インテル® C++ コンパイラー のベクトル化ガイド

目次

注: 本ガイドの Visual Studio* の利用説明では、Visual Studio* 2008 を想定しています。他のバージョンでは異なることがあります。また、サンプルコードのコンパイルには、インテル® コンパイラー バージョン 12.0 のコンパイラーを利用していますが、最新版では結果が異なるものがあります。

1. はじめに

このガイドは、インテル® C++ コンパイラーのベクトル化機能に関するガイドラインを提供します。インテル® ストリーミング SIMD 拡張命令 (インテル® SSE) などの SIMD 命令に対応したインテル® プロセッサー、または互換性のあるインテル以外のプロセッサー・ベースのシステムを使用する C/C++ 開発者を対象としています。インテル® 64 およびほとんどの IA-32 システムが対象となりますが、インテル® Itanium® プロセッサー・ベースのシステムは対象外です。ここで紹介する例ではインテル® SSE を使用しますが、多くの原則はほかの SIMD 命令セットにも適用されます。また、例では C++ プログラムを使用しますが、その概念のほとんどは、Fortran プログラムにもあてはまります。

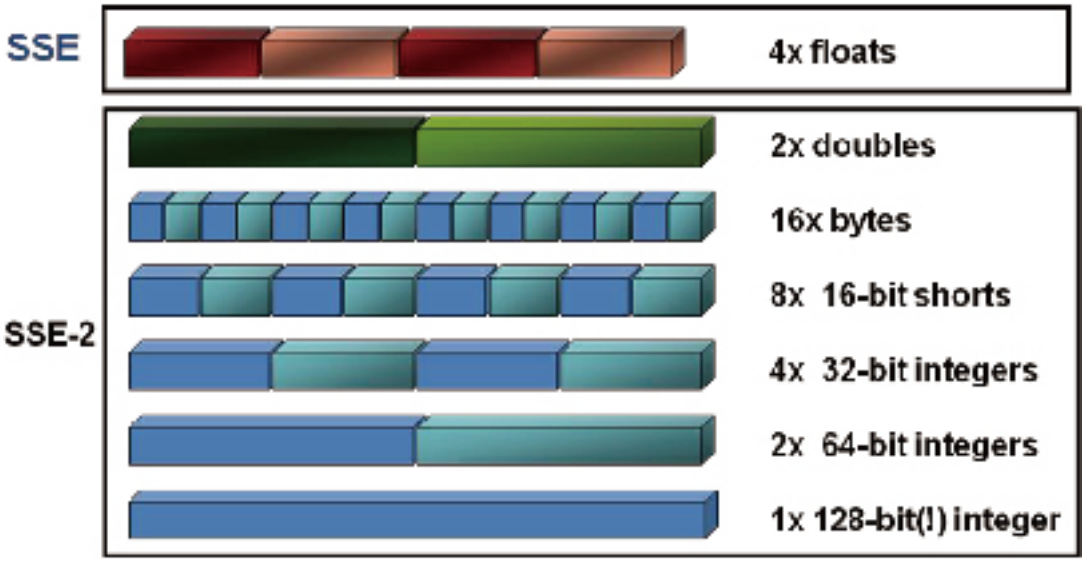
2. インテル® コンパイラーのベクトル化とは?

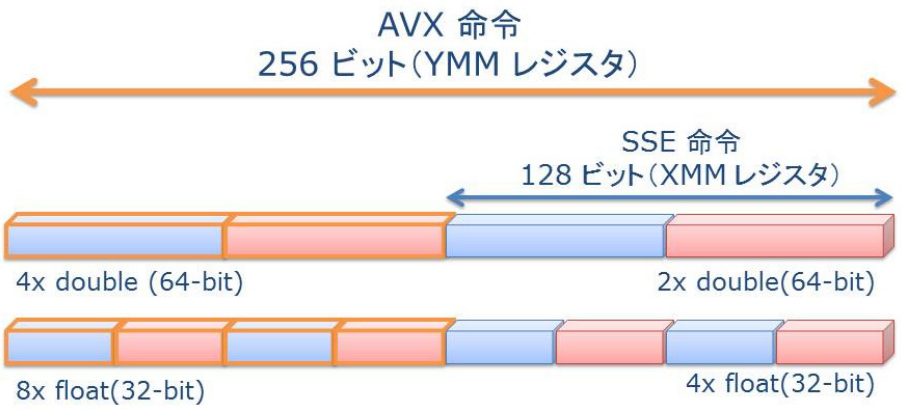
ここでは、コンパイラーによるパックド SIMD 命令の生成を伴うループアンロールを指します。パックド命令は一度に複数のデータ要素の演算を行うため、ループをより効率良く実行できます。このプロセスは、開発者が特別な処理をすることなく、コンパイラーが適したループを自動的に識別し最適化するため、自動ベクトル化と呼ばれることがあります。

この概念に精通している場合は、この節の技術背景をスキップして、次の「2.1 どのような場合にコンパイラーはベクトル化するのか?」へ進んでください。

現在のプロセッサーは、幾つかの異なる並列性レベルを備えた高度な並列プロセッサーです。プロセッサー・コアの並列実行ユニットから SIMD (Single Instruction, Multiple Data) 命令セットや複数のスレッドの並列実行まで、あらゆるところで並列化が採用されています。x86 アーキテクチャーの拡張であるインテル® SSE 命令セットを使用することをベクトル化と呼びます。コンピューター・サイエンスでは、スカラー実装 (1つの命令がオペランドの 1 ペアに対する演算を行う) からベクトル処理 (1 つの命令がベクトル “一連の隣接した値” を参照できる) にアルゴリズムを変換するプロセスをベクトル化と呼びます。SIMD 命令は複数のデータ要素を 1 つの命令で演算し、128 ビット・レジスターを使用します。第二世代インテル® コア・マイクロアーキテクチャー Sandy Bridge で追加された、インテル® AVX では 1 つの浮動所数点命令で、256 ビット・レジスターを使用します。

インテルは、SSE の実装において最初に 8 つの 128 ビット・レジスター (XMM0 から XMM7) を追加し、その後、64 ビットに対応するため、さらに 8 つのレジスター (XMM8 から XMM15) を追加しました。第二世代インテル® コア・マイクロアーキテクチャー Sandy Bridge では、前述の 16 個のレジスターが 256 ビットに拡張され、YMM0 - YMM15 という名称でアクセスできます。XMM0 – XMM15 という名称を使用すると、下位 128 ビットにアクセスできます。開発者は、ベクトル化を利用してコードの特定の部分をスピードアップすることができます。ベクトルコードの記述にはその分時間がかかりますが、大幅なパフォーマンスが見込めるので、それだけの価値はあるでしょう。コンピューター・サイエンスの主要な研究テーマに、自動ベクトル化方法の模索があります。つまり、人的介入なしにコンパイラーによってスカラー・アルゴリズムをベクトル・アルゴリズムに変換する手法を探す研究です。インテル® コンパイラーは自動でインテル® SSE 命令やインテル® AVX 命令を生成することができます。このガイドは、インテル® コンパイラーを使用して自動で SIMD コードを生成することに焦点を当てています (これ以降、この機能を自動ベクトル化と呼びます)。コンパイラーによる自動ベクトル化を支援するガイドラインも紹介します。場合によっては、コードに特定のキーワードや宣言子を追加して、自動ベクトル化が行われるようにします。

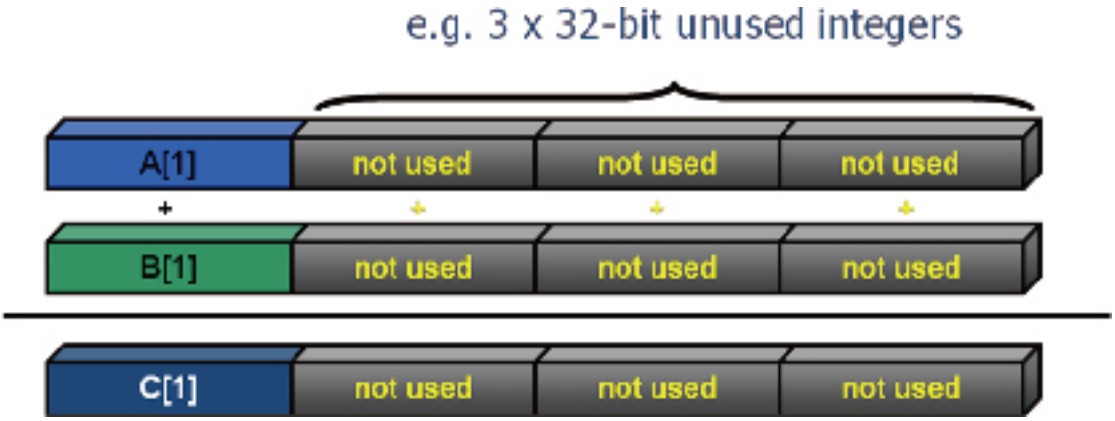




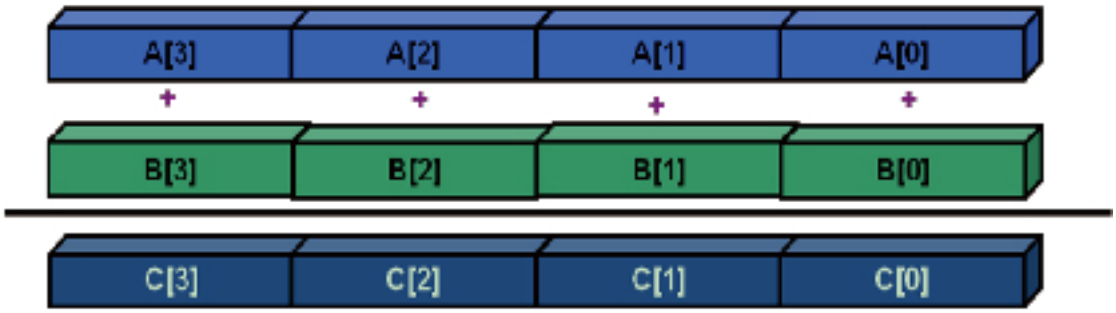
ベクトル化によるスピードアップはどのようにもたらされるでしょうか?
次に示すコード例について考えてみましょう。a、b、c は整数配列です。

```
for(i=0;i<=MAX;i++)
   c[i]=a[i]+b[i];
```

ベクトル化が有効でない場合 (例えば /Od、/O1、または /Qvec- を使用する場合)、コンパイラーは次のように処理します。



SIMD レジスターに未使用の空間があり、さらに 3 つの整数を格納することができます。ベクトル化が有効な場合、コンパイラーは SIMD レジスターの未使用の空間を活用して、1 つの命令で 4 つの加算を実行します。



2.1 どのような場合にコンパイラーはベクトル化するのか?

デフォルトの最適化オプション (/O2) 以上でコンパイルすると、コンパイラーは常にベクトル化の可能性を探します。これらのオプションはインテル製マイクロプロセッサーおよび互換マイクロプロセッサーで利用可能ですが、インテル製マイクロプロセッサーにおいてより多くの最適化が行われる場合があります。ベクトル化されたコードとそうでないコードを比較するため、/Qvec- (Windows*) または -no-vec (Linux* または Mac OS* X) オプションを使用してベクトル化を無効にできます。/O1 以下では、コンパイラーはベクトル化しません。

2.2 ループがベクトル化されたかどうかを確認するには?

/Qvec-report (Windows*) または –vec-report (Linux* または Mac OS* X) オプションでベクトル化レポート (「6. ベクトル化レポート」節で詳しく説明) を有効にすると、次のように、ベクトル化された各ループに対してメッセージが出力されます。

```
    > icl /Qvec-report MultArray.c
    MultArray.c(92):(列 5) リマーク:ループがベクトル化されました。
```

ソースの行番号 (上記の例では 92) はループの開始を指します。Visual Studio 環境では、[プロジェクト] > [プロパティ] > [C/C++] > [診断] > [最適化診断レベル] で [最小 (/Qopt-report:1)] を、[最適化診断フェーズ] で [ハイパフォーマンス・オプティマイザー・フェーズ(/Qopt-report-phase:hpo)] を選択します。

[ベクトライザー診断レベル] で、/Qvec-report2 (-vec-report2) を選択すると、ベクトル化されなかった各ループに対する診断メッセージと、ベクトル化されなかった理由も確認できます。

2.3 ベクトル化の影響は?

一般に、ベクトル化はループのパフォーマンスを改善します。この点について検証してみましょう。まず、インテル® コンパイラーのコマンドライン・ウィンドウを開きます。例えば、Windows* の [スタート] メニューから [すべてのプログラム] > [Intel Parallel Studio XE 2011] > [Command prompt] > [インテル(R) コンパイラー XE 12.1] > [インテル(R)64 Visual Studio 2008 モード] を選択します。Linux* または Mac OS* X の場合は、製品のリリースノートで説明されているように、使用しているコンパイラー・バージョンの bin/intel64 または bin/ia32 ディレクトリーにある初期化スクリプトを source で読み込んでください。

(例題1) ここで使用するコードは、インテル® コンパイラーのサンプルコードの vec_samples に含まれます。この小さなプログラムは、次のループを使用してベクトルに行列を掛けます。

```
   for (j = 0;j < size2; j++) {
     b[i] += a[i][j] * x[j];
   }
```

Windows* システムで次のコマンドを使用してアプリケーションをビルドし、実行します。

```
icl /O2 /Qvec- /Qipo Driver.c Multiply.c /FeNoVectMult

¥> NoVectMult

icl /O2 /Qvec-report /Qipo Driver.c Multiply.c /FeVectMult

¥> VectMult
```

Linux* または Mac OS* X では、相当する次のコマンドを使用します。

```
icc -O2 -no-vec -ipo Driver.c Multiply.c -o NoVectMult

$ ./NoVectMult

icc -O2 -vec-report -ipo Driver.c Multiply.c -o VectMult

$ ./VectMult
```

2 つのバージョンの実行時間を比較してみてください。ベクトル化されたバージョンのほうが高速でしょう。

3. どのようなループがベクトル化されるのか?

ベクトル化の対象となるためには、ループは次の条件を満たしていなければなりません。

3.1. 可算ループ

ループの反復数は、ランタイムにループの入口で判明していなければなりません。ただし、コンパイル時に判明している必要はありません。(つまり、反復数は変数でもかまいませんが、ループの実行中は一定である必要があります。) これは、ループの出口はデータに依存してはならないことを意味します。

3.2. 1 つの入口と 1 つの出口

これは、可算ループであるための条件です。データ依存性を持つ出口があるために、ベクトル化できない次のループの例について考えてみましょう。

```
   void no_vec(float a[], float b[], float c[])
   {
        int i = 0.;
        while (i < 100) {
          a[i] = b[i] * c[i];
   // データ依存性を持つ出口条件
          if (a[i] < 0.0)
             break;
          ++i;
        }
   }

> icl /c /O2 /Qvec-report2 two_exits.cpp
two_exits.cpp(4) (列 9): リマーク: ループはベクトル化されませんでした: 非標準のループはベクトル化候補ではありません。
```

3.3. 直列型コード

SIMD 命令は、オリジナルループと同じ演算を実行するため、反復ごとに異なる制御フローを持つことはできません。つまり、分岐してはなりません。switch 文も使用できません。ただし、if 文はマスクされた代入として実装できる場合のみ使用できます (通常は使用可能)。演算はすべてのデータ要素に対して実行されますが、結果はマスクが TRUE に評価された要素のみ格納されます。次の例はベクトル化されます。

```
#include <math.h>
void quad(int length, float *a, float *b,
    float *c, float *restrict x1, float *restrict x2)
{
  for (int i=0; i<length; i++) {
    float s = b[i]*b[i] - 4*a[i]*c[i];
    if ( s >= 0 ) {
       s = sqrt(s) ;
       x2[i] = (-b[i]+s)/(2.*a[i]);
       x1[i] = (-b[i]-s)/(2.*a[i]);
    }
    else {
       x2[i] = 0.;
       x1[i] = 0.;
    }
```

```
  }
}
> icl /c /Qrestrict /Qvec-report2 quad.cpp
   quad5.cpp(5) (列 3): リマーク:ループがベクトル化されました。
```

3.4. 入れ子の最内ループ

通常ベクトル化は最内ループに対して行われます。唯一の例外は、ループアンロール、ループコラプス、ループ交換など、優先されるその他の最適化フェーズの結果として、オリジナルの外側のループが内側のループに変換される場合です。

3.5. 関数呼び出しがない

関数呼び出しを含むループはベクトル化されません。例外として、算術組込み関数とインライン展開可能な関数です。print 文でさえもループのベクトル化の妨げになります。通常、ベクトル化レポートには「非標準のループはベクトル化候補ではありません」と出力されます。

sin()、log()、fmax() などの算術組込み関数は、コンパイラーのランタイム・ライブラリーにベクトル化されたバージョン(ベクター・マス・ライブラリー)が含まれているため、利用できます。利用可能な関数を表 1 にリストします。これらの多くには float と double の両バージョンがあります。

表 1: コンパイラーにベクトル化されたバージョンがある関数

acos	ceil	fabs	round
acosh	Cos	floor	sin
asin	Cosh	fmax	sinh
asinh	erf	fmin	sqrt
atan	Erfc	log	tan
atan2	Erfinv	log10	tanh
atanh	Exp	log2	trunc
cbrt	exp2	pow	

次の例のループは、sqrtf() がベクトル化可能で、func() がインライン展開されるため、ベクトル化できます。デフォルトの最適化により、同じソースファイル内のすべての関数のインライン展開が有効になります。(インライン展開レポートを出力するには /Qopt-report-phase ipo_inl (Windows*) または -opt-report-phase ipo_inl (Linux* または Mac OS* X) オプションを指定します。)

```
float func(float x, float y, float xp, float yp) {
  float denom;
  denom = (x-xp)*(x-xp) + (y-yp)*(y-yp);
  denom = 1./sqrtf(denom);
  return denom;
}

float trap_int
    (float y, float x0, float xn, int nx, float xp, float yp) {

float x, h, sumx;
int i;
```

```
  h = (xn-x0) / nx;
  sumx = 0.5*( func(x0,y,xp,yp) + func(xn,y,xp,yp) );

  for (i=1;i<nx;i++) {
    x = x0 + i*h;
    sumx = sumx + func(x,y,xp,yp);
  }
  sumx = sumx * h;

return sumx;
}
> icl /c /Qvec-report2 trap_integ.c
trap_int.c(16) (列 3): リマーク:ループがベクトル化されました。
```

4. ベクトル化を妨げる要因

次に示す状況が必ずしもベクトル化を妨げるわけではありませんが、コンパイラーがベクトル化のメリットがないと判断する原因になります。

4.1 連続していないメモリーアクセス

4 つの連続するint 型またはfloat 型、あるいは 2 つの連続する double 型は、1 つの SSE 命令でメモリーから直接ロードされます。しかし、4 つの整数が隣接していなければ、複数の命令を使って別々にロードしなければならず、非効率的です。連続していないメモリーアクセスの最も一般的な例は、次に示すような非ユニットストライドのループや間接アドレス指定のループです。連続していないメモリーアクセスのオーバーヘッドと比較して計算量が多くない限り、コンパイラーはそのようなループをほとんどベクトル化しません。

```
  // ストライド 2 で配列にアクセス
     for (int i=0; i<SIZE; i+=2)  b[i] += a[i] * x[i];

  // ストライド SIZE で内側のループにアクセス
     for (int j=0; j<SIZE; j++)  {
        for (int i=0; i<SIZE; i++)   b[i] += a[i][j] * x[j];
     }

  // 配列 x への間接アドレス指定
     for (int i=0; i<SIZE; i+=2)  b[i] += a[i] * x[index[i]];
```

通常、ベクトル化レポートには次のメッセージが出力されます。

"ベクトル化は可能ですが非効率です"

間接アドレス指定に対しては、次のメッセージが出力されることもあります。

"ベクトル依存関係が存在しています"

4.2 データ依存

各 SIMD 命令は複数のデータ要素を一度に演算するため、ベクトル化ではループ内の演算順序が換わります。この順序の変更が演算結果に影響しない場合にのみ、ベクトル化が可能です。

- 最も単純なケースは、ループで書き込まれる (格納される) データ要素がそのループのほかの反復では使用されない場合です。この場合は、オリジナルのループのすべての反復が互いに独立しており、結果を変更することなくどの順序でも実行できます。このようなループは、ベクトル化を含むあらゆる並列手法で安全に実行することできます。これまでに示した例はすべてこのカテゴリーに分類されます。

- **リードアフターライト:** 変数が 1 つの反復で書き込まれ、後に続く反復で読み取られる場合は、"リードアフターライト" の依存性 (フロー依存性とも呼ばれる) があります。次に例を示します。

```
A[0]=0;

for (j=1; j<MAX; j++)  A[j]=A[j-1]+1;

// これは、次のコードと同じ

A[1]=A[0]+1;   A[2]=A[1]+1;   A[3]=A[2]+1; A[4]=A[3]+1;
```

上記のループは、最初の 2 つの反復が同時に実行されると、最初の反復による演算前に A[1] の値が 2 つ目の反復で使用され、正しくない結果を招く可能性があるため、安全にベクトル化できません。

- **ライトアフターリード:**変数が 1 つの反復で読み取られ、後に続く反復で書き込まれる場合は、"ライトアフターリード" の依存性 (アンチ依存とも呼ばれる) があります。次に例を示します。

```
for (j=1; j<MAX; j++)  A[j-1]=A[j]+1;

// これは、次のコードと同じ

A[0]=A[1]+1;   A[1]=A[2]+1;   A[2]=A[3]+1; A[3]=A[4]+1;
```

この場合、書き込みの反復が、読み取りの反復前に実行される可能性があるため、一般的な並列実行では安全ではありません。しかし、この例では j の値が大きい反復が小さい反復よりも前に完了することはありません。そのため、この場合は安全にベクトル化できます (例えば、非ベクトル化コードと同じ結果がもたらされるなど)。ただし、次の例では、ベクトル化により、A の要素が 2 つ目の反復で使用される前に最初の反復で上書きされ可能性があるため、安全ではありません。

```
for (j=1; j<MAX; j++)  {
    A[j-1]=A[j]+1;
    B[j]=A[j]*2;
}
// これは、次のコードと同じ
A[0]=A[1]+1;   A[1]=A[2]+1;   A[2]=A[3]+1; A[3]=A[4]+1;
```

- **リードアフターリード:** リードアフターリードは、実際には依存ではなく、ベクトル化や並列実行を妨げるわけではありません。変数が書き込まれなければ、いくら読み取られても関係ありません。

- **ライトアフターライト:** 同じ変数が複数の反復で書き込まれるライトアフターライト、または「出力」依存性は、一般にベクトル化を含む並列実行で安全ではありません。

しかし、上記の依存性タイプすべてを含む重要な例外があります。

```
sum=0;
for (j=1; j<MAX; j++)  sum = sum + A[j]*B[j]
```

sum は各反復で読み取りと書き込みの両方が行われますが、コンパイラーはこのようなリダクション・スタイルを認識し、安全にベクトル化することができます。2.3 節の例題 1 で示されるループは、スカラーの代わりにループ不変配列要素を持つリダクションの別の例です。

このようなループ反復間の依存性は、ループ伝搬の依存としても知られています。

上記の例では依存性が証明されています。しかし、コンパイラーは依存性の可能性があると、安全にループをベクトル化できません。

```
for (i = 0; i < size; i++) {
  c[i] = a[i] * b[i];
}
```

上記の例では、コンパイラーは、反復 i で c[i] が別の反復の a[i] または b[i] と同じメモリー位置を参照するかどうかを判別しなければなりません (そのようなメモリー位置は “エイリアス” と呼ばれます)。例えば、a[i] が c[i-1] と同じメモリー位置を指した場合、前述したようにリードアフターライトの依存性があります。この可能性をコンパイラーが排除できない場合、(少なくとも、「6.2 コンパイラーのベクトル化の支援」節で説明するように #pragma ivdep と restrict キーワードを使用しない限り) ループはベクトル化されません。

5．ベクトル化可能なコードを記述するガイドライン

5.1 ガイドライン

以下に、コンパイラーが効率良くコードをベクトル化できるようにするガイドラインを示します。

- できる限り、入口と出口が 1 つずつの可算 “for” ループを使用します。複雑なループの終了条件は避けてください。反復の下限と上限はループ内で不変でなければなりません。ループの入れ子の最内ループは、外側のループ・インデックスの関数にすることができます。

- 分かりやすいコードを記述します (switch、goto または return などの分岐文、関数呼び出し、マスクされた代入として扱えない if 構造は避けてください)。

- ループ反復間の依存性は避けてください。少なくとも、リードアフターライトの依存性は避けてください。

- できる限り、ポインターではなく配列表記を使用します。特に C プログラムはポインターの使用についてほとんど制限がありません。エイリアスされたポインターは予期しない依存性を招きます。何のヒントもないと、コンパイラーは、ポインターが含まれるコードを安全にベクトル化できるかどうかを判断できません。

- 可能な場合は、別のカウンターをインクリメントして配列アドレスとして使用する代わりに、直接ループ・インデックスを配列インデックスに使用してください。

- 効率的なメモリーアクセスを使用します。

    - ユニットストライドを使用する内側ループを優先する

    - 間接アドレス指定を最小限にする

    - インテル® SSE 命令を利用する場合、データを 16 バイト境界にアライメントする

    - インテル® アドバンスト・ベクトル・エクステンション (インテル® AVX) 命令を利用する場合、データを 32 バイト境界にアライメントする

最適なデータレイアウトを選択してください。多くのマルチメディア拡張命令セットはアライメントの影響を受けます。例えば、SSE/SSE2/SSE3/SSSE3/SSE4.1/SSE4.2 のデータ移動命令は、16 バイト境界にアライメントされているデータをより効率良く実行できます。そのため、ベクトル化の成功は、(ループピーリングのような) コード再構築と併用して、適切なデータレイアウトを選択できるかどうかにも依存します。これにより、プログラム全体でアライメントされたメモリーへのアクセスが可能になります。インテル® AVX 命令セット向けにコンパイルする場合は、データを 32 バイト境界にアライメントしてください。これにより、パフォーマンスが向上することがあります。

5.2 アライメントされたデータ構造の使用

データ構造のアライメントとは、他のオブジェクトと関連したデータ・オブジェクトを調整することです。インテル® コンパイラーは高速なメモリーアクセスのため、各変数が特定のアドレスで始まるようにアライメントします。ターゲット・プロセッサーでアライメントされていないメモリーへのアクセスがサポートされていない場合、大幅なパフォーマンスの低下を招きます。アライメントとはメモリーアドレスのプロパティーの 1 つで、数値アドレスモジュロ 2 の累乗 (数値アドレスを 2 のべき乗で除算した剰余) で表します。各データにはアドレスに加えて、サイズも設定されています。

データのサイズに合わせてアドレスがアライメントされている場合、そのデータは「自然にアライメントされている」といえます。そうでない場合は「アライメントされていない」といえます。例えば、8 バイトの倍精度浮動小数点データのアドレスが 8 にアライメントされている場合、このデータは自然にアライメントされています。データ構造は、データを効率良く使用できるようにコンピューターに格納する方法です。適切に設計されたデータ構造では、より効率良いアルゴリズムを可能にします。適切なデータ構造により、さまざまな操作が最小のリソース (実行時間とメモリー空間の両方) で実行できます。

```
struct MyData{
       short   Data1;
       short   Data2;
       short   Data3;};
```

上記のデータ構造では、***short*** 型が 2 バイトのメモリーに格納されるため、データ構造の各メンバーは 2 バイト境界にアライメントされます。Data1 はオフセット 0、Data2 はオフセット 2、Data3 はオフセット4 です。この構造のサイズは 6 バイトです。構造体の各メンバーのデータ型にはアライメント要件があります。つまり、開発者が要求しない限り、あらかじめ決定された境界にアライメントされます。コンパイラーが次善のアライメントを採用しても、開発者は declspec(align(base,offset)) 宣言を使用して、特定の base からの offset にデータ構造を割り当てることができます (0 <= offset < base で、base は 2 の累乗)。

プログラムの実行時間の大部分が費やされている次のようなループの例について考えてみましょう。

```
double a[N], b[N];
...
for (i = 0; i < N; i++){
  a[i+1] = b[i] * 3;
}
```

両方の配列の最初の要素が 16 バイト境界でアライメントされる場合、ベクトル化によりアライメントの合っていない b からの要素のロードか、a への要素のストアのいずれかが行われます。(この場合、反復を小さくしても意味がありません。) しかし、以下のようにアライメントを指定することで、ベクトル化した後に双方ともアライメントの合ったアクセスパターンになります (8 バイト・サイズの double の場合)。

```
_declspec(align(16, 8))  double a[N];
_declspec(align(16, 0))  double b[N];
/* または、ただ単に "align(16)" を使用する */
```

ポインター変数が使用されると、コンパイラーはコンパイル時にアクセスパターンのアライメントを判断できません。次のような fill() 関数について考えてみましょう。

```
void fill(char *x) {
  int i;
  for (i = 0; i < 1024; i++){
      x[i] = 1;
  }
}
```

追加のヒントがない限り、コンパイラーは上記のループでアクセスされるメモリー領域のアライメントについて想定できません。この時点で、コンパイラーはアライメントされていないデータの移動命令によってこのループをベクトル化することを決定するか、またはインテル® C/C++ コンパイラーのように、ランタイムのアライメント最適化を行います。

```
peel = x & 0x0f;
if (peel != 0) {
  peel = 16 - peel;
  /* ランタイム・ピーリング・ループ*/
  for (i = 0; i < peel; i++){
      x[i] = 1;
  }
}
/*アライメントされたアクセス */
for (i = peel; i < 1024; i++){
      x[i] = 1;
}
```

ランタイムの最適化では、コードサイズとテストがやや増えますが、一般にアライメントの合ったアクセスパターンを得る効率良い方法が選択されます。ただし、アクセスパターンが 16 バイト境界にアライメントされていることが事前に分かっていれば、__assume_aligned(x, 16) によるヒントを使用してその情報をコンパイラーに伝え、オーバーヘッドを回避することができます。ヒントは注意して使用してください。アライメントされたデータの移動を誤って使用すると、SSE 命令では例外が発生します。

5.3 構造体配列 (AoS) ではなく配列構造体 (SoA) を使用

一般的によく知られているデータ構造は、インデックスでアクセス可能な、連続したデータ項目の集合を含む配列で、構造体配列 (AoS)、または配列構造体 (SoA) として定義することができます。AoS はカプセル化には最適ですが、ベクトル化を妨げることがあります。

適切なデータ構造を定義することで、ベクトル化された効率良いコードを生成することができます。これを理解する例として、3 次元ポイントのセットの r、g、b コンポーネントを格納する従来の構造体配列 (AoS) と、配列構造体 (SoA) を比較してみましょう。

Point 構造体 (AoS):

```
struct Point{ //AoS
    float r;
    float g;
    float b;
}
```

Points 構造体 (SoA):

```
struct Points{ //SoA
    float* r;
    float* g;
    float* b;
}
```

AoS では、1 つの RGB ポイントのすべての要素にアクセスするようなループは、フェッチされたキャッシュラインにある要素を使用できるため、高い局所性が得られます。AoS の短所は、そのようなループの各メモリー参照は非ユニットストライドとなることです。これは一般に、ベクトル・パフォーマンスに不利に働きます。さらに、1 つの要素にのみアクセスするループは、フェッチされたキャッシュラインの多くの要素が再利用されないため、局所性が低下します。逆に SoA では、ユニットストライドなメモリー参照は、ベクトル化の恩恵がより受けられ、また 3 つのデータストリームのそれぞれで、高い局所性をもたらします。したがって、コンパイラーがベクトル化を行う場合、SoA を使用するアプリケーションは、AoS ベースのアプリケーションよりパフォーマンスが向上します (ただし、このパフォーマンスの差は実装の早い段階では明白でないことがあります)。

ベクトル化を開始する前に、次のガイドラインに従ってください。

- データ構造をベクトル化しやすいようにします。

- 内部ループのインデックスが外側（最後の）配列のインデックスと一致するようにします（行優先順)。

- 構造体配列（AoS）より配列構造体（SoA）を使用します。

例えば、3 次元座標を処理する場合、3 要素からなる構造体配列（AoS）の 1 つの要素を使用する代わりに、各要素に対する 3 つの異なる配列構造体（SoA）を使用します。ベクトル化を阻むループ間の依存性を避けるために、3 次元座標を処理する場合、3 要素からなる構造体配列（AoS）の代わりに、各要素に対する 3 つの異なる配列構造体（SoA）を使用することを推奨します。AoS では、各反復は XYZ を計算することによって 1 つの結果を導きますが、4 つ目の要素が使用されていないため、最大でも SSE の 75% しか使用できません。場合によっては、1 つの要素（25%）しか使用しないこともあります。SoA では、各反復は SSE ユニットを 100% 使用し、XXXX、YYYY、ZZZZ を計算することによって 4 つの結果を導きます。SoA の短所は、コード量がおよそ 3 倍に増えることです。一方、コンパイラーは AoS のコードを全くベクトル化できず、スカラー操作を行う可能性があります。

元々のデータレイアウトが AoS 形式であれば、性能に影響を及ぼすループの前に SoA への動的な変換を検討してください。ベクトル化されれば、それだけの価値はあります。

AoS の例:

```
 3 typedef struct {
 4   float red;
 5   float green;
 6   float blue;
 7   float alpha;
 8 } CData32;
 9
10 void SepiaFilterAoS(float *pImage, float *pResult, unsigned int imageSize)
11 {
12   CData32 *pSrc = (CData32 *)pImage;
13   CData32 *pDst = (CData32 *)pResult;
14 #ifdef __INTEL_COMPILER
15 __assume_aligned(pSrc, 16);
16 __assume_aligned(pDst, 16);
17 #endif
18   for(int i=0;i<imageSize;i++)
19   {
20     pDst[i].red = 0.393f*pSrc[i].red + 0.769f*pSrc[i].green + 0.189f*pSrc[i].blue;
21     pDst[i].green = 0.349f*pSrc[i].red + 0.686f*pSrc[i].green + 0.168f*pSrc[i].blue;
22     pDst[i].blue = 0.272f*pSrc[i].red + 0.534f*pSrc[i].green + 0.131f*pSrc[i].blue;
23     pDst[i].alpha = pSrc[i].alpha;
24     if(pDst[i].red > 255) pDst[i].red = 255;
25     if(pDst[i].green > 255) pDst[i].green = 255;
26     if(pDst[i].blue > 255) pDst[i].blue = 255;
27   }
28 }
```

icl /c AoSvsSoA.cpp /DAOS /Qvec-report2

```
AoSvsSoA.cpp

AoSvsSoA.cpp(18):(列 3): リマーク: ループはベクトル化されませんでした: ベクトル依存関係が存在しています。
```

SoA の例:

```
34 #ifdef SOA
35
36 typedef struct {
37   float red[4];
38   float green[4];
39   float blue[4];
40   float alpha[4];
41 } CData32T4;
42
43 const float fIntensity = 0.0234;
44 extern CData32T4 * GetCurrentSrcPointer32T4(void);
45
46
47 void CrossfadeTransposedFloatKernel()
48 {
49   CData32T4 *pSrc1 = GetCurrentSrcPointer32T4();
50   CData32T4 *pSrc2 = GetCurrentSrcPointer32T4();
51   CData32T4 *pDst  = GetCurrentSrcPointer32T4();
52 #ifdef __INTEL_COMPILER
53 __assume_aligned(pSrc1, 16);
54 __assume_aligned(pSrc2, 16);
55 __assume_aligned(pDst, 16);
56 #endif
57 #pragma ivdep
58   for(int j=0;j<4;j++)
59   {
60     pDst->red[j] = (1.0f-fIntensity)*pSrc1->red[j] + fIntensity*pSrc2->red[j];
61     pDst->green[j] = (1.0f-fIntensity)*pSrc1->green[j] + fIntensity*pSrc2->green[j];
62     pDst->blue[j] = (1.0f-fIntensity)*pSrc1->blue[j] + fIntensity*pSrc2->blue[j];
63     pDst->alpha[j] = pSrc1->alpha[j];
64   }
65 }
66 #endif // SOA
```

icl /c AoSvsSoA.cpp /DSOA /Qvec-report2

AoSvsSoA.cpp

AoSvsSoA.cpp(58):(列 3) リマーク:ループがベクトル化されました。

5.4 データ構造のベクトル化ガイドライン

- SIMD 幅を最大限にするために、必要な精度を提供する最小のデータ型を使用します。(16 ビットのみ必要な場合、int より short を使用すると、4 ウェイと 8 ウェイの SIMD 並列処理のそれぞれ違いが実感できます。)

- 同じループの中にベクトル化できる複数のデータ型を混在させないようにします (配列のインデックスに対する整数演算を除く)。データ型変換を行うベクトル化は、効率が悪くなるか、型と操作によってはベクトル化はサポートされません。インテル® C++ コンパイラー バージョン 12 (Windows* 版/Linux* 版) などの最新のコンパイラーは、機能が大幅に強化されており、複数の型が混在するループの自動ベクトル化もサポートしています。ただし、可能な場合は避けてください。次の例は、データ型が混在し、自動ベクトル化の妨げとなる可能性があります。

```
void mixed(float *restrict a, double *restrict b, float *c)
{
      for(int i = 1; i < 1000; ++i)
      {
         b[i] = b[i] - c[i];
         a[i] = a[i] + b[i];
      }
}
```

- SIMD ハードウェアでサポートされていない演算は避けてください。Linux*での (80 ビットの) long double の演算や剰余演算子 “%” は、OS と SIMD ハードウェアでサポートされない例です。

- 使用するプロセッサーで利用可能なすべての命令セットを使います。使用するプロセッサーに最適なコマンドライン・オプションを使用するか、最適な Microsoft* Visual Studio* IDE (統合開発環境)オプションを選択します。インテル® プロセッサーでのみ実行するアプリケーションは、[プロジェクト] > [プロパティ] > [C/C++] > [コード生成] > [指定された命令セットの専用コード生成]
で /QxSSE4.1、/QxSSE4.2 など (Windows*) を選択します (Linux* または Mac OS* X の場合は、-xSSE4.1、-xSSE4.2 など)。
互換性のあるインテル以外のプロセッサーで実行する可能性のあるアプリケーションは、[プロジェクト] > [プロパティ] > [C/C++] > [コード生成] > [拡張命令セットを有効にする] で /arch:SSE2、/arch:SSE3 (Windows*) を選択します。(Linux* または Mac OS* X の場合は -msse2、-msse3)

  ビルドするプロセッサーと同じ種類のプロセッサーでのみ実行するアプリケーションは、/QxHost (Windows*) または –xhost (Linux* または Mac OS* X) コマンドライン・オプションを選択するか、Microsoft* Visual Studio* IDE (統合開発環境)で、[プロジェクト] > [プロパティ] > [C/C++] > [コード生成] > [指定された命令セットの専用コード生成] で /QxHost を選択します。このオプションはインテル製マイクロプロセッサーおよび互換マイクロプロセッサーで利用可能ですが、両者では生成されるコードが異なります。

  利用可能なプロセッサー固有のオプションの詳細は、次の記事を参照してください。
  http://software.intel.com/en-us/articles/performance-tools-for-software-developers-intel-compiler-options-for-sse-generation-and-processor-specific-optimizations/ (英語)

- ベクトル化をサポートするコンパイラーは、ベクトル化がパフォーマンスを向上する見込みがあるかどうかを判断します。インテル® C/C++ コンパイラーは、アライメントされていないデータ・アクセス・パターン、または非ユニットストライドのデータ・アクセス・パターンがあると、ループのベクトル化を無効にします。ベクトル化によりパフォーマンスが向上することが分かっている場合は、#pragma vector always ヒントをループの前に追加してコンパイラーの解釈を無効にできます。これにより、効率性解析の結果に関係なく、コンパイラーはあらゆるループをベクトル化します。

6. ベクトル化レポート

インテル® コンパイラーのベクトル化レポートは、2 つの重要な情報を提供します。1 つは、コード中のベクトル化されたループの情報です。ベクトル化されたループは、最終的にパックド SIMD 命令を含むループの命令ストリームになります。もう 1 つは、コンパイラーがループをベクトル化できなかった理由です。この情報は、コンパイラーによるベクトル化の妨げになっている要因を明らかにします。インテル® コンパイラーのベクトル化レポートは、デフォルトでは無効になっているため、使用する場合はオプションにより有効にします。Microsoft* Visual Studio* IDE (統合開発環境) では、[プロジェクト] > [プロパティ] > [C/C++] > [診断] > [ベクトライザー診断レベル] で、[ベクトル化に成功したループと成功しなかったループ(2)] を選択します。これは、以下の n=2 に当たります。

/Qvec-report (Windows*) または -vec-report (Linux* または Mac OS* X) オプションは、さまざまな情報を含むベクトル化レポートを生成するようにコンパイラーに指示します。ベクトル化レポートオプション /Qvec-report=<n> は、引数 <n> を使用して /Qvec-report=0 (診断情報なし) から /Qvec-report=3 (詳細な診断情報) まで、レポートに含める情報を指定します。/Qvec-report で設定可能な引数の値は次のとおりです。

n=0:診断情報なし

n=1:ベクトル化されたループ

n=2:ベクトル化されなかったループとその理由

n=3:依存関係の情報

n=4:ベクトル化されなかったループのみ

n=5:ベクトル化されなかったループと依存関係の情報のみ

6.1 ベクトル化レポートのメッセージの例

“**反復数が少なすぎます**”: ベクトル化のメリットを得るのに十分なループ反復数がありません。

“内側のループではありません”: ループの入れ子の内側のループのみベクトル化できます。

“ベクトル依存関係が存在しています”: 依存関係が存在するか、その可能性があるため、コンパイラーはループをベクトル化しませんでした。可能性のある依存関係を無視できることが明らかな場合は、**#pragma ivdep** を使用してコンパイラーに無視するよう指示できます。

"ベクトル化は可能ですが非効率です": コンパイラーは、ベクトル化によってループのパフォーマンスは向上しないと判断しました。ベクトル化が安全な場合は、**#pragma vector always** を使用してコンパイラーの判断を無視し、ベクトル化するようにコンパイラーに指示できます。

“条件は例外を保護します”: if 文を含むループをベクトル化する場合、通常コンパイラーはループ・インデックスのすべての値を使用して右辺の式を評価し、条件文が TRUE のケースだけを基に最終的な評価を下します。場合によっては、コンパイラーは、条件が不正なメモリーアドレスへのアクセスを保護するためである可能性を考慮してベクトル化しないことがあります。

その場合は、**#pragma ivdep**を使用して、条件文がメモリーの例外を保護するためではないことをコンパイラーに知らせることができます。

“**データ型は指定されたターゲット・アーキテクチャーでサポートされていません**”: このメッセージは、例えば、インテル® SSE2 命令セットのみをサポートするターゲット・プロセッサー向けに複雑な演算を含むループをコンパイルしたときに出力されます。複雑なデータ型を含む演算を効率良くベクトル化するには、インテル® SSE3 命令が必要です。

“文をベクトル化できません”: switch 文など、特定の文はベクトル化できません。

“インデックスが複雑すぎます”: 配列のインデックスが複雑すぎて、コンパイラーがメモリー・アクセス・パターンを解析できません。インデックスをメイン・ループ・カウンターの明示的な変数として記述してみてください。

“サポートされていないループ構造”、“トップテストが見つかりませんでした”: これらのメッセージは、可算ループ、1 つの入口と 1 つの出口など、ベクトル化の条件を満たしていないループに対して出力されます。

“ベクトル化には不適切な演算子です”: “%” (モジュロ) など、特定の演算子はベクトル化できません。

6.2 コンパイラーのベクトル化の支援

ループのベクトル化を促進する追加ヒントをコンパイラーに伝える方法を紹介します。

6.2.1 プラグマ

ここでは、主要なプラグマについて説明します。詳細は、コンパイラーのユーザー・リファレンス・ガイドを参照してください。

- **#pragma ivdep:** 可能性のあるデータ依存を無視しても安全であることをコンパイラーに知らせます(コンパイラーは自身が検出した依存性は無視しません)。明らかな依存性がある場合にこのプラグマを使用すると、不正な結果を引き起こします。

開発者には安全にベクトル化できることが分かっていても、コンパイラーのスタティック解析では判断できない場合があります。次のループについて考えてみましょう。

```
void copy(char *cp_a, char *cp_b, int n) {
  for (int i = 0; i < n; i++) {
         cp_a[i] = cp_b[i];
  }
}
```

追加ヒントがなければ、コンパイラーは、ポインター変数 cp_a と cp_b によりアクセスされるメモリー領域が (すべてか一部でも) オーバーラップする可能性があることを想定します。これにより、データ依存の可能性が発生し、このループのベクトル化が妨げられます。この時点でコンパイラーはループをシリアルのままにすることを決定するか、以下のようなランタイムテストを生成し条件を満たすループをベクトル化します。

```
if (cp_a + n < cp_b || cp_b + n < cp_a) // ポインターのオーバーラップをテスト
  /* ベクトル化ループ */
  for (int i = 0; i < n; i++) cp_a[i] = cp_b [i];
else
  /* シリアルループ */
  for (int i = 0; i < n; i++) cp_a[i] = cp_b[i];
```

ランタイムのデータ依存性テストは、コードサイズとテストのオーバーヘッドは増えますが、一般に C または C++ コードで並列処理を活用する効率的な方法を提供します。ただし、関数自身にヒントが指定される場合、次のようにコンパイラーのベクトル化を支援できます。関数が主に小さな値の n に使用されるか、またはオーバーラップされるメモリー領域に使用される場合、#pragma novector ヒントをループの前に挿入してベクトル化を実行しないようにし、ランタイム・オーバーヘッドを抑えられます。逆に、ループがオーバーラップしないメモリー領域で実行されることが明らかである場合、#pragma ivdep ヒントをループの前に挿入して、ベクトル化の妨げとなる可能性のあるデータ依存性を無視できることをコンパイラーに伝えます。これにより、ランタイムのデータ依存性テストなしでループがベクトル化されます。

```
#pragma ivdep
void copy(char *cp_a, char *cp_b, int n) {
  for (int i = 0; i < n; i++) {
         cp_a[i] = cp_b[i];
  }
}
```

あるいは、restrict キーワードを使用することもできます (次の節を参照)。

- **#pragma loop count (n):** コンパイラーにループの反復数を知らせます。これは、ベクトル化のメリットがあるかどうか、あるいはループの代替コードパスを生成するべきかどうかコンパイラーが判断するのに役立ちます。
- **#pragma vector always:** コンパイラーがパフォーマンスの向上につながると判断するかどうかにかかわらず、安全にループをベクトル化できる場合はベクトル化するようコンパイラーに指示します。
- **#pragma vector align:** 後に続くループ内のデータは (例えば、SSE 命令セットでは 16 バイト境界に) アライメントされていることを通知します。
- **#pragma novector:** 後に続くループをベクトル化しないようコンパイラーに指示します。
- **#pragma vector nontemporal:** データは再利用されないため、キャッシュに格納せず直接メモリーに書き込むストリーミング・ストアを使用するようコンパイラーに知らせます。

6.2.2 *restrict* キーワード

restrict キーワードは、ポインター参照されるメモリーにエイリアスがないこと (つまり、他のアクセスがないこと) を表明します。.c または .cpp ファイルでは /Qrestrict (-restrict) コンパイラー・オプション、.c ファイルでは /Qstd=c99 コンパイラー・オプションを指定する必要があります。

上記の #pragma ivdep の例は、restrict キーワードを使用して処理することもできます。

restrict キーワードを上記の **cp_a** と **cp_b** の宣言に加え、各ポインター変数が特定のメモリー領域へ排他アクセスすることをコンパイラーに知らせることができます。引数リストの restrict 指示子は、ポインターが指すメモリーへのエイリアスがほかにないことをコンパイラーに知らせます。つまり、使用されるポインターは、そのポインターが有効なスコープ内でそのメモリーへアクセスする唯一の方法であることを示します。ループが restrict キーワードなしでベクトル化されたとしても、コンパイラーはランタイムにエイリアシングをチェックします。restrict キーワードが使用されると、コンパイラーはランタイムにエイリアシングをチェックしません。このキーワードを使用する場合、追加のコンパイラー・オプション (インテル® C/C++ コンパイラーでは /Qrestrict) が必要です。

```
void copy(char * restrict cp_a, char * restrict cp_b, int n) {
  for (int i = 0; i < n; i++) cp_a[i] = cp_b[i];
}
```

次の例では、ポインター a、b 、c の間でエイリアス問題の可能性があるため、ベクトル化されません。

```
// サポートされていない可能性のあるループ構造
void add(float *a, float *b, float *c) {
    for (int i=0; i<SIZE; i++) {
        c[i] += a[i] + b[i];
    }
}
```

restrict キーワードが追加されると、コンパイラーはそのメモリーにアクセスする他のポインターがないことを想定し、コードをベクトル化します。

```
// restrict を指定してポインターが安全であることをコンパイラーに知らせる
void add(float * restrict a, float * restrict b, float * restrict c)
{
    for (int i=0; i<SIZE; i++) {
        c[i] += a[i] + b[i];
    }
}
```

restrict キーワードは書き込みを行うポインター変数にのみ指定すれば大丈夫です。

ただし、ループ固有の #pragma ivdep ヒントと、ポインター変数固有の restrict ヒントの使用には注意してください。誤った使用は、オリジナルのプログラムのセマンティクスを変更することがあります。

restrict を使用する欠点は、すべてのコンパイラーがこのキーワードをサポートしているわけではないことです。そのため、ソースの移植性が低下します。ソースの移植性を重視する場合は、代わりに /Qalias-args- (-fargument-noalias) または /Qansi-alias (**-ansi-alias**) コンパイラー・オプションの使用を検討してください。/Qalias-args- (-fargument-noalias) オプションは、関数の引数がファイル内のほかのすべての引数とエイリアスしないようにコンパイラーに指示し (ただし、グローバルストレージとはエイリアスの可能性があります)、/Qansi-alias (**-ansi-alias**) オプションは、異なる型同士がエイリアスしないようにコンパイラーに指示します。この手法は、ソース中の多くのループで排他アクセスの可能性があるポインター変数を利用している場合に便利です。ベクトル化が可能なループにそれぞれヒントを加える必要がないためです。ただし、コンパイラー・オプションはソース全体に適用されるので、ほかのコード領域に影響を及ぼさないことを確認しなければなりません。

- __declspec(align(16)) を使用して、データが 16 バイト境界にアライメントされることを保証できます。

6.2.3 オプション

- 複数のソースファイルにわたるプロシージャー間の最適化（IPO）は、/Qipo (-ipo) オプションを使用して有効にします。IPO を有効にすると、ソース全体を解析することが可能になり、反復数、アライメント、データ依存性など、ループに関する追加情報をコンパイラーが知ることができます。また、関数呼び出しのインライン展開も可能になります。「3. どのようなループがベクトル化されるのか?」節の "関数呼び出しがない" 場合の例で、関数 func() と trap_int() が別々のソースファイルにある場合、/Qipo (-ipo) を指定して両ファイルをコンパイルすると、trap_int() 内のループをベクトル化できます。

- ポインターおよび配列の一義化。/Oa (Windows*) または –fno-alias (Linux* または MacOS* X) オプションを使用して、メモリー参照のエイリアシングがないこと、つまり、同じメモリー位置にアクセスする他の配列やポインターがないことを明示します。ほかのオプションはより制限されたヒントを有効にします。例えば、/Qalias-args- (-fargument-noalias) は関数の引数は互いにエイリアスがないこと (つまり、オーバーラップがない) を明示します。/Qansi-alias (-ansi-alias) オプションは、ISO C 規格のエイリアス規則に厳密に準拠することを有効にします。これらのオプションは、実際にメモリーのエイリアシングが存在し不正な結果を招く原因となります。開発者の責任で使用してください。

- ハイレベルの最適化（HLO）は、/O3 (-O3) オプションを使用して有効にします。より高度なループの最適化により、コンパイラーは最適化により変換されたループをより簡単にベクトル化できるようになります。このオプションはインテル製マイクロプロセッサーおよび互換マイクロプロセッサーで利用可能ですが、インテル製マイクロプロセッサーにおいてより多くの最適化が行われる場合があります。

  /Qopt-report-phase:hlo (-opt-report-phase hlo) オプション、または対応する IDE のオプションで得られる HLO レポートは、このような高度な変換が行われたかどうかを示します。

7. コンパイラーによるベクトル化を支援するコンパイラー機能の使用

7.1 ガイド付き自動並列化の使用

ガイド付き自動並列化（GAP）オプション “/Qguide (-guide)” を指定すると、コンパイラーは診断メッセージを生成し、自動ベクトル化、自動並列化、およびデータ変換を高めるアドバイスを示します。“/Qguide” オプションには、最適化レベル “/O2 (-O2)” 以上が必要です。最適化レベルが “/O2” 以上でない場合、オプションは無視されます。個々のアドバイスは、コンパイラーにより多くのヒントを伝える 1 つの方法として捉えることができます。通常、提供するヒントが多ければ多いほど、コンパイラーはより多くの最適化を行い、より優れたパフォーマンスを達成できます。“/Qguide” と “/Qparallel (-parallel)” オプションを同時に使用すると、コンパイラーは並列化に関するアドバイスを示すことがあります。

アドバイスにはソースコードの変更、特定のプラグマの適用、コンパイラー・オプションの追加などが含まれます。アドバイスは、適用する前にそれが安全に適用できることを確認してください。例えば、特定のプラグマを適用するようにアドバイスされた場合は、そのプラグマのセマンティクスを理解して、対象のループに安全に適用できるかどうかを慎重に検証して判断しなければなりません。アプリケーションのデータ・アクセス・パターンに基づいて有効性を検証しないままプラグマを

適用すると、コンパイラーが誤ったコードを生成し、アプリケーションが正しく動作しないことがあります。

“/Qguide” オプションを指定すると、コンパイラーはオブジェクトファイルや実行ファイルを生成しません。“/Qguide” オプションなしでは、コンパイラーは並列化、ベクトル化、データ変換の最適化を向上する方法に関するアドバイスを生成しません。ループがベクトル化されない場合は、“/Qguide” オプションを指定してコードをコンパイルし、(適用可能な場合は) GAP の推奨内容に従ってください。

例:

Scaler_dep.cpp:

```
49 for (i=0; i<n; i++) {
50     if (A[i] > 0) {b=A[i]; A[i] = 1 / A[i]; }
51     if (A[i] > 1) {A[i] += b;}
52   }
```

7.1.1 デフォルト - gap.cpp はベクトル化されない

icl /c /Qvec-report2 gap.cpp

```
      プロシージャー: test_scalar_dep

  gap.cpp(49):(列 1): リマーク: ループはベクトル化されませんでした: ベクトル依存関
  係が存在しています。
```

7.1.2 ベクトル化について GAP のアドバイスを得る

icl /c /Qguide gap.cpp

GAP レポート記録開始 Tue Jun 08 17:10:55 2010

リマーク #30761:コンパイラーに自動並列化を向上させるアドバイスを生成させる場合は -parallel オプションを追加します。
gap.cpp(49): リマーク #30515:(VECT) 行 49 のループ は、次の変数への条件付き代入があるためベクトル化できません: b。このループをベクトル化するには、この変数を各反復の最初で無条件に初期化します。[確認] ループの各反復で変数の値を読み取る場合は、同じ反復でそれ以前に書き込まれた値でなければなりません。このコンパイルセッションで出力されたアドバイス・メッセージの数: 1。

GAP レポート記録終了

7.1.3 GAP の推奨内容に基づく変更と gap.cpp をベクトル化するためのリビルド

Scaler_dep.cpp:

```
41   for (i=0; i<n; i++) {
42         b = A[i];
43     if (A[i] > 0) {A[i] = 1 / A[i];}
44     if (A[i] > 1) {A[i] += b;}
45   }
```

icl /c /Qvec-report1 gap.cpp

gap.cpp(41) (列 3): リマーク:ループがベクトル化されました。

7.2 ユーザー指示によるベクトル化 (#pragma simd)

ユーザー指示によるベクトル化、つまり “#pragma simd によるベクトル化” は、コンパイラーにループをベクトル化するように指示します。#pragma simd は、コードをベクトル化するのに必要なソースコードの変更を最小限に抑えるように設計されています。OpenMP* 並列化が自動並列化を補足するように、#pragma simd は自動ベクトル化を補足します。通常、コンパイラーが自動並列化しないループを並列化させる OpenMP* と同様に、#pragma simd を使用すると、#pragma vector always や #pragma ivdep などのベクトル化のヒントを利用してもコンパイラーが自動ベクトル化しないループをベクトル化できます。
#pragma simd ヒント付きのループのベクトル化に失敗した場合、エラーをレポートするかどうかを指定できます。デフォルトでは、#pragma simd は noassert に設定されており、ループのベクトル化に失敗すると、コンパイラーは警告を発行します。コンパイラーにエラーを発行するよう指示するには、#pragma simd に assert 節を追加します。#pragma simd ヒント付きのループがベクトル化されなかった場合、そのループはシリアル・セマンティクスを保持します。

#pragma simd の対象となる可算ループは最内ループで、OpenMP* のワークシェアリング・ループ構造の (C/C++) for ループ形式に適合していなければなりません。詳細は、http://www.openmp.org/mp-documents/spec30.pdf (英語) の2.5.1 節を参照してください。以下の例は、“#pragma simd” なしでも自動ベクトル化でき、上記のユーザー指示によるベクトル化の条件に合致しません。

```
1 void vec_copy(float *dest, float *src, int len)
2 {
3   float ii;
4
5 #pragma simd
6     for (int i = 0, ii = 0.0f; i < len; i++)
7       dest[i] = src[i] * ii++;
8 }
```

icl /c simd2.cpp /Qvec-report2

```
simd2.cpp
simd2.cpp(6): エラー: simd プラグマに続く for 文は <index> = <expr> 形式の
初期化子を含んでいなければなりません。

      for (i = 0, ii = 0.0f; i < len; i++) {
      ^
simd2.cpp(6): エラー: 無効な simd プラグマです。

#pragma simd
^
```

#pragma simd を使用して、データ依存性やその他の理由でベクトル化できないループのベクトル化を強制すると、コンパイラーは誤ったコードを生成する可能性があります。その場合、“reduction”、“private” などの simd プラグマのオプション節により、コンパイラーが正しいベクトル化コードを生成できることがあります。simd プラグマのオプション節の構文と使用法は、『インテル® C++ Composer 12 ユーザー・リファレンス・ガイド』を参照してください。

次の例でコンパイラーは、ベクトル依存関係があるためループをベクトル化できないことがレポートされます。このループのベクトル化を強制すると、コンパイラーは誤ったコードを生成します。

```
 1 char foo(char *A, int n){
 2
 3   int i;
 4   char x = 0;
 5
 6 #ifdef SIMD
 7 #pragma simd
 8 #endif
 9 #ifdef REDUCTION
10 #pragma simd reduction(+:x)
11 #endif
12 #ifdef IVDEP
13 #pragma ivdep
14 #endif
15   for (i=0; i<n; i++){
16     x = x + A[i];
17   }
18   return x;
19 }
```

icl /c /DIVDEP /Qvec-report2 simd3.cpp

```
simd3.cpp
simd3.cpp(15) (列 3): リマーク: ループはベクトル化されませんでした: ベクトル依存
関係が存在しています。
```

注: この例では、バージョン 12.0 のインテル® コンパイラーを利用していますが、バージョン 12.1 では、simd3.cpp をベクトル化できるように改善されています。

このループのベクトル化を強制すると、誤ったコードが生成されます。

```
>icl /c /DSIMD /Qvec-report2 simd3.cpp
```

```
simd3.cpp
simd3.cpp(15) (列 3): リマーク:SIMD ループがベクトル化されました。
```

7.2.1 正しくないコード

```
46 .B1.5::                         ; Preds .B1.5 .B1.4
47         paddb     xmm0, XMMWORD PTR [r8+rcx]                    ;16.13
48         add       r8, 16                                        ;15.3
49         cmp       r8, rdx                                       ;15.3
50         jb        .B1.5         ; Prob 82%                      ;15.3
51                                   ; LOE rdx rcx rbx rbp rsi rdi r8 r9 r12 r13 r14 r15 xmm0
52 .B1.6::                         ; Preds .B1.5
53         psrldq    xmm0, 15                                      ;4.10
54         movd      eax, xmm0                                     ;4.10
```

次のように reduction 節を追加することで、正しいコードを生成するようにコンパイラーにヒントを与えることができます。

```
>icl /c /DREDUCTION /Qvec-report2 simd3.cpp
```

```
simd3.cpp
simd3.cpp(15) (列 3): リマーク:SIMD ループがベクトル化されました。
```

7.2.2 正しいコード

```
56 .B1.8::                         ; Preds .B1.6 .B1.4
57         movzx     eax, al                                       ;4.10
58         movd      xmm0, eax                                     ;4.10
59                                   ; LOE rdx rcx rbx rbp rsi rdi r8 r9 r12 r13 r14 r15 xmm0
60 .B1.9::                         ; Preds .B1.9 .B1.8
61         paddb     xmm0, XMMWORD PTR [r9+rcx]                    ;16.13
62         add       r9, 16                                        ;15.3
63         cmp       r9, rdx                                       ;15.3
64         jb        .B1.9         ; Prob 82%                      ;15.3
65                                   ; LOE rdx rcx rbx rbp rsi rdi r8 r9 r12 r13 r14 r15 xmm0
66 .B1.10::                        ; Preds .B1.9
67         movdqa    xmm1, xmm0                                    ;4.10
68         psrldq    xmm1, 8                                       ;4.10
69         paddb     xmm0, xmm1                                    ;4.10
70         movdqa    xmm2, xmm0                                    ;4.10
71         psrldq    xmm2, 4                                       ;4.10
72         paddb     xmm0, xmm2                                    ;4.10
73         movdqa    xmm3, xmm0                                    ;4.10
74         psrldq    xmm3, 2                                       ;4.10
75         paddb     xmm0, xmm3                                    ;4.10
76         movdqa    xmm4, xmm0                                    ;4.10
77         psrldq    xmm4, 1                                       ;4.10
78         paddb     xmm0, xmm4                                    ;4.10
79         movd      eax, xmm0                                     ;4.10
```

この例では、ループ伝搬による依存性のため “pragma ivdep” を使用してもループは自動ベクトル化されませんが、“pragma simd” を使用するとベクトル化できます。ベクトル長 4、8、16 ではループ伝搬依存性はないため、“vectorlength(16)” 節を使用して、ベクトル長 4、8、16 のループのベクトル化は問題ないが、それ以外はループ伝搬依存性のため問題があることをコンパイラーに知らせます。

```
13   short x = 0;
14 #ifdef SIMD
15 #pragma simd reduction(+:x)
16 #endif
17   for (i=0; i<n; i++) {
18     x = SAT2SI16(x + p[i]*q[i]);
19   }
20   return x;
21 }
22
23 int main(int argc, char **argv) {
24   short x = 0;
25   const __int64 startTime = __rdtsc();
26
27   for (int i=0; i<32767; i++) {
28     a[i] = 1;
29     b[i] = 1;
30   }
31
32 #ifdef SIMD
33 #pragma simd vectorlength(16)
34 #endif
35   for (int i=0; i<32767; i++) {
36     if (i >= 16 && i < 32767) {
37       b[i] = b[i-16] - 1;
38     }
39   printf("b[%d] = \n", b[i]);
40   }
41
42   x = sat2short(&a[0], &b[0], 32767);
```

>icl /c /Qvec-report2 simd4.cpp

```
インテル(R) 64 対応アプリケーション用 インテル(R) C++ コンパイラー XE、バージョン
12.0.0.063 ビルド 20100721

simd4.cpp(27) (列 3): リマーク:ループがベクトル化されました。
simd4.cpp(35) (列 3): リマーク: ループはベクトル化されませんでした: ベクトル依存
関係が存在しています。
simd4.cpp(35) (列 3): リマーク: ループはベクトル化されませんでした: ベクトル依存
関係が存在しています。
simd4.cpp(42) (列 7): リマーク: ループはベクトル化されませんでした: ベクトル依存
関係が存在しています。
simd4.cpp(17) (列 3): リマーク: ループはベクトル化されませんでした: ベクトル依存
関係が存在しています。
```

```
>icl /c /Qvec-report2 simd4.cpp /DSIMD
```

```
インテル(R) 64 対応アプリケーション用 インテル(R) C++ コンパイラー XE、バージョン
12.0.0.063 ビルド 20100721

simd4.cpp
simd4.cpp(27) (列 3): リマーク:ループがベクトル化されました。
simd4.cpp(35) (列 3): リマーク:SIMD ループがベクトル化されました。
simd4.cpp(35) (列 3): リマーク:SIMD ループがベクトル化されました。
simd4.cpp(42) (列 7): リマーク:SIMD ループがベクトル化されました。
simd4.cpp(17) (列 3): リマーク:SIMD ループがベクトル化されました。
```

以下の例では、変数 “t” が各反復で代入されるかどうか不明なため、コンパイラーはループをベクトル化しません。“private” 節を使用してループの各反復に “t” のプライベート・コピーがあることを通知することで、コンパイラーはこのループを正しくベクトル化できます。“private” 節は、ベクトル化と並列化でよく知られている順序の不整合（決定性の問題）を招くことがあります。必要に応じて、順序の整合性を維持するのは開発者の責任です。例えば、以下の例でシリアル実行した場合の“t” の結果は、SIMD 実行した場合と異なることがあります。SIMD 実行では “t” の最終値はループの最後の反復からの値になりますが、シリアル実行では A[i] と B[i] によっては最後の反復の値になりません。

```
 1 void foo(int *A, int *B, int *restrict C, int n){
 2   int i;
 3   int t = 0;
 4
 5 #ifdef PRIVATE
 6 #pragma simd private(t)
 7 #endif
 8   for (i=0; i<n; i++){
 9     if (A[i] > 0) {
10       t = A[i];
11     }
12     if (B[i] < 0) {
13       t = B[i];
14     }
15     C[i] = t;
16   }
17 }
```

```
icl /c /Qvec-report2 /Qrestrict simd5.cpp
```

```
インテル(R) 64 対応アプリケーション用 インテル(R) C++ コンパイラー XE、バージョン
12.0.0.063 ビルド 20100721

simd5.cpp
simd5.cpp(8) (列 3): リマーク: ループはベクトル化されませんでした: ベクトル依存関
係が存在しています。
```

```
>icl /c /Qvec-report2 /Qrestrict simd5.cpp /DPRIVATE

  インテル(R) 64 対応アプリケーション用 インテル(R) C++ コンパイラー XE、バージョン
  12.0.0.063 ビルド 20100721

  simd5.cpp
  simd5.cpp(8) (列 3): リマーク:SIMD ループがベクトル化されました。
```

7.3 要素関数

要素関数は、スカラー引数または配列要素で並列に呼び出すことができる関数です。要素関数は、関数宣言の前に **“__declspec(vector)”** (Windows* の場合) や **"__attribute__((vector))"** (Linux* の場合) を追加して定義します。

```
__declspec (vector)
 double ef_add (double x, double y){  return x + y;}
```

要素関数を宣言すると、インテル® コンパイラーは一度の呼び出しで複数の引数に対して操作が行えるようにそのショートベクトル・バージョンを生成します。コンパイラーは関数のスカラー実装も生成し、プログラムの異なる位置で 1 つの引数または配列引数で関数を呼び出すことができます。関数のベクトル形式は、次のように並列コンテキストから呼び出すことができます。

1. for ループから。自動並列化されます。要素関数だけを呼び出すループは常にベクトル化可能ですが、コンパイラーはパフォーマンス・ヒューリスティックを適用し、関数をベクトル化しない判断を下すことができます。

2. #pragma simd が指定された for ループから。“#pragma simd” が指定されたループから要素関数が呼び出されると、コンパイラーはパフォーマンス・ヒューリスティックを適用せずに、必ず関数のベクトル形式を呼び出します。ループに C ライブラリーの “printf” 関数が含まれている場合、コンパイラーは次のメッセージを出力します。

   "リマーク: ループはベクトル化されませんでした:ベクトル依存関係が存在しています。"

3. cilk_for から。

4. 配列表記構文から。

```
13  __declspec(noinline)
14  __declspec(vector)
15  int vfun_add_one(int x)
16  {
17    return x+1;
18  }
19
20
21  __declspec(noinline)
22  int checksum()
23  {
24    int i;
25    int sum = 0;
26    for (i = 0; i < N; i++) {
27      sum += a[i];
28    }
29    return sum;
30  }
31
32
33  __declspec(noinline)
34  void d5() {
35    int h,g = 0;
36    init();
37  #pragma simd
38    for (h = 0; h < N; h++) {
39      a[h] = vfun_add_one(a[h]);
40      b[h] = c[h] - a[h];
41      c[h] = a[h] - b[h];
42    }
43    g = checksum();
44    printf("SIMD for loop: passed %d\n",g);
45  }
```

>icl /c /Qvec-report2 vectorFunc.cpp

```
インテル(R) 64 対応アプリケーション用 インテル(R) C++ コンパイラー XE、バージョン
12.0.0.063 ビルド 20100721

vectorFunc.cpp
vectorFunc.cpp(36) (列 3): リマーク:ループがベクトル化されました。
vectorFunc.cpp(38) (列 3): リマーク:SIMD ループがベクトル化されました。
vectorFunc.cpp(16) (列 1): リマーク:関数がベクトル化されました。
vectorFunc.cpp(16) (列 1): リマーク:関数がベクトル化されました。
vectorFunc.cpp(26) (列 3): リマーク:ループがベクトル化されました。
vectorFunc.cpp(10) (列 3): リマーク:ループがベクトル化されました。
```

要素関数とその制約事項については、『インテル® C++ Composer XE 12 ユーザー・リファレンス・ガイド』と「Elemental functions:Writing data parallel code in C/C++ using Intel® Cilk Plus」を（英語）を参照してください。

8．外側のループのベクトル化

コスト予測、コードサイズ、コンパイル時間などの内部ヒューリスティックによっては、コンパイラーは外側のループを自動並列化できることがあります。外側のループの自動ベクトル化は /O3 で有効になります。"#pragma simd" を使用して、外側のループのベクトル化に対するコンパイラーの判断を無効にできます。#pragma simd が指定された外側のループのベクトル化は /O2 以上で有効になります。

```
1  subroutine test1(n, term, x, y)
2           integer n
3           real*8  term(n,3), x(n), y(n)
4           real*8  sum
5           integer i, k
6           do i = 1, n
7              sum = 0
8              do k = 1, 3
9                 sum = sum + term(i,k)
10             end do
11             y(i) = x(i)**sum
12          end do
13        end
14        subroutine test3(n, term, x, y)
15          integer n
16          real*8  term(n,3), x(n), y(n)
17          real*8  sum
18          integer i, k
19          do i = 1, n
20             y(i) = x(i)**sum(term(i,:))
21          end do
22  end
23
```

>ifort /Qvec-report2 /c test.f90 /O3

インテル(R) 64 対応アプリケーション用 インテル(R) Visual Fortran コンパイラー XE、バージョン 12.1.0.233 ビルド 20110811

test.f90(8):(列 12): リマーク: ループはベクトル化されませんでした: ベクトル化は可能ですが非効率です。

test.f90(6):(列 9) リマーク:外側のループがベクトル化されました。

test.f90(20):(列 25): リマーク: ループはベクトル化されませんでした: ベクトル化は可能ですが非効率です。

test.f90(19):(列 9) リマーク:外側のループがベクトル化されました。

8.1 pragma simd による外側のループのベクトル化

```
1  #define M 1024
2  void Loop7_novec_cse(double a[][M], double b[][M],double c[][M])
3  {
4  #ifdef SIMD
5    #pragma simd
6  #endif SIMD
7    for (int i=0; i<M; i++)
8      for (int k=0; k<M; k++)
9        for (int j=0; j<M; j++)
10         c[i][j] = c[i][j] + a[i][k] * b[k+1][j+1];
11 }
12
```

>icl /c outerloop.cpp /O3 /Qvec-report2

インテル(R) 64 対応アプリケーション用 インテル(R) C++ コンパイラー XE、バージョン 12.1.0.233 ビルド 20110811

outerloop.cpp

outerloop.cpp(9):(列 7): リマーク: ループはベクトル化されませんでした: ベクトル依存関係が存在しています。

outerloop.cpp(8):(列 5): リマーク: ループはベクトル化されませんでした: 内部ループではありません。

outerloop.cpp(7):(列 3): リマーク: ループはベクトル化されませんでした: 内部ループではありません。

>icl /c outerloop.cpp /O3 /Qvec-report2 /DSIMD

インテル(R) 64 対応アプリケーション用 インテル(R) C++ コンパイラー XE、バージョン 12.1.0.233 ビルド 20110811

outerloop.cpp

outerloop.cpp(7):(列 3) リマーク:SIMD ループがベクトル化されました。

9. まとめ

自動ベクトル化を確実に行う鉄則はありません。しかし、このガイドの説明に従うことで、ベクトル化される確率が高まります。これまでの経験から、自動並列化が成功すると、大幅なスピードアップが達成できます。SIMD コードを記述する前に、インテル® コンパイラーの自動ベクトル化機能を使用してみてください。コンパイラーを使用するほうが、自分で SIMD コードを記述するよりも短時間に、そして簡単に済みます。さらに、インテル® コンパイラーを使用して自動並列化することで、将来にわたってそのメリットが得られます。SIMD コードを書き直さなくても、新しいプロセッサーをターゲットにしてコードを再コンパイルするだけで、新しいプロセッサー向けに最適化されます。コンパイラーが自動ベクトル化できないケースや、自分で記述したほうが効率良くベクトル化できるなど、場合によっては開発者が SIMD コードを記述しなければならないこともあります。

10. 付録

以下の表は、パフォーマンスと診断に関連するコンパイラー・オプションのリストです。"/Qvec-" など、一部のオプションは診断目的でのみ使用します。

これらのオプションはインテル製マイクロプロセッサーおよび互換マイクロプロセッサーで利用可能ですが、インテル製マイクロプロセッサーにおいてより多くの最適化が行われる場合があります。

表 1: パフォーマンスと診断に関連するコンパイラー・オプション

Linux*	Windows*	説明
-vec-report0	/Qvec-report0	ベクトル化診断を無効にします (デフォルト)。
-vec-report1	/Qvec-report1	ベクトル化されたコードをレポートします。
-vec-report2	/Qvec-report2	ベクトル化されなかったループもレポートします。
-vec-report3	/Qvec-report3	ベクトル化の妨げとなるすべてのデータの依存関係もレポートします。
-no-vec	/Qvec-	コード中のフラグやプラグマに関係なく自動ベクトル化を無効にします。
-vec-threshold0	/Qvec-threshold0	ベクトル化によりパフォーマンスが向上しないとコンパイラーが判断した場合であってもループの自動ベクトル化を試みます。このオプションを指定すると、適切ではない、パフォーマンスを低下させる多数のループがベクトル化されるため、使用する際は注意が必要です。
-O0	/Od	最適化を無効にします。

Linux*	Windows*	説明
-O1	/O1	コードサイズの最適化を有効にします。
-O2	/O2	デフォルトの速度の最適化を有効にします。
-O3	/O3	ループの速度の強力な最適化を有効にします。
-guide	/Qguide	ガイド付き自動並列化とベクトル化を有効にします。コンパイラーは、ループをベクトル化または並列化するためのアドバイスを提供します。インテル® コンパイラー 12.0 以降でのみ利用できます。
-S	/S	アセンブリー・ファイルを生成します。
-unroll=0	/Qunroll:0	ループアンロールを無効にします。
-opt-prefetch-	/Qopt-prefetch-	ソフトウェア・プリフェッチを無効にします。
-ip	/Qip	1 つのソースファイルのプロシージャー間の最適化 (IPO) を有効にします。
-ipo	/Qipo	複数のソースファイルにわたる、インライン展開を含むプロシージャー間の最適化 (IPO) を有効にします。
-fargument-noalias	/Qalias-args-	関数の引数がほかの引数とエイリアスしないようにコンパイラーに指示します。
-fp-model precise	/fp:precise	浮動小数点の一貫性を向上します。実行速度が多少遅くなる場合があります。
-restrict	/Qrestrict	コンパイラーにエイリアシングがないことを知らせる restrict キーワードの使用を有効にします。
-parallel	/Qparallel	自動並列化を有効にします。
-openmp	/Qopenmp	OpenMP* 拡張を使用する並列化を有効にします。
-std=c99	/Qstd=c99	C99 拡張 (ISO 1999) を有効にします。

表 2: コンパイラーによるループの自動ベクトル化を支援するプラグマ

コンパイラー・ヒント	説明
#pragma ivdep	可能性のある (証明されていない) データ依存性を無視します。
#pragma vector always	効率性ヒューリスティックを無視します。

コンパイラー・ヒント	説明
#pragma vector nontemporal	ストリーミング・ストアを使用するようにヒントを与えます。
#pragma vector [un]aligned	アライメントされている [されていない] ことを表明します。
#pragma novector	対象ループのベクトル化を無効にします。
#pragma distribute point	この位置でループを分割するようにヒントを与えます。
#pragma loop count (<int>)	予測される反復数のヒントを与えます。
#pragma simd	ベクトル化を強制します。インテル® コンパイラー 12.0 以降でのみ利用できます。詳細は、コンパイラーのユーザー・リファレンス・ガイドを参照してください。
restrict	ポインターを介した排他アクセスを表明するキーワードです。コマンドライン・オプション “-restrict” を指定する必要があります。このガイドで紹介した例を参照してください。
__declspec(align(<int>,<int>)) (Windows*) __attribute__(align(<int>,<int>)) (Linux*) _declspec(align(<int>,<int>)) (_icc は _declspec を認識するが、_declspec は Windows* 形式の構文。Linux* では __attribute__. を使用。)	アライメントを明示します。
__assume_aligned(<var>,<int>)	アライメントされていることを想定します。

11. 参考資料

インテル® 64アーキテクチャーおよび IA-32 アーキテクチャー最適化リファレンス・マニュアル
http://download.intel.com/jp/developer/jpdoc/248966-024JA.pdf

インテル® ソフトウェア・ドキュメント・ライブラリー (英語)
http://software.intel.com/en-us/articles/intel-software-technical-documentation/

The Software Vectorization Handbook, Aart Bik, Intel Press, 2004.ISBN 0-9743649-2-4

Vectorization with the Intel® Compilers (Part I), Aart Bik,
http://software.intel.com/en-us/articles/requirements-for-vectorizable-loops/

http://software.intel.com/en-us/articles/vectorization-writing-cc-code-in-vector-format/

Image Processing Acceleration Techniques using Intel® Streaming SIMD Extensions and Intel® Advanced Vector Extensions, Petter Larsson, Eric Palmer

http://software.intel.com/en-us/articles/image-processing-acceleration-techniques-using-intel-streaming-simd-extensions-and-intel-advanced-vector-extensions/

インテル® コンパイラーは、互換マイクロプロセッサー向けには、インテル製マイクロプロセッサー向けと同等レベルの最適化が行われない可能性があります。これには、インテル® ストリーミング SIMD 拡張命令 2 (インテル® SSE2)、インテル® ストリーミング SIMD 拡張命令 3 (インテル® SSE3)、ストリーミング SIMD 拡張命令 3 補足命令 (SSSE3) 命令セットに関連する最適化およびその他の最適化が含まれます。インテルでは、インテル製ではないマイクロプロセッサーに対して、最適化の提供、機能、効果を保証していません。本製品のマイクロプロセッサー固有の最適化は、インテル製マイクロプロセッサーでの使用を目的としています。インテル® マイクロアーキテクチャーに非固有の特定の最適化は、インテル製マイクロプロセッサー向けに予約されています。この注意事項の適用対象である特定の命令セットの詳細は、該当する製品のユーザー・リファレンス・ガイドを参照してください。

改訂 #20110804