

VTune™パフォーマンス拡張環境の利用による、 Pentium® III プロセッサのインターネット・ストリーミング SIMD 拡張命令を活用するためのプログラミング手法

Joe H. Wolf - インテル社、マイクロプロセッサ・プロダクツ・グループ(MPG)

インデックス・ワード: VTune™、Intel® C/C++コンパイラ、組み込み関数、ベクトル・クラス・ライブラリ、ベクトル化、イベント・ベース・サンプリング、インテル・パフォーマンス・ライブラリ集

摘要

この記事では、Pentium® III プロセッサのインターネット・ストリーミング SIMD 拡張命令が持つ、優れた処理性能を有効に活用するためのプログラミング手法について解説します。独自の機能を備えた VTune™パフォーマンス拡張環境 4.0 の各種ツールを使うと、インターネット・ストリーミング SIMD 拡張命令への理解が深まり、ソフトウェアの開発にあたって新命令の機能を十分に活かしたアプリケーションを開発でき、チューニングによるパフォーマンスの最適化が図れます。

VTune パフォーマンス拡張環境 4.0 は、Intel® C/C++コンパイラ、VTune 性能アナライザ、インテル・アーキテクチャ・パフォーマンス・トレーニング・センター、およびレジスタ・ビューイング・ツールで構成されています。これらのツールでは、以下のようなプログラミング手法を利用できます。

- (1) **組み込み関数** - アプリケーションのコードに挿入して使用する、通常関数と同様の呼び出しルーチンです。組み込み関数は、C/C++コンパイラによってインライン・コードとして生成されます。
- (2) **ベクトル・クラス・ライブラリ** - 組み込み関数を C++ のクラスとして抽象化したものです。
- (3) **ベクトル化** - コンパイラの特長な最適化機能で、char、short、int、または float の配列を操作するループのコードを、SIMD 命令を使用した効率の高いコードに変換します。
- (4) **インテル・パフォーマンス・ライブラリ集** - インターネット・ストリーミング SIMD 拡張命令を活用して最適化されたルーチンにより、一般に多用される各種アルゴリズムの高速実行を実現するライブラリ集です。このライブラリ集にはインテル信号処理ライブラリ、インテル・イメージ処理ライブラリ、インテル音声/信号認識ライブラリ、インテル数値演算ライブラリ、インテル JPEG ライブラリが含まれています。

さらに、VTune 性能アナライザを使用すれば、アプリケーションのパフォーマンスをさまざまな方法で分析しながら、Pentium III プロセッサに最適化するためのチューニング・ア

ドバイスが得られます。この記事では、そのような機能の具体的な使用例も紹介します。

はじめに

インテル・アーキテクチャに Intel® MMX®テクノロジーが導入されたのは 1996 年のことでした。MMX®テクノロジーは、整数データ・タイプに対応した SIMD (Single-Instruction, Multiple-Data)命令セット・アーキテクチャ(ISA)により、飛躍的なパフォーマンスの向上を実現しました。MMX®テクノロジーは、整数ベースのコードのパフォーマンスを高める上では現在でも十分に有効な手段です。しかし、MMX®テクノロジーが導入されてから約 2 年間は、SIMD テクノロジーを利用するには、アセンブリ・ファイルや C または C++ のインライン・コードの形でアセンブリ・コードを使用するしか方法がない、という状況が続きました。パフォーマンス的には、アセンブリ・コードの方が高級言語を使用してコンパイルされたコードよりも優れていることが多いのは事実ですが、アセンブリ言語によるプログラミングは、コーディング、パフォーマンス・チューニング、保守、新しい命令セット・アーキテクチャへの移行のいずれの面においても困難かつ非効率的です。MMX®テクノロジーや、Pentium III プロセッサのインターネット・ストリーミング SIMD 拡張命令などの SIMD 命令セット・アーキテクチャにおいて、開発者が高級言語のサポートを必要としていることは明らかでした。

VTune™パフォーマンス拡張環境 4.0 は、そのような高級言語のサポートへの要望に応えて開発されたものです。プログラマは、VTune の優れた開発方法を採用することにより、Intel® C/C++コンパイラ、VTune アナライザ、パフォーマンス・ライブラリ集の高級言語サポート機能を使用して、SIMD 命令セット・アーキテクチャの性能を余すところなく活用できるようになります。

この記事の Intel C/C++コンパイラに関するセクションでは、最初に簡単なループのコード例を示し、その例を用いながら、インターネット・ストリーミング SIMD 拡張命令をサポートするいくつかの方法について、その方法で最適なパフォーマンスを実現するためのアドバイスと併せて説明していきます。VTune アナライザのセクションでは、同じコード例を使って、アプリケーションのコードの中から、インターネット・ストリーミング SIMD 拡張命令による再コーディングの効果が大

VTune™パフォーマンス拡張環境の利用による、Pentium® III プロセッサの
インターネット・ストリーミング SIMD 拡張命令を活用するためのプログラミング手法

きい箇所を見つけ出す方法を示します。また、VTune アナライザを使用して、アプリケーションの修正やチューニングに役立つアドバイスを取得する方法と、プリフェッチ命令やストリーミング・ストア命令を挿入する場合の分析に欠かさないキャッシュ使用率の情報を取得する方法についても説明します。

ここで示すいくつかのプログラミング手法の間にはパフォーマンスの差はほとんどありませんが、いずれの方法でも、スカラ浮動小数点による実装に比べればパフォーマンスは大幅に向上します。しかも、そのパフォーマンスの向上を実現するためのコストは、アセンブリ・コードを書く場合の数分の1で済みます。結論として、SIMD 命令セット・アーキテクチャを活用するためのプログラミング手法にはいくつかの選択肢があり、各手法の違いはコーディング・スタイルや実装効率の問題に過ぎない、ということが言えます。

INTEL® C/C++ コンパイラ

Intel® C/C++ コンパイラは、Microsoft® Developer Studio 環境でプラグインとして使用できる、高度な最適化機能を備えたコンパイラです。Intel® C/C++ コンパイラは C++ 標準に準拠しており、言語仕様、デバッグ機能、オブジェクト・フォーマットにおいて、バージョン 4.2 以上の Microsoft Visual C++ と互換性があります。

Intel® C/C++ コンパイラは、プログラマがインターネット・ストリーミング SIMD 拡張命令を活用できるように、インライン・アセンブリ、組み込み関数、ベクトル・クラス・ライブラリ、ベクトル化の機能をサポートしています。インターネット・ストリーミング SIMD 拡張命令のパフォーマンスを最大限に高めるにはデータを 16 バイト境界にアライメントする必要があるため、データを正しくアライメントするための方法も数種類用意されています。このセクションでは、それらの方法について詳しく説明します。

データとスタックのアライメント

Pentium® III プロセッサのインターネット・ストリーミング SIMD 拡張命令を使用して最高のパフォーマンスを実現するには、データを 16 バイト境界に合わせてアライメントする必要があります。また、正しくアライメントされていないデータに対してアライメント整合データ用の移動命令を使用すると、例外が発生する場合があります。Intel® C/C++ コンパイラでは、このような問題に対処するために、次のようなメカニズムが用意されています。

- 新しい `_m128` データ・タイプ。このデータ・タイプは、4 つの単精度浮動小数点値を含む構造体、あるいは 1 つの XMM レジスタを表すものと考えてください。このタイプとして宣言されたデータは、グローバル・データとローカル・データの区別に関わらず、自動的に 16 バイト境界にアライメントされます。
- C++ コードの中で新しいデータ・オブジェクトとして使用できる `F32vec4` クラス。これは、上記の `_m128` タイプのデータをメンバとして持つクラス・オブジェクトです。コンパイラは、このクラスのオブジェクトを `_m128` データ・タイプと同じように扱います。
- データの宣言時に、そのデータをアライメントするようにコンパイラに指示する `__declspec(align(16))` 指定子。こ

の指定子は、インターネット・ストリーミング SIMD 拡張命令が使われているルーチンに渡される可能性があるグローバル・データを宣言する場合に特に有用です。次に使用例を示します。

```
__declspec(align(16)) float buffer[400];
```

この宣言により、`buffer` 変数は、100 個の `_m128` データ・タイプまたは 100 個の `F32vec4` クラス・オブジェクトの配列と同様に使用できるようになります。次に示す例では、`F32vec4` オブジェクトの `x` がアライメントの合ったデータを使用して生成されます。この場合、`buffer` が `__declspec(align(16))` 指定子なしで宣言されていると、エラーが発生する場合があります。

```
void foo() {
    F32vec4 x = *(_m128 *) buffer;
    ...
}
```

- 一部のケースでは、パフォーマンスの向上を目的として、コンパイラが、`_m64` (MMX テクノロジーで導入された整数の SIMD データ・タイプ)、または `double` データを扱うルーチンをデフォルトで 16 バイト境界にアライメントする場合があります。また、コンパイラのコマンドライン・スイッチとして `-Qsfalign16` を指定すれば、アライメントを合わせる対象を、インターネット・ストリーミング SIMD 拡張命令のデータを扱うルーチンのみに限定できます。デフォルトでは、`-Qsfalign8` スイッチを指定した場合と同様に、8 バイトまたは 16 バイトのデータ・タイプを扱うルーチンが 16 バイト境界にアライメントされます。

これらの拡張データ・タイプが使用されている関数のスタック・フレームは、デバッグ用コードの場合も含め、コンパイラによって自動的にアライメントされます。スタック・フレームの実際の配置は、参考資料[1]に詳しく示されています。これらの拡張機能を効果的に使用するための詳細情報や使用例については、参考資料[2]および[5]を参照してください。

組み込み関数

組み込み関数は C の関数と同様の呼び出しルーチンで、コンパイラによって最適なインライン・コードとして生成されます。組み込み関数はそれぞれ、特定のインターネット・ストリーミング SIMD 拡張命令、または MMX® テクノロジー命令に対応付けられています。組み込み関数の大部分は、引数として `_m128` データ・タイプ、または `_m64` (整数) データ・タイプを取ります。それぞれの組み込み関数とそれに対応するアセンブリ命令の間には 1 対 1 の対応関係がありますが、レジスタの割り当てや命令のスケジューリング管理をコンパイラに任せられる分、プログラマとしては組み込み関数を使用した方がずっと効率的です。また、データ初期化用の組み込み関数も多数用意されているので、`_m128` データ・タイプ (または 1 つの XMM レジスタ) のローディングも容易に行えます。

図 1 に、C++ で書かれた簡単なループの例を示します。このコード例は、VTune アナライザの機能を説明するこの後のセクションでも、例として使用します。

* 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。

```
float xa[ARRAY_SIZE], xb[ARRAY_SIZE],
      xc[ARRAY_SIZE];

float q;

void do_c_triad() {

    for (int j = 0; j < ARRAY_SIZE; j++) {
        xa[j] = xb[j] + q * xc[j];
    }
}
```

図 1: C++で書かれた、3つ組データのループ

この図のループでは、3つの単精度浮動小数点データを使用した簡単な演算を実行しています。1つのベクトルをスケーリングし($q * xc[j]$)、それを別のベクトルに加えて、結果を格納しているだけです。ループ内ではデータが再利用されていないことに注意してください。

図 2 は、図 1 の例をコーディングする際に使用できる組み込み関数の構文例を示したものです。

```
__m128 _mm_set_ps1(float f)

__m128 _mm_load_ps(float *mem)

__m128 _mm_mul_ps(__m128 x, __m128 y)

__m128 _mm_add_ps (__m128 x, __m128 y)

void _mm_store_ps(float *mem, __m128 x)
```

図 2: 組み込み関数の構文

`_mm_set_ps1()`は、float のスカラ変数または定数のデータを、`__m128` 変数全体にレプリケートまたはブロードキャストするために使用します。

`_mm_load_ps()`は、float 配列などの特定のメモリ位置のデータを `__m128` 変数にロードするために使用します。

`_mm_mul_ps()`と `_mm_add_ps()`はそれぞれ、`__m128` タイプの 2 つのオペランドを乗算または加算し、結果を `__m128` データ・タイプとして返します。

`_mm_store_ps()`は、第 2 引数の `__m128` 変数を、第 1 引数として指定されたメモリ位置に格納します。

参考資料[2]および[6]には、Pentium® III プロセッサの全命令の一覧と、すべての組み込み関数を網羅したリストが記載されています。

```
#define VECTOR_SIZE 4

__declspec(align(16)) float xa[ARRAY_SIZE],
      xb[ARRAY_SIZE], xc[ARRAY_SIZE];

float q;

void do_intrin_triad() {
    __m128 tmp0, tmp1;

    tmp1 = _mm_set_ps1(q);
    for (int j = 0; j < ARRAY_SIZE; j+=VECTOR_SIZE){

        tmp0 = _mm_mul_ps(*((__m128 *) &xc[j]), tmp1);
        *(__m128 *) &xa[j] =

            _mm_add_ps(tmp0, *((__m128 *) &xb[j]));
    }
}
```

図 3: 組み込み関数による3つ組データ・ループのエンコーディング

図 3 は、組み込み関数を使用して図 1 のループをエンコードしたコード例です。このコードでは特に次のような点に注目してください。

1. この例のループではグローバル・データに対する演算を実行しているため、データを確実に 16 バイト境界に合わせてアライメントしておかなければなりません。そのため、グローバル・スコープでの float 配列の宣言時に、`__declspec(align(16))`指定子を使用する必要があります。
2. この例に限らず、SIMD 命令を使用してループをエンコードする場合、ベクトル・サイズ(1つの SIMD 命令で同時に処理可能な要素の数)に応じてループの反復回数を調整する、ストリップ・マイニングと呼ばれる操作が必要です。この例では、インターネット・ストリーミング SIMD 拡張命令の XMM レジスタまたはデータ・タイプのサイズが 4 であるため、ループのインデックス変数を 4 ずつインクリメントする `j+=VECTOR_SIZE` の操作によって、ループの反復回数を元の 4 分の 1 に減らしています。
3. `_mm_set_ps1()`組み込み関数を使用して、スカラ `q` の値を `__m128` データ・タイプの `tmp1` 変数全体にブロードキャストしています。 `tmp1` の値はループ内では不変であるため、`_mm_set_ps1()`組み込み関数をループの外で使用していることにも注意してください。
4. 配列 `xb` および `xc` のデータは、`_mm_load_ps()`組み込み関数を使用して `__m128` タイプの変数に明示的にロードするのではなく、`__m128` タイプにキャストした上で直接 `_mm_mul_ps()`および `_mm_add_ps()`組み込み関数の引数として使用しています。この方法を採用することにより、コンパイラがレジスタの割り当てを完全に制御できるようになるだけでなく、データのロードも、そのデータが本当に必要な時点で合わせて的確に実行されるようになります。同様に、組み込み関数による加算の結果を格納する操作でも、`_mm_store_ps()`組み込み関数を使用する代わりに、結果を格納する配列 `xa` をキャストしてそこに直接

代入しています。こうするだけで、コンパイラがプログラムに代わって適切なストア命令を生成してくれます。

このように組み込み関数ではコンパイラが多くの作業を代行してくれるため、プログラマは、組み込み関数を使用することにより、SIMD アルゴリズムのコーディングを大幅に効率化できます。

SIMD 組み込み関数の使用ガイド

このセクションでは、組み込み関数を使用して最大限のパフォーマンスを得るためのガイドラインを紹介します。このガイドラインの内容は、すべて Intel® C/C++コンパイラ 4.0 のリリース・ノートから抜粋したものです。

- ローカル変数が使用できる場合には、static 変数や extern 変数の使用は避けてください。static 変数や extern 変数は、通常、レジスタ上には保持されません。また、C 言語のエイリアス規則により、ポインタを使用して代入を行うと、一般に static 変数や extern 変数はエイリアスされるため、命令のスケジューリングが制限される原因となります。

効率の良くないコード

```
void foo (m128 *dst, m128 *src, m128 junk)
{
    static m128 t;
    int i;

    for (i = 0; i < 1000; i++, dst++, src++)
    {
        t = _mm_mul_ps(*src, junk);
        *dst = _mm_add_ps(*dst, t);
    }
}
```

効率的なコード

```
void foo (__m128 *dst, __m128 *src, m128
junk)
{
    m128 t;
    int i;

    for (i = 0; i < 1000; i++, dst++, src++)
    {
        t = _mm_mul_ps(*src, junk);
        *dst = _mm_add_ps(*dst, t);
    }
}
```

- 変数やパラメータのアドレスを参照することは避けてください。アドレス演算子&を使用して変数やパラメータのアドレスを参照すると、その変数はレジスタに保持されなくなります。したがって、その変数は必ずメモリ上に置かれることになり、パフォーマンスの低下を招きます。また、static 変数や extern 変数の場合と同様、ポインタを使用した代入を行うとその変数がエイリアスされるため、命令のスケジューリングが制約される原因となります。Intel C/C++コンパイラ 4.0 の実装では、1 つのパラメータのアドレスが参照されると、ほかのパラメータもすべてエイリアスされるため、パラメータのアドレス参照は特に禁物です。

効率の良くないコード

```
void f(float *dst, float dscale, int n)
{
    m128 t1; int i;
    t1 = _mm_load_ps1(&dscale);

    for (i = 0; i < n; i++)
        *(__m128 *)dst = _mm_mul_ps(*(__m128
        *)dst, t1);
    dst += 4;
}
```

効率的なコード

```
void f(float *dst, float dscale, int n)
{
    m128 t1; int i;
    t1 = _mm_set_ps1(dscale);

    for (i = 0; i < n; i++)
        *(__m128 *)dst = _mm_mul_ps(*(__m128
        *)dst, t1);
    dst += 4;
}
```

- ループの条件式では、できる限り、動的に変化しない値を使用してください。それができない場合は、ループの条件式で参照する変数を、アドレスを参照する必要のないローカル変数だけに限定してください。この方針を守れば、ループの終了条件の判定のために不要な処理が発生するのを避けられます。

効率の良くないコード

```
int i, n;
get_bounds(&n);
/* このコード例では、ループが回転するたびに毎回、
   変数 n のロードと除算が必要になるため、
   パフォーマンスが低下します。*/
for (i = 0; i < n / 4; i++) { ... }
```

効率的なコード

```
int i, n, l_end;
get_bounds(&n);
l_end = n / 4;
for (i = 0; i < l_end; i++) { ... }
```

- 組み込み関数を使用するループでは、メモリへの値の書き込みがループの最後で実行されるようにコーディングし、中間的な計算結果はローカル変数に格納してください。この方法を採用すれば、スケジューラがコードの実行順序を自在に変更できるようになり、メモリ参照の回数を最小限に抑えることができます。ポインタを使用した参照を行うと、ポインタによるその他の参照がエイリアスされてしまうのは、C/C++言語の仕様による一般的な問題です。

効率の良くないコード

```
m128 *dst, *src, c; int i, n;
for (i = 0; i < n; i += 2)
    dst[i] = _mm_add_ps(src[i], c);
    dst[i+1] = _mm_add_ps(src[i+1], c);
}
```

効率的なコード

```
m128 *dst, *src, c, t1, t2; int i, n;
for (i = 0; i < n; i += 2)
    t1 = _mm_add_ps(src[i], c);
    t2 = _mm_add_ps(src[i+1], c);
    dst[i] = t1;
    dst[i+1] = t2;
}
```

5. ループ内では、以下の組み込み関数の使用は避けてください。

```
_mm_set_ps()
_mm_setr_ps()
_mm_set_ps1()
_mm_set_ss()
```

`_mm128` データ・タイプのデータの初期化に使用されるこれらの組み込み関数は、機械語命令に直接対応したものではありません。これらの組み込み関数は複数の機械語命令を使用して実装されているため、実行時の負荷が高くなる場合があります。これらの組み込み関数は、ループに入る前に、別の組み込み関数の戻り値を `_mm128` タイプのローカル変数にセットするために使用し、ループの中ではそのローカル変数の値を使用するのが最良の方法です。前ページの図 3 のコード例でも、この方法が使用されています。

6. 短いループで、パフォーマンス向上のためにアンロールすることが望ましいと思われる場合には、ソース・コードで明示的にアンロールしてください。ループ・アンロールは、ループ内で同じ処理を複数回実行することによりループの反復回数を減らす技法です。ループ・アンロールの詳細情報と使用例については、参考資料[8]を参照してください。

効率の良くないコード

```
m128 *a, *b, *c; int i;
for (i=0; i < 16; i++)
    a[i] = _mm_add_ps(b[i], c[i]);
}
```

効率的なコード

```
m128 *a, *b, *c, t1, t2; int i;
for (i=0; i < 16; i+=2)
/* This loop has been unrolled twice */
    t1 = _mm_add_ps(b[i], c[i]);
    t2 = _mm_add_ps(b[i+1], c[i+1]);
    a[i] = t1;
    a[i+1] = t2;
}
```

ベクトル・クラス

ベクトル・クラスは、C++のコードで組み込み関数を簡単かつ効率的に使用する手段を提供します。F32vec4 は、浮動小数点インターネット・ストリーミング SIMD 拡張命令を使用するために定義されたクラスです。また、I32vec2、I16vec4、I8vec8 の各クラスは、それぞれ、MMX[®]テクノロジーで使用される 3 種類のデータ・タイプ(char, short, int)に対応しています。これらの各クラスは、`_mm64` と `_mm128` のデータ・タイプと、そのデータ・タイプをサポートしている組み込み関数をクラスとして抽象化したものです。これらのクラスの実装は、それぞれ `ivec.h` (整数 SIMD)、`fvec.h` (浮動小数点 SIMD) ヘッダ・ファイルとして、Intel C/C++コンパイラとともに提供されています。各クラスのメンバ関数は、*、+、-、/、平方根、比較などの基本的な演算子をオーバーロードしたものです。必要に応じて、ユーザ側でこれらのクラスを再定義したり拡張することができます。

次の図 4 は、F32vec4 クラスを使用して、前ページの最初に示した 3 つ組データのループをエンコードした例です。

```
#define VECTOR_SIZE 4

__declspec(align(16)) float xa[ARRAY_SIZE],
    xb[ARRAY_SIZE], xc[ARRAY_SIZE];

float q;

void do_fvec_triad() {

    F32vec4 q_xmm = (q, q, q, q);

    F32vec4 *xa_xmm = (F32vec4 *) &xa;
    F32vec4 *xb_xmm = (F32vec4 *) &xb;
    F32vec4 *xc_xmm = (F32vec4 *) &xc;

    for (int j = 0;
         j < (ARRAY_SIZE/VECTOR_SIZE); j++) {
        xa_xmm[j] = xb_xmm[j] +
            q_xmm * xc_xmm[j];
    }
}
```

図 4:ベクトル・クラスを使用した、3 つ組データ・ループのエンコーディング

ベクトル・クラスを使用する場合は、以下の事項に注意してください。

- 各クラスでは、定数、変数、およびポインタで参照されるデータ(配列など)を、それぞれ SIMD クラス・オブジェクトに変換できるように、数種類のコンストラクタが定義されています。図 4 の例では、スカラの `q_xmm` に値をロードするために、ブロードキャスト(レプリケーション)コンストラクタを使用しています。また、コンストラクタの使用とメモリの参照を最小限に抑えるために、グローバル変数である 3 つの配列は、ループでの使用前に F32vec4 オブジェクトへのポインタにキャストしてあります。

2. float タイプの配列を *F32vec4* オブジェクトへのポインタ (*xa_xmm*, *xb_xmm*, *xc_xmm*) にキャストしているため、ループ内でのメモリ参照は、*__m128* タイプのデータを持つ *F32vec4* オブジェクトの配列に対する参照になっています。したがって、ループの 1 回転ごとに *F32vec4* オブジェクトを 1 つずつたどることになるので、ループ・インデックスは 1 ずつインクリメントする必要があります。一方、ループの終了条件式では、ループの 1 回転ごとに元の配列の要素がベクトル・サイズの個数だけ処理されることを考慮して、元の配列のサイズをベクトル・サイズで除算した値を使用しています。

ベクトル・クラスを使うと、非常に簡潔なコードで SIMD 命令を実装できます。ベクトル・クラスには一般的な演算子の大部分がオーバーロードされた形で含まれているため、プログラム自体をほとんど変更することなく、float データ・タイプを *F32vec4* クラスとして再定義するだけで、そのクラスが使用されるあらゆる場所でインターネット・ストリーミング SIMD 拡張命令の機能を活用できるようになります。

ベクトル化

Intel C/C++ コンパイラが SIMD のコーディングを支援するもう 1 つの方法は、ベクトル化によるものです。ベクトル化の機能を使用するように指示すると、コンパイラでは、指定されたループ内の配列操作に合わせて適切な SIMD コードを生成するようになります。実際にベクトル化を指定するには、C または C++ の *#pragma* ディレクティブ、コマンドライン・スイッチ、またはその両者の組み合わせを使用します。*#pragma* ディレクティブとコマンドライン・スイッチはどちらも種類が多いため、その詳細については、参考資料[2]の Intel C/C++ コンパイラ・ユーザズ・ガイドを参照してください。以下の図 5~図 7 では、一般的に最もよく使われる *#pragma* ディレクティブの使用例を示します。

```
#define VECTOR_SIZE 4
__declspec(align(16)) float xa[ARRAY_SIZE],
    xb[ARRAY_SIZE], xc[ARRAY_SIZE];
float q;

void do_vector_triad() {
    #pragma vector aligned
    for (int j = 0; j < ARRAY_SIZE; j++) {
        xa[j] = xb[j] + q * xc[j];
    }
}
```

図 5: コンパイラによる、3 つ組データ・ループのベクトル化

図 5 のコードは、*#pragma vector aligned* のみを使用した簡単なベクトル化の例です。*#pragma vector aligned* は、コンパイラにデータを正しくアライメントするように指示するディレクティブで、アライメント整合データ移動命令を使用できるようにする効果があります。このディレクティブ、または同等のコマンドライン・オプションを指定しない場合、コンパイラはアライメント不整合データ用の移動命令を生成すること

になり、アライメント整合データに対応した命令が使用できる場合に比べてパフォーマンスが大幅に低下します。

図 6 は、3 つ組データ・ループのコードを一部変更して、データをパラメータとして関数に渡すように変えたものです。

```
#define VECTOR_SIZE 4
__declspec(align(16)) float xa[ARRAY_SIZE],
    xb[ARRAY_SIZE], xc[ARRAY_SIZE];
float q;

void do_vector_triad(float *a,
    float *b,
    float *c) {
    #pragma vector aligned
    for (int j = 0; j < ARRAY_SIZE; j++) {
        a[j] = b[j] + q * c[j];
    }
}
```

図 6: ポインタを使用した 3 つ組データ・ループ

図 6 の例のように、ループのルーチンに配列を渡す方法は、ルーチンのベクトル化に大きな影響を与えます。これはコンパイラが、配列への単純な参照の代わりに、ポインタが指しているデータを扱うことになるからです。その結果、コンパイラは、ループの中で書き込まれたデータが次のループの実行時に使用されるといったメモリ参照の競合を想定しなければなくなり、SIMD 命令を使用してループを単純にエンコードすることは不可能になります。この問題は、ポインタがエイリアスされる可能性があることに起因しています。コンパイラはプログラムの整合性を保証するために、例えば、*xa* が *xb[1]* をポインタしていて、ループの反復によって *xa[j]* に順次格納されていく値が次のループの中で *xb[j]* として再利用される、という状態も想定する必要があります。

コンパイラがこのようなケースを考慮しないようにするには、新たに導入された *restrict* キーワードを使用します。このキーワードは、あるポインタが指しているデータが、現在のスコープ内ではそのポインタ変数以外からはアクセス不可能であることをコンパイラに対して明示するものです。図 7 に、このキーワードの使用例を示します。

```

#define VECTOR_SIZE 4
__declspec(align(16)) float xa[ARRAY_SIZE],
    xb[ARRAY_SIZE], xc[ARRAY_SIZE];
float q;

void do_vector_triad(float *restrict a,
    float *restrict b,
    float *restrict c) {
#pragma vector aligned
    for (int j = 0; j < ARRAY_SIZE; j++) {
        a[j] = b[j] + q * c[j];
    }
}

```

図 7: restrict キーワードの使用法

このように restrict キーワードを追加すれば、コンパイラは、各ポインタが別々の配列または別々のメモリ位置を指しているとして想定できるようになります。

ベクトル化に関する制約事項

ベクトル化の機能を使用する場合は、以下の制約事項に注意してください。

1. ループはカウント可能でなければなりません。言い換えれば、ループの反復カウンタはループの途中では変更できません。

正しいコード: `for (i=0; i<N; i++) ...`

```
while (i<100) { ... i = i + 2; ... }
```

不適切なコード: `while (p) { ... p=p->next ... }`

2. ループ内のコードは、1つの基本ブロックだけで構成されている必要があります。したがって、if 文や内部で分岐するコードは使用できません。また、ループは開始点と終了点をそれぞれ1つずつしか持てません。
3. サポートされているデータ・タイプは `float`、`char`、`short`、`int` です。ループ内ではこれらのデータ・タイプを混用しないでください。
4. インターネット・ストリーミング SIMD 拡張命令で使用されるデータは、ユーザが明示的にアライメントする必要があります。ループごとに `#pragmavector aligned` ディレクティブを追加するか、あるいは、コマンドライン・オプションの `-Qvec_alignment2` を指定して、ベクトル化できるすべてのデータを正しくアライメントするようにコンパイラに指示してください。ベクトル化できるデータがアライメントされていない(または、アライメントできない)場合、コンパイラは、アライメント不整合データ用の移動命令である `movups` を使用します。
5. ループ内でのデータ・アクセスは、データの1単位ごとに行う必要があります。つまり、1ずつインクリメントしながら連続的にアクセスしなければなりません。

6. スカラ・データへの代入は許されていません。スカラ・データを参照している変数は、必ず式の右辺になければなりません。
7. ループ内では関数を呼び出すことはできません。組み込み関数だけでなく、`sqrt()` や `cos()` などの外部関数もコールできません。

コンパイラ・スイッチの `-Qvec_verbose(0,1,2,3)` を指定すると、コンパイラは、ループがベクトル化されたかどうかを通知するメッセージを表示するようになります。ベクトル化が成功しなかった場合には、その理由も表示されます。表示されるメッセージの詳細レベルは、スイッチに付ける 0~3 の数字で指定します。上記のような制約があるため、ベクトル化できるループのタイプはある程度限られますが、このメッセージ表示のスイッチを指定すると、コンパイラは自動的にベクトル化を試行するようになります。この方法でベクトル化を成功させるには、メッセージの情報に基づいて、ループ・コードの修正や pragma の追加を行いながら、何度かコンパイルを繰り返す必要があるかもしれません。しかし、それでも、組み込み関数を使用してループを再コーディングするよりはずっと短時間で済む場合が多いでしょう。

パフォーマンスの比較

ここで説明したいいくつかのプログラミング方法の間には、パフォーマンスの差はほとんどありません。ベクトル・クラスを使用した実装では、C++ のオーバーヘッドのために、組み込み関数を使用した場合よりパフォーマンスがわずかに落ちる場合がありますが、それは全体から見れば稀なケースです。ここで紹介した各方法のパフォーマンスは、通常、手作業でコーディングされた最適なアセンブリ・コードのパフォーマンスよりは 10% ~ 15% 程度低くなります。しかし、コーディング、保守、移植作業の効率を考慮すれば、C や C++ を使用するプログラミング方法は、そのパフォーマンスの差を補って余りある優位性を備えていると言えます。

VTUNE™ 性能アナライザ

VTune™ 性能アナライザ (VTune アナライザ) は、アプリケーションのパフォーマンスをさまざまな方法で分析し、その結果をグラフや表などの形でグラフィカルに表示するツールです。また、アプリケーションのチューニングに役立つアドバイスを提示する機能も備えています。このセクションでは、VTune アナライザの主要機能について概説します。

イベント・ベース・サンプリング

イベント・ベース・サンプリングは、VTune アナライザによるアプリケーションのパフォーマンス分析で、最もよく使われる分析方法です。分析対象とするイベントは、プロセッサで発生する多数のイベントの中から任意に選択できます。イベントを選択することにより、アプリケーションによる CPU の使用状況を、クロックティック・イベントや CPU 時間、リタイアされた特定タイプの操作、発生したペナルティなど、特定の面に焦点を絞って調査できるようになります。

イベント・ベース・サンプリングを実行すると、アナライザは選択されたイベントが指定回数発生した時点でプロセッサ・イベントのサンプルを収集し、そのときのプログラム・カウンタのアドレスを記録します。そして、そのイベントが、ユーザ・プログラムまたはシステム上で実行されていたいずれかのプログラムのどの位置で発生したのかをユーザに報告します。

発生したイベントに関するこの情報は、モジュール・レポートとして表示されます。モジュール・レポートでは、サンプリング中にプロセッサを使用していたすべてのアプリケーションとモジュールについて、分析対象のイベントの発生回数が棒グラフで示されます。また、ホットスポット・レポートでは、特定のモジュールについて、モジュール・レポートと同様の情報がさらに詳しく表示されます。

インターネット・ストリーミング SIMD 拡張命令を使用したパフォーマンス・チューニングを行う場合は、*Clockticks* および *Floating-point operations retired* イベントを対象としてイベント・ベース・サンプリングを実行し、その結果のイベント比率の値を手掛かりにして、浮動小数点演算の実行に特に多くの時間を費やしたモジュールを調べます。この方法により、

チューニング対象の候補として、インターネット・ストリーミング SIMD 拡張命令によるパフォーマンス向上の効果が大きいと予想されるモジュールを選び出すことができます。

ホットスポット・レポートの棒グラフをダブルクリックすると、そのグラフが示しているイベントの発生位置のソース・コードが表示されます。図 8 は、ごく簡単なアプリケーションでのクロックティック・イベントのサンプリング結果を示すモジュール・レポートとホットスポット・レポートの画面です。また、図 9 は、ホットスポットのグラフをダブルクリックすると表示される、アプリケーションのソース・コード表示画面です。ソース・コードとともに、コードの各セクションでの *Clockticks* および *Floating-point operations retired* イベントの発生回数が表示されています。

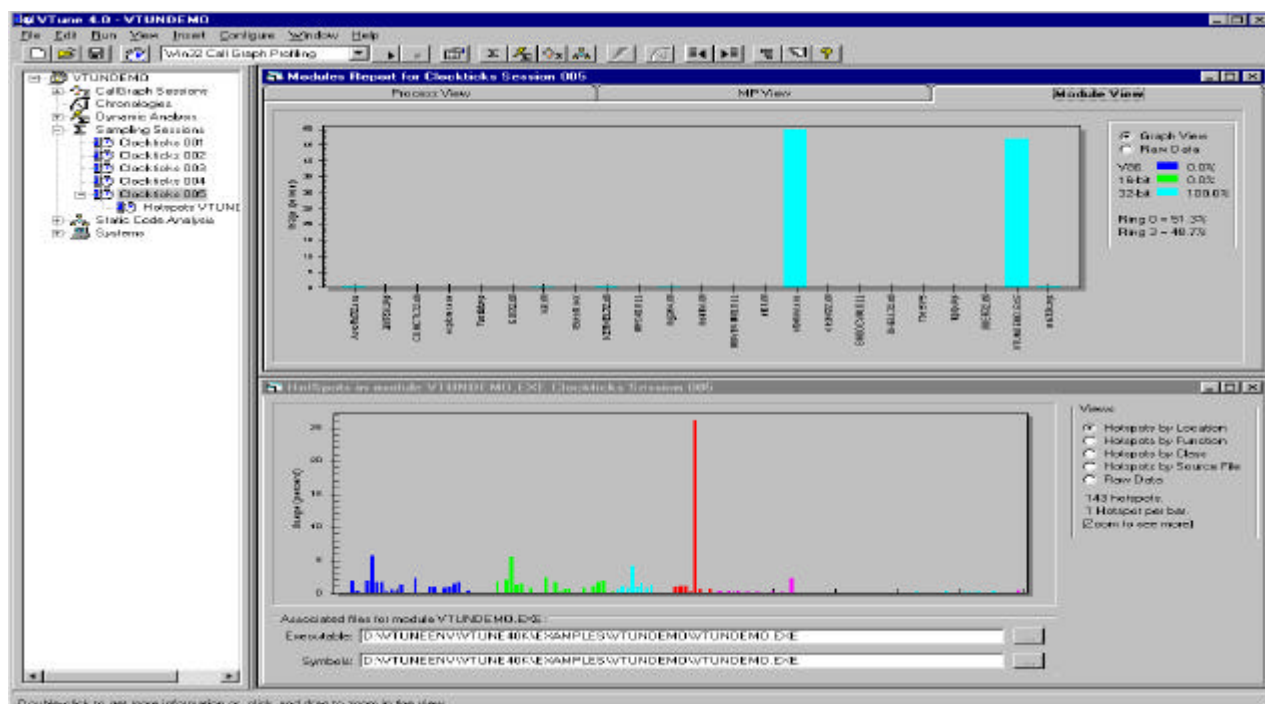


図 8:モジュール・レポートとホットスポット・レポート

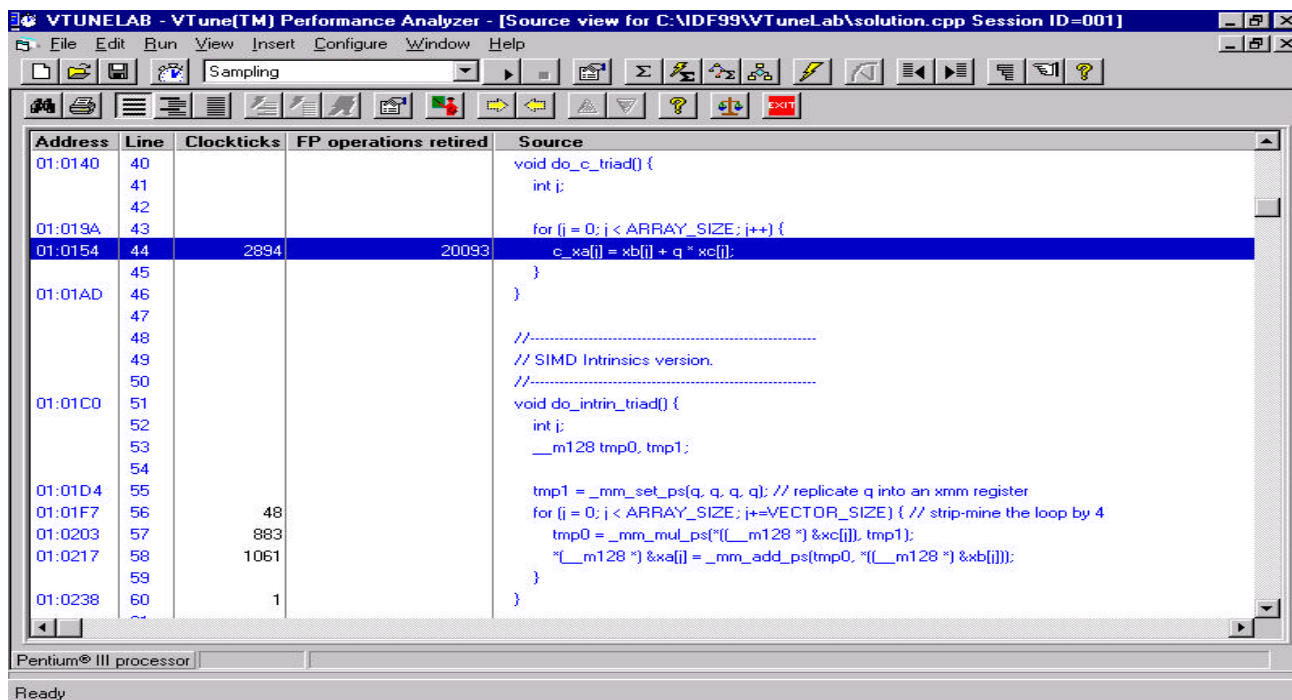


図9:ソース・コード表示ウィンドウ

コード・コーチ

VTune アナライザのコード・コーチは、コンパイラからの情報を使用してソース・コードを分析し、アプリケーションのチューニングに役立つアドバイスを表示します。表示されるアドバイスは、検索アルゴリズムを効率化するためのヒントから、不要なキャスト(データ・タイプの変換操作)に関する注意、インターネット・ストリーミング SIMD 拡張命令や MMX®テクノロジーを最大限に活用するための組み込み関数や

パフォーマンス・ライブラリ集の効果的な使用方法まで、多岐にわたります。

コード・コーチによるアドバイスを表示するには、ソース・コード表示ウィンドウの中で、アドバイスが必要なコードの文をダブルクリックします。図10の画面には、図1のコード例に含まれている `do_c_triad()` 関数についてのアドバイスが表示されています。

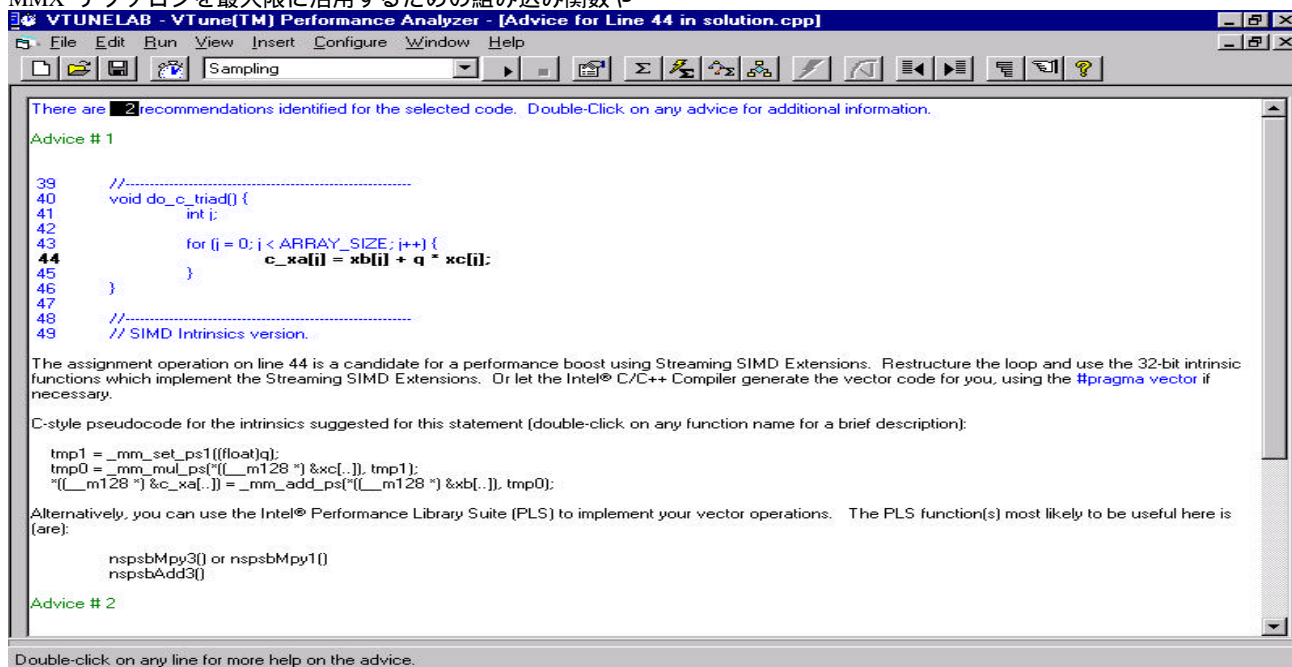


図10:コード・コーチによる、インターネット・ストリーミングSIMD 拡張命令に関するアドバイス

VTune™パフォーマンス拡張環境の利用による、Pentium® III プロセッサのインターネット・ストリーミングSIMD 拡張命令を活用するためのプログラミング手法

ダイナミック解析

VTune アナライザのダイナミック解析では、Pentium® II プロセッサや Pentium® III プロセッサの設計過程で実際に使用されたものと同じソフトウェア・シミュレータが使われます。ダイナミック解析は、イベント・ベース・サンプリングで見つかったホットスポットに関して、ペナルティやリタイアメント時間など、マイクロアーキテクチャに関わる特定の情報を

得るために有用です。また、分岐予測ミスやキャッシュ使用率を調査・分析する上でも非常に大きな力を発揮します。

コードの一部を分析対象として選択し、部分的にシミュレーションを実行するのが、ダイナミック解析の基本的な使い方です。図 11 の画面には、図 3 のコード例の `do_intrin_triad()`関数をダイナミック解析した結果が表示されています。

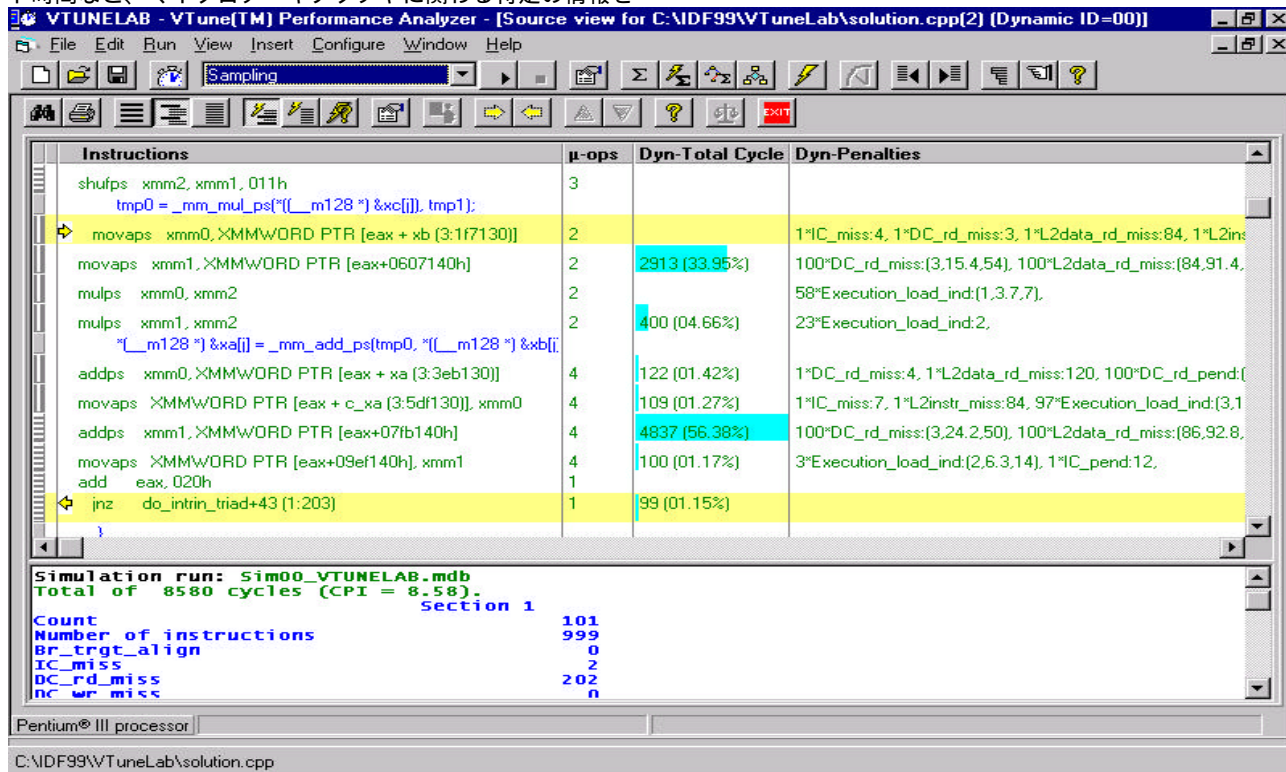


図 11:ダイナミック解析の結果表示

図 11 のダイナミック解析の結果例は、キャッシュ・ミスの頻発がパフォーマンスの低下を招いている可能性が高いことを示しています。このことから、ループのパフォーマンスを高めるには、プリフェッチ命令、またはインターネット・ストリーミング SIMD 拡張命令に含まれるストリーミング・ストア命令が有効であることがわかります。

まとめ

この記事では、Pentium® III プロセッサのインターネット・ストリーミング SIMD 拡張命令が持つ、優れたパフォーマンスを最大限に活用するための特別なプログラミング手法をいくつか紹介しました。どの方法も、パフォーマンス的には手作業でコーディングした最適なアセンブリ・コードと大差なく、しかも、開発コストはアセンブリ・プログラミングの場合よりもかなり低い水準に抑えられます。VTune™パフォーマンス拡張環境には、ここで取り上げたツールに加えて、Intel® パフォーマンス・ライブラリ集、Intel® アーキテクチャ・パフォーマンス・トレーニング・センター(詳しくは参考資料[7]をご覧ください)、レジスタ・ビューイング・ツールが含まれています。どのツールも、さらなるパフォーマンスの向上と、インターネット・ストリーミング SIMD 拡張命令

の使用に役立つものばかりです。これらの優れたツール群を備えた VTune パフォーマンス拡張環境 4.0 は、インターネット・ストリーミング SIMD 拡張命令の活用には不可欠であり、決定版と言うにふさわしい開発環境です。

参考資料

記事の中で参考資料として挙げられているドキュメントは次のとおりです。これらの参考資料には、記事内容の理解に役立つ関連情報が含まれています。

1. AP-589 ストリーミング SIMD 拡張命令のソフトウェア規則(日本語 PDF ファイル: 61KB)
2. Intel® C/C++ コンパイラ V4.0(ストリーミング SIMD 拡張命令対応): ユーザーズ・ガイド(日本語 PDF ファイル: 1,417KB)
3. C++ SIMD 命令 クラス・ライブラリ: リファレンス・マニュアル(日本語 PDF ファイル: 612KB)
4. AP-814 ストリーミング SIMD 拡張命令テクノロジーのソフトウェア開発戦略(日本語 PDF ファイル: 42KB)

VTune™パフォーマンス拡張環境の利用による、Pentium® III プロセッサのインターネット・ストリーミング SIMD 拡張命令を活用するためのプログラミング手法

5. AP-833 Intel® C/C++コンパイラを使用したストリーミング SIMD 拡張命令のデータ・アライメントとプログラミングの問題(日本語 PDF ファイル: 39KB)
6. インテル・アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル、中巻: 命令セット・リファレンス(日本語 PDF ファイル: 6,873KB)
7. インテル・アーキテクチャ・パフォーマンス・トレーニング・センター
<http://www.intel.co.jp/jp/developer/vtune/cbts/index.htm>
8. インテル・アーキテクチャ最適化リファレンス・マニュアル(日本語 PDF ファイル: 2,934KB)

著者紹介

Joe Wolf、マイクロプロセッサ・プロダクツ・グループ(MPG)、プラットフォーム・ツール・オペレーション所属スタッフ・ソフトウェア・エンジニア。1996年にインテルに入社し、コンパイラ開発、テクニカル・マーケティング、カスタマ・サポートを担当。以前は、スーパーコンピュータ業界のコンパイラ開発者として9年間、主にベクトル、多重処理、コード生成に関連する開発に携わる。1984年にアリゾナ大学で経営情報システム学の理学士号を取得。1987年にカリフォルニア州立科学技術大学でコンピュータ・サイエンスの理学修士号を取得。

電子メール・アドレス: joe.wolf@intel.com