Bulletin of Aichi Univ. of Education, 72（Natural Science）, pp. 22 − 25, March, 2023

Raspberry Pi 4のCPUのSIMD並列演算機能の一考察

安本 太一

情報教育講座

A Study of SIMD Instructions on Raspberry Pi 4

Taichi YASUMOTO

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

1　はじめに

筆者は，過去に，Raspberry Pi 2とRaspberry Pi 3に使用されているARM系CPUのSIMD並列演算機能の評価をLisp言語処理系の関数の並列化によって行った［1］。SIMD命令使用による速度向上の評価に加えて，SIMD命令などのパイプライン処理などによる速度向上の効果も評価した。この時，2つの世代のCPUを用いたが，Raspberry Pi 2のCPUは32bitのみに対応，Raspberry Pi 3のCPUは32bitと64bitの双方に対応していた。Raspberry Piの公式OSは32bit版しかなかったので，評価はCPUの32bitモードのみで行った。64bitモードにおける評価は今後の課題となっていた。

その後年月が経過し，Raspberry Pi 3の後継であるRaspberry Pi 4（CPUは64bitにも対応）の発表とRaspberry Pi 3とRaspberry Pi 4に対応した公式OSの64bit版の正式発表があったので，今回，64bitモードにおける評価を過去の研究と同様の方法で行った。その結果，過去の研究とは異なる傾向の結果が得られたので本稿で報告する。以下，記述を簡潔にするために，Raspberry Pi 2，Raspberry Pi 3，Raspberry Pi 4について，それぞれPi 2，Pi 3，Pi 4という略語を用いる。

2　過去の研究

筆者は，2017年，Raspberry Piの公式OSが32bit版のみの時に，Pi 2（CPUはARM Cortex-A7）とPi 3（CPUはARM Cortex-A53）を対象に，ARMアーキテクチャのCPUのSIMD並列演算命令NEONを用いて，Common Lisp言語処理系であるKCL（Kyoto Common Lisp）［2］の組み込み関数**max**の並列化の試みを行なっている［1］。2世代のRaspberry Piを用いて，筆者が提案したSIMD命令を用いた組み込み関数**max**の速度向上並びにパイプライン処理などによるさらなる速度向上を32bitモードのOSの下で比較した。それ以前の2015年にも，64bit版の OS X（現在のmacOS）の下で，Intel Core i7-4771のSIMD並列演算命令AVX2を用いて，64bit版KCLを対象に同様なことを行っている［3］。

詳細は文献［1, 3］に譲るが，ARM 32bit，Intel 64bitいずれの場合も，SIMD並列演算命令を用いた単なる並列版で明らかな速度向上が得られていた。パイプライン処理などの効果を狙った並列版（2分割，4分割）では，いずれの場合もわずかではあるがさらなる速度向上が得られていた。単なる並列版，並列版（2分割，4分割）のプログラムは本稿の主題であるARM 64bitの場合と同様であり後に示すので，ここでは省略する。

3　本研究の背景

2016年2月に発表されたPi 3，2019年6月に発表されたPi 4ともに，64bit CPU（前者はARM Cortex-A53，後者はARM Cortex-A72）を搭載しているが，公式OSは32bit版のみのままであり，両CPUとも32bitモードで使用されていた。

2022年3月に，Pi 3，Pi 4など向けに，公式OSであるRaspberry Pi OSの64bit版が正式に公開されたの機に，KCLの64bit版［4］を64bit版公式OSで動くようにして，組み込み関数**max**の並列化の試みを行うことにした。すなわち，2017年に32bit版公式OSの下でPi 2とPi 3で行ったのと同様のことを，今回は64bit版公式OSの下でPi 3とPi 4の下で行うことにした。しかしながら，KCLの64bit化は完了しているが［4］，64bit版公式OSの下の新しいバージョンのgccでは，古いC言語仕様で記述されているKCLのソースプログラム全体を完全にコンパイルすることはできなかった。2016年ぐらいまでのgccではコンパイルできたのだが［5］，最近のgccは今日のC言語仕様に準拠していることをソースプログラムに要求するようになったからである。KCLのソースプログラム全体を今日のC言語仕様に厳密にあうように修正するには相当の時間を要するので，本研究の評価実験では

KCLの組み込み関数**max**の本体のC言語ソースの部分（今日のC言語仕様に適合）だけ取り出してコンパイルして実行することにした。本体とは最大値を求める処理を実質的に行っている部分である。このようにしても，本研究の目的は損なわれない。

4　Cortex-A53 や Cortex-A72 の NEON と 64bit 版 KCL

Cortex-A53やCortex-A72のアーキテクチャであるARMv8のSIMD演算は，128bitのSIMD命令セットを基本としている。例えば，32bitの整数4つからなるベクタ2つを引数としてとり，対応する要素の大きい方の要素からなる1つのベクタを返すC言語の関数**int32x4_t vmaxq_s32(int32x4_t a, int32x4_t b)**が用意されており，並列度は4である。**int32x4_t**は符号付き32ビット整数4つのベクタである。過去の32bit版のKCLにおける研究では，最大値を求めるCommon Lispの組み込み関数**max**をこの**vmaxq_s32**を用いて並列化して，速度向上を評価した。過去の研究では，**vmaxq_s32**だけを使う版（max版）のほかに，**vcgtq_s32**, **vandq_u32**, **vbicq_u32**, **vorrq_u32**を組み合わせて**vmaxq_s32**と同様の動作を実現した版（cgt版）も実装して，速度向上を評価した。最初の命令でベクトルの要素の大小比較**a > b**を行い，その結果と残りの3つの命令（ビット毎の論理積**a AND b**，論理積の変形**a AND NOT b**，論理和**a OR b**）を用いて，**vmaxq_s32**と同じ結果を得るのである。

64bit版のKCL［4］では，即値データの固定長整数（fixnum）が64bitのポインタを使って実装されていることから，並列演算のためのベクタの要素は64bitである必要がある。NEONを用いて，組み込み関数**max**を並列化するためには，**int64x4_t vmaxq_s64(int64x4_t a, int64x4_t b)**というような命令が使用できれば良いが，そのような命令は存在しない［6］。存在するのは，64ビット符号付き整数2つのベクタを引数とする関数で，大小比較**uint64x2_t vcgtq_s64(int64x2_t a, int64x2_t b)**，論理積**uint64x2_t vandq_u64(uint64x2_t a, uint64x2_t b)**，論理積の変形**uint64x2_t vbicq_u64(uint64x2_t a, uint64x2_t b)**，論理和**uint64x2_t vorrq_u64(uint64x2_t a, uint64x2_t b)**である。128bitのSIMD命令セットを基本としているいるからであろう。理由は不明であるが，**int64x2_t vmaxq_s64(int64x2_t a, int64x2_t b)**というような命令は存在しない。算術比較とビット演算の命令を組み合わせて**vmaxq_s64**のような機能を実現することはできるが，複数命令になってしまう，引数のベクタの長さが2なので並列度は2となり32bit版と比べて半減，引数のセットアップなどSIMD命令を使用するためのオーバーヘッドがあることを考えると，十分な速度向上が得られないことが想定される。それでも，今日のCPUはパイプライン処理などの機能を備えているので，プログラムの実装によっては，速度向上が得られる場合もあると考え，筆者の過去の研究と同様，単なる並列版のプログラムを変形した並列版（2分割，3分割，…）の評価実験も行った。

5　評価実験

NEONのSIMD命令の使用により，組み込み関数**max**の本体の実行時間がどのように変化するか，Pi 3（Cortex- A53 @ 1.2GHz）とPi 4（Cortex-72 @ 1.5GHz）で実験を行った。過去の研究［1, 3］と同様，**max**の引数は**(max 1 2 .... 14999 15000)**を想定し，**max**の本体を100000回実行したときの実行時間を計測した。その結果を表1と表2に示す。並列版（単なる並列版），並列版（2分割），並列版（4分割）の**max**の本体のプログラムの概要をそれぞれ図1，図2，図3に示す。

今回用いた64bit版Raspberry Pi OS（公式OS）のカーネルバージョンは5.15，gccのバージョンは10.2.1である。C言語ソースのコンパイル時に用いるgccの最適化オプションは**-O3**である。C言語ソースをコンパイルして生成される実行形式ファイル及び64bit版Raspberry Pi OSのバイナリは，Pi 3とPi 4で全く同じものを用いた。

並列版（3分割），並列版（5分割），並列版（6分割）は，過去の研究では行っていない。評価実験の過程で，表2にみられるように単なる並列版では十分な速度向上が得られなかったが，並列版（2分割）になって十分な速度向上が得られたことから，分割の効果をさらに確認するために3，5，6分割を追加した。3，5，6分割のプログラムの概要は省略するので2,4分割のプログラム（図2，3）から類推されたい。

表1：オリジナルの max と並列版の max の比較
ARM Cortex-A53 @ 1.2GHz cgt 版

max のバージョン	実行時間	比 1	比 2
オリジナル (逐次版)	4.572	1.0	
並列版	5.192	1.136	1.0
並列版 (2 分割)	5.026	1.099	0.968
並列版 (3 分割)	5.155	1.128	0.993
並列版 (4 分割)	5.213	1.140	1.004
並列版 (5 分割)	5.700	1.247	1.098
並列版 (6 分割)	7.401	1.619	1.425

(実行時間の単位は秒)

表 2：オリジナルの max と並列版の max の比較
ARM Cortex-A72 @ 1.5GHz cgt 版

max のバージョン	実行時間	比 1	比 2
オリジナル (逐次版)	2.618	1.0	
並列版	2.512	0.960	1.0
並列版 (2 分割)	1.396	0.533	0.556
並列版 (3 分割)	1.321	0.505	0.526
並列版 (4 分割)	1.400	0.535	0.557
並列版 (5 分割)	1.266	0.484	0.504
並列版 (6 分割)	1.370	0.523	0.545

(実行時間の単位は秒)

```
uint64x2_t  t1, t2;
int64x2_t *base = (int64x2_t *)arg;
//arg は max の引数が積まれたスタックの先頭アドレス
//max の引数は 64 ビットの整数

for(int i=1; i<max の引数の数 /2; ++i){
  t1 = vcgtq_s64(*base, *(base + i));
  t2 = vandq_u64((uint64x2_t)*base, t1);
  t1 = vbicq_u64((uint64x2_t)*(base + i), t1);
  *base = (int64x2_t)vorrq_u64(t1, t2);
}

//base が指しているベクタの要素から大きい方を取り出して
//max の値とする ( 詳細は省略 )
```

図 1：max の並列版（単なる並列版）の本体の概要

```
uint64x2_t  t11, t12, t21, t22;
int64x2_t *base1, *base2;
base1 = (int64x2_t *)arg;
base2 = (int64x2_t *)arg + max の引数の数 /2/2;
//arg は max の引数が積まれたスタックの先頭アドレス
//max の引数は 64 ビットの整数

for (i = 1; i < max の引数の数 /2/2; ++i){
  t11 = vcgtq_s64(*base1, *(base1 + i)); // 前半
  t12 = vandq_u64((uint64x2_t)*base1, t11);
  t11 = vbicq_u64((uint64x2_t)*(base1 + i), t11);
  *base1 = (int64x2_t)vorrq_u64(t11, t12);

  t21 = vcgtq_s64(*base2, *(base2 + i)); // 後半
  t22 = vandq_u64((uint64x2_t)*base2, t21);
  t21 = vbicq_u64((uint64x2_t)*(base2 + i), t21);
  *base2 = (int64x2_t)vorrq_u64(t21, t22);
}

//base1, base2 が指しているベクタの要素から最大値をみつけ
//max の値とする ( 詳細は省略 )
```

図 2：max の並列版（2 分割）の本体の概要

```
uint64x2_t t11, t12, t21, t22, t31, t32, t41, t42;
int64x2_t *base1, *base2, *base3, *base4;
base1 = (int64x2_t *) arg;
base2 = (int64x2_t *) arg + max の引数の数 /2/4*1;
base3 = (int64x2_t *) arg + max の引数の数 /2/4*2;
base4 = (int64x2_t *) arg + max の引数の数 /2/4*3;
//arg は max の引数が積まれたスタックの先頭アドレス
//max の引数は 64 ビットの整数

for (i = 1; i < size/ max の引数の数 /4/2; ++i){
  t11 = vcgtq_s64(*base1, *(base1 + i)); //4 分の 1
  t12 = vandq_u64((uint64x2_t)*base1, t11);
  t11 = vbicq_u64((uint64x2_t)*(base1 + i), t11);
  *base1 = (int64x2_t)vorrq_u64(t11, t12);

  t21 = vcgtq_s64(*base2, *(base2 + i)); //4 分の 2
  t22 = vandq_u64((uint64x2_t)*base2, t21);
  t21 = vbicq_u64((uint64x2_t)*(base2 + i), t21);
  *base2 = (int64x2_t)vorrq_u64(t21, t22);

  t31 = vcgtq_s64(*base3, *(base3 + i)); //4 分の 3
  t32 = vandq_u64((uint64x2_t)*base3, t31);
  t31 = vbicq_u64((uint64x2_t)*(base3 + i), t31);
  *base3 = (int64x2_t)vorrq_u64(t31, t32);

  t41 = vcgtq_s64(*base4, *(base4 + i)); //4 分の 4
  t42 = vandq_u64((uint64x2_t)*base4, t41);
  t41 = vbicq_u64((uint64x2_t)*(base4 + i), t41);
  *base4 = (int64x2_t)vorrq_u64(t41, t42);
}

//base1～base4 が指しているベクタの要素から最大値をみつけ
//max の値とする ( 詳細は省略 )
```

図 3：max の並列版（4 分割）の本体の概要

逐次版において Pi 4（Cortex-A53）の実行時間は Pi 3（Cortex-A72）の 0.573 倍となっている。商業誌の解説記事［7］では Pi 4 の CPU のコア性能は Pi 3（正確には 3B+）の約 2 倍であるから，本稿と商業誌では評価プログラムが全く異なるとはいえ，本評価実験の逐次版において Pi 4 は Pi 3 に比べて実行性能が大幅に高い結果が出ていることは誤りではないと考えられる。

6 考察

Pi 3（Cortex-A53）では SIMD による速度向上は全く得られない。Pi 4（Cortex-A72）では，単なる並列版では十分な速度向上が得られないが，並列版を変形したもの（2 分割以上）で明らかな速度向上が得られている。このような結果になったのは，Pi 3 と Pi 4 のパイプライン処理などの機能の違いによると考えられる。Pi 3 の Cortex-A53 は 8 ステージのイン・オーダ

で1サイクル2命令同時発行，Pi 4のCoretex-A72は15ステージのアウト・オブ・オーダで1サイクル2命令同時発行である［7］。

Pi 3とPi 4ともに，使用したSIMD命令の並列度がもともと低く，最大値を直接求めるSIMD命令がなく4つのSIMD命令になってしまい，単なる並列版では全くあるいはほとんど速度向上が得られなかった。並列版（分割版）の場合，分割された部分間（例えば，図2の前半，後半，図3の4分の1から4分の4）は独立実行でき，Pi 4ではアウト・オブ・オーダが可能なSIMD命令は並行実行されたため，速度向上が得られたと考えられる。一方，Pi 3はイン・オーダなので，速度向上が得られなかったのであろう。

並列版の分割版の分割数による実行時間の違いは，パイプライン処理やスーパースカラー実行の状況によるものだと推測される。表1のPi 3の並列版（6分割）で大幅に実行時間が増えているのは，パインプライン処理を大幅に阻害するパイプラインストールやパイプラインハザードが起こっていることが疑われる。

7　まとめ

過去の研究からの継続で，Common Lispの組み込み関数を並列化することを通じて，Pi 4のCPUの64bitモードにおけるSIMD命令の評価を行った。SIMD命令の単なる使用だけでは速度向上が得られない場合でも，パイプライン処理，アウト・オブ・オーダ実行，（SIMD命令の）スーパースカラー実行を意識したプログラムの変形により明らかな速度向上が得られる場合がある。このような速度向上が得られるか否かは，CPUの機能の有無に依存することが，2世代のRaspberry Piによる比較でわかった。

CPUの新製品発表の際に，CPUの機能強化が行われたり，低価格帯CPUに従来の高価格帯CPUの機能が搭載されるようになっているが，その機能を十分活かすには，コンパイラ任せではく，プログラムを書き換えないといけないことがあることがわかった。

今後の課題としては，過去からの研究の継続として，アップル社のM1や新しい世代のIntelのx64のCPUで，本研究と同様な評価を行うことがあげられる。

参考文献

［1］安本太一：ARM NEON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，愛知教育大学，六十六輯，pp.21-24（2017）.

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

［3］安本太一：Intel AVX2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，愛知教育大学，六十四輯，pp.15-19（2015）.

［4］安本太一：Common Lisp言語処理系の64bit化，愛知教育大学研究報告 自然科学編，愛知教育大学，五十三輯，pp.22-32（2004）.

［5］安本太一：Common Lisp言語処理系によるBroadwellの任意精度整数演算命令の評価，愛知教育大学研究報告 自然科学編，愛知教育大学，六十五輯，pp.25-28（2016）.

［6］Arm: *Arm Neon Intrinsics Reference for ACLE Q2 2021*, 02 July（2021）.

［7］中森章：ラズパイ4性能の研究2…新型Coretex- A72プロセッサBCM2711，インターフェース，第45巻，第10号，pp.28-34（2019）.

（2022年9月26日受理）