インテル® Cilk™ Plus を使用したグラフィック処理パフォーマンスの向上

はじめに

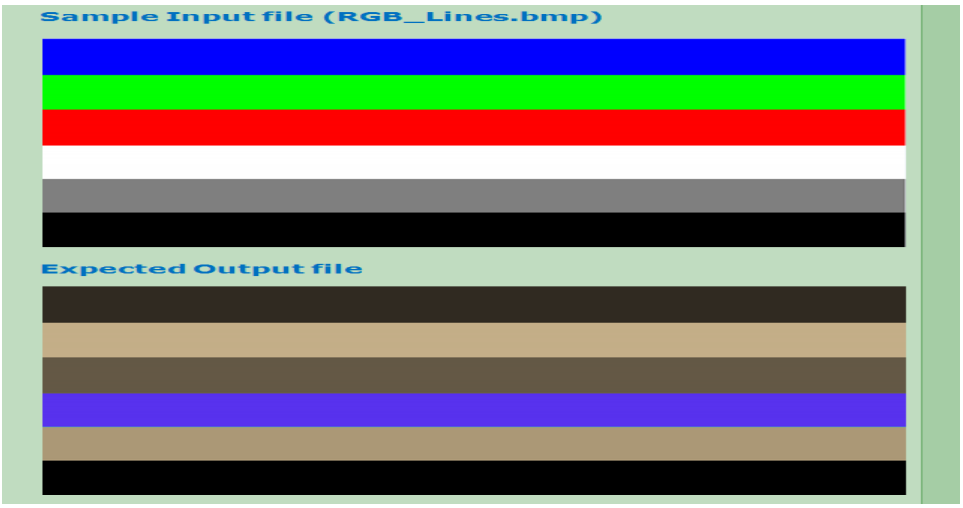
インテル® Cilk™ Plus は、データ並列とタスク並列をサポートする C/C++ 言語拡張であり、タスク並列を実装する 3 つの新しいキーワードとデータ並列を表現する配列表記構文を提供します。このドキュメントでは、インテル® Cilk™ Plus を利用してグラフィック処理のパフォーマンスを向上させる方法を説明します。サンプルは、ビットマップ・ファイルをカラーイメージからセピア調イメージに変換するプログラムを使用します。セピア調イメージとは茶褐色のモノクローム・イメージのことで、白黒フィルムの時代に写真で別のトーンを表現するために利用されていました。プログラムは、ビットマップ・ファイルの各ピクセルをセピア調に変換します。

概要

セピアフィルターはカラーイメージを、(ダークブラウンとグレーの 2 色で構成される) セピア色のイメージに変換します。各カラーピクセルの変換には、次の式を用います。

$$R_s = 0.393 * R + 0.769 * G + 0.189 * B$$
$$G_s = 0.349 * R + 0.686 * G + 0.168 * B$$
$$B_s = 0.272 * R + 0.534 * G + 0.131 * B$$

R、G、B は入力イメージの各ピクセルの赤 (Red)、緑 (Green)、青 (Blue) の値を表し、Rs、Gs、Bs は出力イメージの出力ピクセルに対応します。これは、出力イメージの位置 (i,j) のピクセルの値が、入力イメージの位置 (i,j) のピクセルにのみ依存する、データ並列性の高いアルゴリズムです。ループ構造に含まれる複数のデータ項目をベクトルレジスターにロードし、それらを 1 つの命令で同時に処理できる SIMD (Single Instruction Multiple Data) を利用するのに理想的なアルゴリズムと言えます。セピア変換を適用する前と後のビットマップ・ファイルを以下に示します。



最初に、セピアフィルター・アルゴリズムのシリアル実装のパフォーマンスを測定します。次にインテル® Cilk™ Plus を利用して、インテル® C++ コンパイラーのベクトル化機能と並列化機能によりパフォーマンスを向上させます。

最適化ステップ

次のステップで最適化を行います。

- Visual Studio* コンパイラーを使って、デフォルトのオプション (Release ビルド) でセピアフィルターのシリアルバージョンをビルドおよび実行し、パフォーマンスのベースラインを測定します。
- インテル® C++ コンパイラーを使って、デフォルトのオプション (Release ビルド) でプロジェクトをリビルドし、パフォーマンスの向上を測定します。
- インテル® Cilk™ Plus の配列表記 (アレイ・ノーテーション) を使ってフィルターを実装します。
- インテル® Cilk™ Plus の Cilk_for 構造を使ってスレッドレベルの最適化を実装します。
- 構造体配列 (AOS) を配列構造体 (SOA) に置換して、さらにパフォーマンスを高めます。

システム要件

このドキュメントの例と演習をコンパイルおよび実行するには、インテル® Parallel Composer XE 2013 Update 1 以降と、インテル® SSE2 以降の命令拡張をサポートするインテル® Pentium 4 以降のプロセッサーが必要です。**このドキュメントの演習は、256 ビットのベクトルレジスターを搭載する第 3 世代インテル® Core™ i5 システムでテストされました。**また、このドキュメントの手順は、Microsoft* Visual Studio* 2012 による方法を示しています。ビルド環境によってはコマンドが異なる場合がありますのでご注意ください。以前のバージョンの Visual Studio* でサンプルコードをビルドできるように、Visual Studio* 2008 プロジェクトが提供されています。この例は、Windows*、Linux*、OS X* のコマンドラインから、以下のコマンドライン・オプションを指定してビルドすることもできます。

Windows*:
```
icl /Qvec-report2 /Qrestrict /fp:fast SepiaFilterCilkPlus.cpp
```

Linux* および OS X*:
```
icc -vec-report2 -restrict -fp-model -fast SepiaFilterCilkPlus.cpp
```

Linux* および OS X* のシステム要件は、インテル® C++ Composer XE 2013 のリリースノートを参照してください。

注: このドキュメントで示すサンプルコードは、拡張子が .bmp の RGB (24 ビット) 形式のイメージのみ読み込むことができます。ソリューションには、サイズの異なる 3 つのサンプルイメージが含まれています。

サンプルの場所
サンプルコードをビルドするには、sepiafiltercilkplusvs2008.zip を展開します。このドキュメントでは次のファイルを使用します。

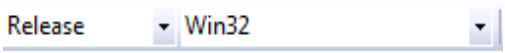
SepiaFilterCilkPlus ディレクトリー以下にあるサンプル入力イメージ:

- `RGB_Lines.bmp`
- `Test.bmp`
- `Blackbuck.bmp`

展開したディレクトリーにある以下のファイル:

- `SepiaFilterCilkPlus.sln`
- `SepiaFilterCilkPlus.cpp`
- `SepiaFilterCilkPlus.h`
- Microsoft* Visual Studio* のソリューション・ファイル `SepiaFilterCilkPlus.sln`。このファイルを開き、次の手順に従って演習で使うプロジェクトを準備します。

1. 「Release」、「Win32」構成を選択します。



2. **[ビルド] > [ソリューションのクリーン]** を選択してソリューションを消去します。

消去することで、関連するファイルや一時ファイルがすべて削除され、次回のビルドが既存のファイルの変更ではなくフルビルドになります。

ソースコードの内容

プログラムの main 関数は、コマンドライン引数として入力ファイルと出力ファイルを受け取り、

read_process_write() 関数を呼び出します。この関数は .bmp 入力ファイルを読み込みます。最初に、入力イメージファイルからヘッダー情報 (イメージの種類、圧縮の有無、入力イメージの幅と高さ) を読み込み、その情報を取得したら、動的データ構造を作成し、そこへピクセルレベルで処理できるようにペイロード・イメージ・データをコピーします。このプログラムでは、構造体配列 (AOS) と配列構造体 (SOA) の両方を実装し、それぞれのパフォーマンスを比較します。

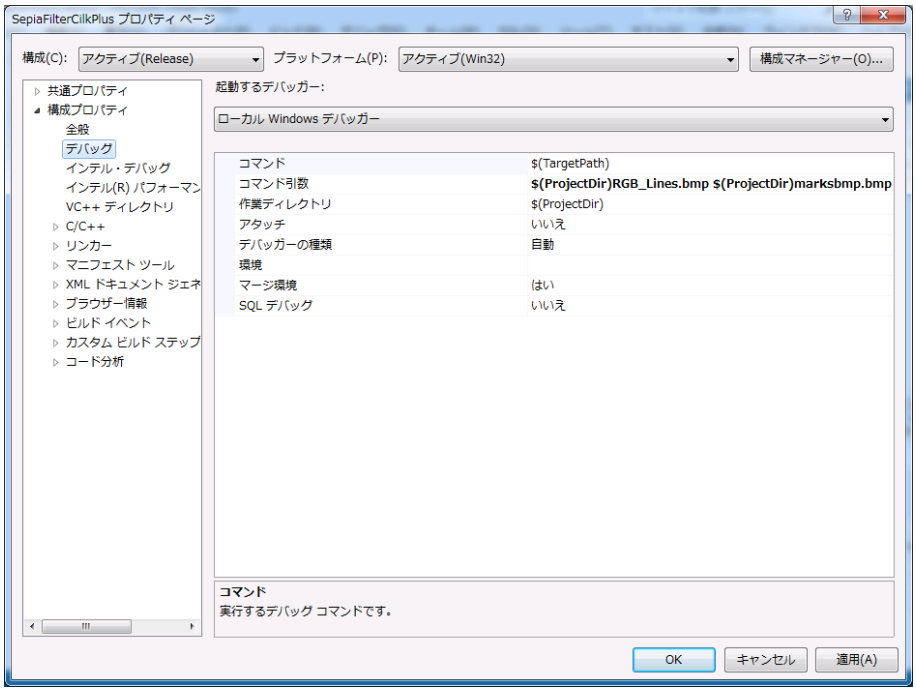
主要部であるセピアフィルター・カーネルは process_image() という名前で、コンパイルフェーズで用いられるマクロに応じて、対応するセピアカーネルの実装 (SOA および AOS データ構造により実装された配列表記バージョンまたは cilk_for バージョン) が有効になります。

コマンドラインでプログラムを実行する場合、次の形式を指定してください。

<実行ファイル> <入力ファイル> <出力ファイル>

Visual Studio* では、次のように入力イメージと出力イメージをコマンドライン引数として指定できます。

プロジェクトを右クリックして、[プロパティ] > [構成プロパティ] > [デバッグ] > [コマンド引数] を選択します。

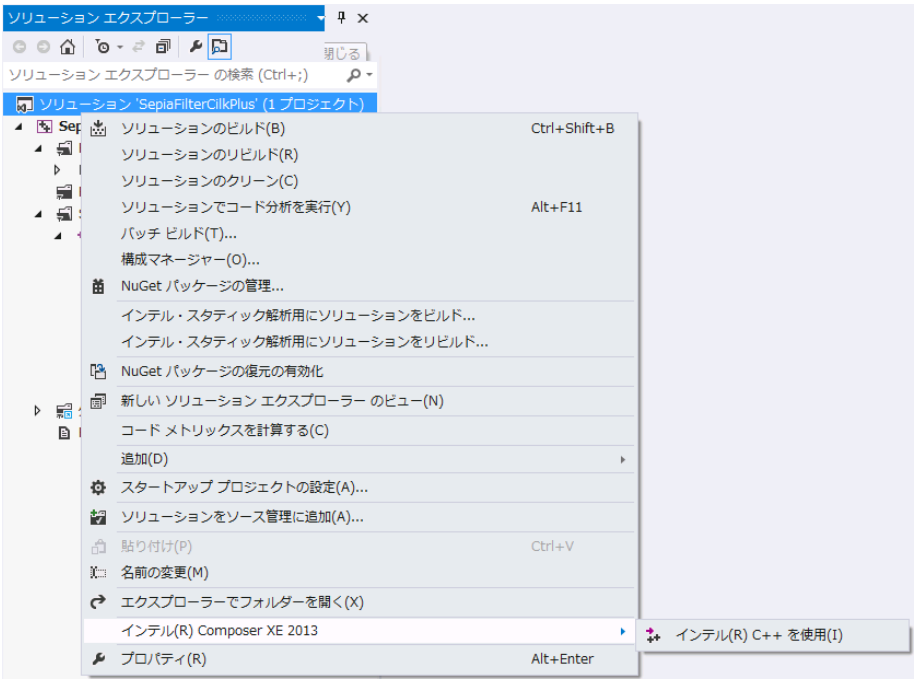


パフォーマンスのベースラインを測定する

パフォーマンスのベースラインを測定するには、Visual Studio* の Microsoft* C++ コンパイラーでプロジェクトをビルドし、プログラムを実行します (**[デバッグ] > [デバッグなしで開始]**)。プログラムを実行すると、ウィンドウにプログラムの実行時間 (クロック数) が表示されるので、この出力結果を記録します。

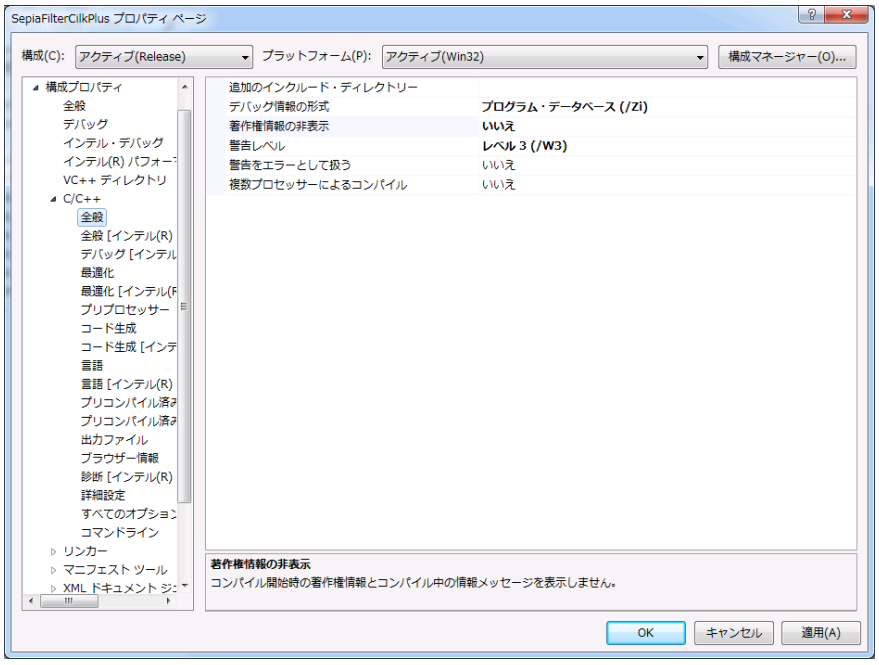
インテル® C++ コンパイラーでプロジェクトをビルドする

インテル® C++ コンパイラーを使用するようにプロジェクトを変換します。ソリューションを右クリックして、**[インテル(R) Composer XE 2013] > [インテル(R) C++ を使用]** を選択します。

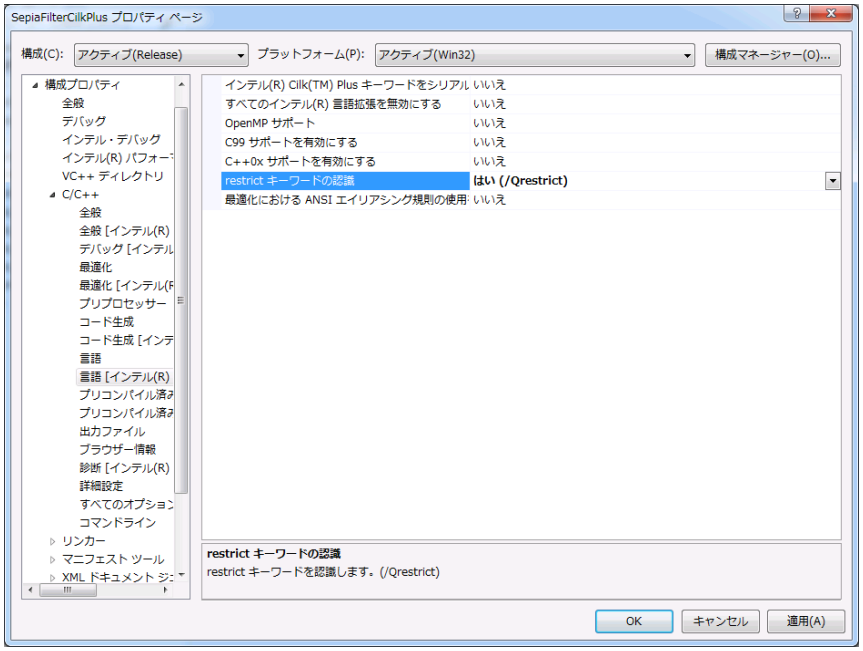


インテル® C++ コンパイラー用にプロジェクトを変換したら、以下の手順に従ってプロジェクトのプロパティーを設定します。

1. **[プロジェクト] > [プロパティ] > [C/C++] > [全般] > [著作権情報の非表示] > [いいえ]** を選択します (この設定はパフォーマンスには影響しません)。

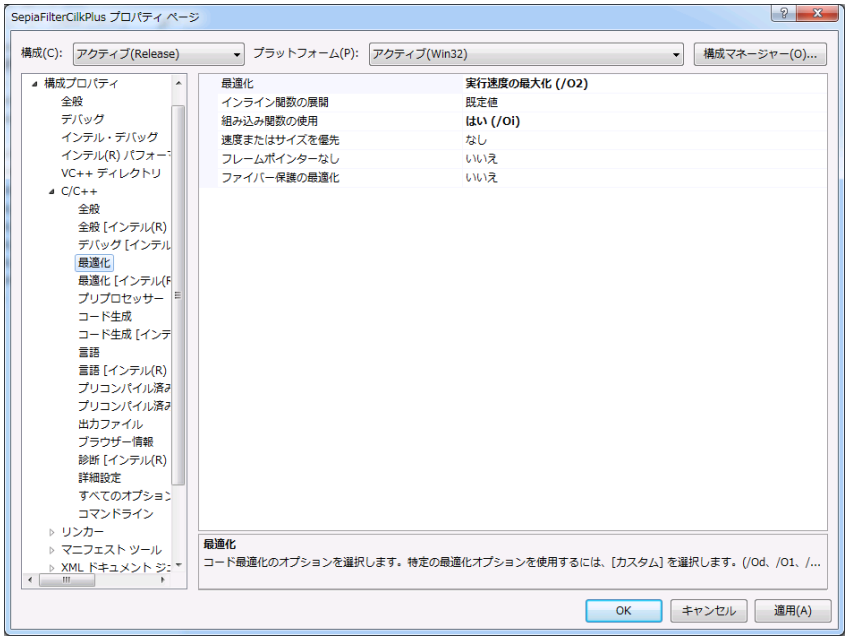


[言語 [インテル(R) C++]]> [restrict キーワードの認識] > [はい (/Qrestrict)] を選択します。

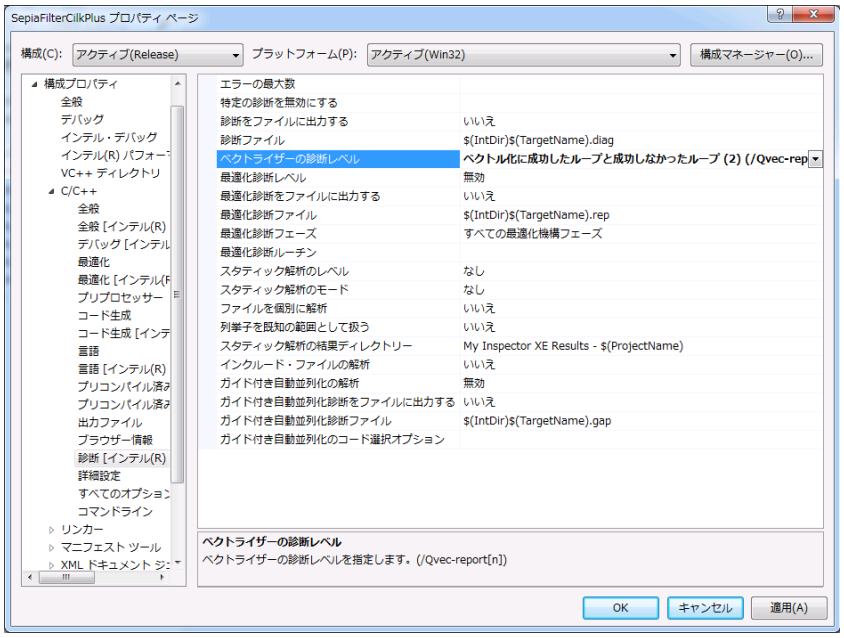


「restrict」キーワードは C99 拡張ですが、インテル® C++ コンパイラーは C++ でこのキーワードをサポートしています。このキーワードはデータポインターに適用され、そのポインターによりアクセスされるデータは他のポインターとエリアスがないことをコンパイラーに知らせます。restrict キーワードは、オブジェクトがほかのポインターによって変更されないという前提に基づいて、コンパイラーによる特定の最適化を有効にします。restrict キーワードを持つポインターは意図したとおりに使用されることを確認しなければいけません。そうでない場合、未定義の動作を引き起こします。

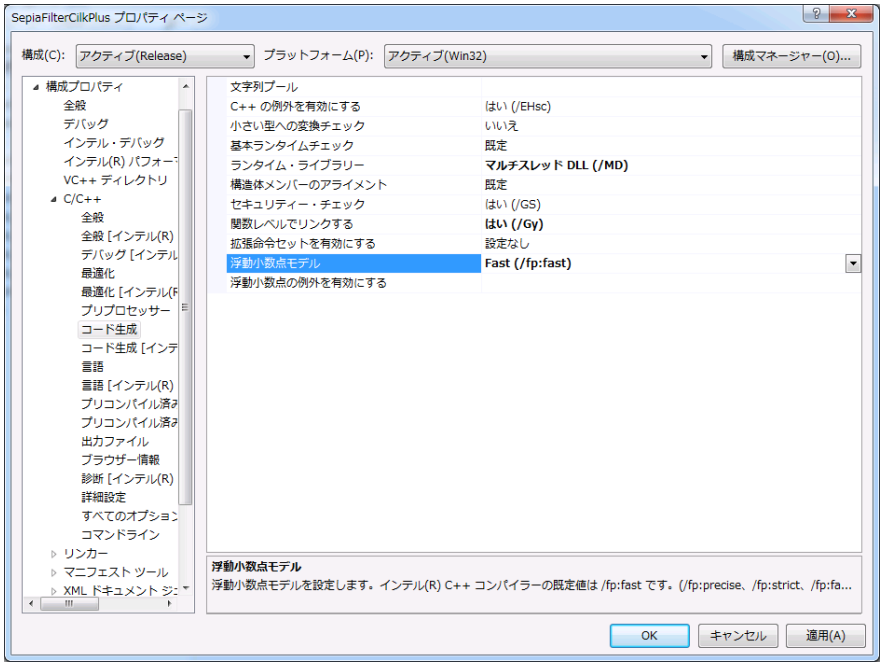
2. **[プロジェクト] > [プロパティ] > [C/C++] > [最適化] > [最適化] > [実行速度 (/O2)]** を選択します。



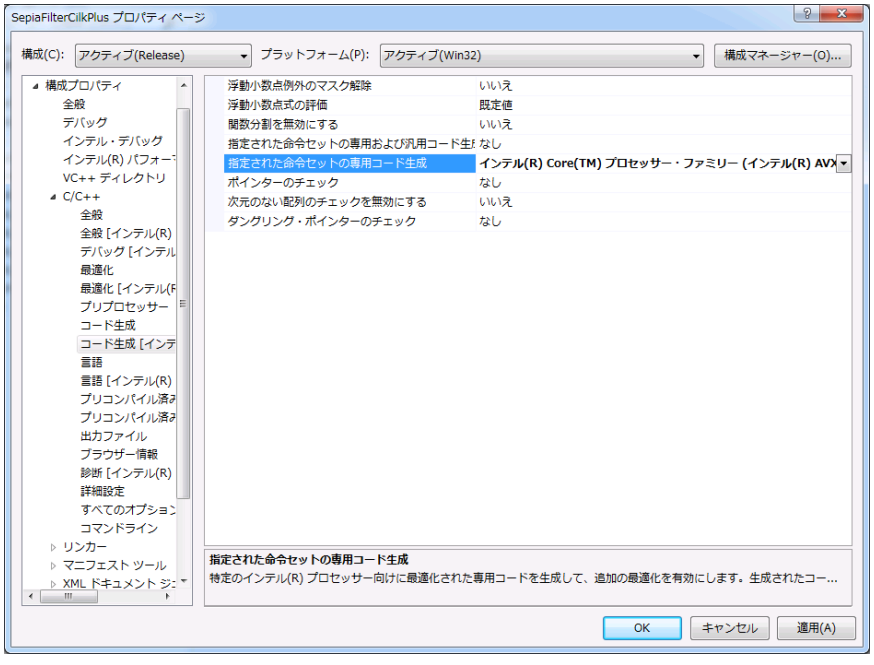
3. **[プロジェクト] > [プロパティ] > [C/C++] > [診断[インテル(R) C++]] > [ベクトライザー診断レベル] > [ベクトル化に成功したループと成功しなかったループ (2) (/Qvec-report2)]** を選択します。



4. **[プロジェクト] > [プロパティ] > [C/C++] > [コード生成] > [浮動小数点モデル)] > [Fast (/fp:fast)]** を選択します。

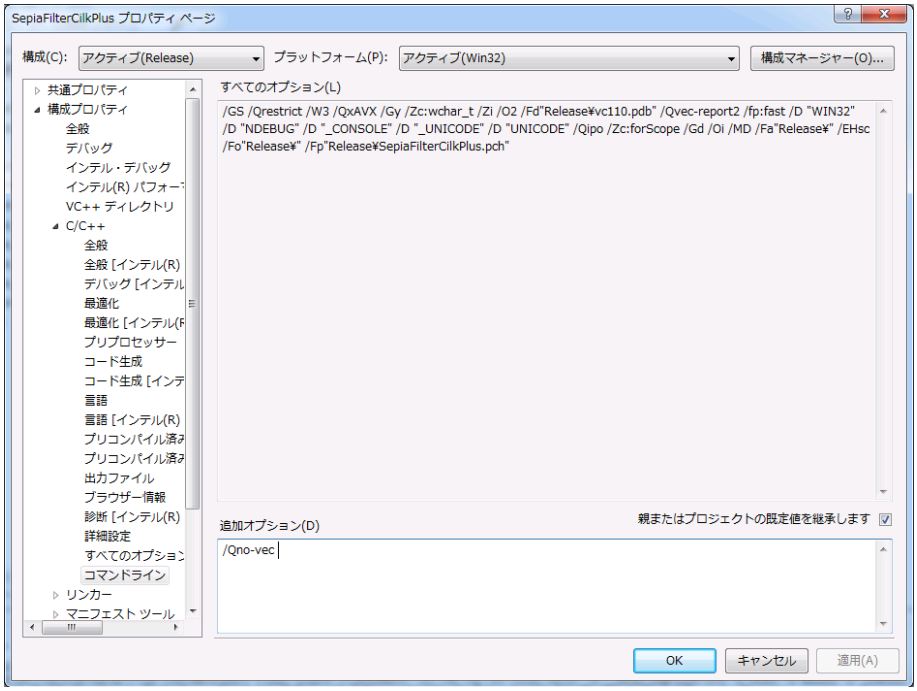


5. **[プロジェクト] > [プロパティ] > [C/C++] > [コード生成] > [指定された命令セットの専用および汎用コード生成] > [インテル(R) Core(TM) プロセッサー・ファミリー (インテル(R) AVX 対応) (/QaxAVX)]** を選択します。



6. ベクトル化は、インテル® C++ コンパイラーではデフォルトで有効であり、ほとんどのアプリケーションでパフォーマンスを大幅に向上します。ベクトル化によるセピアフィルターのパフォーマンスへの効果を検証するため、一時的にベクトル化を無効にし、実行時のパフォーマンスを見てみましょう。次の操作を行ってください。

[プロジェクト] > [プロパティ] > [C/C++] > [コマンドライン] を選択し、[追加オプション] に /Qno- vec と入力します。



プロジェクトをリビルドしてプログラムを実行します。この出力結果の実行時間を記録します。

7. /Qno-vec オプションを削除して、ベクトル化を再度有効にします。プロジェクトをリビルドしてプログラムを実行し、この出力結果の実行時間を記録します。これで、ベクトル化によりパフォーマンスが向上していることが分かるでしょう。これは、以降のパフォーマンス向上を測定する際のベースラインとなります。ベースラインのパフォーマンスを測定するときに、/O2 および /O3 最適化レベルで /Qvec-report2 の結果を比較することを推奨します。通常は、/O3 のほうがより多くのベクトル化候補が示されます。ただし、この例では /O2 と /O3 はほぼ同じ結果になります。

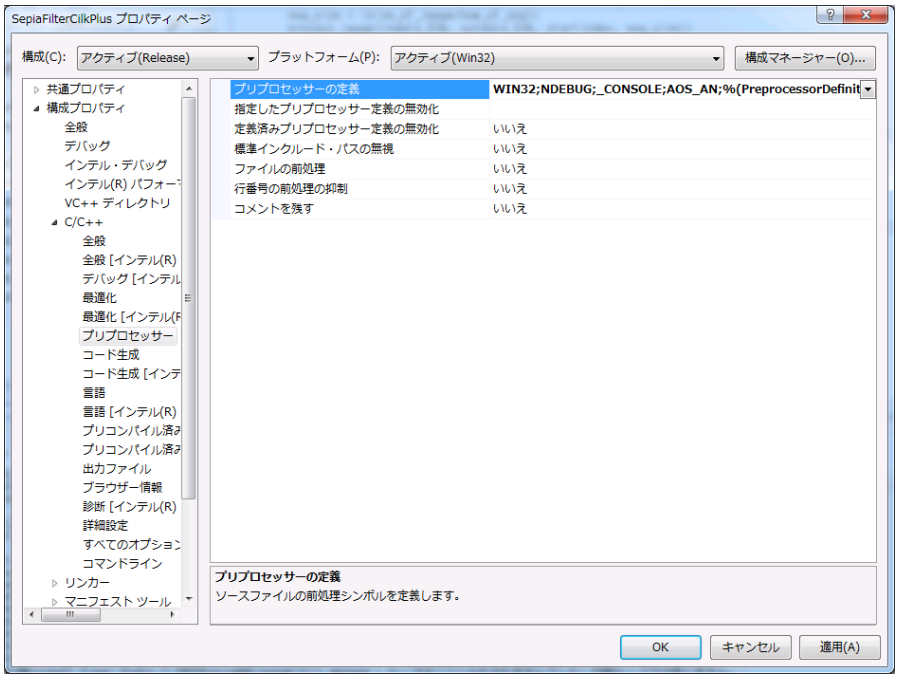
```
SepiaFilterCilkPlus.cpp(202): (列 2) リマーク: ループがベクトル化されました。
```

ベクトル化レポートから、SepiaFilterCilkPlus.cpp の上記の行でループがベクトル化されたことが分かります。これは process_image 関数を呼び出す for ループで、ここではインライン展開されています。コンパイラーは、SIMD レジスターを利用して関数本体をベクトル化しています。オリジナルのシリアル実装は、構造体配列 (AOS) 実装です。このデータ構造は、アルゴリズムに連続しないメモリーアクセスを含んでいるため、ベクトル化に適していません。連続しないメモリーアクセスのオーバーヘッドは、しばしばベクトル化の利点を損なったり、効率良いベクトル化の妨げとなりますが、この例では、コンパイラーはベクトル化により利点がもたらされると判断しています。

配列表記を利用してセピアフィルター・カーネルを実装する

ここでは、デフォルトのベクトル長の配列表記を利用して、オリジナルのループを書き直します。ベクトルレジスターが 128 ビットのプロセッサーの場合、デフォルトのベクトル長は 4 です (例えば、4 つの 32 ビット単精度浮動小数点データ要素をベクトルレジスターにロードできます)。

1. **[プロジェクト] > [プロパティ] > [C/C++] > [プリプロセッサー] > [プリプロセッサーの定義]** を選択して、新しいマクロ「**AOS_AN**」を追加します。



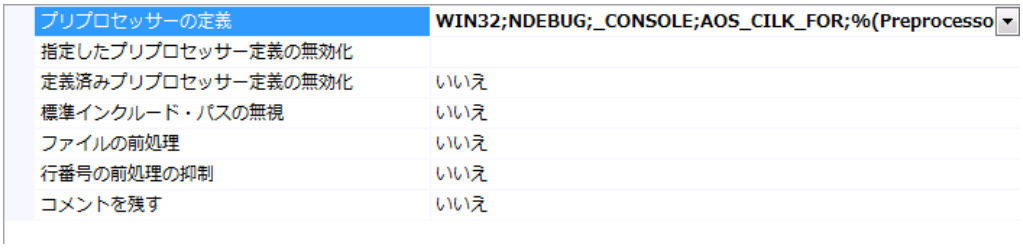
2. プロジェクトをリビルドして、プログラムを実行し (**[デバッグ] > [デバッグなしで開始]**)、この出力結果の実行時間を記録します。配列表記バージョンは、SIMD レジスターと SIMD 命令セットを利用して、ベクトルオペランドで演算を行います。ベクトル化レポートに、配列表記バージョンのループがベクトル化されたことを示すメッセージが出力されます。

```
SepiaFilterCilkPlus.cpp(173): (列 5) リマーク: ループがベクトル化されました。
```

このセピアフィルターのサンプルコードでは、配列表記バージョンのパフォーマンスは前出の自動ベクトル化バージョンとほぼ同じになります。自動ベクトル化バージョンでは、コンパイラーがベクトル化するコードを判断するため、常にベクトル化が行われる保証はありませんが、配列表記ではベクトル化が保証されます。

Cilk_for を指定してパフォーマンスを向上する

ここでは、cilk_for によるタスクレベルの並列化を紹介します。cilk_for 文は、指定した通常の C/C++ の for ループを置き換えて、ループの各反復を複数のコア上で並列に処理するように指示します。このサンプルコードでは、以下のように cilk ヘッダーファイルをインクルードし、「for」ループを「cilk_for」ループに置換するだけでマルチスレッド化できます。cilk_for バージョンを有効にするには、次のように「AOS_CILK_FOR」マクロを追加します。




```
#if defined(AOS_CILK_FOR) || defined(SOA_CILK_FOR)
#include<cilk\cilk.h>
#endif

#if defined(AOS_CILK_FOR)
    starttime = __rdtsc();
    cilk_for(int i = 0; i < size_of_image; i++)
    {
        process_image(indata[i], outdata[i]);
    }
    endtime = __rdtsc();
```

これらの変更を行った後にプロジェクトをリビルドすると、セピアフィルター・カーネルが SIMD レジスター (自動ベクトル化) を利用するだけでなく、複数のコアを活用し、for ループのワークロードを複数のタスクに分配して、さらなるスピードアップが得られます。ワークロードが大きいほど、スピードアップは理論上の限界に近くなります。サンプルコードとともに提供される入力イメージを使ってテストできます。入力イメージは、blackbuck.bmp、RGB_Lines.bmp、test.bmp の順にワークロードが大きくなります。これらのイメージに対するマルチスレッド・バージョンのパフォーマンスは、ワークロードの大きさに比例して向上します。これは、ワークロードが大きいほど、複数のコアによるスピードアップが高まることを裏付けています。

配列構造体 (SOA) を利用してさらにパフォーマンスを向上する

ここまでのデフォルトの実装では、連続しないメモリーアクセスを含むため、ベクトル化にあまり適していない構造体配列アルゴリズムを使ってきました。連続しないメモリーアクセスは、レイテンシーの長いギャザー/スキャッター命令を生成するため、効率良いベクトル化の妨げとなります。それにもかかわらず、インテル® Cilk™ Plus により優れたパフォーマンスが得られました。ベースライン実装を配列構造体 (SOA) に書き直すことで、ベクトル化しやすいユニットストライド方式 (連続した) のメモリーアクセスにより、さらにパフォーマンスを向上させることができます。この情報に基づいて、コンパイラーはレイテンシーの長いギャザー/スキャッター命令を生成しないで、より高速な線形ベクトル (メモリー) ロード/ストア命令 (インテル® SIMD でサポートされている movaps や movups など) を生成します。構造体配列 (AOS) 実装は、次のデータ構造を使用します。

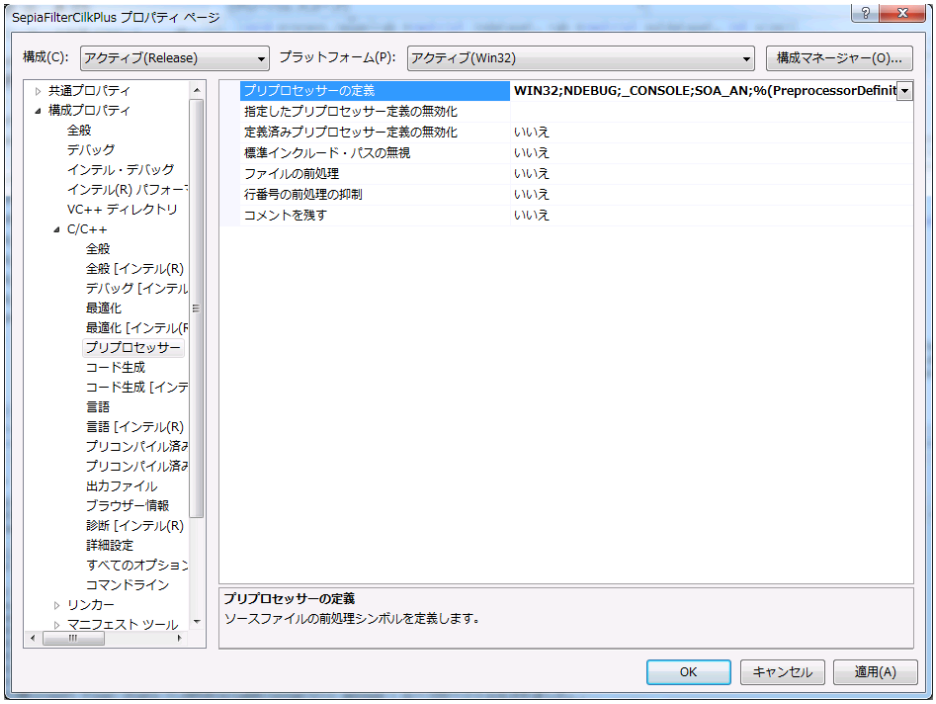
```
typedef struct {
    unsigned char blue;
    unsigned char green;
    unsigned char red;
} rgb;
```

配列構造体 (SOA) 実装は、次のデータ構造を使用します。

```
typedef struct {
    unsigned char *blue;
    unsigned char *green;
    unsigned char *red;
} SOA_rgb;
```

SOA を利用してパフォーマンスを向上させる方法は 2 つあります。1 つは配列表記を用いて SIMD 機能を活用し、もう 1 つは SIMD 機能とマルチスレッド化の両方を利用します。

サンプルコードでこのコード領域を有効にするには、次のように「SOA_AN」マクロを追加します。



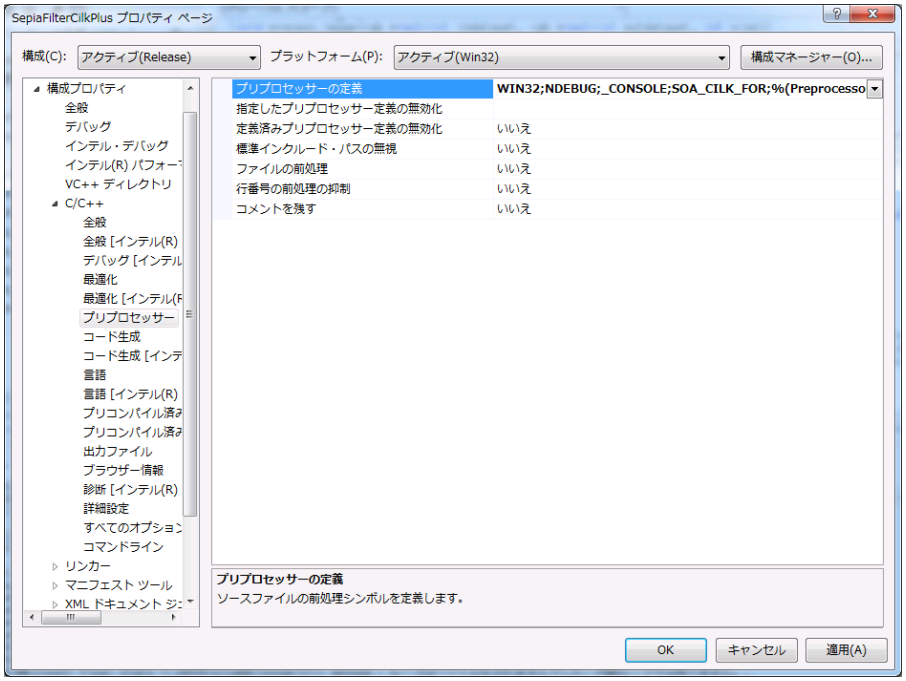
プロジェクトをリビルドし、コードをベクトル化します。

```
SepiaFilterCilkPlus.cpp(141): (列 2) リマーク: ループがベクトル化されました。
SepiaFilterCilkPlus.cpp(147): (列 2) リマーク: ループがベクトル化されました。
SepiaFilterCilkPlus.cpp(182): (列 3) リマーク: ループがベクトル化されました。
SepiaFilterCilkPlus.cpp(178): (列 2) 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。
SepiaFilterCilkPlus.cpp(210): (列 2): 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。
SepiaFilterCilkPlus.cpp(210): (列 2) 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。
```

配列表記のコードを含む process_image() 関数が呼び出されてベクトル化されます。「配列表記を利用してセピアフィルター・カーネルを実装する」セクションで述べた内容はすべて、このセクションにも当てはまります。ただし、使用するデータ構造は異なります (ここでは、ユニットストライド方式のメモリーアクセスをサポートするデータ構造を使います)。前出の AOS 実装と比べて、パフォーマンスは大幅に向上するでしょう。

Cilk_for (SOA) を指定してパフォーマンスを向上する

サンプルコードで、このコード領域を有効にするには、次のように「SOA_CILK_FOR」マクロを追加し、「for」ループを「cilk_for」ループに置換します。




```
#elif defined(SOA_CILK_FOR)
    starttime = __rdtsc();
    cilk_for(int i = 0; i < size_of_image; i++)
    {
        process_image(indata_SOA.blue[i], indata_SOA.green[i], indata_SOA.red[i], outdata_SOA.blue[i], outdata_SOA.green[i], outdata_SOA.red[i]);
    }
    endtime = __rdtsc();
```

プロジェクトをリビルドすると、配列表記バージョンの場合と同じベクトル化レポートが出力されます。ただし、このバージョンでは複数のスレッドにワークロードが分配され、複数のコアで実行されるため、AOS バージョンよりもパフォーマンスが向上します。

Cilk_for と配列表記の併用

cilk_for と配列表記を併用するには、配列を複数のセグメントに分割し、複数の cilk ワーカースレッドに分配する必要があります。しかし、そうすることで cilk ランタイムのヒューリスティックがオーバーライドされ、一般にパフォーマンスが低下します。このサンプルでは、特にこれが顕著です。cilk ランタイムにロードバランスを任せたほうが良いパフォーマンスが得られます。これを検証するため、前述のように「SOA_AN」マクロを用いて配列表記コード領域を有効にします。デフォルトでは、次に示すように SOA_AN コード領域に cilk_for はありません。また、num_of_seg = 1 を使用し、1 つのスレッドが配列全体を処理します。

```
175 #elif defined(SOA_AN)
176       int startindex, seg_size, num_of_seg = 1; //Variable num_of_seg is
177       starttime = __rdtsc();
178       for(int i = 0; i < num_of_seg; i++)
179       {
180           startindex = i*(size_of_image/num_of_seg);
181           seg_size = (size_of_image/num_of_seg);
182           process_image(indata_SOA, outdata_SOA, startindex, seg_size);
183       }
184       endtime = __rdtsc();
```

cilk_for と配列表記を併用するには、for ループを cilk_for に置換し、作成する配列セグメントの数を num_of_seg に設定します。num_of_seg を大きくするとパフォーマンスが低下します。これは、すべてのスレッドに分配するだけのワークがなく、オーバーヘッドが増えるためです。
cilk_for と配列表記を併用する場合は、ショートベクトルを利用することを推奨します。つまり、部分配列の長さがベクトルレジスターのサイズと同じか、またはその倍数になるようにします。そうすることで、ピーリング (データがアラインされている場合) とクリーンアップ・ループが不要なベクトル化が可能になります。

要素関数を利用してセピアフィルター・カーネルを実装する

インテル® Cilk™ Plus の要素関数は、スカラー引数または (コンパイラーが内部的に) 配列要素で並列に呼び出すことができる通常の関数です。要素関数を使わなければ、ループのベクトル化の妨げとなるループ内の関数呼び出しをベクトル化します。サンプルコードでは、コンパイラーはループ内の process_image() 関数への呼び出しをインライン展開して、ベクトル化できます。そのため、このサンプルコードで要素関数は必要なく、要素関数を利用してもパフォーマンスは変わりません。要素関数を使う場合は、次のように関数を宣言するだけです。

```
// process_image() 関数を要素関数として宣言する
_declspec(vector) void process_image(rgb &indataset, rgb &outdataset)
```

要素関数の詳細は、「参考文献」セクションの「Elemental functions: Writing data parallel code in C/C++ using Intel® Cilk™ Plus (要素関数: インテル® Cilk™ Plus を使用した C/C++ でのデータ並列コードの記述)」(英語) を参照してください。

参考文献

SIMD ベクトル化、インテル® コンパイラーの自動ベクトル化、要素関数、インテル® Cilk™ Plus 構造の使用例については、以下の情報を参照してください。

「A Guide to Autovectorization Using the Intel® C++ Compilers (インテル® C++ コンパイラーによる自動ベクトル化ガイド)」(英語)
「ループをベクトル化するための条件」
「#pragma SIMD を使用してループをベクトル化するための条件」
「インテル® Cilk™ Plus の配列表記 (アレイ・ノーテーション) 入門」
「配列表記 (アレイ・ノーテーション) による SIMD 並列処理」
「Intel® Cilk™ Plus Language Extension Specification (インテル® Cilk™ Plus の言語拡張仕様)」(英語)
「Elemental functions: Writing data parallel code in C/C++ using Intel® Cilk™ Plus (要素関数: インテル® Cilk™ Plus を使用した C/C++ でのデータ並列コードの記述)」(英語)

「インテル® Cilk™ Plus を使用したデータとスレッドの並列化」

最適化に関する注意事項

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