Bulletin of Aichi Univ. of Education, 64（Natural Sciences）, pp. 15 − 19, March, 2015

Intel AVX2 によるCommon Lisp言語処理系の最適化の一考察

安本 太一

情報教育講座

A Study of Optimization for Common Lisp System Using Intel AVX2

Taichi YASUMOTO

Department of Information Sciences, Aichi University of Education, Kariya 448-8542, Japan

1 はじめに

インテル社の第4世代Coreプロセッサ（Haswellマイクロアーキテクチャ）から実装されているベクトル演算命令AVX2（Advanced Vector eXtensions 2）[1] を用いて，Common Lisp [2] の組み込み関数の最適化を試みたので，その結果を報告する．対象としたCommon Lisp言語処理系は，64ビット版のKCLである [3, 4, 5, 6]．

筆者は，前年，AVX2の前のバージョンであるAVXを用いて，同様のことを行った [7]．AVXにおいては，ベクトル演算（SIMD演算）は，倍精度浮動小数点数（float64）が4つのベクトル演算だけのサポートであった．AVX2では，64ビット整数4組のベクトル演算もサポートするなどの機能向上が行われている．

筆者の前年の開発及び実験環境は第2, 3世代Core i7プロセッサ（Sandy BridgeやIvy Bridgeマイクロアーキテクチャ）を搭載したコンピュータであったが，今年のそれは第4世代Core i7プロセッサを搭載したコンピュータである．コンパイラやOSも，前年とは異なる．

このように，前年とは異なる環境下で，前年提案した手法の再評価，前年提案した手法のAVX2の64ビット整数ベクトル演算への適用とその評価を行った．

2 AVX2の概要

AVXの概要については，前年の研究報告で述べた．ここでは，AVX2に固有で，本論文でとりあげることに焦点を絞って，説明する．

AVXの機能と性能を拡張したAVX2では，256ビットのベクトルレジスタYMM（YMM0からYMM15）を，64ビット整数4組のベクトル演算でも使えるようになった．AVXでは，256ビットのベクトルレジスタYMMが使えるのは，float64が4つのベクトル演算のためだけだった，加えて，浮動小数点数と整数のベクトル演算ユニットは，AVXでは128ビット幅だったものが，AVX2では256ビット幅に拡張された．

ただし，64ビット整数4組のベクトル演算が，float64が4つのベクトル演算と全く同じように用意されているわけではない．例えば，64ビット整数4組の加算と減算はベクトル演算が用意されているが，64ビット整数4組の乗算と除算のベクトル演算は用意されていない [8]．乗算や除算が用意されないのは，桁数の事情であると，容易に推測できる．

また，float64が4つの中から最大値を求める並列演算`_mm256_max_pd`はAVXで用意されていたが，64ビット整数4組の中から最大値を求めるベクトル演算はAVX2になっても用意されていない．用意されなかったのは，ハードウェア設計の事情によるものと推測される．64ビット整数4組の中から最大値を求めるベクトル演算は，64ビット整数4組のベクトル演算として用意されている比較命令，AND（論理積）命令，AND NOT（否定論理積）命令，OR（論理和）命令を組み合わせて，図1のように実現できる．

このプログラム例は，64ビット整数（long）の大きさ4の配列`a1,b1,c1`があって，`a1`と`b1`の添字が同じ要素の大小比較を行い，大きい方の値を`c1`の添字が同じ要素に代入するものである．

大きさ4の配列を1つの256ビットの64ビット整数4組ベクトルデータ`__m256i`にキャストするために`a2,b2,c2`を用いているが，説明を簡単にするため，説明ではキャスト前の`a1,b1,c1`を用いることにする．`temp`は，`__m256i`型の一時的な変数であるが，説明を簡単にするため，大きさ4の配列のように添字をつけて扱うことがある．

1. `_mm256_cmpgt_epi64`を用いて，`a1`と`b1`の対応する要素の大小比較を行う．`a1`の要素`a1[i]`が`b1`の対応する要素`b1[i]`より大きい場合，`temp[i]`のビットを全て1に設定する．さもなければ，`temp[i]`のビットを全て0に設定する．

```
// 32 バイトアドレス境界で
// 64 ビット整数の大きさ 4 の配列 a1,b1,c1 を確保
  __attribute__ ((aligned(32)))
    long a1[4], b1[4], c1[4];

// 配列 a1,b1 に値をセット
  ... 詳細省略...

// ベクトル演算用のデータ (256 ビット) への
// ポインタにキャスト
  __m256i *a2 = (__m256i *)a1,
          *b2 = (__m256i *)b1,
          *c2 = (__m256i *)c1, temp;

// a2 と b2(a1 と b1) の間の最大値を並列に求める
  temp = _mm256_cmpgt_epi64(*a2, *b2);
  *c2 = _mm256_and_si256(temp, *a2);
  temp = _mm256_andnot_si256(temp, *b2);
  *c2 = _mm256_or_si256(*c2,temp);

// ベクタ c2(c1) の 4 つの要素は,
// c1[0], c1[1], c1[2], c1[3] のように,
// 個別にアクセスできる
  ... 詳細省略...
```

図1：64ビット整数4組の最大値

2. 対応する要素のビット単位のAND（論理積）を行う `_mm256_and_si256` 命令と = を用いて，`a1[i]` のうち，`b1[i]` より大きい方については，`c1[i]` にコピーする．そうでない方については `c1[i]` のビットを全て0に設定する．
3. 対応する要素のビット単位のAND NOT（否定論理積）を行う `_mm256_andnot_si256` 命令と = を用いて，`b1[i]` のうち，`a1[i]` より大きい方あるいは `a1[i]` と等しい方については，`temp[i]` にコピーする．そうでない方については `temp[i]` のビットを全て0に設定する．
4. 対応する要素のビット単位のOR（論理和）を行う `_mm256_or_si256` 命令と = を用いて，上記2の `c1[i]` と上記3の `temp[i]` の値を合流させて，その結果を `c1` に代入する．

図1のように分岐命令を使わない方法は，命令のパイプラインを壊さないので，典型的なものであろう．目的を直接実装する単独のベクトル演算命令がないため，複数のベクトル演算命令の組み合わせの列で実現する例であるが，速度向上が得られるかどうか関心があるところである．

3 KCLにおけるAVX2の利用の試み

前年はAVXのfloat64が4つのベクトル演算を利用する試みを行ったので，今年はAVX2の64ビット整数4組のベクトル演算を利用する試みを行った．

3.1 64ビット整数のベクトル演算の利用

図2に，64ビット版のKCLのデータオブジェクトのフォーマットを示す．固定長整数（fixnum）は，即値データとして実現されている[9]．即値データとは，ポインタの中に直接オブジェクトを埋め込むことによって，表現されたデータである．

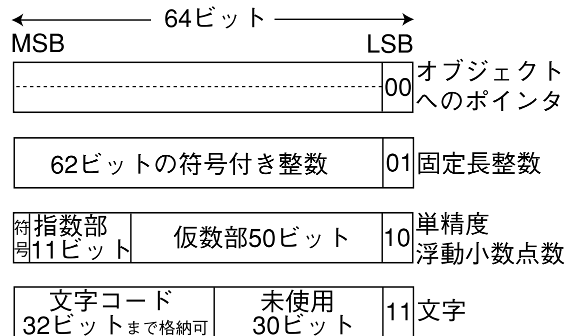

図2：64ビット環境下のオブジェクトの表現

KCLは，C言語とLisp言語で記述されている．固定長整数の場合，LispオブジェクトからC言語のデータへの変換は，右に2つ右へ算術シフトする．C言語のデータからLispオブジェクトへの変換は，左に2つ算術シフトし，下位2ビットを（固定長整数を示すタグとして)01にする．0か否かの判定，符号を調べる，大小比較といったことであれば，LispオブジェクトからC言語データへの変換は，不要である．

AVX2の64ビット整数4組のベクトルデータを構成する要素と，第4世代Coreプロセッサの64ビット整数のスカラーデータのフォーマットは同じである．2節で説明したように，64ビット整数のスカラーデータを連続したメモリ領域に4つ並べて，64ビット整数4組のベクトルデータにキャストすると，ベクトル演算命令から利用できる．したがって，KCLの固定長整数を，キャストして64ビット整数のスカラーデータとみなせば，その4つの集まりをAVX2のベクトルデータ（256ビットのベクトルデータ）とすることが可能である．

64ビット版のKCLにおいて，Lispの関数への引数（Lispオブジェクト）は，64ビットのポインタ配列で表現されたスタックに積まれる．この配列に格納されたままで，AVX2のベクトル演算の引数として渡すことができるならば，非常に効率が良い．

```
for(int i=1; i<max の引数の数/4; ++i){
  temp1=_mm256_cmpgt_epi64(vec[0],vec[i]);
  temp2=_mm256_and_si256(temp1,vec[0]);
  vec[0]=_mm256_andnot_si256(temp1,vec[i]);
  vec[0]=_mm256_or_si256(vec[0],temp2);
}

// vec[0] を，要素が 4 つの 64 ビットの
// 固定長整数の配列とみなして，
// 最大値の要素を求める.
  ...詳細省略...
```

図3：引数が固定長整数の場合のmaxのベクトル演算部分その1

3.2 引数が固定長整数の場合の組み込み関数maxの並列化

前年の62ビット単精度浮動小数点数の場合に引き続いて，今年は引数が固定長整数の場合の組み込み関数maxの並列化を行った．その結果，maxは，引数が全て固定長整数あるいは全て62ビット単精度浮動小数点数である場合は，AVXによるベクトル演算を利用し，いずれでもなければ，オリジナルのKCLのmaxのコードを実行するようになった．

ここでは，引数が全て固定長整数の場合について，説明する．maxのベクトル演算は，maxの引数4つ毎に（キャストにより）AVX2の256ビットのベクトルデータに対応させ，図3のようになった．C言語で記述してあり，vecはAVX2の256ビットのベクトルデータ（64ビット整数4組からなるベクトル）を要素とする配列である．temp1，temp2は，計算の途中結果を格納するための，AVX2の256ビットのベクトルデータ型の一時的な変数である．

maxの引数が32バイト境界から始まらない場合は，maxの先頭アドレスから最初に32バイト境界になるところから，vec[0]が始まるようにする補正は，前年と同様である．

maxの引数の数が4の倍数でない場合は，4の倍数に達するのに不足している個数だけ，KCLの固定長整数の最小値を，maxの引数の最後（あるいは引数の開始番地の調整のために移動した引数の後）に付け加えることも，前年と同じである．

図3に示したベクトル演算の列の繰り返しが終了すると，vec[0]に最大値の候補4つが得られる．vec[0]を要素が4つの固定長整数の配列とみなして，最大値を求める演算は，逐次的な比較によって行う．

4 評価実験

大きくわけて，2つの評価実験を行った．

一方は，引数が全て62ビット単精度浮動小数点数である場合，すなわち，前年行ったAVXを用いて並列化を行ったmaxの性能を，前年より新しい別の環境（CPU，コンパイラ，OS）の下で再評価することである．

他方は，引数が全て固定長整数である場合，すなわち，AVX2を用いて並列化を行ったmaxの性能の評価である．

今年の環境は，アップル社のiMac 27-inch, Late 2013（CPU：インテル Core i7-4771 3.50GHz，メモリ：16GB 1600MHz DDR3）である．OSはOS X 10.9.4で，コンパイラはApple LLVM version 5.1（clang-503.0.40）（based on LLVM 3.4svn）である．Core i7-4771はHaswellマイクロアーキテクチャであり，AVX2を備えていて，AVXの機能も使用できる．

4.1 引数が全て62ビット単精度浮動小数点数である場合（前年の再評価）

前年行った，引数が全て62ビット単精度浮動小数点数である場合のmaxの並列化のソースコードはそのままで，新しい環境，すなわち，APPLE LLVM version 5.1でコンパイルし直し，Haswellマイクロアーキテクチャのの CPUで実行した．評価用プログラムは，前年と同じで，(max 1.0 2.0....14999.0 15000.0)を10000回を行う．その実行時間を，表1に示す．実行時間は，10回計測して，その平均値をとったもので，これ以降の実験でも同様である．

表1：オリジナルのmaxと並列版のmaxの比較 Core i7-4771 CPU @ 3.5GHz単精度浮動小数点

maxのバージョン	実行時間	比
オリジナル（逐次版）	1.235	1.0
並列版	0.533	0.432
並列版（2分割）	0.550	0.445
並列版（4分割）	0.520	0.421

（単位は秒）

並列版（2分割）と並列版（4分割）は，前年と同じで，ベクトル演算を2回あるいは4回続けて行う代わりに，ループの回数を2分の1あるいは4分の1に減らすものである．ベクトル演算が2回あるいは4回確実に連続実行されるようになり，機械語レベルにおける命令の実行順序を予測しやすくして，パイプライン処理やアウトオブオーダ実行の効果を期待したものである．

参考までに，前年（2013年）の環境，すなわち，MacPortsのgccバージョン4.8でコンパイルし，Sandy

表2：オリジナルのmaxと並列版のmaxの比較
Core i7-3740 CPU @ 2.70GHz単精度
浮動小数点数　2013年に計測

maxのバージョン	実行時間	比
オリジナル（逐次版）	1.898	1.0
並列版	0.845	0.445
並列版（2分割）	0.795	0.419
並列版（4分割）	0.781	0.411

（単位は秒）

BridgeマイクロアーキテクチャのCPUで実行した場合の実行時間を表2示す．

新環境下で実行した場合も，AVXによる並列化の効果が表れており，並列版による速度向上は，新しい場合の方がわずかではあるが優れている．その一方で，新環境下では，並列版（2分割）が並列版より遅くなり，並列版（4分割）は並列版より速くなるが，2013年の場合ほどの速度向上は得られていない．

4.2　引数が全て固定長整数の場合その1

引数が全て固定長整数で，図3のようにAVX2の命令を利用して並列化された場合の性能を評価した．評価用プログラムは，`(max 1 2 .... 14999 15000)`を10000回を行う．62ビット単精度浮動小数点数の場合と異なるのは，引数が整数になっていることである．引数の値や，繰り返しの回数は変わっていない．実行時間を表3に示す．

表3：オリジナルのmaxと並列版のmaxの比較
Core i7-4771 CPU @ 3.5GHz固定長整数

maxのバージョン	実行時間	比
オリジナル（逐次版）	1.186	1.0
並列版	0.517	0.436
並列版（2分割）	0.515	0.434
並列版（4分割）	0.483	0.407

（単位は秒）

固定長整数の場合も，62ビット浮動小数点数の場合と同様に，AVX2のベクトル演算命令利用による並列化による，速度向上が得られている．並列版から並列版（2分割）への速度向上は誤差ともいえるほどわずかなのものであるが，並列版（2分割）から並列版（4分割）への速度向上は明確に表れている．

4.3　引数が全て固定長整数の場合その2

変則的ではあるが，（64ビット整数ベクトルではなく）float64ベクトルを対象にした（AVX2ではなく）AVXの命令`_mm256_blendv_pd`命令を用いて，比較後の最大値の代入を行うことも試してみた（図4参照）．代入の部分が3行から1行になり，機械語レベルでステップ数（命令数）が減ることに，関心を持ったからである．

```
for(int i=1; i<max の引数の数/4; ++i){
  temp=mm256_cmpgt_epi64(vec[i],vec[0]);
  vec[0]
    =(__m256i)
     _mm256_blendv_pd((__m256d)vec[0],
                      (__m256d)vec[i],
                      (__m256d)temp);
}

// vec[0] を，要素が 4 つの 64 ビットの
// 固定長整数の配列とみなして，
// 最大値の要素を求める.
   ... 詳細省略...
```

図4：引数が固定長整数の場合のmaxのベクトル演算部分その2

`_mm256_blendv_pd`は，3つ引数をとる．3つ目の引数のベクトルの各要素の最上位ビットに応じて，1番目と2番目の引数（float64が4つのベクトル）の条件付きマージを行う．3つ目の引数（float64が4つのベクトル）の各要素の最上位ビットが1の場合，2番目の引数の対応する要素を結果のベクトルにコピーする．最上位のビットが0の場合，1番目の引数の対応する要素を結果のベクトルにコピーする．

`_mm256_blendv_pd`は，本来，float64ベクトルを対象とした命令であるので，引数の64ビット整数ベクトルをflot64ベクトルにキャストして呼び出し，結果のfloat64ベクトルを64ビット整数ベクトルにキャストするという変則的な使い方をしている．

その1と同じ評価プログラムを実行した場合の実行時間は，並列版，並列版（2分割），並列版（4分割）のいずれとも，その1とほぼ同じであった．その1と全く同じではないが，その違いは誤差といえるほどわずかであり，ここに表にして提示するほどのものではない．平均をとるために実行時間を10回計測することを何回も行うと，その1と同じ実行時間がかなりの割合で出現する．`_mm256_and_si256`，`_mm256_andnot_si256`，`_mm256_or_si256`を用いた複数行からなるコードを，`_mm256_blendv_pd`を用いた一行のコードに置き換える効果はないと判断される．

5　考察

前年に提案したAVXを用いたmaxの並列化の手法は，実行環境や開発環境が変わっても，引数が62ビット単精度浮動小数点数のみの場合でも，引数が固定長

整数のみの場合でも，速度向上の効果があることがわかった．

2節では述べていない別の実験によると，KCLをコンパイルするコンパイラにより，引数が62ビット単精度浮動小数点数のみ場合の実行効率の向上の傾向に多少違いがあることがわかっている．コンパイラにApple LLVM version 5.1を用いた場合は，Sandy BridgeとHaswellの双方において，表3のように，並列版（2分割）では並列版より性能が悪くなり，並列版（4分割）では並列版と並列版（2分割）の双方より性能が良いといった，順当ではない傾向が得られている．コンパイラにMacPortsのgccバージョン4.8を用いた場合は，並列版，並列版（2分割），並列版（4分割）の順に，性能が良くなるという順当な傾向が得られている．ここに掲げた速度向上の傾向の違いは，コンパイラによる最適化の違いによるものだと，推測される．いずれにせよ，並列版は，コンパイラがどのバージョンであっても，オリジナル（逐次版）より，大幅に速いことに変わりはない．

引数が固定長整数のみの場合は，AVX2には64ビット整数ベクトルの最大値を求める直接のベクトル演算命令はないため，64ビット整数ベクトルの比較命令1つとビット演算命令3つを組み合わせたが，62ビット単精度浮動小数点数の場合と，遜色ない速度向上が得られた．スカラー命令と同様に，ベクトル演算においても，ビット演算命令の解釈実行に必要なサイクル数が，少ないことが推測される．

ビット演算命令3つの代わりに，（64ビット整数ベクトルではなく）float64ベクトルを対象としたマージ命令`_mm256_blendv_pd`を用いてみたが，速度向上（実行時間）に違いは得られなかった．`_mm256_blendv_pd`に対応するアセンブラ命令（機械語命令）は`VBLENDVPD`であり，4つのオペランドをとる命令である．このマージ命令は，（単純なビット演算命令に比べて）動作が複雑なこともあり，解釈実行に必要なサイクル数が多いことが推測され，コード量削減に見合った速度向上が得られなかったのであろう．速度向上が得られないのであれば，プログラムの保守性の観点から好ましくないので，キャストまでして，ベクトルのデータ型が異なるマージ命令を使う必要はない．

6　まとめ

前年提案したIntel社のベクトル演算命令AVXを用いて，Lispの組み込み関数の実行効率を向上させる試みを，今年はAVX2を搭載したHaswellマイクロアーキテクチャのプロセッサで行った．

62ビット単精度浮動小数点数を引数とする場合にAVXを使う手法を，固定長整数を引数とする場合にAVX2を使う場合に用いたら，同程度の速度向上が得られた．したがって，前年から引き続いて提案している，マイクロプロセッサに内蔵のベクトル演算命令を用いたLispの組み込み関数の実行効率を向上させる手法は，汎用性があるといえる．Lisp言語処理系における，ベクトル演算活用の範囲が広まった．

今後の課題として，ベクトル演算命令を活用できるLispの組み込み関数を多く見つけることや，ベクトルレジスタが512ビットに拡張されたAVX512［10］への対応があげらえる．

参考文献

［1］Intel Corporation: *Intel Advanced Vector Extensions 2 and Bit Manipulation New Instructions*, ARCS005, Intel Developer Forum 2012（2012）.

［2］Steele, G. L. Jr.: *Common Lisp the language*, Digital Press（1984）.

［3］Yuasa, T. and Hagiya, M.: *Kyoto Common Lisp Report*, Teikoku Insatsu Publishing（1985）.

［4］Yuasa, T. : Design and Implementation of Kyoto Common Lisp, *Journal of Information Processing*, Vol. 13, No. 3, pp. 284–295（1990）.

［5］安本太一：Common Lisp言語処理系の64ビット化，愛知教育大学研究報告自然科学編，五十三輯，pp. 27–32（2004）.

［6］安本太一：Common Lisp言語処理系による64ビット環境の評価，愛知教育大学研究報告自然科学編，五十五輯，pp. 9–13（2006）.

［7］安本太一：Intel AVXによるCommon Lisp言語処理系の最適化の一考察， 愛知教育大学研究報告 自然科学編，六十三輯，pp. 25–30（2014）.

［8］Intel Corporation: *Intel C++ Compiler 12.1 User and Reference Guides*, Document number: 323271-012US（2011）.

［9］湯淺太一，安本太一：KCLにおける即値データの実装とその評価，電子情報通信学会春季全国大会講論集，D-357（1989）.

［10］Intel Corporation : *Intel Architecture Instruction Set Extensions Programming Reference*, Document number: 319433-017（2013）.

（2014年9月22日受理）