インテル® TBB を使用したインテル® IPP のイメージリサイズのスレッド化

この記事は、インテル® デベロッパー・ゾーンに公開されている「Threading Intel® Integrated Performance Primitives Image Resize with Intel® Threading Building Blocks」の日本語参考訳です。

はじめに

インテル® インテグレーテッド・パフォーマンス・プリミティブ (インテル® IPP) ライブラリーは、ベクトル化されたさまざまな信号/イメージ処理関数を提供します。インテル® スレッディング・ビルディング・ブロック (インテル® TBB) は、C++ プログラムで並列化を表現する単純で強力な抽象化を提供します。この記事は、これらのツールを併用し、ベクトル化とスレッド化の利点を組み合わせてイメージリサイズのパフォーマンスを向上させます。

インテル® IPP 8.2 から、パフォーマンスとほかのスレッドモデルとの互換性のため、マルチスレッド (内部的にスレッド化された) ライブラリーは非推奨になりました。しかし、マルチスレッド・プログラミングは現在の主流であり、インテル® TBB のように豊富なスレッド化ツールのエコシステムも存在します。ほとんどの場合、アプリケーション・レベル (外部/プリミティブの上) でのスレッド化は多くの利点をもたらします。多くのアプリケーションはすでに独自のスレッドモデルを利用していますが、アプリケーション・レベル/外部スレッド化により開発者は高い柔軟性と制御能力を手にすることができます。少しだけ労力をかけてアプリケーションをスレッド化することで、内部スレッド化と同じまたはそれ以上のパフォーマンスを達成できます。また、複数スレッド間のローカル・キャッシュ・データの再利用のように、より高度な最適化手法を行うことも可能になります。これが、インテル® IPP の最新リリースで内部スレッド化を非推奨にした大きな理由です。

***parallel_for* 入門**

インテル® TBB の *parallel_for* テンプレートは、簡単に並列処理を開始する、インテル® TBB で最も多く使用される機能の 1 つです。アプリケーションの for() ループにおいて、各反復を独立して行うことができ、実行の順序が重要ではない場合、インテル® TBB の *parallel_for* は、そのケースに最適なスレッドプールとスケジューラーなどの詳細な制御を行います。ユーザーは、個別のスレッドまたはコアで実行する分割手法とコードを提供します。さらに高度なアプローチも可能ですが、この記事とサンプルコードの目的は、あらゆる状況で最良のスレッド化を行うことではなく、単純な起点を示すことです。

インテル® TBB の *parallel_for* の引数は 2 つまたは 3 つです。

parallel_for (range, body, optional partitioner)

単純なラインベースの分割では、**range** を次のように指定します。

blocked_range<int>(begin, end, grainsize)

これはイメージが処理しているライン情報を各スレッドに提供し、range の begin から end までを grainsize のチャンクで自動的に分割します。インテル® TBB では、範囲が均等に分割されない場合、grainsize が自動的に調整されるため、任意のサイズを提供することができます。

body は並列化するコード本体です。これは個別に (クラスの一部として含めて) 実装することができます。これは単純なケース用ですが、ラムダ式を使用すると便利な場合があります。ラムダを適用する場合、関数の本体全体が *parallel_for* 呼び出しの一部となります。この匿名関数に渡す変数は括弧 [*alg, pSrc, pDst, stridesrc_8u, …*] 内にリストされ、範囲情報は *blocked_range<int>& range* で渡されます。

これは、さまざまな問題に適用できる、一般的なスレッド化の抽象です。配列演算のような単純なループを含む *parallel_for* の例を示したサンプルは多く存在します。リサイズの変更も同じパターンに従います。

インテル® IPP リサイズの外部並列化

スレッド化されたリサイズは任意の形のタイルに分割することができます。しかし、タイルがイメージの幅である場合、行のグループを使用すると便利です。

各スレッドは、イメージバッファーのオフセットを決定するために *range.begin*()、*range.size*() などを問い合わせることができます。注: この実装では、イメージ全体がメモリーの 1 つのバッファー内で利用可能であると仮定しています。

インテル® IPP 7.1 以降で提供されるイメージリサイズ関数には多くの利点があります。

- IppiResizeSpec は入出力解像度の組み合わせに基づいて事前に計算した係数を保持します。それらの係数を再計算することなく、複数のリサイズを完了できます。
- 補間手法ごとに別の関数が用意されています。
- スタティック・リンクで実行ファイルのサイズが大幅に小さくなります。
- スレッド化およびタイル化されたイメージ処理のサポートが向上しました。
- 詳細は、「インテル® IPP 7.1 のリサイズの変更」(英語) を参照してください。

リサイズを開始する前に、オフセット (各スレッドの領域の開始点を計算するためにソースポインターとデスティネーション・ポインターに追加するバイト数) を計算しておく必要があります。インテル® IPP には、この計算を行う便利な関数が用意されています。

ippiResizeGetSrcOffset

この関数は、デスティネーション・イメージの位置と対応するソースイメージのオフセット/位置を計算します。この場合、デスティネーション・オフセットはスレッドのブロック範囲の始めです。

この関数から返る値を基に、各スレッドの現在の作業単位のソースアドレスとデスティネーション・アドレスを計算するのは簡単です。

```
pSrcT=pSrc+(srcOffset.y*stridesrc_8u);
pDstT=pDst+(dstOffset.y*stridedst_8u);
```

これらは、次のようにリサイズ関数で使用できます。

```
ippiResizeLanczos_8u_C1R(pSrcT, stridesrc_8u, pDstT, stridedst_8u, dstOffset, dstSizeT,
ippBorderRepl, 0, pSpec, localBuffer);
```

これは、スレッドがイメージのラインのサブセットでどのように動作しているかを示します。ソースバッファーとデスティネーション・バッファーの先頭を指定する代わりに、*pSrcT* と *pDstT* は、スレッドが動作している

領域の先頭を示します。各スレッド領域の高さは *dstSizeT* によりリサイズに渡されます。もちろん、1 スレッドの特別なケースでは、これらの値はスレッド化されていない実装の値と同じです。

呼び出しのもう 1 つの違いは、各スレッドが独自のリサイズを同時に行っているため、同じ作業バッファーをすべてのスレッドに使用できないことです。各スレッドに事前にバッファーを割り当てて、さらに効率を高めることもできますが、単純にするため、作業バッファーは scalable_aligned_malloc でラムダ関数内に割り当てました。

次のサンプルコードは、*parallel_for* ラムダ関数内のリサイズをセットアップする方法と、これまで説明した概念をともに実装する方法を示したものです。

ソースコードはここから (英語) ダウンロードできます。このサンプルコードをダウンロードすると、エンド・ユーザー・ライセンス契約書 (英語) に同意したものとします。

```
parallel_for( blocked_range<int>( 0, pnminfo_dst.imgsize.height, grainsize ),
    [pSrc, pDst, stridesrc_8u, stridedst_8u, pnminfo_src,
    pnminfo_dst, bufSize, pSpec]( const blocked_range<int>& range )
    {
        Ipp8u *pSrcT,*pDstT;
        IppiPoint srcOffset = {0, 0};
        IppiPoint dstOffset = {0, 0};

        // サイズ変更領域の幅はイメージの幅
        // 高さは range.size() を使用して TBB が設定
        IppiSize  dstSizeT = {pnminfo_dst.imgsize.width,(int)range.size()};

        // スレッドのサイズ変更用の作業バッファーをセットアップする
        Ipp32s localBufSize=0;
        ippiResizeGetBufferSize_8u( pSpec, dstSizeT,
            pnminfo_dst.nChannels, &localBufSize );

        Ipp8u *localBuffer =
            (Ipp8u*)scalable_aligned_malloc( localBufSize*sizeof(Ipp8u), 32);

        // デスティネーション・オフセットを指定してソースイメージの
        // オフセットを計算する
        dstOffset.y=range.begin();
        ippiResizeGetSrcOffset_8u(pSpec,dstOffset,&srcOffset);

        // このスレッドが読み書きするバッファー内の先頭のポインター
        pSrcT=pSrc+(srcOffset.y*stridesrc_8u);
        pDstT=pDst+(dstOffset.y*stridedst_8u);

        // グレースケールまたはカラーのサイズ変更を行う
        switch (pnminfo_dst.nChannels)
        {
        case 1: ippiResizeLanczos_8u_C1R(pSrcT,stridesrc_8u,pDstT,stridedst_8u,
            dstOffset,dstSizeT,ippBorderRepl, 0, pSpec,localBuffer); break;
        case 3: ippiResizeLanczos_8u_C3R(pSrcT,stridesrc_8u,pDstT,stridedst_8u,
            dstOffset,dstSizeT,ippBorderRepl, 0, pSpec,localBuffer); break;
        default:break; // 1 および 3 チャネルイメージのみ
        }

        scalable_aligned_free((void*) localBuffer);
    });
```

ご覧のとおり、スレッド化された実装はシングルスレッド版とよく似ています。主な違いは、複数のスレッドで処理するためにインテル® TBB によってイメージが分割され、各スレッドがイメージラインのグループを担当していることです。これは、複数のコアまたはスレッドにイメージリサイズのタスクを分割する比較的簡単な方法です。

まとめ

インテル® IPP は、SIMD 最適化関数を提供します。インテル® TBB は、インテル® IPP アプリケーションのスレッド化を制御する単純で強力な方法を提供します。インテル® IPP とインテル® TBB を併用すると、複数のコアに作業を効率的に分割して、各コアで適切にベクトル化されたパフォーマンスを得ることができます。外部スレッド化を利用した詳細な制御を行うことで、より効率的な処理が可能になり、より優れたパフォーマンスを達成できます。

サンプルコード: ほかのインテル® IPP サンプルコードと同様に、サンプルコードをダウンロードすると、エンド・ユーザー・ライセンス契約書 (英語) に同意したものとします。

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。