

**ストリーミング SIMD 拡張命令 2(SSE2)を使用した、  
倍精度浮動小数点ベクトルの最大/最小要素と  
そのインデックスの検出**

**バージョン 2.0**

**2000 年 7 月**

資料番号: 248602J-001

**【輸出規制に関する告知と注意事項】**

本資料に掲載されている製品のうち、外国為替および外国為替管理法に定める戦略物資等または役務に該当するものについては、輸出または再輸出する場合、同法に基づく日本政府の輸出許可が必要です。また、米国産品である当社製品は日本からの輸出または再輸出に際し、原則として米国政府の事前許可が必要です。

**【資料内容に関する注意事項】**

- ・本ドキュメントの内容を予告なしに変更することがあります。
  - ・インテルでは、この資料に掲載された内容について、市販製品に使用した場合の保証あるいは特別な目的に合うことの保証等は、いかなる場合についてもいたしかねます。また、このドキュメント内の誤りについても責任を負いかねる場合があります。
  - ・インテルでは、インテル製品の内部回路以外の使用にて責任を負いません。また、外部回路の特許についても関知いたしません。
  - ・本書の情報はインテル製品を使用できるようにする目的でのみ記載されています。
- インテルは、製品について「取引条件」で提示されている場合を除き、インテル製品の販売や使用に関して、いかなる特許または著作権の侵害をも含み、あらゆる責任を負わないものとします。
- ・いかなる形および方法によっても、インテルの文書による許可なく、この資料の一部またはすべてを複製することは禁じられています。

本資料の内容についてのお問い合わせは、下記までご連絡下さい。

インテル株式会社 資料センター

〒305-8603 筑波学園郵便局 私書箱115号

Fax: 0120-47-8832

\*一般にブランド名または商品名は各社の商標または登録商標です。

Copyright © Intel Corporation 1999 - 2001

## 目次

1	はじめに .....	5
2	最大/最小アルゴリズム .....	5
2.1	最大/最小アルゴリズムの実現.....	5
2.1.1	パフォーマンス向上の手法.....	5
2.1.2	ヒント .....	8
3	パフォーマンス .....	9
3.1	測定されたパフォーマンス向上.....	9
3.2	考察 .....	9
4	結論 .....	9
5	C/C++によるコード例.....	10
6	SSE2 テクノロジを使用した DVEC コード例 .....	11
7	SSE2 テクノロジを使用したアセンブリ・コード例 .....	15
付録 A	パフォーマンス・データ .....	A-1
テスト・システム構成 .....		A-3

## 改訂履歴

改訂	改訂履歴	日付
2.0	インテル® Pentium® 4 プロセッサに関する更新	2000年7月
1.0	初版	1999年9月

## 参考資料

このアプリケーション・ノートでは次の資料を参考にした。この資料には、ここで取り上げた事項を理解するための有用な情報が含まれている。

1. Thomas H. Cormen、Charles E. Leiserson、Ronald L. Rivest 著、“*Introduction to Algorithms*”、The MIT Press、Cambridge、Massachusetts、1991年
2. 『ストリーミングSIMD拡張命令(Streaming SIMD Extensions)を使用した単精度浮動小数点ベクトルの最大/最小要素とそれに対応するインデックスの検出』、AP-805、インテル、資料番号 243639J-002

## 1 はじめに

ストリーミング SIMD 拡張命令 2(SSE2、Streaming SIMD Extensions 2)では、SIMD(Single Instruction Multiple Data)倍精度浮動小数点命令、および SIMD 整数命令が IA-32 インテル® アーキテクチャに新しく導入された。倍精度 SIMD 命令による機能拡張の方法は、ストリーミング SIMD 拡張命令(SSE)で導入された単精度 SIMD 命令による機能拡張とよく似ている。128 ビットの整数 SIMD 拡張命令は、64 ビットの整数 SIMD 拡張命令の完全なスーパーセットで、より多くの整数データ型、整数と浮動小数点間のデータ型変換、キャッシュとシステム・メモリの効果的使用をサポートする命令が追加されている。これらの命令は、3D グラフィックス、リアルタイムの物理的な現象、空間的(3D)オーディオ、ビデオ・エンコーディング/デコーディング、暗号化、および科学計算アプリケーションによく使用される演算を高速化する。このアプリケーション・ノートでは、SSE2 を使用して倍精度浮動小数点ベクトルの最大/最小要素とそのインデックスを検出する方法を説明し、そのコード例を示す。

## 2 最大/最小アルゴリズム

これは、倍精度浮動小数点値が入った 1 次元配列(ベクトル)を検索して、最大/最小要素を見つけ出すアルゴリズムである。最大/最小要素が見つかったら、その値と、その要素のベクトル内の位置を示すインデックスの両方を返す。

SSE2 テクノロジを使用すると、2 つの倍精度浮動小数点値を一度にロードして比較できるので、このアルゴリズムのパフォーマンスが大きく向上する。

このアプリケーション・ノートでは、最大値を検出する方法を中心に説明する。わずかな修正を行うだけで、最大値と最小値の両方を見つけ出すアルゴリズムに変更できる。

### 2.1 最大/最小アルゴリズムの実現

最大/最小アルゴリズムを C 言語で実現するときは、一般に、最大値の初期値として、ベクトルの第 1 要素を採用する。この最大値を残りの要素と順に比較して行き、より大きい値が見つかったら、最大値とそのインデックスを更新する。この操作は、ベクトル内の全要素を最大値と比較するまで続ける。

このアルゴリズムを C 言語で実現すると、次の理由で効率が悪い。新しい要素と比較するたびに、次のような分岐を使用して、最大値とそのインデックスを更新するかどうかを判定する。

```
if(maxDouble < the_array[i])
{
    maxDouble = the_array[i];
    maxIndex = i;
}
```

このアルゴリズムを C 言語で実現する場合、最低 N-1 回の比較を行う必要がある。

#### 2.1.1 パフォーマンス向上の手法

この C プログラムのパフォーマンスを向上させる手法の 1 つに、ループ内の分岐(if 文)を排除する方法がある。

その方法の1つとして、ベクトル内の要素をすべて比較して最大値を見つけ出す(それには、N-1回の比較が必要である)。次に、見つかった最大値をベクトル内の各要素と比較してそのインデックスを見つけ出す。

ベスト・ケースは、ベクトルの第一要素が最大値の場合で、比較はN回で済む。ワースト・ケースは、ベクトルの最終要素が最大値の場合で、比較は2N-2回必要になる。アルゴリズム上では、ベクトル内を2回検索すると、それに比例して複雑さが増すが、分岐の排除によるパフォーマンス向上は、ベクトル内を2回検索する手間以上の価値がある。

パフォーマンスを向上させるもう1つの手法として、SSE2を使用して、同時に複数の倍精度浮動小数点値を比較する方法がある。

コード例では、最高のパフォーマンス向上を実現するために、2つの手法を組み合わせで使用している。この2つの手法を組み合わせる方法について、次に説明する。

SSE2 アーキテクチャでは、8つの128ビット・レジスタを使用する。各レジスタに64ビットの倍精度浮動小数点値を2つずつ入れられる。SSE2には、次の操作を行う大変便利なSIMD命令がある。

1. アライメントされた2つの倍精度浮動小数点値をレジスタにロードする。これはアセンブリ命令の `movapd` で行う。
2. 2つの128ビット・レジスタ(それぞれが倍精度浮動小数点値を2つ持つ)を比較し、2つの最大値を一方のレジスタにストアする。これはアセンブリ命令の `maxpd` で行う。
3. 2つの128ビット・レジスタ(それぞれが倍精度浮動小数点値を2つ持つ)を等しいかどうか比較し、その結果を一方のレジスタにストアする。これはアセンブリ命令の `cmpeqpd` で行う。
4. 128ビット・レジスタの値を32ビット・レジスタに移動する。これはアセンブリ命令の `movmskpd` で行う。
5. 128ビット・レジスタの論理ORを行う。これはアセンブリ命令の `orpd` で行う。
6. この他に、アライメントされていない値をロードしたり、レジスタ間で値をシャッフルする命令がある。

ベクトル内の最大値を見つける最初のループで、7つのレジスタに続けて14要素をロードして比較する(常に `xmm0` レジスタに最大値が入るようにする)。このループは次のようなアセンブリ・コードになる。

```
maxloop:
    movapd xmm1,[edi - 16]
    movapd xmm2,[edi - 32]
    movapd xmm3,[edi - 48]
    movapd xmm4,[edi - 64]
    movapd xmm5,[edi - 80]
    movapd xmm6,[edi - 96]
    movapd xmm7,[edi - 112]
    sub    edi,112
    maxpd  xmm0,xmm1
    maxpd  xmm2,xmm3
```

```

maxpd   xmm4, xmm5
maxpd   xmm6, xmm7
maxpd   xmm0, xmm2
maxpd   xmm4, xmm6
maxpd   xmm0, xmm4
sub     ecx, 14
cmp     ecx, 14
jge     maxloop

```

このループは、最後に1つしかジャンプ命令がないことに注意する。プロセッサはこの分岐を正確に予測できるので、パフォーマンス上のペナルティはない(C言語の `for` ループと同様である)。アーキテクチャ上では、このループは `maxpd` 命令のスケジュールという上限があるだけなので、非常に高速に実行できる。

最大値を見つけたら、次にそのインデックスを特定する必要がある。そのためのアセンブリ・コードは次のようになる。

```

indexloop:
    cmp     ecx, 12
    jle     indexlast
    movapd  xmm0, [edi]
    movapd  xmm1, [edi + 16]
    movapd  xmm2, [edi + 32]
    movapd  xmm3, [edi + 48]
    movapd  xmm4, [edi + 64]
    movapd  xmm6, [edi + 80]
    add     edi, 96
    add     edx, 12
    sub     ecx, 12

    cmpeqpd xmm0, xmm5
    cmpeqpd xmm1, xmm5
    cmpeqpd xmm2, xmm5
    cmpeqpd xmm3, xmm5
    cmpeqpd xmm4, xmm5
    cmpeqpd xmm6, xmm5
// OR our registers to see if the maximum value was found
    orpd   xmm0, xmm1
    orpd   xmm2, xmm3
    orpd   xmm4, xmm6
    orpd   xmm0, xmm2
    orpd   xmm0, xmm4
// Move the result of the OR into eax
movmskpd  eax, xmm0
    cmp     eax, 0
    jz     indexloop

```

このループは最大値を見つけるためのループとよく似ている。ただ1つ違うのは、続けて6つのレジスタにロードしている点である。これは、1つのレジスタにビット・マスクが入っていることと、比較回数を偶数にしたいため(`orpd` 命令を最大限に活用するため)である。

`movmskpd` 命令は負荷が大きいため、この例ではインデックスが見つかったかどうか調べるのに `orpd` 命令を使用している(毎回 `movmskpd` 命令を使用するのを避けている)。次に、ループを抜けてからインデックスを解決する。`orpd` 命令と `movmskpd` 命令は同じ実行ユニットを使用する。したがって、これらの命令のスケジュールと実行という上限があるだけなので、このループも非常に高速である。

この2つのループを組み合わせるにより、SSE2 を活用して最大/最小アルゴリズムを高速化できる。ベクトル内を2度も検索するのは不利なように思えるが、SSE2 テクノロジーを最大限に活用できるので、それによって検索時間がリニアに増大するのを軽減できる。

### 2.1.2 ヒント

1. ベクトルがメモリ上で16バイト境界にアライメントされていれば、パフォーマンスは大きく向上する。このアプリケーション・ノートに示すコード例では、ベクトルが8バイトまたは16バイト境界にアライメントされていると仮定している。アライメントを確認する方法については、コンパイラのマニュアルで確認のこと。
2. インデックス・ループの前で、ベクトルの最後に最大値がないかチェックする。ベクトルの最初でも同じことを行う。ここでインデックスが特定できれば、`indexdone` というラベルにジャンプする。この方法には次のような利点がある。
  - ベクトルの最初と最後のアライメントされていない要素をチェックしておけば、ループ本体でアライメントを心配しなくてよい。
  - ベクトルの終わりまで進まないうちに、最大値とそのインデックスが見つかるのがわかっている状態でインデックス・ループに入ることができる。
3. 最小値を見つけたい場合は、アセンブリ・コードの `maxpd` 命令を `minpd` 命令に置き換えるだけでよい。SSE2 C++倍精度ベクトル・クラス(DVEC)コードも同様に `simd_max` 命令を `simd_min` 命令に置き換えるだけでよい。
4. 最初のループはベクトルの末尾から先頭に向けて進み、2番目のループはベクトルの先頭から末尾に向けて進む。これは、キャッシュを最も効果的に活用するためである。
5. ワorst・ケースは、最大値がベクトルの最後から3要素目にある場合である。最後の2要素は、ループを開始する前に必ずチェックしている。

## 3 パフォーマンス

### 3.1 測定されたパフォーマンス向上

SSE2 テクノロジを使用したアセンブリ・コードは、平均的ケースならば、同じ機能を実行する C プログラムより 2 倍以上高速になった。これには次のような理由が考えられる。

1. SSE2 の使用により、2 つの倍精度浮動小数点要素を一度に処理できる。SSE2 は、従来の x87 命令よりはるかに高速である。
2. 次のアルゴリズムの最適化
  - C コードから条件分岐(`if` 文)を排除
  - メモリから要素をロードするときのパイプライン
  - 1 回のループ実行で、複数要素を持つ複数レジスタを操作

このようなアルゴリズムの最適化は、新しい SSE2 命令を使用しないと実現できない。

### 3.2 考察

このアプリケーション・ノートに示す SSE2 コード例は、多くの場合有効に機能する。ただし、ワースト・ケース(最大値がベクトルの最後の方にある)でベクトル・サイズが大きい(10,000 要素以上)場合は十分な高速化が得られない可能性がある。その場合の解決方法を次に示す。

1. ベクトルをいくつかに分割して、そのサイズ(バイト数)が第 1 レベル・キャッシュ内に収まるようにする。分割した各部に対して同じアルゴリズムを使用する。
2. 別のプログラミング方法を採用する。そのコード例については、アプリケーション・ノート AP-805 を参照のこと。

## 4 結論

SSE2 を使用することにより、最大/最小アルゴリズムのパフォーマンスが大きく向上した。このアルゴリズムで使用した `maxpd` 命令が、そのよい例である。SSE2 を使用すると、2 つの倍精度浮動小数点要素を同時に処理できるので、より強力なアルゴリズム開発の可能性が開かれる。

## 5 C/C++によるコード例

```
double max_c(double *the_array, int array_size, int *index)
{
    int maxIndex = 0;
    // Initialize maxDouble with the value of the first item in the vector
    double maxDouble = the_array[0];
    for(int i=1; i<array_size; i++)
    {
        // Compare maxDouble with remaining vector elements
        // Keep track of the maximum value and its index
        if(maxDouble < the_array[i])
        {
            maxDouble = the_array[i];
            maxIndex = i;
        }
    }
    *index = maxIndex;
    return(maxDouble);
}
```

## 6 SSE2 テクノロジを使用した DVEC コード例

```
double maxw_dvec_unrolled(double *the_array, int array_size, int *index)
{
    // Assume 8 or 16 byte alignment
    assert((((unsigned int)&the_array[0] & (0x07)) == 0));
    // Use C code if array size is small
    if (array_size<=18)
    {
        int    maxIndex = 0;
        double maxDouble = the_array[0];

        for(int i=1; i<array_size; i++)
        {
            if(maxDouble < the_array[i])
            {
                maxDouble = the_array[i];
                maxIndex = i;
            }
        }
        *index = maxIndex;
        return(maxDouble);
    }

    F64vec2 r1(0.0,0.0), r2(0.0,0.0), r3(0.0,0.0), r4(0.0,0.0), r5(0.0,0.0),
    r6(0.0,0.0), r7(0.0,0.0), r8(0.0,0.0);
    double max = 0.0;
    int i=0;
    int j, mask;
    F64vec2 *aligned_front_of_array;
    F64vec2 *aligned_end_of_array;
    *index = 0;
    int front_alignment,back_alignment;
    // Calculate alignments and compensate if 8 byte aligned
    if((((unsigned int)&the_array[0]) & (0x0F)) == 0) front_alignment = 1;
    else front_alignment = 0;

    if((((unsigned int)&the_array[array_size - 1]) & (0x0F)) == 0)
    back_alignment = 1;
    else back_alignment = 0;

    if(!back_alignment) {
        aligned_end_of_array = (F64vec2 *)&the_array[array_size - 2];
    } else aligned_end_of_array = (F64vec2 *)&the_array[array_size - 1];
}
```

```
if(!front_alignment) {
    aligned_front_of_array = (F64vec2 *)&the_array[1];
} else aligned_front_of_array = (F64vec2 *)&the_array[0];

r1 = _mm_loadu_pd(&the_array[array_size - 2]);
r2 = _mm_loadu_pd(&the_array[0]);
r1 = simd_max(r1,r2);
Loop through the vector and find the maximum value
j = array_size/16 * 8;
for(i=1; i<j; i+=8) {
    r1 = simd_max(r1,*(aligned_end_of_array - i));
    r2 = simd_max(r2,*(aligned_end_of_array - (i+1)));
    r3 = simd_max(r3,*(aligned_end_of_array - (i+2)));
    r4 = simd_max(r4,*(aligned_end_of_array - (i+3)));
    r5 = simd_max(r5,*(aligned_end_of_array - (i+4)));
    r6 = simd_max(r6,*(aligned_end_of_array - (i+5)));
    r7 = simd_max(r7,*(aligned_end_of_array - (i+6)));
    r8 = simd_max(r8,*(aligned_end_of_array - (i+7)));
}

r1 = simd_max(r1,*(aligned_front_of_array));
r2 = simd_max(r2,*(aligned_front_of_array+1));
r3 = simd_max(r3,*(aligned_front_of_array+2));
r4 = simd_max(r4,*(aligned_front_of_array+3));
r5 = simd_max(r5,*(aligned_front_of_array+4));
r6 = simd_max(r6,*(aligned_front_of_array+5));
r7 = simd_max(r7,*(aligned_front_of_array+6));
r8 = simd_max(r8,*(aligned_front_of_array+7));
r1 = simd_max(r1,r2);
r3 = simd_max(r3,r4);
r5 = simd_max(r5,r6);
r7 = simd_max(r7,r8);
r1 = simd_max(r1,r3);
r5 = simd_max(r5,r7);
r1 = simd_max(r1,r5);

// Create a mask of maximum values in r5
r5 = unpack_low(r1,r1);
r1 = unpack_high(r1,r1);
r5 = simd_max(r5,r1);

_mm_store_sd(&max,r5); // Store the max value

// Calculate the index now (starting from the front of array in cache)
if(!front_alignment) {
```

```
r1 = _mm_loadu_pd(&the_array[0]);
r1 = cmpeq(r1,r5);
mask = move_mask(r1);
// If we are lucky, the max is in the front of the array
if(mask) {
    if(mask == 3) mask = 1;
    *index = mask-1;
    return(max);
}
*index = 1;
}

// Last two doubles to look at
r1 = _mm_loadu_pd(&the_array[array_size - 2]);
r1 = cmpeq(r1,r5);
mask = move_mask(r1);
if(mask) {
    if(mask == 2) *index = array_size - 1;
    else *index = array_size - 2;
    return(max);
}

i = 0;
// Go through array from the front and look for index
while(!mask) {
    r1 = cmpeq(*(aligned_front_of_array+i),r5);
    i++;
    r2 = cmpeq(*(aligned_front_of_array+i),r5);
    i++;
    r3 = cmpeq(*(aligned_front_of_array+i),r5);
    i++;
    r4 = cmpeq(*(aligned_front_of_array+i),r5);
    i++;
    r6 = cmpeq(*(aligned_front_of_array+i),r5);
    i++;
    r7 = cmpeq(*(aligned_front_of_array+i),r5);
    i++;

    r1 = _mm_or_pd(r1,r2);
    r3 = _mm_or_pd(r3,r4);
    r6 = _mm_or_pd(r6,r7);
    r1 = _mm_or_pd(r1,r3);
    r1 = _mm_or_pd(r1,r6);

    mask = move_mask(r1);
    if((i*2+12) >= array_size) break;
```

```
}

i -= 6;
mask = 0;
while(!mask) {
    r1 = cmpeq(*(aligned_front_of_array+i),r5);
    mask = move_mask(r1);
    i++;
}
i--;
if(mask == 3) mask = 1;
*index += 2*i + (mask-1);
return(max);
}
```

## 7 SSE2 テクノロジを使用したアセンブリ・コード例

```
double maxw_asm(double *the_array, int array_size, int *index)
{
    double maximum;
    int indexvalue = 0;
    double *end_of_array,*aligned_end_of_array,*aligned_front_of_array;

    // Assume 8 or 16 byte alignment
    assert(((unsigned int)&the_array[0] & (0x07)) == 0);

    // Array size must be at least 18 elements or we use the C code
    if (array_size<=18)
    {
        int  maxIndex = 0;
        float maxFloat = the_array[0];

        for(int i=1; i<array_size; i++)
        {

            if(maxFloat < the_array[i])
            {
                maxFloat = the_array[i];
                maxIndex = i;
            }
        }
        *index = maxIndex;
        return(maxFloat);
    }

    end_of_array = &the_array[array_size - 1];
    int front_alignment,back_alignment;

    if((((unsigned int)&the_array[0]) & (0x0F)) == 0) front_alignment = 1;
    else front_alignment = 0;

    if((((unsigned int)&the_array[array_size - 1]) & (0x0F)) == 0)
    back_alignment = 1;
    else back_alignment = 0;

    if(!back_alignment) {
        aligned_end_of_array = &the_array[array_size - 2];
    } else aligned_end_of_array = &the_array[array_size - 1];
}
```

```
if(!front_alignment) {
    aligned_front_of_array = &the_array[1];
    indexvalue = 1;
} else aligned_front_of_array = &the_array[0];

__asm {
    mov     esi,the_array
    mov     ecx,array_size
    mov     edx,ecx
    mov     edi,aligned_end_of_array
    mov     eax,end_of_array

    movupd  xmm0,[eax - 8]
    movupd  xmm1,[esi]
    maxpd   xmm0,xmm1
    sub     ecx,1

    // Loop where we find the maximum
maxloop:

    movapd  xmm1,[edi - 16]
    movapd  xmm2,[edi - 32]
    movapd  xmm3,[edi - 48]
    movapd  xmm4,[edi - 64]
    movapd  xmm5,[edi - 80]
    movapd  xmm6,[edi - 96]
    movapd  xmm7,[edi - 112]
    sub     edi,112

    maxpd   xmm0,xmm1
    maxpd   xmm2,xmm3
    maxpd   xmm4,xmm5
    maxpd   xmm6,xmm7
    maxpd   xmm0,xmm2
    maxpd   xmm4,xmm6
    maxpd   xmm0,xmm4

    sub     ecx,14
    cmp     ecx,14
    jge     maxloop
```

```
mov     edi,aligned_front_of_array

maxdone:

movapd  xmm2,[edi]
maxpd   xmm0,xmm2

add     edi,16
sub     ecx,2
cmp     ecx,0
jg     maxdone

mov     edi,aligned_front_of_array

shufpd  xmm5,xmm0,3
maxpd   xmm5,xmm0
shufpd  xmm5,xmm5,1
maxpd   xmm5,xmm0    // Created mask of maximum values in xmm5
movsd   maximum,xmm5 // Stored maximum value

sub     edx,2
movupd  xmm0,[eax - 8]
cmpeqpd xmm0,xmm5
movmskpd eax,xmm0
cmp     eax,0
jne     indexdone

xor     edx,edx
movupd  xmm0,[esi]
cmpeqpd xmm0,xmm5
movmskpd eax,xmm0
cmp     eax,0
jne     indexdone

mov     ecx,array_size

// Loop where we find the index
indexloop:

cmp     ecx,12
jle     indexlast
movapd  xmm0,[edi]
movapd  xmm1,[edi + 16]
```

```
movapd xmm2,[edi + 32]
movapd xmm3,[edi + 48]
movapd xmm4,[edi + 64]
movapd xmm6,[edi + 80]
add    edi,96
add    edx,12
sub    ecx,12
```

```
cmpeqpd xmm0,xmm5
cmpeqpd xmm1,xmm5
cmpeqpd xmm2,xmm5
cmpeqpd xmm3,xmm5
cmpeqpd xmm4,xmm5
cmpeqpd xmm6,xmm5
```

```
orpd xmm0,xmm1
orpd xmm2,xmm3
orpd xmm4,xmm6
orpd xmm0,xmm2
orpd xmm0,xmm4
movmskpd eax,xmm0
cmp    eax,0
jz    indexloop
```

```
sub    edx,12
sub    edi,96
```

indexlast:

```
movapd xmm0,[edi]
cmpeqpd xmm0,xmm5
movmskpd eax,xmm0
add    edi,16
add    edx,2
cmp    eax,0
jz    indexlast
```

```
sub    edx,2
add    edx,indexvalue
```

indexdone:

```
cmp    eax,2
```

```
    jne    end
    add    edx,1

end:

    mov    indexvalue,edx
}

    *index = indexvalue;
    return(maximum);
}
```

## 付録 A - パフォーマンス・データ

表 1: 最大値検出のパフォーマンス・データ

パフォーマンス・データ(単位はマイクロ秒)		
ケース	インテル® Pentium® III プロセッサ (733 MHz)	インテル® Pentium 4 プロセッサ (1.2 GHz)
C コード、ミドル・ケース*	7.74	4.94
SSE2 ASM、ベスト・ケース*		1.09
SSE2 ASM ミドル・ケース*		1.74
SSE2 ASM ワースト・ケース*		2.22
SSE2 DVEC、ベスト・ケース*		1.14
SSE2 DVEC、ミドル・ケース*		1.86
SSE2、ワースト・ケース*		2.48

\* ベスト・ケースは最大要素がベクトルの先頭にあるケース。ミドル・ケースは、最大要素がベクトルの中央にあるケース。ワースト・ケースは、最大要素がベクトルの最後にあるケース(アルゴリズムを参照)。

表 2: 表 1 のパフォーマンス・データからの高速化

コード・タイプとプラットフォーム	高速化
インテル® Pentium 4 プロセッサ(SSE2 ASM vs.C コード、ミドル・ケース)	2.84
インテル Pentium 4 プロセッサ(SSE2 DVEC vs.C コード、ミドル・ケース)	2.66
インテル Pentium 4 プロセッサで実行した C コード vs. Pentium III プロセッサで実行した C コード	1.57

表 1 と 2 は、1.2 GHz インテル® Pentium® 4 プロセッサおよび 733 MHz インテル Pentium III プロセッサで、最大/最小要素検出プログラムを実行したときのパフォーマンス測定値である。測定に使用した Pentium 4 および Pentium III プロセッサ・システムの詳細については、「テスト・システム構成」を参照のこと。どちらのシステムでも、第 1 レベル・キャッシュ内に収まるサイズのテスト・データを使用してパフォーマンスを測定した。

代表的なベクトル長は、1000 要素とした。C コードでは明らかに、最大/最小要素の位置によってパフォーマンスに変化はない。C コードのコンパイルでは、インテル® C/C++コンパイラによる最高レベルの最適化を行った。アセンブリ(ASM)コードと、ベクトル・クラス・ライブラリ(DVEC)コードでは、最大/最小要素の位置がパフォーマンスに影響している。

最大/最小要素の検出では、SSE2 の SIMD 効果により、ベクトル化しない場合の 2 倍の高速化が期待されたかもしれない。実際には、SSE2 テクノロジーの採用により、2.84 倍の高速化が実現された(標準的ケース – 最大/最小要素がベクトルの中央にある場合)。これは、SSE2 テクノロジーを最大限に活用する全般的アルゴリズム改良の結果である。

## テスト・システム構成

表 3: インテル® Pentium III プロセッサのシステム構成

プロセッサ	Pentium III プロセッサ(733 MHz)
システム	インテル® Desktop Board VC820
BIOS バージョン	VC82010A.86A.0028.P10
2次キャッシュ	256 KB
メモリ・サイズ	128 MB RDRAM PC800-45
Ultra ATA ストレージ・ドライバ	製品候補 6.00.012
ハード・ディスク	IBM DJNA-371800 ATA-66
ビデオ・コントローラ/ バス	Creative Labs 3D Blaster <sup>†</sup> Annihilator <sup>†</sup> Pro AGP nVidia GeForce256 <sup>†</sup> DDR -32MB
ビデオ・ドライバの リビジョン	NVidia Reference Driver 5.22
オペレーティング・ システム	Windows <sup>†</sup> 2000 ビルド 2195

表 4: インテル® Pentium 4 プロセッサのシステム構成

プロセッサ	Pentium 4 プロセッサ(1.2 GHz)
システム	インテル® Desktop Board D850GB
BIOS バージョン	GB85010A.86A.0014.D.0007201756
2次キャッシュ	256 KB
メモリ・サイズ	128 MB RDRAM PC800-45
Ultra ATA ストレージ・ ドライバ	製品候補 6.00.012
ハード・ディスク	IBM DJNA-371800 ATA-66
ビデオ・コントローラ/ バス	Creative Labs 3D Blaster Annihilator Pro AGP nVidia GeForce256 DDR -32MB
ビデオ・ドライバの リビジョン	NVidia Reference Driver 5.22
オペレーティング・ システム	Windows 2000 ビルド 2195