Bulletin of Aichi Univ. of Education, 63（Natural Sciences）, pp. 25 − 30, March, 2014

Intel AVX 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 AVX

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2世代Coreプロセッサ（Sandy Bridgeアーキテクチャ）から実装されているベクトル演算命令AVX（Advanced Vector eXtensions）[1]を用いて，Common Lisp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．

筆者は，過去に，アップル社とIBM社で共同開発されたプロセッサPowerPC970のベクトル演算器AltiVec［2］を用いたCommon Lisp［3］の組み込み関数の最適化の試みを行った．AltiVecは128ビットのレジスタを32本搭載し，4組の32ビットの整数の演算を同時にできるなどの特徴を持つ．対象としたCommon Lisp言語処理系は，32ビット版のKCL（Kyoto Common Lisp）であった［4, 5］．この試みでは，組み込み関数maxの実行効率が約2倍（実行時間が約半分）になる結果が得られた［6］．

今回は，インテル社のCore i7プロセッサ（Sandy BridgeおよびIvy Bridgeアーキテクチャ）のAVXで同様の試みを行う．対象とするCommon Lisp言語処理系は，64ビット版のKCLである[7, 8]．過去の研究とは，プログラム実行形式（32ビット，64ビット）とCPUアーキテクチャ（AltiVecを備えたPowerPC970, AVXを備えた第2世代第3世代Intel Core）が異なる条件で，過去に提案した手法に汎用性があるかどうかを検証する．

提案しているベクトル演算命令利用の手法，すなわち過去の試みと今回の試みは，過去に行ったKCLにおける即値データ（immediate data）の実装［9］に大きく依存しているので，即値データについても改めて言及する．

2 AVXの概要

ベクトル演算用として，16個の256ビットのレジスタが用意されている．そして，4組の倍精度浮動小数点数（64ビット）の並列演算を行う命令，8組の単精度浮動小数点数（32ビット）の並列演算を行う命令が用意されている．Sandy BridgeやIvy Bridgeでは，256ビット幅のベクトル演算器はなく，128ビット幅のベクトル演算器が2つあり，倍精度浮動小数点数のベクトル演算はこれらの演算器を同時に利用して行われる．AVXでは，4組の64ビット整数の並列演算や，8組の32ビット整数の並列演算といった整数演算はサポートされていない．

AVXにおけるベクトル演算の概要を，C言語からの利用を想定して述べる．AVXの機能を利用する際は，引数の256ビットのベクトルデータを32バイト境界のアドレスから配置してから，ベクトル演算命令を呼び出すことになっている．32バイト境界は，CPUのハードウェアの都合によるものであろう．この配置は，例えば，32バイト境界から始まる4つの64ビットデータが連続したものを確保して，1つの256ビットのベクトルデータにキャストすることによってできる．ベクトル演算の結果は，256ビットのベクトルデータとして得られる．ベクトルデータの要素は，キャスト前の配列の要素として，個別にアクセスできる．

gccにおいて，4組の倍精度浮動小数点数の最大値を並列に求める例を，図1に示す．この例は，倍精度浮動小数点数（double）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．`__attribute__((aligned(32)))`はデータを32バイト境界に配置する修飾子，`__m256d`はfloat64ベクトルに，`_mm256_max_pd`はAVX命令VMAXPDに対応する［10］．

gccでは，float64ベクトルへのキャストを用いることによって，ベクトルレジスタへのロードやストアを明示的に記述することなく，AVXを用いることができる．

```
// 32 バイトアドレス境界で 大きさ 4 の配列を確保
  __attribute__ ((aligned(32)))
    double a1[4], b1[4], c1[4];

// 配列 a,b 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 */
  __m256d *a2 = (__m256d *)a1,
          *b2 = (__m256d *)b1,
          *c2 = (__m256d *)c1;

// a と b の間の最大値を並列に求める
  c2[0] = _mm256_max_pd(a2[0], b2[0]);

// ベクタ c2[0] 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1：4組の倍精度浮動小数点数の最大値

3　64ビット版のKCLの即値データ

図2に示すフォーマットのように，64ビット版のKCLでは，Lispのデータオブジェクトのうち，固定長整数（fixnum）と単精度浮動小数点数（short-float）と文字（character）は即値データとして実現されている．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．

一般には，単精度浮動小数点数は32ビット表現を指すことが多いが，64ビット版のKCLの即値データでは，単精度浮動小数点数（short-float）は62ビットとなっている．即値データでない倍精度浮動小数点数（long-float）は64ビットとなっている．単精度と倍精度の間には仮数部2ビットの差しかないが，プロセッサの64ビットの倍精度浮動小数点数の仮数部の下位2ビットを削ってKCLの即値データとしたからである．

図2：64ビット環境下のオブジェクトの表現

常識的な考え方からは，プロセッサの32ビットの単精度浮動小数点数をKCLの即値データとして表現する選択肢もあるが，ポインタの幅が64ビットであるため30ビットほど無駄になるので得策ではない．即値データのメモリ効率と実行性能の特徴を，精度が高いプロセッサの64ビットの倍精度浮動小数点数の演算の方で生かした方が良いと考え，図2の表現形式に落ち着いた．

4　KCLにおけるAVXの利用の試み

4.1　64ビットの倍精度浮動小数点数のベクトル演算の利用

64ビット版のKCLにおけるAVXの利用の試みは，KCLの62ビットの単精度浮動小数点数について行うことにした．理由は次の通りである．

1. AVXのベクトル演算の引数は，要素数が4の64ビットの配列に格納して，256ビットのベクトルデータにキャストすることが基本となっているので，64ビットで収まるデータ型である必要がある．このようなデータ型は，即値データで実装されているデータ型である．
2. 64ビットの整数のベクトル演算を，AVXはサポートしていない．文字型も文字コードを演算の対象とするならば，整数演算となるので，整数演算をサポートしていないAVXは使えない．即値データとして，実装されたデータ型で残っているのは，KCLの62ビットの単精度浮動小数点数である．
3. 64ビット版のKCLにおいて，Lispの関数への引数（Lispのデータオブジェクト）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，AVX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．
4. 即値データとして表現されたKCLの単精度浮動小数点数の下位2ビットのタグは10であるが，これを00にマスクしないで，そのままAVXの演算に持ち込んでしまうという手もある．大小関係の演算ならば，下位2ビットのタグはそのままで，全く問題がない．

4.2　組み込み関数maxの並列化

AVXによるベクトル演算の機能を有効に利用できるのは，引数が可変長で，同じデータ型の引数を多数とる可能性があるものであろう．そのような組み込み関数の中から，与えられた引数の最大値を求めるmaxを，AVXを利用して高速化することを試みることにした．

先述のように，maxの引数は64ビットのポインタを

```
for(int i=1; i < max の引数の数/4; ++i)
  vec[0] = _mm256_max_pd(vec[0], vec[i]);

// vec[0] を，要素が 4 つの 64 ビットの
// 浮動小数点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：maxのベクトル演算部分

要素とする配列に積まれる．引数が全て62ビットの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，さもなければ，オリジナルのKCLのmaxのコードを実行することにした．

AVXのベクトル演算を用いたmaxの実装は，maxの引数4つ毎に（キャストにより）AVX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AVXの256ビットのベクトルデータの配列である．

maxの引数が32バイト境界から始まらない場合は，先述したAVXの命令の制限により，vec［0］が32バイト境界から始まるようにする必要がある．図4に示すように，maxの引数のうち，アドレスの低位から最初に32バイト境界になるところから，vec［0］を始める．vec［0］より前になったmaxの引数は，maxの最後の引数の後ろにコピーする．

さらに，maxの引数の数が4の倍数でない場合は，図5に示すように，4の倍数に達するのに不足している個数だけ，KCLの62ビットの単精度浮動小数点数の最小値を，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コピーされた引数の後）に付け加える．最小値を付け加えても，maxの結果に影響を与えない．図5に示す例では，4の倍数に，2つ引数が足りず，2個のKCLの62ビットの単精度浮動小数点数の最小値を付け加えている．引数の開始番地と数の調整を経て，_mm256_max_pdによるベクトル演算を行うことができる．

_mm256_max_pdによるベクトル演算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つの浮動小数点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．並列的な比較で行わなくても，maxの引数の数が非常に多い場合には，並列化したmaxの性能には影響を与えない．筆者が調べた範囲では，AVXでは，ベクトルを構成する要素の最大値を求める命令が用意されていないようである．

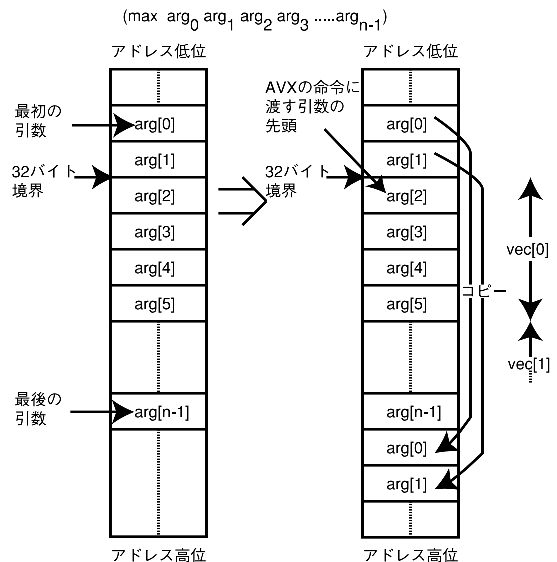

図4：引数の開始番地の32バイト境界への調整

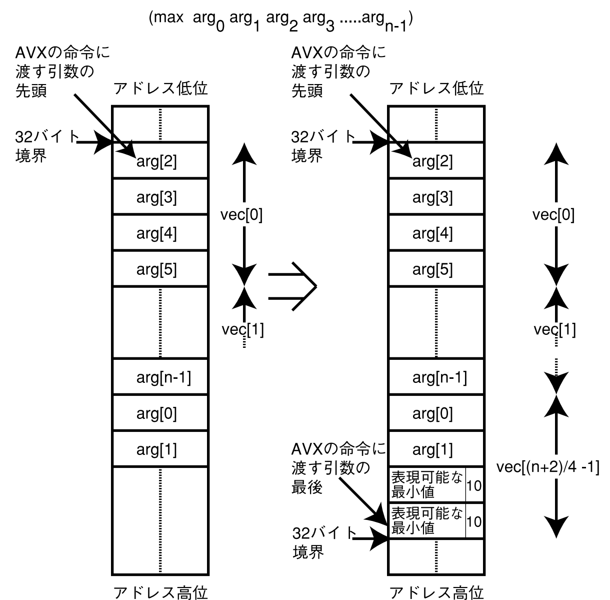

図5：引数の数の調整（図4から続く）

5 評価実験

提案した方法でAVXを用いて並列化を行ったmaxの性能を評価するために，評価用プログラムを実行して，並列版maxとオリジナルのmax（逐次版max）でその実行時間を比較した．

評価実験に用いたコンピュータはアップル社のMacのデスクトップあるいはノート，OSはOS X Mountain Lionバージョン10.8.4である．コンパイラは，MacPortsのgccバージョン4.8である．この評価実験の時点では，アップル社が提供する開発環境Xcodeのgccでは，AVXのC言語の組込み関数をサポートしていなかったからである．OS，コンパイラの生成するコード，ベースとしたKCLともに，64ビット対応である．

評価用プログラムは，過去に行った研究とほぼ同じもので，(max 1.0 2.0 .... 14999.0 15000.0) を10000回を行う．違いは引数の型だけである．評価用プログラムの実行は，インタプリタではなく，コンパイルしてから行った．10000回の繰り返しは特殊形式dotimesで行うが，KCLのコンパイラにより，C言語のifとgotoを使った繰り返しに展開される．1.0から15000.0の浮動小数点数は，即値データとして表現される．リーダーによるこれらの浮動小数点数の文字列表現から即値データへの変換は，コンパイル時に完了している．maxの呼出しのたびに，即値データに変換された浮動小数点数が，maxの引数として（C言語の64ビットのポインタ配列で表現されたスタックに）セットされる．したがって，即値データに変換された浮動小数点数は，10000回の繰り返しにおいて，何度も使われ，同じ値の浮動小数点数の即値データが新たに生成されることはない．

評価用プログラムの実行時間を次に示す．表1はSandy Bridge（L3キャッシュ6MB）のMac，表2はIvy Bridge（L3キャッシュ6MB）のMacである．実行時間は，10回計測して，その平均値をとったものである．いずれのCPUにおいても，並列版においては，実行時間が逐次版に比べて約44.6%に短縮されており，2倍強の速度向上である．

並列版（2分割）と並列版（4分割）は，ループの回数を2分の1あるいは4分の1に減らして，AVXのベクトル演算を2回あるいは4回続けて行うものである．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，パイプラインやアウトオブオーダなどの効果を期待したものである．AVXには，倍精度浮動小数点数ベクタのための演算器は実質一つしかないが，ベクトルレジスタへのベクトルデータのロードなどが効率的に行われることを期待した．並列版（2分割）の概要を図6に示す．この概要には示していないが，maxの引数が8の倍数でない場合は，引数が8の倍数になるように，必要な個数だけKCLの単精度浮動小数点数の最小値を補う必要がある．並列版（4分割）の場合も，並列版（2分割）と同様である．

```
for(int i=1; i < max の引数の数/4/2; ++i){
  vec[0] = _mm256_max_pd(vec[0], vec[i]);
  vec[n/2] = _mm256_max_pd(vec[n/2],
                           vec[n/2+i]);
}
vec[0] = _mm256_max_pd(vec[0], vec[n/2]);

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

図6：並列版（2分割）

並列版（2分割）の実行時間はオリジナルの約42.1%，並列版（4分割）の実行時間はオリジナルの約41.2%である．これらは，並列版より実行時間が短くなっている．並列版（2分割）から並列版（4分割）は，並列版から並列版（2分割）より，実行時間の短縮の度合いが小さい．

表1：オリジナルのmaxと並列版のmaxの比較
Core i7-2635QM CPU @ 2.00GHz

maxのバージョン	実行時間	比
オリジナル（逐次版）	2.433	1.0
並列版	1.088	0.447
並列版（2分割）	1.027	0.422
並列版（4分割）	1.005	0.413

（単位は秒）

表2：オリジナルのmaxと並列版のmaxの比較
Core i7-3740QM CPU @ 2.70GHz

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）

6 考察

AVXを用いたmaxの並列版では，先述のように，2倍強の速度向上（実行時間が44.7%に短縮）が得られた．過去に，PowerPC970のAltiVecでmaxの並列化を同様に行ったが，このときは4つの32ビットの整数を要素とする128ビットベクタのベクトル演算を行う機能を用いて，引数が全て固定長整数（fixnum）のときのmaxの高速化を試みた．表3に示すように50%程度の時間短縮（2倍程度の速度向上）が得られた．

AVXの方が5%ほど実行時間の短縮の度合いが高いことがわかる．5%という差であるが，扱うベクトルデータのビット数の違い（AVXは256ビット，AltiVec

表3：オリジナルのmaxと並列版のmaxの比較
PowerPC970 2GHz（2006年に計測）

maxのバージョン	実行時間	比
オリジナル（逐次版）	5.333	1.0
並列版	2.66	0.499
並列版（2分割）	2.48	0.465
並列版（4分割）	2.42	0.454

（単位は秒）

は128ビット）を考えると，AltiVecの場合はAVXのそれに比べてベクトル演算器の恩恵を受けておらず，逆の言い方をすればAVXの場合はAltiVecのそれに比べてベクトル演算器の恩恵を受けていると考えることもできる．このように考えられる理由として，メモリ転送速度の差が考えられる．

PowerPC970とCore i7-2635QMにおいて，メモリ転送速度を計測した結果を図7に示す．メモリ転送速度の計測に用いたC言語プログラムは，代入文の繰り返しを行うものである．32ビットはint（32ビットの整数）の代入文の繰り返しを行う32ビットの実行形式，64ビットはlong int（64ビットの整数）の代入文の繰り返しを行う64ビットの実行形式を用いたことを示す．PowerPC970についてはgcc4.01を用いて2006年に計測したものであり，Core i7-2635QMについてはgcc4.2.1を用いて今回計測したものである．メモリ転送速度計測プログラムをgccでコンパイルするときに用いた最適化オプションは，-Oである．メモリサイズの増加とともに，急激に転送速度が落ちるところは，キャッシュのサイズに関係していることがうかがえる．Core i7-2635QMのメモリサイズの始めの方にデータがないのは，メモリ転送時間が非常に短かったため，(転送速度計算の分母が小さくなり）正確なメモリ転送速度が計測できなかったと判断して，データを除外したからである．

Core i7-2635QMの方が，PowerPC970より，圧倒的にメモリ転送速度が速い．メモリ転送速度の差と，maxの速度向上の差およびベクトルレジスタのビット数の差を対比すると，ベクトル演算器の並列計算の恩恵を受けるためには，メモリ転送速度が速いことが極めて重要であることがわかる．

メモリ転送速度に加えて，Lispのデータをベクトルレジスタに値をセットするまでの過程が簡素であることが大切であることも，今回のAVXおよび過去のAltiVecの評価実験の比較から推測される．筆者が提案したAVXとAltiVecの利用では，即値データとして表現されたLispデータを利用したことが，maxにおける速度向上に大きく寄与していると考えられる．

筆者が過去の研究で提案した即値データの表現形式は，プロセッサのデータ表現との変換が極めて単純なことが特徴である．即値データから，プロセッサのデータ表現に変換せずに，そのまま演算を行って，実質差し支えないケースもある．即値データとして表現されていないLispのデータを，ベクトルレジスタにセットしようとした場合は，その過程がいくつかのステップに及び，そのステップのコストが無視できずベクトル演算による恩恵が得られないことは容易に推測できる．Lispのデータをベクトルレジスタにセットするときの転送速度を，“実質的なメモリ転送速度”というならば，即値データとして表現されていれば“実質的なメモリ転送速度”は速く，即値データとして表現されていなければ“実質的なメモリ転送速度”は遅いといえるだろう．

並列版（2分割）や並列版（4分割）では，わずかではあるが，AltiVecの場合と同様に，並列版に比べて，一定の速度向上がみられた．プロセッサ内部で（ベクトル演算命令を含んだ）機械語命令が並列に実行される過程は承知していないが，gccが生成した並列版（2分割）や並列版（4分割）のmaxの機械語（アセンブリ言語）のコードを筆者が観察した限りでは，ベクトル演算を2回ないし4回続けて行っている部分の機械語をプロセッサ内部で並列的に実行できる可能性があることを確認している．

7　まとめ

過去に提案した，CPUに内蔵のベクトル演算機能を用いてLispの組み込み関数の実行効率を向上させる試みを，今回はアーキテクチャが異なる別のプロセッサで行った．過去にPowerPC970に試みた手法は，Intel Core i7（Sandy BridgeとIvy Bridge）においても有効で，実行効率が向上した．過去の試みと今回の試みの実行効率の向上の度合いの違いから，ベクトル演算機能の恩恵を受けるためには，メモリ転送速度が速いことが重要であることがわかった．

Lispのデータをベクトルレジスタに転送する際の十分な“実質的なメモリ転送速度”の確保には，筆者らが過去に提案した即値データの実装が寄与している．即値データの実装を提案した時には，ベクトル演算への応用は思いもよらなかったが，プロセッサのデータ表現と即値データのデータ表現の間の変換のコストが極めて低いことが重要であることがわかった．

2013年6月に発売された第4世代Intel Coreプロセッサ（Haswellアーキテクチャ）では，AVXの後継であるAVX2が搭載されている．AVX2では，ベクトル演

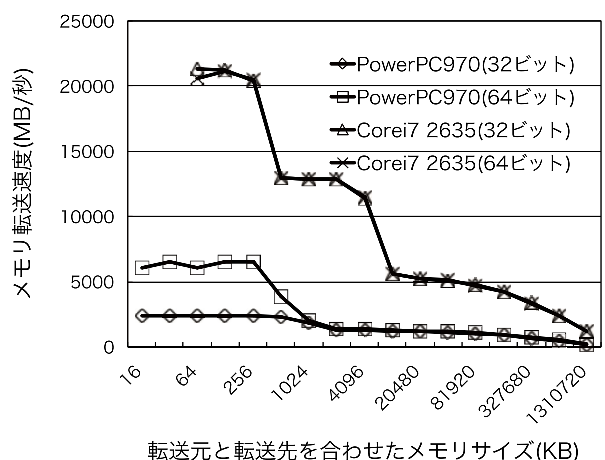

図7：PowerPC970とCore i7-2635QMのメモリ転送速度

算器が256ビットの実装になり，4つの64ビット倍精度浮動小数点数の演算を真に一度に行うことができるようになったほか，64ビットの整数のベクトル演算もサポートされた．

今後の課題として，HaswellアーキテクチャのIntel Coreプロセッサに，提案してきたベクトル演算のLispの組み込み関数への利用を試みて，その評価を行うことがあげられる．

参考文献

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

[2] Apple Computer, inc.: *Velocity Engine Introduction and Overview*, http://developer.apple.com/hardwaredrivers/ve/.

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

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

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

[6] 安本太一：マルチメディア命令による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，五十六輯，pp. 13–17（2007）.

[7] 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）.

[8] 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）.

[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 C++ Compiler12.1 User and Reference Guides*, Document number: 323271-012US（2011）.

（2013年9月30日受理）