"の AVX/SSE2 の比

Figure 9　AVX speedup ratio (Performance of AVX/SSE2)

キャッシュ，メモリアクセスがボトルネックとなっていると考えられる．また，スレッド数が多いほど早くメモリボトルネックに達する傾向が見られた.

4.6　“dot"の性能分析

1 スレッドで"dot"をベクトルサイズ N を 10^3 から 8.0×10^5 まで 10^3 ずつ増加させたときの結果を図 10 に示す．"dot"は"axpy"とほぼ同様の傾向が見られたが，"axpy"より性能が高く，ベクトルがキャッシュに収まるとき，AVX では約 18.0GFLOPS, SSE2 では約 7.5GFLOPS となっていた

"dot"の内部は，"axpy"と同様の FMA マクロ関数を用いている．”axpy"は $\boldsymbol{y}_i = a * \boldsymbol{x}_i + \boldsymbol{y}_i$ を計算するのに対し，"dot"は $val = \boldsymbol{x}_i * \boldsymbol{y}_i + val$ を計算する. 違いはマクロ関数に与える引数だけである.

コンパイラにより生成されたアセンブリコードを解析すると，AVX で性能が異なる原因はコンパイラにより，スカラー値である val のストアが基本ループの外に移動され，Store が基本ループ内で行われないようになっていた.

FMA マクロ関数の理論ピーク性能である約 18.3GFLOPS と比較すると, "dot"は AVX において約 98.5%の性能がでており，演算器の性能が限界まで引き出されている．スレッド化の影響は"axpy"とほぼ同様であった.

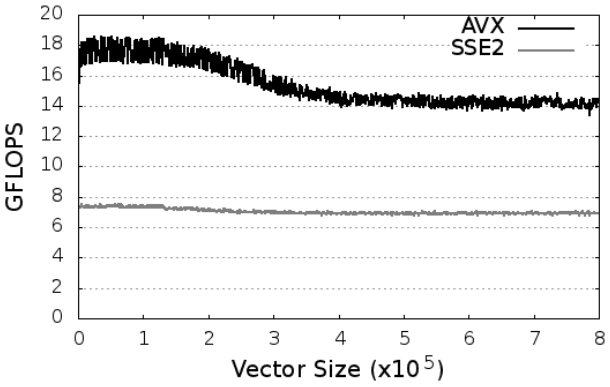

図 10　"dot"の性能 (1 スレッド)

Figure 10　Performance of "dot" operation (1 Thread)

5. 疎行列ベクトル積

本研究では，倍精度の疎行列と倍々精度のベクトルの積に対する測定を行った. 実際の反復法ライブラリにおいて，入力として倍精度の行列とベクトルが与えられることから，計算の現実的な安定化の観点では，係数行列は倍精度のままでよいと考えた.

分析には The University of Florida Sparse Matrix Collection (フロリダコレクション)[10]から入手した 15 種の疎行列と"random"を用いた．"random"は非零要素の分布による影響を調べるため，行列サイズ N が 10^5，1 行あたり 64 個の非零要素がランダムに分布している疎行列である．結果は行列サイズごとに 500 回反復計測した相加平均を用いた.

疎行列の格納形式は，Compressed Row Storage (CRS)形式[11]である．CRS 形式とは，非零要素数を nnz とすると，①サイズ N×N の正方行列 A の非零要素の値を行方向に沿って格納する長さ nnz の倍精度配列 value，②配列 value に格納された非零要素の列番号を格納する長さ nnz の整数配列 index，③配列 value と index の各行の開始位置を格納する長さ N+1 の整数配列 ptr からなる．図 11 に CRS 形式のデータ構造を, 図 12 に CRS 形式の倍精度の疎行列 A と倍々精度ベクトルx_{DD}の積を示す. 倍々精度数間の FMA マクロ関数と比べ, 乗算と加算が 1 回ずつ少なく, 演算数は 33flops である．疎行列ベクトル積の演算数は 33*nnz とした.

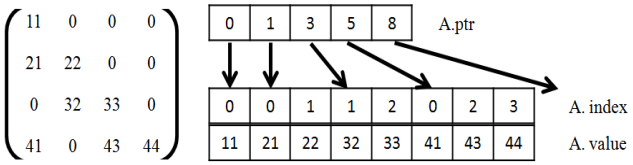

図 11　CRS 形式のデータ構造

Figure 11　Data structure of CRS format

```
for( i=0 ; i<N ; i++ ){
        for( j=A.ptr [value[i]] ; j<A.ptr[i+1] ; j++ ){
                yDD[i] =A.value[j] * xDD[A.index[j]] + yDD[i];
        }
}
```

図 12　CRS 形式の疎行列とベクトルの積

Figure 12　Product of sparse matrix of CRS format and vector

5.1 疎行列ベクトル積の性能

平均非零要素数の順に並べた 1 スレッドにおける結果を図 13 に，4 スレッドにおける結果を図 14 に示す．括弧内は平均非零要素数を示す.

実験の結果，SSE2 と比較して AVX は約 1.1 倍から 1.9 倍となった．また，"random"以外の疎行列では，平均非零要素数の数が多い疎行列は性能向上が大きい傾向がみられた．"random"は，平均非零要素数がほぼ同一の"olafu"と比べ性能が低く，非零要素の分布も性能を決定する要因であると考えられる. 疎行列の格納形式は CRS 形式なので連続したメモリアクセスとなるため，この原因は倍々精度ベクトルx_{DD}に対するキャッシュミスであると考えられる．帯行列やフロリダコレクションの疎行列のような非零要素の列番号が連続している場合が多いとき，x_{DD}は L3 キャッシュに残り続ける．しかし，"random"のような規則性のないものではx_{DD}に対するアクセスが不連続になり，x_{DD}に対するキャッシュミスが発生していると考えられる.

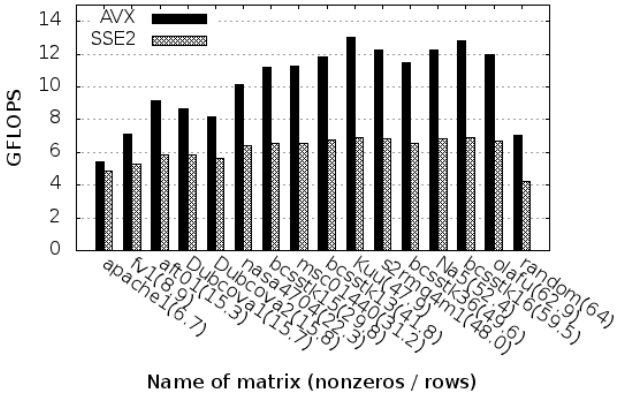

図 13　疎行列ベクトル積の性能 (1 スレッド)

Figure 13　Performance of sparse matrix vector product (1 Thread)

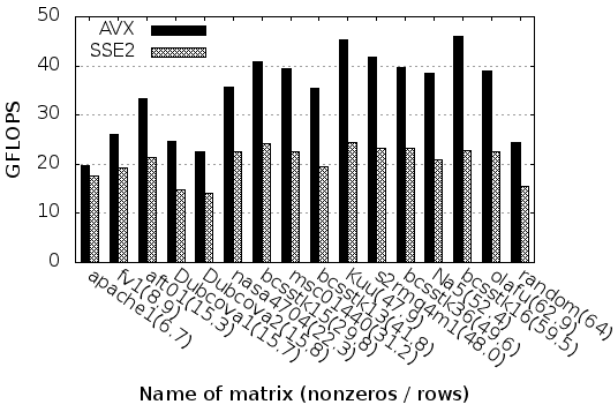

図 14　疎行列ベクトル積の性能 (4 スレッド)

Figure 13　Performance of sparse matrix vector product (4 Threads)

4 スレッドの計測結果において，SSE2 と比較して AVX は約 1.1 倍から 2.0 倍となった．ここでも平均非零要素数が多い疎行列ほど向上が大きい傾向が見られた.

4 スレッドと 1 スレッドを比べると，AVX は約 2.7 から 3.7 倍，SSE2 は約 2.5 から 3.7 倍となった．マルチスレッド化による高速化は有効である.

5.2 疎行列のサイズについての分析

疎行列のサイズ，平均非零要素数，非零要素の分布が性能に与える影響を分析するため，帯幅を m とした

- if($0 \leq j - i \leq m$) a_{ij}=value
- else $a_{ij} = 0$

を満たすテスト用帯行列を作成し，CRS 形式で格納した．

疎行列サイズが性能に与える影響を分析するため，帯幅 m = 32 のテスト用帯行列に対し，1 スレッドで行列サイズ N を 10^3 から 4.0×10^5 まで 10^3 ずつ増加させた結果を図 15 に示す． L3 キャッシュには CRS 形式の倍精度の行列と倍々精度のベクトルが，約 1.9×10^4 まで格納できる．テスト用帯行列では，キャッシュにデータが収まるとき，AVX は SSE2 の約 1.9 倍，キャッシュに収まらないとき約 1.8 倍となった．

次に，4 スレッドで，行列サイズ N を 10^3 から 4.0×10^5 まで 10^3 ずつ増加させた計測結果を図 16 に示す．キャッシュに収まるとき，AVX は SSE2 の約 1.8 倍，キャッシュに収まらないとき約 1.7 倍となった．また，1 スレッドと 4 スレッドを比較すると，AVX はデータがキャッシュに収まるとき約 3.7 倍，SSE2 は約 3.8 倍となり，キャッシュに収まらないとき AVX は約 3.5 倍, SSE2 は約 3.6 倍となった．

疎行列ベクトル積では，ベクトル演算と違い AVX の性能が SSE2 の性能まで落ちることはない．これは，係数行列を倍精度としたためメモリへの要求が低下したと考えられる．CPU が処理するデータ量は 4 スレッドでキャッシュに収まるとき 18GB/s, キャッシュに収まらないとき 15GB/s であり，いずれもメモリ帯域の理論値最大である 21.2GB/s に達しておらず，メモリはボトルネックになっていないと考えられる．ただし，ベクトル $\boldsymbol{x}$ はキャッシュにヒットすると仮定し，1 度だけメモリから読み込むことと仮定している．

これらの結果から，倍精度疎行列と倍々精度ベクトルの積では行列サイズ N による性能への影響は少ないことがわかる．

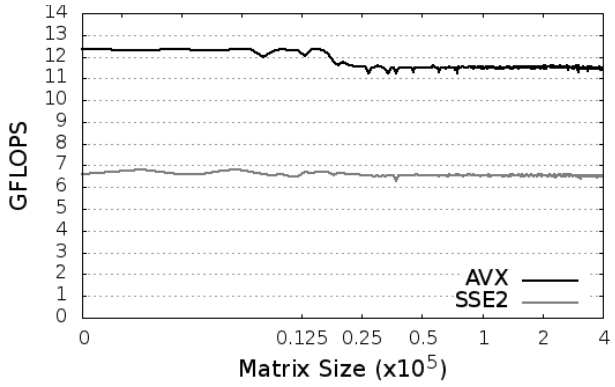

図 15　テスト用帯行列の性能 (1 スレッド)

Figure 15　Performance of band matrix (1 Thread)

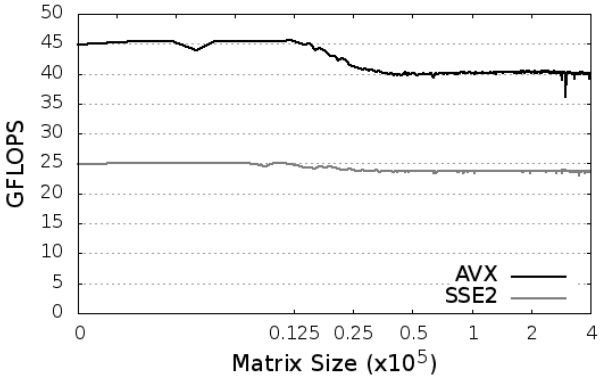

図 16　テスト用帯行列の性能 (4 スレッド)

Figure 16　Performance of band matrix (4 Threads)

5.3 平均非零要素数についての分析

平均非零要素数の影響を分析するため，テスト用帯行列の行列サイズを固定し，帯幅 m を変化させた．

1 スレッドにおいて行列サイズを 10^5 に固定し，帯幅を 1 から 100 まで変化させた性能を図 17 に示す．参考のため，平均非零要素数を帯幅とみなしてフロリダコレクションと "random" も同一グラフ上に示した．

実験の結果，帯幅の増加に従って性能が階段状に増加するが, 帯幅 m = 40 前後から性能の増加の傾きが緩やかになり，帯幅 m = 65 以上ではほぼ一定の性能になった．帯幅 m = 4 のとき，1 スレッドにおいて，AVX は 5.9GFLOPS と理論ピーク性能の約 21%，SSE2 は 4.8GFLOPS と理論ピーク性能の約 18%となり，AVX は SSE2 の約 1.1 倍であるのに対し，帯幅 40 では 1 スレッドにおいて AVX が約 11.2GFLOPS と理論ピーク性能の約 41%，SSE2 が約 6.5GFLOPS と理論ピーク性能の約 48%となり，AVX は SSE2 の約 1.7 倍の向上となった．倍精度行列と倍々精度ベクトルの積は，乗算と加減算のバランスが悪く，乗算と加減算のバランスを考慮した 1 コアの理論ピーク性能は 18GFLOPS となるため, AVX では理論ピーク性能の 62.3%, SSE2 は 73.2%の性能がでている．

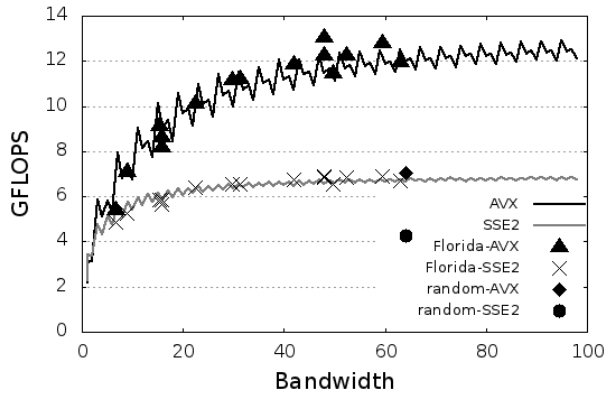

図 17　帯幅と性能の関係 (帯行列, 1 スレッド)

Figure 17　Performance of sparse matrix vector product with bandwidth (Band matrix, 1 Thread)

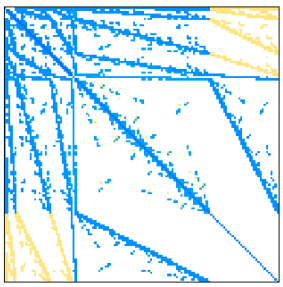

図 18　疎行列"Dubcova2"
Figure 18　Sparse matrix "Dubcova2"

性能が階段状に変化しているのは, SSE2 では 1 回の演算で倍精度 2 つ, AVX では倍精度 4 つのデータを処理するため, 端数処理を行う必要があるためと考えられる. AVX は, 同時演算数が増えたことにより端数処理が発生しやすく, テスト用帯行列とフロリダコレクションの性能差が大きい.

フロリダコレクションの疎行列の平均非零要素数と同じ帯幅のテスト用疎行列は, "random"を除きほぼ同様の性能だった. ”random”は, AVX では約 7GFLOPS, 帯幅 64 のテスト用疎行列と比べて 57.8%, SSE2 では約 4.25GFLOPS (64.1%)であった.

フロリダコレクションの疎行列の中で性能が低かった平均非零要素数 15.8 の疎行列"Dubcova2"の構造を図 18 に示す. この疎行列は行列サイズ N が 65025, 非零要素数が 1030225 の対称行列で, テスト用帯行列と比べ複雑な構造をしている. このような疎行列においても, テスト用帯行列と比較して AVX では約 81.1%, SSE2 では 91.7%となった. このことから, "random"のような特殊ケースを除き, 性能が平均非零要素数に大きく依存していることがわかる. 実際のアプリケーションではある程度データが連続した疎行列を扱うことが多いと考えられるため, 平均非零要素数から性能の予測が可能であると考えられる.

6. まとめ

本研究では, 倍々精度演算による疎行列とベクトルの基本演算を AVX 命令を用いて高速化する際の効果と問題点を論じた.

キャッシュに収まる範囲のベクトル間の演算では 1 スレッドにおいて AVX は SSE2 と比べて性能比が約 1.7 倍から 2.3 倍となった. AVX は 3 オペランド命令を実行できるため, SSE2 と比べて 2 倍以上の性能がでたが, 倍々精度演算の加減算と乗算のバランスが悪いことにより, "axpy"演算において, AVX は 16.8GFLOPS, SSE2 は 7.3GFLOPS となり, 理論ピーク性能の 21.7GFLOPS と比べ約 61.8%しか性能がでることはなかった. さらなる性能向上のためには, 加減算と乗算のバランスを考慮して改善する必要がある.

データがキャッシュに収まらない場合は, メモリアクセスが大きなボトルネックになり性能が低下する. マルチスレッド化を行った場合, メモリへの要求が高まり, メモリ性能による制約を受けることで, AVX は SSE2 と同様の性能となった.

CRS 形式で格納した倍精度の疎行列と倍々精度のベクトルの積では, AVX は SSE2 と比較して約 1.1 倍から 1.9 倍となった. テスト用に作成した帯行列において, 4 スレッドでデータがキャッシュに収まるとき約 1.8 倍, キャッシュに収まらない場合 1.7 倍となり, 疎行列ベクトル積においてはキャッシュにデータが収まらない場合においてもメモリアクセスはネックとならなかった.

CRS 形式で格納した帯行列のサイズを固定し, 平均非零要素数を変化させたところ, 帯幅の増加に伴い性能が増加するが, 帯幅が約 40 を超えると性能の増加の傾きが緩やかになり, 帯幅 65 以上になるとほぼ一定の性能だった. 計測結果を総合すると, 疎行列ベクトル積において, 性能は平均非零要素数に依存していると考えられる,

疎行列と平均非零要素数と同じ帯幅の帯行列と比較した性能は約 81.1%以上となり, 乱数によって非零要素を配置した疎行列においても約 57.8%であった.

これらのことから, SandyBridge マイクロアーキテクチャで AVX を用いた倍々精度演算は, キャッシュに収まるベクトル演算では, 理論ピーク性能の約 60%, キャッシュに収まらない場合はメモリ帯域の制約を受け, メモリ性能を上限とした性能になる. また, 倍精度疎行列と倍々精度ベクトルの積において, 平均非零要素数が 40 以上の一般的な疎行列では, 理論ピーク性能の約 40%, 性能がでにくいランダムな疎行列においても約 20%以上の性能になると予測できる.

今後の課題として, 他のマシンやより多くの疎行列に対して検証, 分析を行うことでより詳細な予測ができると考えている. また, 倍々精度演算の加減算と乗算のバランスや, プリフェッチなどによるメモリアクセスの改善を行うことで性能向上が期待できると考えられる.

謝辞　HPCS2013 の査読者から有益なコメントを頂きました. ここに感謝の意を表します.

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 Linear Algebra (2003).
[2] Bailey, D ,H.: High-Precision Floating-Point Arithmetic in Scientific Computation, computing in Science and Engineering, pp. 54–61 (2005).
[3] Bailey, D, H.:A fortran-90 double-double library. http://crd-legacy.lbl.gov/~dhbailey/mpdist/

[4] 反復解法ライブラリ Lis, http://www.ssisc.org/lis/
[5] 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 (June 2008)
[6] 菱沼利彰，浅川圭介，藤井昭宏，田中輝雄，長谷川秀彦: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, 情報処理学会研究報告, Vol.2012-HPC-135, No.16, pp.1-6 (2012, 8)
[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] インテル Advanced Vector Extensions プログラミング・リファレンス, http://download.intel.com/jp/software/AVE/319433-006JA.pdf
[10] The University of Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices/
[11] Barrett, R., et al.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, SIAM pp. 57–65 (1994)