XAPP793 (v1.0) 2012 年 9 月 20 日

Vivado HLS ツールを使用したビデオ処理用メモリ構造のインプリメント

著者 : Fernando Martinez Vallina

概要

このアプリケーション ノートでは、Vivado™ 高位合成 (HLS) ツールを使用してビデオ/イメージ処理アルゴリズムをインプリメントする際の主な注意点について説明します。このようなアルゴリズムの大部分は計算量が多いため、HLS ツールを用いたハードウェア インプリメンテーションに適しています。このアプリケーション ノートでは、HLS ツールでビデオ アルゴリズムをインプリメントする場合の同期信号の処理およびメモリ アーキテクチャ関する基本事項について説明します。実際のユーザー アルゴリズムで実行される計算処理にかかわらず、この種のアプリケーションでは多くのメモリ アクセスが発生します。メモリ アーキテクチャをアルゴリズム内でどのように記述するかは、全体的なシステム性能とハードウェア リソースの使用率に直接影響を与えます。このアプリケーション ノートで推奨するメモリ バッファー アーキテクチャは、すべてのザイリンクス FPGA をターゲットにした Vivado HLS デザインに適用できます。

はじめに

ビデオ/イメージ処理ブロックは、Vivado HLS ツールを使用してザイリンクス FPGA にインプリメントできます。HLS ツールを使用することで、1 つのアルゴリズム コードからさまざまな性能レベルや FPGA をターゲットにしたアクセラレータが作成可能です。ビデオ/イメージ処理でのターゲット性能は、イメージの最大解像度や fps (フレーム/秒) で指定できます。

このアプリケーション ノートでは、HLS ツールを用いたビデオ処理 IP の作成に関し、次について説明します。

- ビデオ フレーム フォーマットの基礎と C/C++ での記述に関する概要
- 処理スループットを高めるためのメモリ アーキテクチャの記述方法

ビデオ処理 IP コアは、HLS ツールで生成した API (アプリケーション プログラム インターフェイス) を使用してプロセッサから制御できます。Vivado HLS IP のプロセッサ制御の詳細は、『Vivado HLS デザインにおけるプロセッサ制御』[参照 1] を参照してください。

プログラミング環境について

このアプリケーション ノートは、ザイリンクスの Vivado HLS ツールとビデオ IP ソリューションに関する全般的な知識を持つ設計者向けとなっています。ツールの詳細は、『Vivado Design Suite ユーザー ガイド : 高位合成』[参照 2] を参照してください。また、その入門ビデオは、japan.xilinx.com/training/vivado で紹介しています。ザイリンクスが提供するビデオ IP の詳細は、japan.xilinx.com を参照してください。

ビデオ処理の基礎

ビデオ処理は、演算粒度の異なる次の 3 つのレベルでインプリメントできます。

- ピクセル
- マクロ ブロック
- フレーム

どの演算粒度が最適かはアルゴリズムによって異なりますが、いずれの場合もビデオ処理パイプラインの設計で最初に問題となるのは、同期信号をどのように扱うかという点です。基本的には、すべてのビデオ フレームに次の 3 つの領域があります。

- 垂直ブランキング

- 水平ブランキング
- アクティブ

アクティブ領域とは、フレーム内でイメージ ピクセルが存在する場所です。ユーザー アルゴリズムは、データのこの部分を対象に動作します。設計者がビデオ処理について考える場合のデータとは、暗黙的にこの部分のデータを指します。水平および垂直ブランキング領域は、ほかのビデオ機器がコンテンツを正しく検出して表示するために必要な同期スペースを提供します。つまり、ビデオ フレームの基本構造は図 1 のようになります。

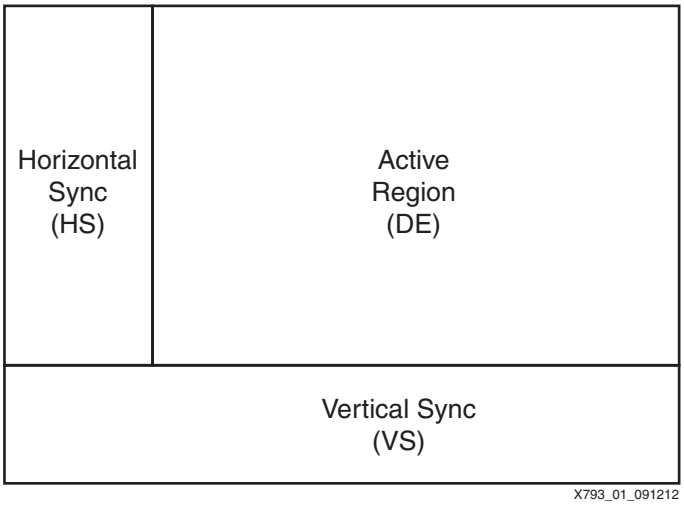

図 1 : 基本的なビデオ フレームのフォーマット

適切にフォーマットされたビデオ ストリームでは、図 1 に示した同期信号 (VS、HS、DE) は必須要素です。これらの信号はコアのユーザー アルゴリズムの一部ではありませんが、その結果がほかのビデオ処理 IP あるいはビデオ機器で正しく受け入れられるように適切に扱う必要があります。HLS ツールを使用したデザインでは、システム統合レベルまたは Vivado HLS ツールのコード内でこれらの信号を扱うことができます。

Vivado HLS ツールで同期信号を処理するには、デザインの C/C++ コードでこれらの信号を明示的に使用する必要があります。Vivado HLS ツールは汎用のアルゴリズム合成ツールであり、ビデオ処理と一般的な計算の違いについての情報は持っていません。したがって、Vivado HLS ツール用のユーザー コードは次に示す例のような形式とする必要があります。

```
void video_proc(input_video, output_video){

  for(i=0; i < MAX_FRAME_SIZE; i++){
    if(VS){
      ......
    }
    if(HS){
      ......
    }
    if(DE){
      //Pixel Processing Algorithm
    }
    ....
  }
}
```

このサンプル コードからわかるように、コアのユーザー アルゴリズムは同期信号処理を組み込むと複雑になってしまいます。つまり、同期信号処理を加えるとアルゴリズムの修正が難しくなります。また、これまでビデオ処理アルゴリズムを C/C++ で記述する場合に一般的だった、行と列に対するループが使用できなくなります。

ザイリンクスは、Vivado HLS ツールでインプリメントするデザインにおける同期信号の処理を不要にするための IP ソリューションを提供しています。これらの IP コアは「Video In to AXI4-Stream」[参照 3] [参照 4] および「AXI4-Stream to Video Out」[参照 5] として提供され、ピクセル ストリームから有効なビデオ ストリームへのフォーマット変換を行います。これらのコアをシステム統合レベルで使用すると、ユーザーは次のサンプル コードに示すような行と列の形式でビデオ処理アルゴリズムを記述できます。

```
void video_proc(input_pixels, output_pixels, height, width){

  for(i=0; i<height; i++){
    for(j=0; j<width; j++){
      //Pixel processing algorithm
    }
  }
}
```

ビデオ処理アルゴリズムを Vivado HLS ツールで記述する方法としては、このサンプル コードのようなアプローチが推奨されます。ザイリンクスのビデオ IP を使用した推奨アプローチでのハードウェアのブロック図は、図 2 のようになります。

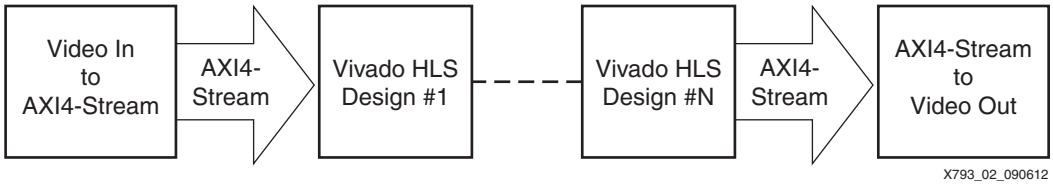

図 2 : Vivado HLS によるビデオ処理パイプラインのブロック図

図 2 のブロック図は、Vivado HLS ツールでビデオ処理パイプラインを構築し、IP で同期信号を処理する理想的な方法を示しています。これ以外にも、Vivado HLS ツールを使用したビデオ処理ではユーザー アルゴリズム内でのメモリ バッファーのインスタンシエートおよび処理が重要になります。

メモリ アーキテクチャ

メモリ バッファーはビデオ処理アルゴリズムに欠かせない基本要素です。アプリケーションが現在のピクセルを処理する際に必要な時間的、空間的データ コンテクストはメモリ構造に格納されます。ビデオおよび DSP アプリケーションのメモリとしては、シフト レジスタ、メモリ ウィンドウ、ライン バッファーの 3 つが一般的です。これらメモリ構造はすべて、Vivado HLS ツールによって生成されるハードウェアのレイテンシ、計算順序、機能の正しさに影響を与えます。

中でも最も大きいのが、デザインにおける機能の正しさ対する影響です。Vivado HLS ツールは新しいメモリをユーザー コードに自動的には挿入しません。したがって、RTL を生成するコードにユーザーがシフト レジスタ、メモリ ウィンドウ、ライン バッファーの動作を明示的にコーディングする必要があります。これで初めて、Vivado HLS ツールはこれらのメモリ構造を RTL で作成します。

シフト レジスタ

シフト レジスタは、DSP 処理で使用されるメモリ構造として最もシンプルで、広く使用されています。シフト レジスタの基本的な考え方は、ストリーミング インターフェイスから入力されたサンプルを一時的に格納しておくための一次元データ バッファーを提供するというものです。サンプルをどれだけの期間シフト レジスタに格納しておくかは、インプリメントするアルゴリズムによって決まります。シフト レジスタを使用してインプリメントできる代表的な DSP アルゴリズムの例として、有限インパルス応答 (FIR) フィルターがあります。図 3 に、シフト レジスタを使用した FIR フィルターの概念図を示します。

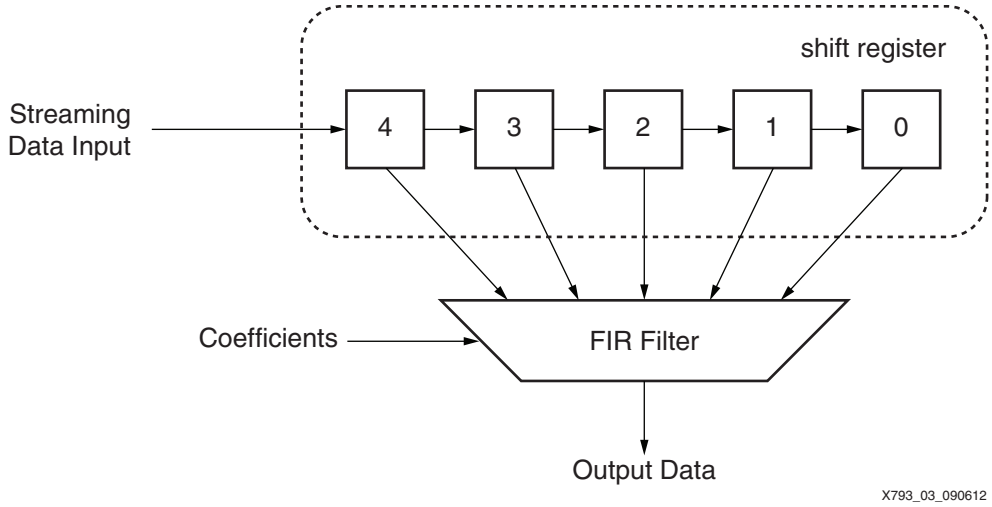

図 3 : FIR フィルターの概念図

図 3 に示したように、ストリーミング インターフェイスからの各データ要素はシフト レジスタの 4 の位置に格納され、FIR フィルターによってさらに 4 回再利用されてから破棄されます。その性質上、ストリーミング データ インターフェイスからは個々のデータ サンプルは一度しか生成されません。したがって、データ サンプルは使用する前に保存しておく必要があります。

Vivado HLS ツールでシフト レジスタを宣言する最もシンプルで直接的な方法は、配列を使用することです。図 3 の例では、入力ストリーミング インターフェイスをどのように定義するかによって、次の 2 通りの方法でシフト レジスタを記述できます。

```
int A[5];
int new_data[5];
int i;
for(i = 0; i < 4; i++){
#pragma HLS unroll
  A[i] = A[i+1];
}
A[4] = new_data[k];
```

```
int A[5];
hls::stream<int> new_data;
for(i = 0; i < 4; i++){
#pragma HLS unroll
  A[i] = A[i+1];
}
A[4] = new_data.read();
```

このコード例は、配列 A からシフト レジスタを作成する方法を示しています。この中の for ループによって、シフト レジスタのエレメント 4 からエレメント 0 へデータが移動します。ハードウェアでは、シフト レジスタ内のすべてのエレメントは毎クロック サイクルで同時に移動します。この理想的なハードウェア特性は、配列 A におけるデータ移動を指定しているループを完全に展開することで Vivado HLS ツールに指示できます。Vivado HLS ツールでループを展開する方法はいくつかありますが、上記の例では次を使用しています。

```
#pragma HLS unroll
```

pragma をコード内に直接用いることで、性能やインプリメンテーションのターゲットにかかわらず、配列 A がシフト レジスタとして動作するように設計者がが明示的に指定できます。これにより、Vivado HLS ツールで性能に基づいた展開ストラテジを使用する場合に生じる曖昧さが排除されます。ユーザー コード内の pragma は、すべてのインプリメンテーションで自動的に実行されます。

上に挙げた 2 つのコード例の違いは、入力インターフェイスから new_data を取得する方法にあります。左側のコードは、配列 new_data の k 番目の位置からデータを取得することにより、暗黙的なストリーミングを使用します。この方法の利点は、元の C/C++ テストベンチ コードに修正の必要がないことです。また、コード開発時にシフト レジスタをアルゴリズムに挿入して問題が発生しても、配列 new_data にアクセスして問題をデバッグできます。

すべての動作が正しいことを確認後、次のいずれかのコマンドを使用して new_data をストリーミング インターフェイスに変換できます。

```
set_directive_interface -mode ap_fifo foo new_data  (Vivado HLS directive)
```

または

```
#pragma HLS interface ap_fifo port=new_data  (Vivado HLS pragma)
```

ストリーミング インターフェイスの場合、ネイティブなハンドシェイク/ストリーミング プロトコルとして Vivado HLS ツールは ap_fifo および ap_hs をサポートしています。これらのモードの詳細は、『Vivado Design Suite ユーザー ガイド : 高位合成』[参照 2] を参照してください。

暗黙的ストリーミングを使用する場合は、C と Vivado HLS ツールで生成した RTL シミュレーション モデルの概念的な違いに注意する必要があります。配列からストリーミング インターフェイスへの new_data の変換は、Vivado HLS ツールでデザインを合成した後にのみ RTL シミュレーション モデルで可視化されます。C からの出力と RTL シミュレーション モデルが一致するのは、逐次かつ線形順序で new_data にアクセスした場合のみです。したがって、配列へのアクセスがこのように行われるように配慮して設計する必要があります。

これに対し、右側のコード例は hls::stream クラスを使用した明示的ストリーミングを用いています。hls::stream クラスを使用することで、ストリーミング FIFO 順の読み出し/書き込み動作が C シミュレーション レベルで強制されます。また、このクラスを使用すると自動的に RTL レベルでストリーミング インターフェイスが生成されます。hls::stream の詳細は、『Vivado Design Suite ユーザー ガイド : 高位合成』を参照してください。暗黙的ストリーミングを用いてストリーミング I/O をアルゴリズムに追加した場合は、次に明示的ストリーミングによってこの動作をロックすることを推奨します。暗黙的ストリーミングから明示的ストリーミングに変更するとインターフェイス レベルでコーディングの変更が不要になり、アルゴリズムの機能の正しさに対して影響を与えることがなります。

メモリ ウィンドウ

ビデオ/イメージ処理において、メモリ ウィンドウとは中心点のピクセル P に隣接する N ピクセルと定義されます。メモリ ウィンドウは、複数のシフト レジスタを組み合わせて二次元のデータ記憶素子を構成したものと捉えることもできます。次のような理由から、この種のメモリは通常フリップフロップとしてインプリメントされます。

- データ要素の数が少ない。格納するのは、P の特性を計算するのに必要なピクセルのみ (例 : エッジ検出で使用する 3x3 メモリ ウィンドウなど)。
- P の値を計算する際、隣接するすべてのピクセルの値も同時に利用できなければならない。

C/C++ でメモリ ウィンドウを定義する方法として最も基本的なのは、二次元配列です。たとえば 3x3 のメモリ ウィンドウ B は次のように定義できます。

```
int B[3][3];
```

メモリ ウィンドウには、すべてのデータ要素を同時に利用できるという特長があります。Vivado HLS では、配列を個別の要素に分割することでこれを実現でき、分割しないと配列はブロック RAM としてインプリメントされます。配列を分割するには、指示子または pragma を使用します。メモリ ウィンドウ B を分割する場合、次のいずれかのコマンドを実行します。

```
set_directive_array_partition -type complete -dim 0 foo B (Vivado HLS
directive)
```

または

```
#pragma HLS ARRAY_PARTITION variable=B complete dim=0 (Vivado HLS pragma)
```

説明

- complete : 配列を次元ごとに個別の要素に分割します。
- dim = 0 : 配列のすべての次元を対象に実行される分割コマンドです。

指示子またはプラグマのいずれかの方法で配列を分割すると、B[3][3] は RTL レベルで 9 個の独立したレジスタに分解されます。

メモリ ウィンドウにおけるデータ移動も、シフト レジスタと同様に扱うことができます。ピクセル フレーム内での垂直方向の移動には、メモリ ウィンドウへのデータ供給に用いられるライン バッファーのみを使用するように制限できます。このように単純化した状態を前提とすると、ウィンドウの水平移動は次のように記述できます。

```
for(i = 0; i < 3; i++){
#pragma HLS unroll
  B[i][0] = B[i][1];
  B[i][1] = B[i][2];
}
```

次に、ライン バッファー D からメモリ ウィンドウ B へデータをコピーする方法の例を示します。

```
for(i = 0; i < 3; i++){
#pragma HLS unroll
  B[i][2] = D[i][col];
}
```

ライン バッファー

ライン バッファーは数行のピクセル データを格納できる多次元のシフト レジスタです。このバッファーは一般にオフチップの DRAM メモリとの通信にレイテンシが生じないようにするためにブロック RAM としてインプリメントされます。また、ライン バッファーは読み出しと書き込みのアクセスを同時に行う必要がありますが、これにはブロック RAM のデュアルポートが活用できます。メモリ ウィンドウもライン バッファーの一種ですが、ビデオ/イメージ処理アルゴリズムではライン バッファーは直接使用できません。図 4 に、ピクセル データが入力されてからアルゴリズム計算カーネルに到達するまでの経路を示します。

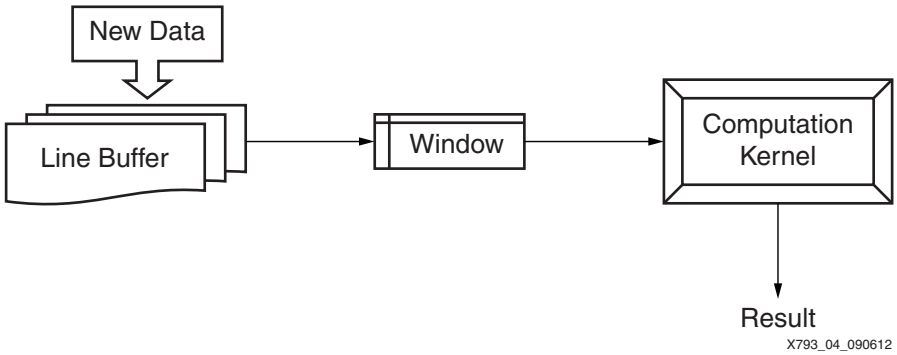

図 4 : ピクセル データが計算カーネルに到達するまでのメモリ経路

ウィンドウの場合と同様に、ライン バッファーも C/C++ で多次元配列として宣言します。ライン バッファーの高さを決定する簡単な目安として、メモリ ウィンドウの高さに揃えるようにします。たとえば、FPGA 内のメモリ ウィンドウ B に対応するライン バッファーは、次のように宣言します。

```
rgb_pixel D[3][MAX_COLS]
```

バッファーの格納行数を最大値の 3 行として宣言すると FPGA リソースの使用効率に無駄が生じますが、C/C++ でのコーディングは最も容易です。ウィンドウ処理アルゴリズムは N 行すべてのデータがなくても計算を開始できるため、エリアの使用効率が悪くなります。N-1 行と余りのピクセルではなく、N 行すべてを格納すると、ブロック RAM 数個分のオーバーヘッドが生じます。デバイスの使用率によっては、このアプリケーション ノートで紹介した C コードの例をさらに最適化して使用するブロック RAM を減らすことができます。

ライン バッファーを使用する際は、シフト レジスタと同様にデータ移動を指定するのに加え、アルゴリズム内の処理ループの境界について再考する必要があります。アルゴリズムでは、ライン バッファー

の 2 つの影響を考慮に入れます。1 つは、最初の出力値を計算するのに十分なデータをライン バッファーに格納するために必要な時間です。もう 1 つは、入力サンプルがなくなってすべての出力サンプルが生成されてもアルゴリズムの反復実行を続けなければならないという点です。

先に挙げたライン バッファー D の例では、アルゴリズムの計算ループを次のように拡張します。

```
for(row = 0; row < MAX_ROW; row++){
  for(col = 0; col < MAX_COL; col++){
    ....
  }
}
```

これを次のように拡張します。

```
For(row = 0; row < MAX_ROW+1; row++){
  for(col = 0; col < MAX_COL+1; col++){
    ....
  }
}
```

境界条件を 1 だけ拡張しているのは、ライン バッファーの挿入によって生じた入力データと出力データのオフセット分です。入力データ セットの境界を越えて処理ループを拡張したため、ユーザー側でゲーティング条件を追加してアルゴリズムの機能の正しさを維持する必要があります。

最初に示したコード例では、アルゴリズムへの入力は元々の境界によって制限されています。2 番目のコード例のデータ フローでは、すべての入力データは最初にライン バッファーへ書き込まれます。このため、ライン バッファーへの入力データの書き込みに対してゲーティング条件を適用する必要があります。次に、入力データのゲーティング条件の例を示します。

```
if(col < MAX_COL & row < MAX_ROW){
  D[2][col] = new_data[k];
}
```

結果を出力インターフェイスに書き込む際は、ライン バッファーの遅延も考慮して遅延を加える必要があります。次に、出力に対するゲーティング条件の例を示します。

```
if(col > 0 & row > 0){
  output_pixel[k] = result;
}
```

ライン バッファー内のデータ移動は垂直シフトで、新しい行が追加されると古い行が破棄されます。ライン バッファーでの列方向の水平移動は、col に対する内側の処理ループの結果として自動的に実行されます。ライン バッファーの垂直シフトは、次のようにコーディングできます。

```
if(col < MAX_COL){
  D[0][col] = D[1][col];
  D[1][col] = D[2][col];
}
```

まとめ

メモリ アーキテクチャおよび使用モデルをアルゴリズムの一部としてコーディングすることにより、Vivado HLS ツールで生成されるハードウェアの種類をユーザーが完全に制御できるようになります。これによってユーザーは、それぞれのメモリ レイアウトがエリアと性能にどのような影響を与えるかを短時間で検討できます。このアプリケーション ノートで紹介したコード例およびツール設定を利用することで、ビデオ処理アルゴリズムの実装で一般的に使用されるメモリ構造が Vivado HLS ツールでインプリメント可能になります。

参考資料

このアプリケーション ノートでは、次の参考資料が使用されています。

1. XAPP745 : 『Vivado HLS デザインにおけるプロセッサ制御』
2. UG902 : 『Vivado Design Suite ユーザー ガイド : 高位合成』
3. PG043 : 『LogiCORE IP Video In to AXI4-Stream v2.00.a』
4. UG761 : 『AXI リファレンス ガイド』
5. PG044 : 『LogiCORE IP AXI4-Stream to Video Out v2.00.a』

改訂履歴

次の表に、この文書の改訂履歴を示します。

日付	バージョン	内容
2012 年 9 月 20 日	1.0	初版リリース

Notice of Disclaimer