Xcell journal
SOFTWARE

SOFTWARE SOLUTIONS FOR
A PROGRAMMABLE WORLD

2号 2016

C/C++ を使用して プログラマブル ロジックに 画像処理をオフロード

Zynq SoC で AES 暗号化の
アクセラレーションに
SDSoC を利用

SDNet で NTT の Lagopus
FPGA リプログラマブル
プラットホームの作成を支援

NI 社が Vivado ツールで、
マシンビジョンのコンセプトから
開発までを支援

PLDA 社の QuickPlay 高レベル
ワークフローにより、効果的な
FPGA アプリケーションを作成

XILINX
ALL PROGRAMMABLE.

japan.xilinx.com/xcell

Xcell SOFTWARE journal

発行人	Mike Santarini mike.santarini@xilinx.com +1-408-626-5981
編集	Diana Scheben
アートディレクター	Scott Blair
デザイン/制作	Teie, Gelwicks & Assoc.
日本語版統括	神保 直弘 naohiro.jinbo@xilinx.com
制作進行	周藤 智子 tomoko.suto@xilinx.com
日本語版 制作・広告	有限会社エイ・シー・シー

Xcell Software Jour nal 日本語版 2 号

2016 年 1 月 28 日発行

Xilinx, Inc
2100 Logic Drive
San Jose, CA 95124-3400

ザイリンクス株式会社
〒141-0032
東京都品川区大崎 1-2-2
アートヴィレッジ大崎セントラルタワー 4F

japan.xilinx.com/xcell/

Letter from the Publisher

一世代先をリードする Zynq MPSoC

エンベデッド ソフトウェアの開発に携わる皆様は、最適化されたエンベデッド システムを開発するために、どのようなシステム ハードウェア リソースを利用できるかをよく理解している必要があります。アプリケーション ソフトウェアの開発担当者にとって、システム リソースに関する詳細な知識はそれほど重要ではありませんが、コードのパフォーマンス向上に役立つハードウェア リソースのオプションを理解しておくことは、間違いなく有益と言えます。

この目的のため、ザイリンクスは 2015 年第 3 四半期にシリコン開発史における重要なマイルストーンを達成しました。9 月下旬に、ザイリンクスは数々の受賞歴がある Zynq-7000 All Programmable SoC をさらに進化させた Zynq® UltraScale+™ MPSoC (ビデオを参照) の最初のサンプル出荷を発表しました。Zynq SoC がデュアルコア ARM® Cortex™ A-9 プロセッシング システムとプログラマブル ロジックおよびオンボード ペリフェラル コントローラーを接続してワンチップ SoC に搭載した製品であるのに対し、Zynq UltraScale+ MPSoC は、SoC で利用可能な処理能力をさらに拡張し、合計 7 個のプロセッサ (64 ビット クワッドコア ARM Cortex-A53 プロセッサおよびデュアルコア ARM Cortex-R5 リアルタイム プロセッサと ARM Mali™-400 MP GPU)、H.265/264 ビデオ コーデック、システムの電力効率を最適化する先進の動的電力管理ユニット、コンフィギュレーション セキュリティ ユニット、DDR4/LPDDR4 メモリ インターフェイスのサポート、大容量のオンチップ プログラマブル ロジックを集積した製品です。

2011 年のリリース以来、ザイリンクスの顧客が Zynq SoC を活用して開発してきたイノベーションの数々は、まさに注目すべきものです。ここではハードウェアの詳細に踏み込んだ説明は控えますが、Zynq SoC デバイス独自の特長は、プロセッサとオンチップ プログラマブル ロジック間の 3,000 を超える接続にあります。これらの多数の接続リソースにより、FPGA ロジックにインプリメントされた機能とプロセッサ間の通信は、2 チップ構成やシステム イン パッケージ構成のデバイスよりもはるかに高速化されています。これにより、従来は実現不可能だったシステムの開発も可能となります。Zynq SoC のリリース以来、顧客各社は、無線通信から宇宙/防衛分野に至るまで、ザイリンクスが製品を供給しているほぼすべての市場で Zynq SoC ベースのイノベーションを実現してきました。

これらのイノベーションの多くは、FPGA エンジニアリング チームがハードウェア設計ツールのザイリンクス Vivado® Design Suite で開発したものです。2015 年前半、ザイリンクスは、C/C++/OpenCL™ ベースの SDx™ 開発環境 (SDSoC™、SDAccel™、SDNet™) を発表し、設計効率の飛躍的な向上を実現しました。SDSoC は Zynq SoC の設計、SDAccel は FPGA による処理の高速化、SDNet はソフトウェア定義ネットワーキング システムの開発に使用されます。SDSoC 開発環境は、リリースからまだ日が浅いにもかかわらず、既に従来の FPGA の専門技術者に加え、新規ユーザー (エンベデッド ソフトウェア開発者) にも新たな機会を提供しています。まず C または C++ でシステムのシステム レベル表現を作成し、次に SDSoC 環境により実行速度の遅いコード セグメントを特定し、アクセラレーションのために FPGA ロジックにオフロードすることが可能となり、さらなるイノベーションの機会が生まれています。

今後、SDSoC 環境と Zynq UltraScale+ MPSoC ハードウェアの組み合わせにより、ますます多くのザイリンクス ユーザーが、さらに多くの注目すべきシステム イノベーションを生み出していくことに間違いはありません。

今回の Xcell Software Journal 日本語版 2 号には、SDSoC 環境および SDNet 環境で、これまでにないレベルのイノベーションをどのように実現できるかに関する記事が掲載されています。本号をぜひともお読みになり、新しい開発環境の活用にお役立てください。もちろん本誌に寄稿して、各社の開発者と設計体験を共有していただければ幸いです。

— Mike Santarini
発行人

CONTENTS

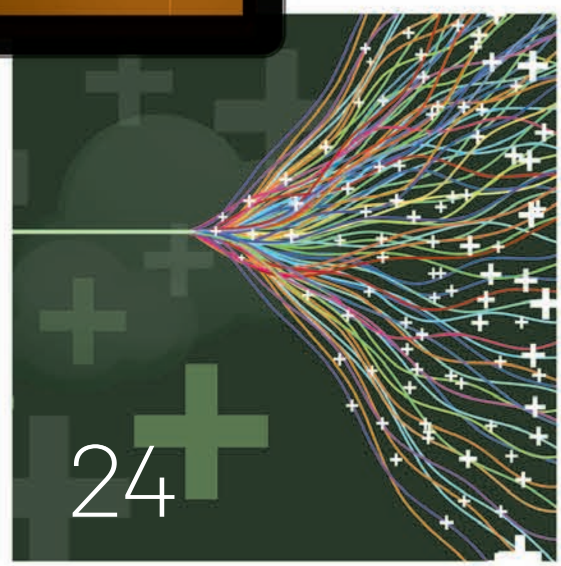
目次

2号
2016

VIEWPOINT

COVER STORY

30



24

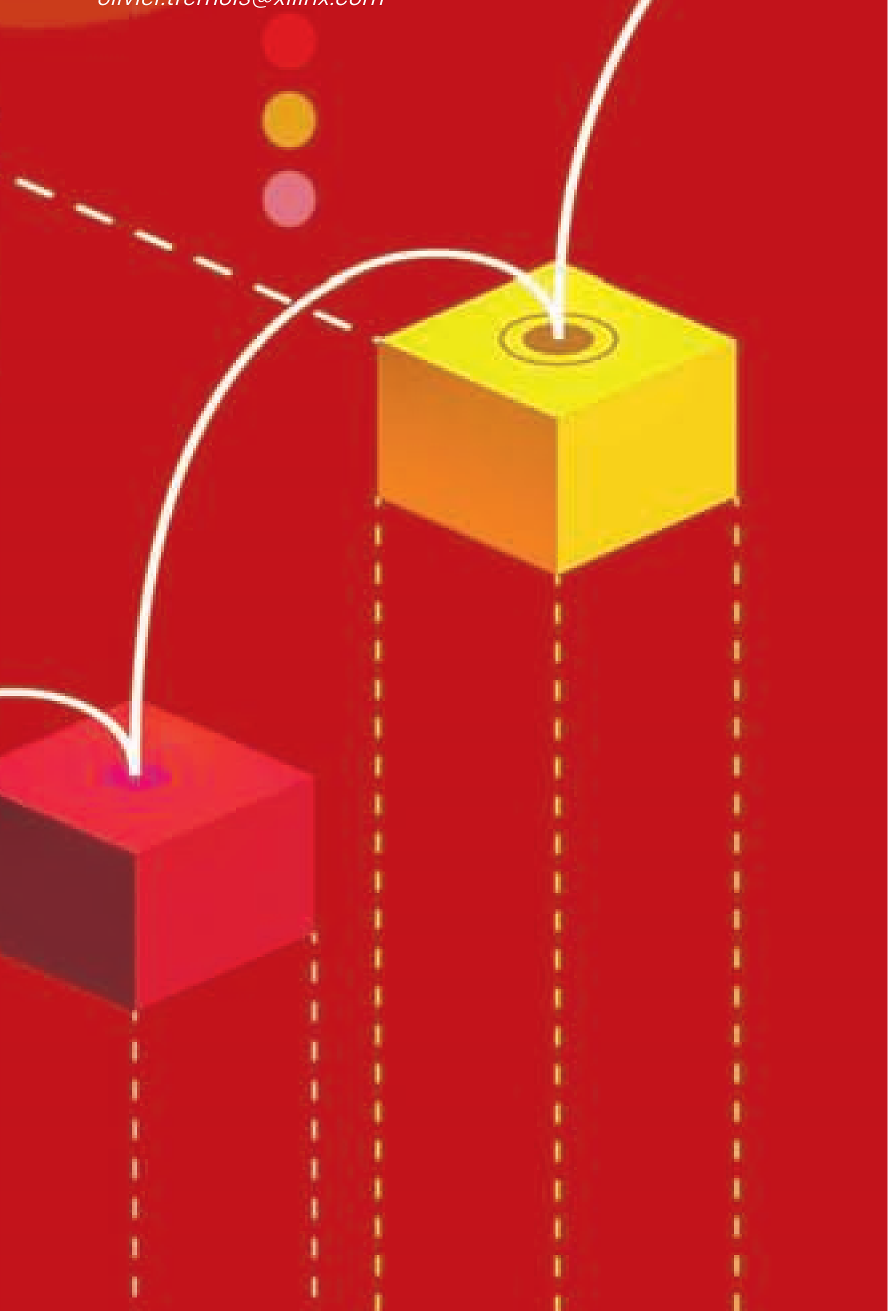
38

Use C/C++ to Offload Image Processing to Programmable Logic

C/C++ を使用して プログラマブル ロジックに 画像処理をオフロード

SDSoC により、プログラム開発者は、パフォーマンスを犠牲にせずに完全なハードウェア/ソフトウェア システムを構築できます。

Olivier Tremois
DSP Specialist FAE
Xilinx, Inc.
olivier.tremois@xilinx.com



医療用や産業用など、増え続ける応用分野における「標準的」画像処理システムは、今日ますます高度化が進んでいます。画像処理の複雑性は、多くの場合、GPU アクセラレータを搭載した PC の処理能力を既に超えています。設計チームは、画像処理の品質に関する新たな基準の確立と製品機能の追加に取り組む一方で、バッテリ電源を搭載した最終製品の小型化とモバイル化を求める顧客の要求に対応していかなければなりません。

多くの既存プラットフォームの開発者は、こうした複雑な要件への対応に苦慮しています。幸いなことに、ザイリンクス Zynq®-7000 All Programmable SoC と新しいザイリンクス SDSoC™ 開発環境を利用すれば、C/C++ で、先進の画像処理システムを搭載した小型、低消費電力、多機能型の製品を開発できます。本稿では、このような目的で SDSoC 環境を使用して画像パイプライン処理システムを高速化する手法について説明します。筆者はこのプロジェクトを 1 週間以内で完成し、サンプル システムを桁違いに高速化することに成功しました。

バッチ画像処理

筆者らのサンプル システムは、特定のカメラを使って取り込んだ画像をバッチ モードで処理するものです。画像サイズは最大 3,000 × 2,000 ピクセル (6 メガピクセル) です。処理された画像はライブ ビデオではありませんが、このプロジェクトの目的は、できるだけ迅速に画像パイプラインに画像を送り込むことです。このパイプラインは非常にシンプルで、RGB 画像をグレースケールに変換し、ソルトアンドペッパー ノイズ (ごま塩ノイズ) を追加し、Dilate (膨張)、Median (メディアン)、Erode (収縮) の 3 種類のフィルターを使ってノイズの多い画像をフィルタリングします。

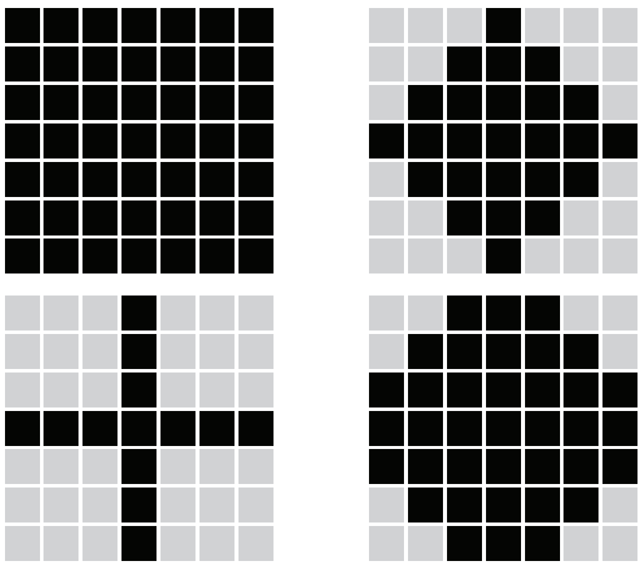


図 1 — 7 × 7 境界ボックスの枠内の構造化要素

膨張フィルター、メディアン フィルター、収縮フィルターは、ランク フィルター ファミリに所属します。ランク フィルターは、主に画質向上のためのインパルス ノイズの除去などの用途に使用されます。これらは算術演算を全く含まない非線形フィルターで、機能はデータの並べ替えとピッキングに限定されます。アルゴリズムはそれほど複雑ではありませんが、プロセッサにはシーケンシャルに実行する性質があり、一定の期間に 1 ピクセルずつ処理するため、大きな画像にアルゴリズムを適用すると、かなりの時間がかかります。

ランク フィルターは、ピクセルごとに出力画像を計算する非線形フィルターです。この計算では、構造化要素と呼ばれる指定された形状内で入力画像ピクセルに隣接するピクセルを取得し、それらのピクセルを並べ替えて、p 番目の順位のピクセルを選び出します。収縮フィルターは最小値 (p = 1) を選択します。膨張フィルターは最大値 (p = N、N は構造化要素のピクセル数) を選択します。メディアン フィルターは中央値 (p = [N/2]) を選択します。典型的には、構造化要素は正方形、ひし形、または十字形です (図 1)。

筆者らのバッチ画像処理システムは、SD カードに格納された画像を読み出し、ノイズ レベルおよび構造化要素として使用される形状ごとに異なるパラメーターを使用して、それらの画像を処理します。この計算は、Zynq-7000 SoC のデュアルコア ARM® Cortex™-A9 プロセッサ (動作周波数 667MHz) によって実行されます。

ソフトウェア インプリメンテーション

最初に、Cortex-A9 上での演算性能を推定できるように、アプリケーション全体を C++ で作成します。このアプリケーションには多くの関数が含まれており、SD カード上の BMP 画像の読み出しと書き込み、輝度の計算、ノイズの追加、および各種のフィルター機能を実行します。SDSoC 開発環境の SDDebug コンフィギュレーションで

	#Size	#Shape	SW Latency
Test 1	1,920 x 1,080	49 (square)	29 s
Test 2	1,920 x 1,080	25 (diamond)	8.5 s
Test 3	3,000 x 2,000	25 (diamond)	8.5 s

図 2 — Zynq プロセッシング システムのみの実行時間

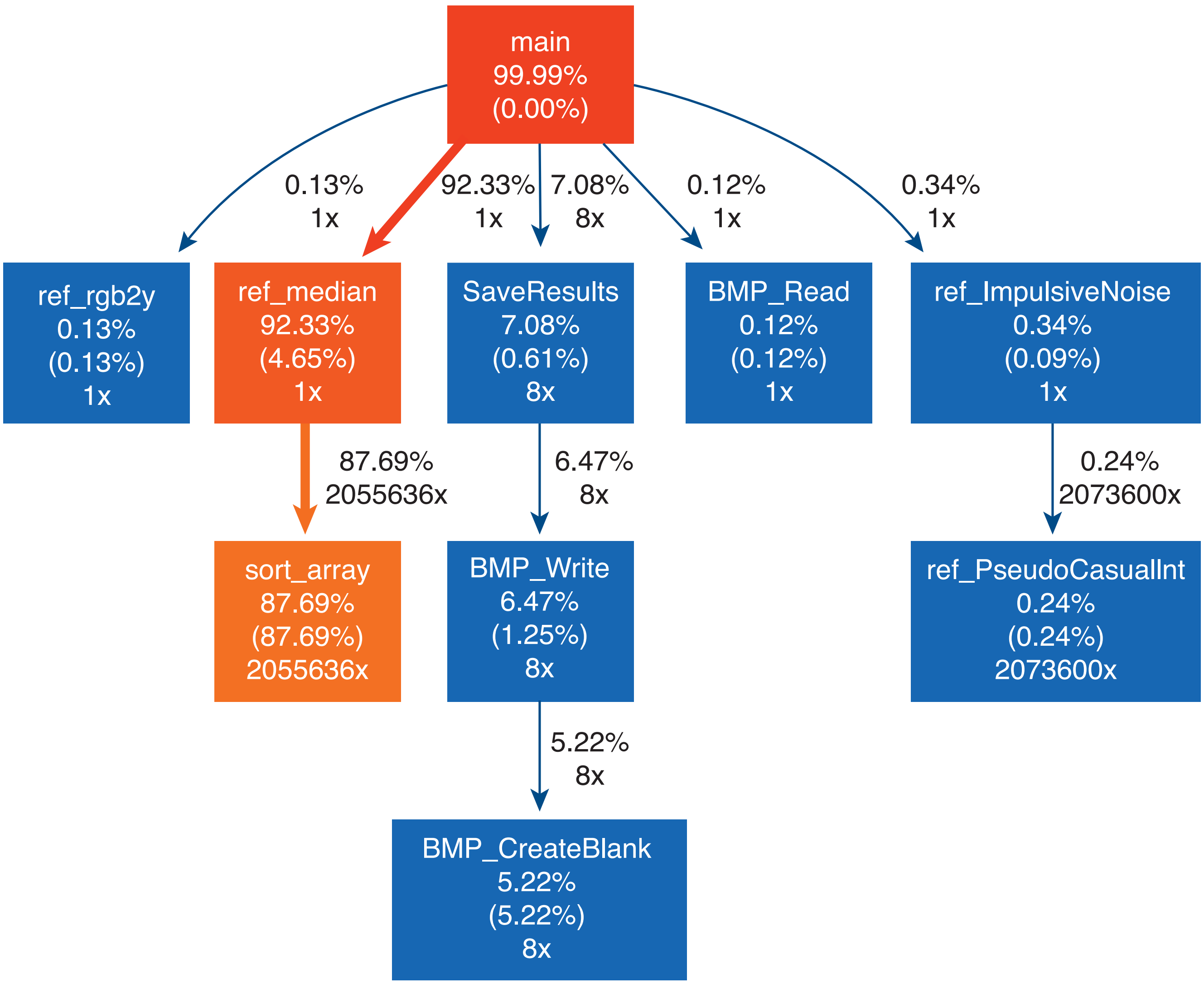


図 3 — 当初のソフトウェアのプロファイリング結果

作業すれば、Linux オペレーティング システムを搭載したザイリンクスの ZC702 評価プラットフォーム上にアプリケーションを素早くインプリメントできます。

フルに機能する実行ファイルを生成するために、オプション -O3 を選択して、コンパイラのすべての最適化機能をオンにします。構造化要素の形状はアプリケーションのパラメーターであり、7 × 7 ピクセルの境界ボックスの枠内であれば、どのような構造化要素を適用してもかまいません。

パイプラインのレイテンシに影響を与えるパラメーター (図 2) は、画像のサイズ (#Size) と構造化要素のアクティブ ピクセルの数 (#Shape) です。これらのレイテンシを最小限に抑えると、システム性能が向上します。FPGA は、多数の加算および乗算を含む信号処理アルゴリズムで驚異的な性能を発揮します。筆者らのサンプル システムでは、プログラマブル ロジックは演算量の多い計算だけでなく標準的なデータ処理にも優れていることが示されます。

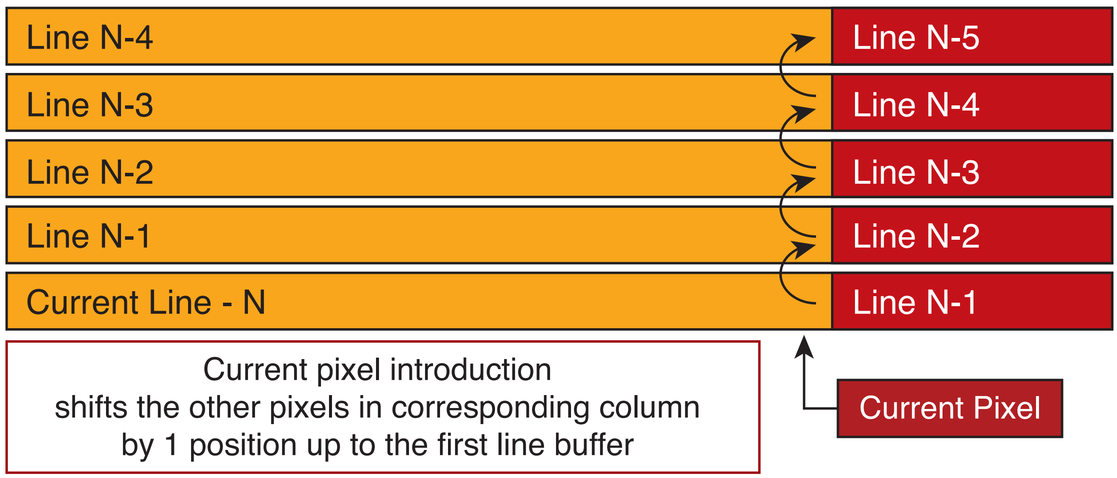


図 4 — 新しいピクセルを受信したときのライン バッファ内のデータの移動

基本的プロファイリング (図 3) を見ると、RGB 値からの輝度の計算 (0.13%) とピクセルへのノイズの追加 (0.34%) はソフトウェア内で非常に高速に実行されています。総処理時間の大半を占めているのは、メディアン フィルター (92.33%) です。総処理時間に影響を与えているその他の機能は、ファイルの読み出しと保存です。

ハードウェアへの機能の移行

このアクセラレーションの第 1 の目的は、各クロック サイクルで新しいサンプルを 1 つ処理できるようにすることです。若干のコードの書き換えとインターフェイスの再検討により、大幅なアクセラレーションが可能です。オンチップ プログラマブル ロジック (PL) のクロック レートがプロセッシング システム (PS) のクロック レートよりはるかに低い場合でも、1 クロック当たり 1 入力ピクセルを処理できれば、大きなアクセラレーションを実現できます。

ハードウェアに移行する機能は、メディアン フィルターだけです。SDSoC 環境では、Project Explorer 内で右クリックするだけで機能を簡単に PL に移動できますが、これに伴って性能向上の目的で指示子の追加 (インターフェイス側を除く) やコードの変更が行われることはありません。これらの変更を行うのは、エンベデッド プログラマの役割です。このため、最初のアクセラレーションでは、一般的にあまり劇的な効果は得られません。

上記のメディアン フィルター関数には、画像全体を処理するネストされた 2 つのループが含まれます。この関数には、構造化要素を処理し、すべての要素を並べ替えるサブループも含まれます。この例では、標準的なバブル ソート アルゴリズムを使用します。マイクロプロセッサにインプリメントする場合は、これ以外にも複雑性を軽減したアルゴリズムがありますが、ハードウェアにインプリメントする場合は、バブル ソート アルゴリズムの規則性が適しています。

```
for ( i=0; i<HeightOfImage; i++)
for ( j=0; j<WidthOfImage; j++)
 {
 Some Code
 for ( s=0; s<NumberOfStages; s++)
 for ( k=0; k<HeightOfStructElem; k++)
 for ( l=0; l<WidthOfStructElem; l++)
 {
 Swap pixels if not correctly ordered
 }
 }
```

1 クロック サイクル当たり 1 ピクセルの出力画像を処理するには、指示子を追加して、クロックごとにピクセル ベクターの並べ替えを開始させる必要があります。画像の列を処理する 2 番目のループを、開始インターバル (II) を 1 にしてパイプライン化します (II はループの新しいイテレーションを開始するまでに必要なクロック サイクル数)。SDSoC 環境は、この指示子 1 つを使用して残りの内側ループを自動的にアンロールし、すべてのイテレーションをハードウェアに並列処理させます。

シングルコア プロセッサでは、各種のプロセッサ機能によって外部メモリとプロセッサの間でスムーズなデータ転送が行われるため、シングルコア プロセッサにインプリメントされる画像処理アルゴリズムのコーディングは比較的簡単です。後で再利用されそうなデータは、メモリ キャッシュ L1 および L2 に一時的に格納されるので、データ アクセスのレイテンシが短縮されます。

このようなメカニズムは、デフォルトでは FPGA に存在しません。このため、同じ C/C++ のソース コードを使ってハードウェア アクセラレータを作成することはできません。しかしこれは、アプリケーションに最適な性能とサイズを備えたメモリ キャッシュを設計する良い機会です。この例の場合、同じ機能を維持するためではなく、要件を満たすレベルまで性能を引き上げるために、C/C++ のソース コードを変更する必要があります。ザイリンクスのVivado® 高位合成 (HLS) は、C/C++ のコードから RTL (Register Transfer Level) IP コアを生成する SDSoC のエンジンであり、指定された指示子を考慮に入れて、筆者らのコードに合わせて調整されたハードウェア アーキテクチャを生成します。このため、画像処理コードの解析時に、ライン バッファと解析ウィンドウは自動的に生成されません。Vivado HLS は書かれたとおりのコードを順守するので、HLS ツールが開発者の承諾なしに実行した可能性がある最適化を隠蔽することはありません。

ハードウェア内での画像処理に慣れた設計者は、ライン バッファと解析ウィンドウについてよく理解しています。外部メモリから同じピクセルが何回も読み出されないように、ピクセルは内部メモリ (ブロック RAM) に一時的に格納され、その後の実行に不要になった時点で上書きされます。

```
pix_t line_buffer[KMED][MAX_WIDTH];
#pragma HLS ARRAY_PARTITION variable=line_buffer complete dim=1

pix_t window[KMED*KMED];
#pragma HLS ARRAY_PARTITION variable=window complete dim=0
```

図 5 — ライン バッファ、解析ウィンドウ、構造化要素の配列の宣言

```
// Line Buffer fill
if(col < width)
BufferFill:for(int ii = 0; ii < KMED-1; ii++)
        pixel[ii]=line_buffer[ii][col]=line_buffer[ii+1][col];

//There is an offset to accommodate the active pixel region
if((col < width) && (row < height))
{
        pix = in_pix[index_in++];
        pixel[KMED-1] = line_buffer[KMED-1][col] = pix;
}
```

図 6 — ライン バッファ間のデータ転送

ブロック RAM は、メモリの読み出し、メモリへの書き込み、またはその両方に使用できる 2 つのポートを備えています。アクセラレータが行 L および 列 C に対応するピクセルを受け入れると、列 C および行 (L-1 … L-6) に対応するすべてのピクセルがライン バッファから読み出され、他の位置に再び書き込まれます (図 4 を参照)。1 クロック当たり 1 ピクセルの目標を達成するためには、すべてのデータ移動を 1 クロック サイクルのスループットで実行する必要があります。

さらに、ピクセルの近傍および構造化要素に属するすべてのピクセルにも、同じように 1 クロック サイクルでアクセスする必要があります。そこで筆者らは、問題の特定のピクセルを含む (また、ピクセルごとに変化する) 解析ウィンドウも定義します。SDSoC 環境および Vivado HLS 内では、このコードのタイミングは全く調整されません。SDSoC ツールは、使用されるリソース

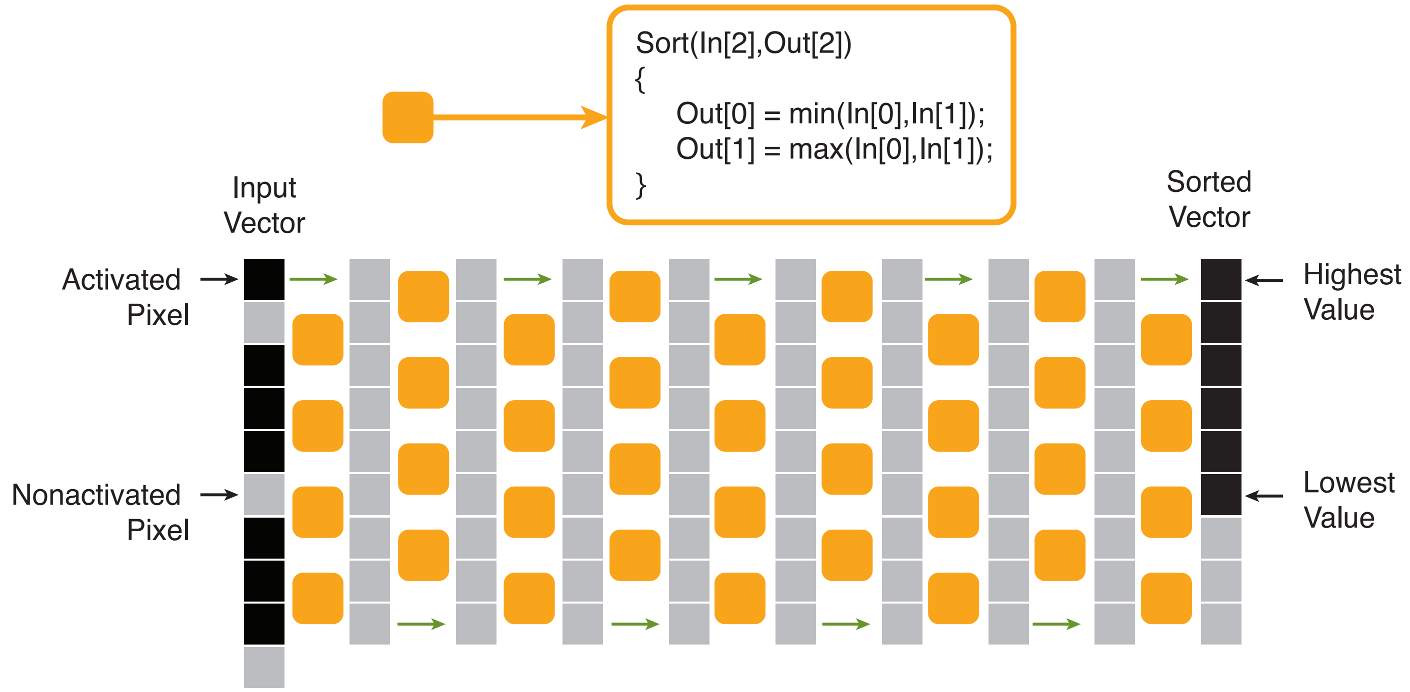


図 7 — 10 要素ベクターのソーティング ネットワーク

と指定された指示子に基づいて、並列化できる処理はすべて並列化します。この例のバッチ画像処理システムのコードでは、適切な分割指示子を使用して 2 つの配列を宣言することにより、コードにライン バッファと解析ウィンドウを追加します (図 5)。次に、これらの配列に対する読み出し/書き込みアクセスとして、データの移動を記述します (図 6)。

この手法は配列内のデータ アクセスを利用するので、ピクセル値並べ替えプロシージャをハードウェア アーキテクチャにインプリメントする手順が複雑になることがあります。ソフトウェア インプリメンテーション用の C のコードは、構造化要素から検証済みピクセルのベクターを取得し、標準バブル ソートを使って並べ替えます。より効率的なアルゴリズムもありますが、より大きなベクターでなければあまりメリットがありません。このアルゴリズムの複雑さは、構造化要素のピクセル数の 2 乗に比例し、このサンプル デザインでは最大で $(7 \times 7)^2$ です。

ハードウェア内で、アーキテクチャの寸法はワースト ケースに合わせて決める必要があります。1 クロック当たり 1 ピクセルの目標を達成するには、非常に規則的な構造をインプリメントする必要があります。このために、入力ベクターのサイズが常に最大サイズの (7 × 7) になるように指定し、すべての未検証ピクセルの値が 0 になり、ソートされたベクターの一番下に置かれるように指定します。また、アクティブなピクセルの数が少ない構造化要素についてパイプラインの段数を減らせる場合でも、ワースト ケースに合わせて段数を決定します。各段で同じベクターが再利用されない場合にのみ、異なる段の並列化が行われます。その結果、初期ベクターの入口点が列インデックス 0、出口点が列インデックス 7 × 7 = 49 になる配列が得られます (図 7 および図 8)。

SDSoC システム コンパイラ

SDSoC は、単純なフルシステム コンパイラではありません。SDSoC は、ハードウェア内に置く必要がある関数

にはどのデータ ムーバーが最適であるか、またデータ ムーバーをどのポートに接続するかを決めるために、広範囲にわたるコード解析を実行します。関数の各パラメーターについて、ARM® AMBA® AXI4-Lite、AXI4 フル メモリマップド、AXI4-Stream のうちどのデータ ムーバーを使用するのが最適かを決める必要があります。

また、AXI4 High Performance (HP) ポート、General Purpose (GP) ポート、または Accelerator Coherency

```
void sorting_network(pix_t window[KMED*KMED],mask_t shape[KMED*KMED],
                kmed2_t NShape,kmed2_t CompNShape,
                pix_t *pixmin,pix_t *pixmed,pix_t *pixmax)
{
static const int N = KMED*KMED;
pix_t tmin,tmax,t0,t1;

static pix_t z[N][N+1]; // Array that contains the sorting network
#pragma HLS ARRAY_PARTITION variable=z complete dim=0

unsigned int i, k, stage;

// Initialization of the first row of the network
// pixels that do not belong to the mask are set to 0
L1:for (i=0; i<N; i++)
        if(shape[i])    z[i][0] = window[i];
        else    z[i][0] = 0;

//sorting_network: This description is correct for KMED odd
L2:for (stage = 1; stage <= N; stage++)
{
        k = (stage&1)^1; // stage odd -> k=0     stage even -> k=1
        L3:for (i = k; i<N-1; i=i+2)
        {
                t0  = z[(i)][stage-1];
                t1  = z[(i+1)][stage-1];
                tmin = MIN(t0,t1);
                tmax = MAX(t0,t1);
                z[(i  )][stage] = tmin;
                z[(i+1)][stage] = tmax;
        }
        // Copy the value that has not been sorted to the next stage
        if(k==0) z[N-1][stage] = z[N-1][stage-1];
        else z[0][stage] = z[0][stage-1];
}
*pixmin = z[CompNShape][N];
*pixmed = z[CompNShape+NShape/2][N];
*pixmax = z[N-1][N];
return;
}
```

図 8 — C で記述されたソーティング ネットワーク

```
#pragma SDS data sys_port(in_pix:AFI, out_pix_min:AFI, out_pix_med:AFI, out_pix_max:AFI, median_mask:ACP)
#pragma SDS data data_mover(in_pix:AXIDMA_SIMPLE, out_pix_min:AXIDMA_SIMPLE, out_pix_med:AXIDMA_SIMPLE, out_pix_max:AXIDMA_SIMPLE)
#pragma SDS data mem_attribute(in_pix:CACHEABLE|PHYSICAL_CONTIGUOUS)
#pragma SDS data mem_attribute(out_pix_min:CACHEABLE|PHYSICAL_CONTIGUOUS)
#pragma SDS data mem_attribute(out_pix_med:CACHEABLE|PHYSICAL_CONTIGUOUS)
#pragma SDS data mem_attribute(out_pix_max:CACHEABLE|PHYSICAL_CONTIGUOUS)
void hw_median(pix_t in_pix[MAX_HEIGHT*MAX_WIDTH],
        pix_t out_pix_min[MAX_HEIGHT*MAX_WIDTH],
        pix_t out_pix_med[MAX_HEIGHT*MAX_WIDTH],
        pix_t out_pix_max[MAX_HEIGHT*MAX_WIDTH],
        pix_t median_mask[KMED][KMED],
        short height, short width);
```

図 9 — デフォルト動作を無効にする SDSoC 環境の指示子

Port (ACP)、あるいは他のアクセラレータからのポート (SDSoC 環境内から構築されるか、ボード サポート パッケージ (BSP) に含まれるもの) のうち、どのコネクタを使用するかを決める必要もあります。

次に、SDSoC 環境はデザインを作成し、すべての必要な IP コアを追加してフルに機能するシステムを構築します

Summary

Performance estimates for 'main' function

SW-only (Measured cycles)	34833713710
HW accelerated (Estimated cycles)	6936128677
Estimated speedup	5,02

Details

Performance estimates for 'hw_median' function

SW-only (Measured cycles)	27937991386
HW accelerated (Estimated cycles)	40406353
Estimated speedup	691,43

Resource utilization estimates for hardware accelerators

Resource	Used	Total	% Utilization
DSP	0	220	0
BRAM	12	140	8,57
LUT	23105	53200	43,43
FF	11192	106400	10,52

図 10 — SDEstimate フェーズで得られるパフォーマンスの推定値

(AXI4 Stream データ ムーバー用のダイレクト メモリ アクセス (DMA) コアなど)。また、ハードウェアを呼び出すために (当初の C++ のコードの代わりに) C のソース コードを変更します。筆者らのデザインでは、インターフェイスは非常にシンプルであり、2 つの入力アレイと 3 つの出力アレイが AXI4-Stream および DMA を介してアクセスされ、少数のスカラーが AXI4-Lite を介して設定されます。DMA の設定について検討したり、スカラー レジスタがアクセス可能になるアドレスを参照したりする必要はありません。SDSoC 環境はこれらの項目をすべて内部で自動的に管理します。

サンプル システムの構築時には、まずソース コードが Vivado HLS に準拠していることを検証し、次に VHLS 指示子を追加しました。特定の SDSoC 指示子を使用して、データが物理空間に (関数 sds_alloc を使って割り当てられたメモリに) 連続的に格納され、DMA がそのデータにアクセスするように指定しました (図 9)。

次に、達成可能なアクセラレーションの最初のおおよその推定値を計算するために、ビルド コンフィギュレーションを SDEstimate に切り替えました (図 10)。この段階ではハードウェアはまだ構築されていないため、この手順は短時間で完了しました。

SDSoC 環境は、(コンパイラの最適化機能を -O0 に設定して、ハードウェア用に調整されたコードを使って計算される) プロセッサの実行時間と、(VHLS によってハードウェア アクセラレータのレイテンシとして計算される) クロック

	#Size	#Shape	Pure SW Latency	SW + HW Latency	Acceleration
Test 1	1,920 x 1,080	49 (square)	29.2 s	154.8 ms	189x
Test 2	1,920 x 1,080	25 (diamond)	8.6 s	154.7 ms	56x
Test 3	3,000 x 2,000	25 (diamond)	25 s	447 ms	56x

図 11 — プログラマブル ロジック内でアクセラレートした関数を使用したときの Zynq プロセッシング システムの実行時間

サイクル数に基づいて、スピードアップの推定値を計算します (このコードは、プロセッサ用に調整された元のコードよりも低速)。このレイテンシはハードウェア アクセラレータの最大レイテンシなので、この概算の結果はあるがままに、おおよその推定値として解釈する必要があります。ハードウェア アクセラレータそれ自体については、このアクセラレーションはほぼ 700 倍です。「main」レベルでは、時間のかかるファイル アクセスが多数発生します。このため、全体的なアクセラレーションはわずか 5 倍です。実際には、グローバルなアクセラレーションが計算されるトップ レベルの関数を選択できるので、より有意義なアクセラレーションの値が得られます。

フローの最後の手順では、システム全体を構築します。このフェーズでは、すべてのアクセラレータが構築され、プロセッサに接続されます。次に、これらのアクセラレータの起動および制御のために、(元の C の関数を呼び出す代わりに) C++ のソース コードが変更されます。この段階では、DDR との間のすべてのデータ転送を考慮に入れて、ハードウェア アクセラレータによって得られるアクセラレーションの正確な値を計算できます。データはメモリのキャッシュ可能部に格納されるため、このアクセラレーションの値は、キャッシュの消去にかかる時間も考慮に入れています。

ハードウェア アクセラレータの処理にかかる時間は、構造化要素のサイズではなく、画像のサイズに比例します。このため、構造化要素のアクティブ ピクセルの数が多いほど、アクセラレーション率は高くなります。図 11 で参照しているレイテンシは、ソフトウェア要素とハードウェア要素を含む画像パイプライン全体のレイテンシです。

このプロジェクトに着手してみると、ソフトウェア アプリケーションの構築が最も長いフェーズとなることが判明しました。そこから 2 時間以内でコードを変更し、スループットの最適化のために適切な指示子が追加された、Vivado HLS 完全準拠のコードが完成しました。このデザインはハードウェア部分の規模が大きいため (チップのルックアップ テーブルの 2 分の 1)、最後の段階 (合成、配置配線、ビットストリーム、SD カード) を完了するのに 2 時間以上かかりました。

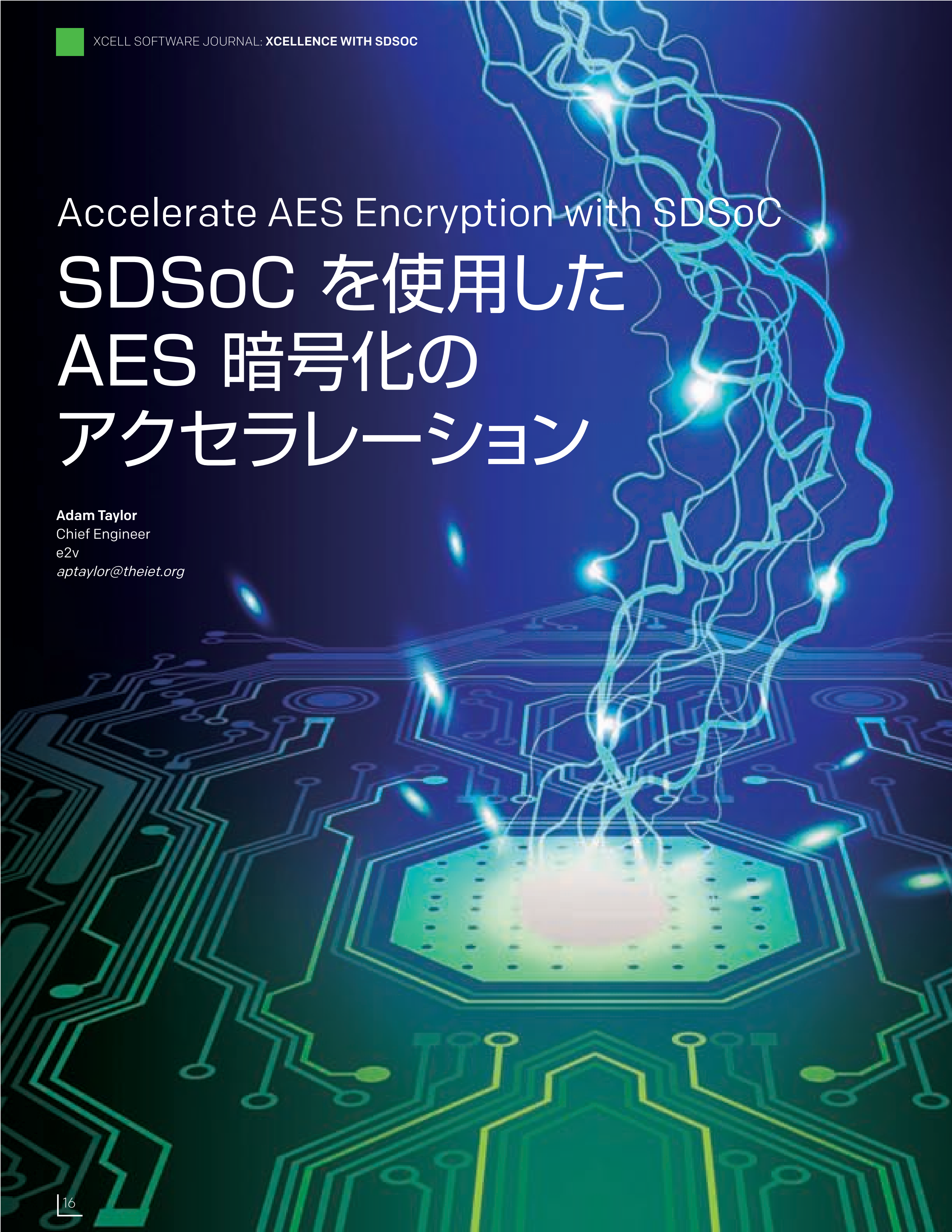
SDSoC 環境の統合型ツールは、システム レベルのプロファイリング、プログラマブル ロジック内での自動化されたソフトウェア アクセラレーション、システム全体を最適化するコンパイル (適切な接続を自動的に生成し、メモリ アクセスのボトルネックを最小限に抑える) の機能を備えています。これらのツールを使用して、筆者はこのサンプル プロジェクトを 1 週間以内で完了しました。

標準的な RTL フローを使ってアクセラレータを作成し、筆者自身のプログラミング能力に頼ってさまざまなドライバーを利用して C のコードを変更していたとしたら、これほど短期間でプロジェクトを成功させることはできませんでした。■

Accelerate AES Encryption with SDSoC

SDSoC を使用した AES 暗号化のアクセラレーション

Adam Taylor
Chief Engineer
e2v
aptaylor@theiet.org



AES256 暗号アルゴリズムを C で記述し、ハードウェアで性能を加速できます。

Advanced Encryption Standard (AES : 高度暗号化標準) は、エンベデッド システムなど多くのアプリケーションの暗号化仕様として広く採用されています。2002 年に米国国立標準技術研究所 (NIST) が標準規格として採用して以来、プロセッサ、マイクロコントローラー、FPGA、そしてSoC アプリケーションの開発者は、システムの入力/出力/格納データのセキュリティ確保に AES を利用してきました。AES アルゴリズムは、高い抽象化レベルで非常に効率的に記述できますが (従来のソフトウェア開発での使用方法)、含まれる演算の性質上、FPGA にインプリメントするのが最も効率的です。実際に、一部のオペレーションは配線リソースを使用せずに実装できます。

これらの理由で、AES は、開発者がアルゴリズムを C で記述し、そのインプリメンテーションをハードウェアで加速することにより、どのようにザイリンクス SDSoC™ 開発環境の恩恵を受けるかを示す優れた例と言えます。本稿では実際にこの作業を行い、まず AES アルゴリズムについて詳しく説明します。次に、ザイリンクス Zynq®-7000 All Programmable SoC のプロセッシング システム (PS) 側に AES256 (キー長 256 ビット) をインプリメントし、ソフトウェア性能のベースラインを確立します。さらに、オンチップ プログラマブル ロジック (PL) 内でアクセラレーションを実行します。得られるメリットを完全に理解するために、SDSoC 環境でサポートしている 3 つのオペレーティング システム (Linux、FreeRTOS、BareMetal) すべてでこれらの手順を実行します。

アルゴリズム

AES は、128/192/256 ビットの可変キー長で使用できる対称ブロック暗号です。キー長により、データの暗号化または復号に必要な処理手順の数が決まります。ブロック暗号アルゴリズムは、その名前が示すように、データをブロック単位で処理します。AES アルゴリズムは、一度に 16 バイトの固定サイズのブロックを処理します。したがって、16 バイト未満のデータを暗号化する場合は、未使用バイトをパディングする必要があります。

AES は対称暗号であるため、情報の暗号化と復号に同じ動作とキーが使用されます。それに対して、RSA 暗号などの非対称アルゴリズムは、データの暗号化と復号に異なるキーを使用します。

AES アルゴリズムの 4 つのステージは、ステートと呼ばれるバイト配列に対して適用されます。4 つの AES ステージの組み合わせをラウンドと呼びます。必要なラウンド数はキー長によって決まります。

非常にシンプルに、AES ステートは暗号化の対象となる 16 バイトから始まります。ステートは新しいステップごとに更新されます。ステートの処理を始める前に、入力バイトストリングを 4 x 4 行列の初期ステートに正しくフォーマッティングする必要があります (図 1)。

初期 16 バイトを 4 x 4 グリッドの初期ステートに再構成したら、各ステップが入力ステートをどのように操作するかを検討します。

AddRoundKey: 暗号化キーを使用するステップはこれだけです。既に述べたように、暗号化アルゴリズムの必要なラウンド数は、キーのサイズ (128 ビット、192 ビット、または 256 ビット) によって決まります。各ラウンドの実行中にキー内のバイトが再利用されないように、暗号化キーを使用する前にキー拡張を実行する必要があります。当然のことですが、拡張されたキー長はキーサイズごとに異なります。拡張されたキーサイズは次の式で計算されます。

*拡張されたキーサイズ (バイト) = 16 * (ラウンド数 + 1)*

このステップに含まれる演算は簡単です。入力ステートバイトと拡張されたキーの 16 バイトの XOR (排他的論理和) を求めます。各ラウンドは、拡張されたキーの異なるセクションを使用します。ラウンド 0 はバイト 0 ～ 15 を使用し、ラウンド 1 はバイト 16 ～ 31 を使用します (以下同様)。各ラウンドについて、ステートのバイト 1 と拡張されたキーの最下位バイトの XOR を求め、バイト 2 と最下位バイト + 1 の XOR を求めます (以下同様)。

B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	B14	B15	B16

16-byte Input

B1	B5	B9	B13
B2	B6	B10	B14
B3	B7	B11	B15
B4	B8	B12	B16

Initial State – 4 x 4 Grid

図 1 — 16 バイトの初期ステートを 4 x 4 グリッドに変換

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	CB	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

S-box for Encryption

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	23	4E
3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

S-box for Decryption

図 2 — AES S-box の内容

SubBytes: このステップは、バイト置換を使ってステートの値を他の値で置き換えます。置換ボックス内の値は事前に定義済みであり、入力ビットと出力ビットの相関が小さくなるように設計されています。置換ボックス (S-box) は 16 x 16 行列です。置換テーブルへのインデックスとして、置換されるバイトの上位ニブルと下位ニブルを使用します。たとえば、図 2 の S-box を使用する場合、最初の初期ステート バイトが 0 x 69 であるとすると、この値は 0 x F9 で置き換えられます。ステート バイトの上位ニブルは置換ボックスの行を選択し、下位ニブルは列を選択します。図 2 で、暗号化と復号には別個の置換ボックスが使用され、各ボックスの内容は異なることに注意してください。

ShiftRows: このステップは、各行に循環バイト シフトを実行し、入力ステート行列を再構成します。行ごとに、異なる係数だけ右にローテートします (図 3)。行 1 は変更せず、行 2 は 1 バイト、行 3 は 2 バイト、行 4 は 3 バイトだけローテートします。復号時にも同じ操作を実行しますが、右ではなく左にローテートします。

MixColumns: これはラウンド内で最も複雑なステップであり、16 回の乗算と 12 回の XOR 演算を必要とします。これらの演算は入力ステート行列の列ごとに実行され、固定された行列との乗算によって新しいステート列を作成します (図 4)。入力ステートの列の各要素に、固定された

行列の行を掛けます。各乗算の結果をまとめて XOR を求め、新しいステート値を生成します。図 4 では、最初に乗算される列と行がハイライトされています。

　最初の列の MixColumns の式を次に示します。

B1' = (B1 * 2) XOR (B2 * 3) XOR (B3 * 1) XOR (B4 * 1)
B2' = (B1 * 1) XOR (B2 * 2) XOR (B3 * 3) XOR (B4 * 1)
B3' = (B1 * 1) XOR (B2 * 1) XOR (B3 * 2) XOR (B4 * 3)
B4' = (B1 * 3) XOR (B2 * 1) XOR (B3 * 1) XOR (B4 * 2)

　すべての入力ステート列の処理が完了するまで、入力ステートの次の列と同じ乗算行列の間でこのプロセスが繰り返されます。

　AES 暗号化/復号アルゴリズムに必要な手順の詳細が理解できたら、次に、ラウンド内で各ステップをどのような順番で適用するか、各ラウンドですべてのステップを適用する必要があるかどうかを理解する必要があります。AES

B1	B5	B9	B13
B2	B6	B10	B14
B3	B7	B11	B15
B4	B8	B12	B16

ShiftRows Input State

B1	B5	B9	B13
B6	B10	B14	B2
B11	B15	B3	B7
B16	B4	B8	B12

Resultant Output State

図 3 — ShiftRows 操作

B1	B5	B9	B13
B2	B6	B10	B14
B3	B7	B11	B15
B4	B8	B12	B16

Input State First Column to be Multiplied

2	3	1	1
1	2	3	1
1	1	2	3
3	1	1	2

Constant Multiplication Matrix

E	B	D	9
9	E	B	D
D	9	E	B
B	D	9	E

Decryption Constant Multiplication Matrix

図 4 — 暗号化および復号用の MixColumns 関数

暗号化の各ラウンドは 4 ステップすべてで構成され、各ステップは次の順番で実行されます。

1. SubBytes
2. ShiftRows
3. MixColumns (ラウンド 1 ～ N – 1 のみ)
4. AddRoundKey (拡張されたキーを使用)

　もちろん、暗号化された情報を利用するには、このプロセスを逆転して、読めない暗号文を平文に戻す必要があります。これを行うには、ステップの順番を次のように変更します。

1. ShiftRows を反転する
2. SubBytes を反転する
3. AddRoundKey (拡張されたキーを使用)
4. MixColumns を反転する (ラウンド 1 ～ N - 1 のみ)

暗号化と復号のいずれについても、暗号化の最初のラウンドを実行する前に、初期 AddRoundKey 操作を実行する必要があります。

AddRoundKey ステップの実行回数に対して十分なキー ビットを用意するために、キーの拡張に使用しなければならないアルゴリズムを見てみましょう (図 5)。キーのサイズが16 バイト、24 バイト、32 バイトの場合、キーの拡張にはそれぞれ 44 ラウンド、52 ラウンド、60 ラウンドが必要です。拡張されたキーの最初のバイトは、初期キーに等しくなります。つまり、筆者らの AES256 の例では、拡張されたキーの最初の 32 バイトはキーそれ自体になります。キー拡張を実行すると、各イテレーションで拡張されたキーに 32 ビットの追加ビットが生成されます。

キー拡張のステップを次に示します。

RotateWord: このステップは、ShiftRows と同じように、最上位バイトが最下位バイトになるように 32 ビット ワードを再構成します。

SubWord: このステップは、暗号化処理でバイト置換に使用したのと同じ置換ボックスを使用します。

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
    word temp

    i = 0

    while (i < Nk)
       w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
       i = i+1
    end while

    i = Nk

    while (i < Nb * (Nr+1)]
       temp = w[i-l]
       if (i mod Nk = 0)
           temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
       else if (Nk > 6 and i mod Nk = 4)
           temp = SubWord(temp)
       end if
       w[i] = w[i-Nk] xor temp
       i = i + 1
    end while
end
```

図 5 — キー拡張アルゴリズム

```
void aes_enc(uint8_t state[4][4],uint8_t cipher[4][4],uint8_t ekey[240])
{

    uint8_t iteration = 0;
    //uint8_t x,y;

    uint8_t sub[4][4];
    uint8_t shift[4][4];
    uint8_t mis[4][4];
    uint8_t round[4][4];

    addroundkey(state,0,sub,ekey);

    loop_main : for(iteration = 1; iteration < nr; iteration++)
       {
        subbytes(sub,shift);
        shift_row_enc(shift,mix);
        mixcolumn(mix,round);
        addroundkey(round,iteration,sub,ekey);
       }
       subbytes(sub,shift);
       shift_row_enc(shift,round);
       addroundkey(round,nr,cipher,ekey);

}
```

図 6 — アクセラレートされる関数

rcon: このステージは、ユーザー定義の値に対する 2 のべき乗を実行します。MixColumns ステージと同じように、rcon はガロア体 (28) に対して実行されます。したがって、このステップには事前に計算されたルックアップ テーブルを使用するのが一般的です。

EK: このステップは、拡張されたキーから 4 バイトを返します。

K: EK のように、このステップはキーから 4 バイトを返します。

暗号化およびキー拡張アルゴリズムを正しくインプリメントできたかどうか、どのように確認できるでしょう。NIST の AES 仕様には、筆者らのインプリメンテーションのチェックに役立つ多くの作業例が含まれています。

コードの作成

Zynq SoC の PL 側で AES コードの暗号化部分を確実に高速化するには、最初からこの目的を念頭に置いてコードを開発する必要があります (このブログのコーディング

規則を参照)。最初に、アルゴリズムのアーキテクチャを検討します。このアーキテクチャは適切に分割する必要があります。各ステージについて記述した関数を必要に応じて呼び出せるため、AES にはこの手法が効果的です。また、高速化する関数は、その関数のファイル内で記述する必要があります。このソフトウェア アーキテクチャには、次のファイルが含まれます。

main.c: このファイルには、キー拡張アルゴリズム、暗号化キー、平文入力と、AES 暗号化関数の呼び出しが含まれます。

aes_enc.c: このファイルは、暗号化を実行します。AES ラウンドで必要に応じて呼び出せるように、各ステージを独自の関数としてコーディングします。複数のプロセッサでインプリメントされたデザインが共通であることを確実にするように、ミックスド ステップの乗算にはルックアップ テーブルを使用します。

aes_enc.h: このファイルは、aes_function の定義と、サイズの決定に使用されるパラメーター (mk、nb、nr など) を格納します。

sbox.h: このファイルには、置換バイトに使用される置換ボックス、キー拡張を実行する rcon 関数用のルックアップ テーブル、MixColumns 乗算に使用される乗算ルックアップ テーブルが含まれます。

この構造内で関数を右クリックして [Toggle HW/SW] を選択するだけで、アクセラレートしたい AES 暗号化関数 (図 6) を選択できます。

ベースライン性能と、関数のアクセラレーションから得られる効果を計算するには、関数の実行時間を計測する必要があります。これを行うには、sds_lib.h 内の sds_clock_counter を使用します。

ソース コード (github に掲載) の作成後、筆者は Zynq SoC のシングルコアARM® Cortex™-A9 プロセッサ上で動作するソフトウェアで AES アルゴリズムを実行し、36,662 プロセッサ サイクルの時間を記録しました。

アクセラレーションのための最適化

AES アルゴリズムのアクセラレーションは、Xcell Software Journal 日本語版 1 号で説明した行列乗算アルゴリズムよりも多少複雑です。これは、AES アルゴリズムのメイン ループが互いに依存するステージで構成されているためです。

AES アルゴリズムを高速化するために、筆者はループを調べてアンロールできる箇所を発見し、メモリ帯域幅を最適化し、データ移動クロック周波数に適切な周波数を選択し、ハードウェア機能に適切な周波数を選択しました。

AES 暗号化関数のメイン ループは、各 AES ステップを実行する関数で構成されます。AES アルゴリズムは、各関数が完了して結果が計算されなければ、次の関数を実行

Data Motion Network

Accelerator	Argument	IP Port	Direction	Declared Size (bytes)	Pragmas	Connection
aes_enc_0	state	state_PORTA	IN	16*1		S_AXI_ACP:AXIDMA_SIMPLE
	cipher	cipher_PORTA	OUT	16*1		S_AXI_ACP:AXIDMA_SIMPLE
	ekey	ekey_PORTA	IN	240*1		S_AXI_ACP:AXIDMA_SIMPLE

図 7 — PS と PL 間のデータ移動ネットワーク

Operating System	PS Only	PS with PL Acceleration	Reduction
BareMetal	28574	7102	75%
FreeRTOS	28368	7104	75%
Linux	36662	16544	54.8%

図 8 — Zynq PS および PL 上での OS の性能。FreeRTOS と BareMetal はほぼ同じようにスピードアップしています。

できません。こうした相互依存性のため、筆者らは、個別の関数として作成される AES ステップに集中して作業する必要がありました。これらのステップには、最適化の大きな可能性が秘められています。

性能向上のために、AddRoundKey、SubBytes、MixColumns の各ステップをパイプライン化できます。これらの関数内で、最初のループの中にプラグマを追加し、HLS のパイプライン コマンドを実行します。内側ループはアンロールする必要があります。ルックアップ テーブルから読み出されるこれらの関数の一部は、通常はブロック RAMから構築されます。メモリ帯域幅を拡張する必要があるため、この例ではプラグマ パラメーター "complete" を指定しました。これにより、メモリの内容は BRAM ではなく個別のレジスタとしてインプリメントされます。

また、Zynq SoC 上の PS と PL 間のデータ転送能力には、性能向上の鍵となる重要性があります。筆者は最初の手順として、データ移動クロック ネットワークを最高のクロック周波数 (200MHz) に設定しました。2 番目の手順として、PS と PL 間のデータ転送にダイレクト メモリ アクセスが使用されるように指定しました。これを行うのに、インターフェイスを多少書き換え、sds_alloc 関数を使用して、データが DMA 転送の要件に従ってメモリ内に連続的に配置できるようにする必要がありました (図 7)。

3 番目の (最後の) 最適化手順として、ハードウェア機能のクロック レートを、このアプリケーションでサポートできる最高の周波数 (166.67MHz) に設定しました。

サポートしているオペレーティング システムでの結果

最後にすべての手法をまとめてこのデザインを構築したとき、PL でアクセラレートされた AES コードの Linux 上での実行時間は、16,544 プロセッサ クロック サイクルになりました。この結果は、ソフトウェアのみで AES コードを実行したときに必要なサイクル数の 45% (16,544 / 36,662) に相当します。相当に複雑で相互依存性の強いアルゴリズムで、実行時間を 55% も大きく短縮できたことになります。

もちろん、SDSoC 環境では BareMetal または FreeRTOS オペレーティング システムも選択できます。BareMetal および FreeRTOS のプロジェクトを作成し、コードを再利用することで、サポートしている 3 種類のオペレーティング システム間の性能比較が可能となります。特定のプロジェクトの OS は、ミッションの要件、性能バジェット、応答時間に基づいて選択されます。

図 8 に、Zynq SoC 上で PS のみを使用した場合と PL で PS をアクセラレートした場合について、3 種類のオペレーティング システムの性能を示します (図 8)。

予想どおり、FreeRTOS と BareMetal はいずれもフル Linux OS よりもはるかにシンプルなインプリメンテーションですから、両者はほぼ同じようにスピードアップしています。

この結果が示すように、SDSoC 開発環境を使った AES 暗号化のアクセラレーションにより、実際に性能が向上しています。しかもこの手法は、FPGA デザインの詳しい知識がなくても容易に実現できます。■

Innovating a Reprogrammable Network with SDNet

SDNet を用いた再プログラム可能なネットワークのイノベーション

Koji Yamazaki
Researcher
NTT Labs
yamazaki.k@lab.ntt.co.jp

Yoshihiro Nakajima
Researcher
NTT Labs

Takahiro Hatano
Researcher
NTT Labs

Hirokazu Takahashi
Researcher
NTT Labs

Akihiko Miyazaki
Researcher
NTT Labs

Katsuhiro Shimano
Researcher
NTT Labs

Lagopus FPGA は通信/クラウドインフラストラクチャ用 SDN/NFV の機能強化を実現します。

日本電信電話株式会社 (NTT) は、世界的な通信事業グループの経営戦略の策定と研究開発の推進を担う持株会社です。筆者らは NTT の研究開発部門でソフトウェア定義ネットワーキング (SDN) とネットワーク機能仮想化 (NFV) に関連する 2 つの革新的なプロジェクトに取り組んでいます。1 つのプロジェクトでは、Lagopus (ラゴパス) [1] という高性能 SDN/OpenFlow ソフトウェア スイッチを開発しました。Lagopus は、現在までにオープンソース ソフトウェアとしてリリースされた OpenFlow 1.3 準拠スイッチの中でも最高クラスの性能を実現しています。もう 1 つのプロジェクトでは、Lagopus FPGA と呼ばれるソフトウェア パケット処理対応の 40Gbit/秒 (Gbps) FPGA ネットワーク インターフェイス カード (NIC) を開発しました。

ザイリンクスの SDNet™ (Software Defined Specification Environment for Networking) を早くから採用したことは、これらの開発プロジェクトの成功の鍵となりました。本稿では、筆者らがどのように SDNet を使用してプロジェクトの目的を達成したかを説明します。

Lagopus FPGA で優れた SDN/NFV を実現

クラウド サービス プロバイダーやネットワーク サービス事業者は、自動化されたプロビジョニング システムの実現の鍵となる技術として SDN に注目しています。NFV は、独自規格のハードウェアに基づく機器で構成されたネットワーク システムを、PC サーバー、市販のシリコンを搭載したスイッチ、およびソフトウェア アプライアンスを主体とした汎用ハードウェア ベースのシステムで置き換えることを可能にする技術であり、通信事業者の設備投資 (CAPEX) と運用コスト (OPEX) の削減に大きく貢献します。多くのクラウド サービス プロバイダーや通信事業者が、自社の次世代商用ネットワークに SDN と NFV の導入を推進しています。

NTT グループは、商用サービスと研究開発の両分野で SDN と NFV を主導しています。NTT ほか数社は、総務省の補助金を受けて O3 プロジェクトと呼ばれる SDN/NFV に関連する先端研究プロジェクトを立ち上げました。O3 プロジェクトにおける NTT の取り組みは、Open Networking Foundation (ONF) の OpenFlow 1.3 プロトコルと汎用インテル x86 サーバーおよび汎用 NIC を組み合わせて、高性能なソフトウェア パケット処理と柔軟な

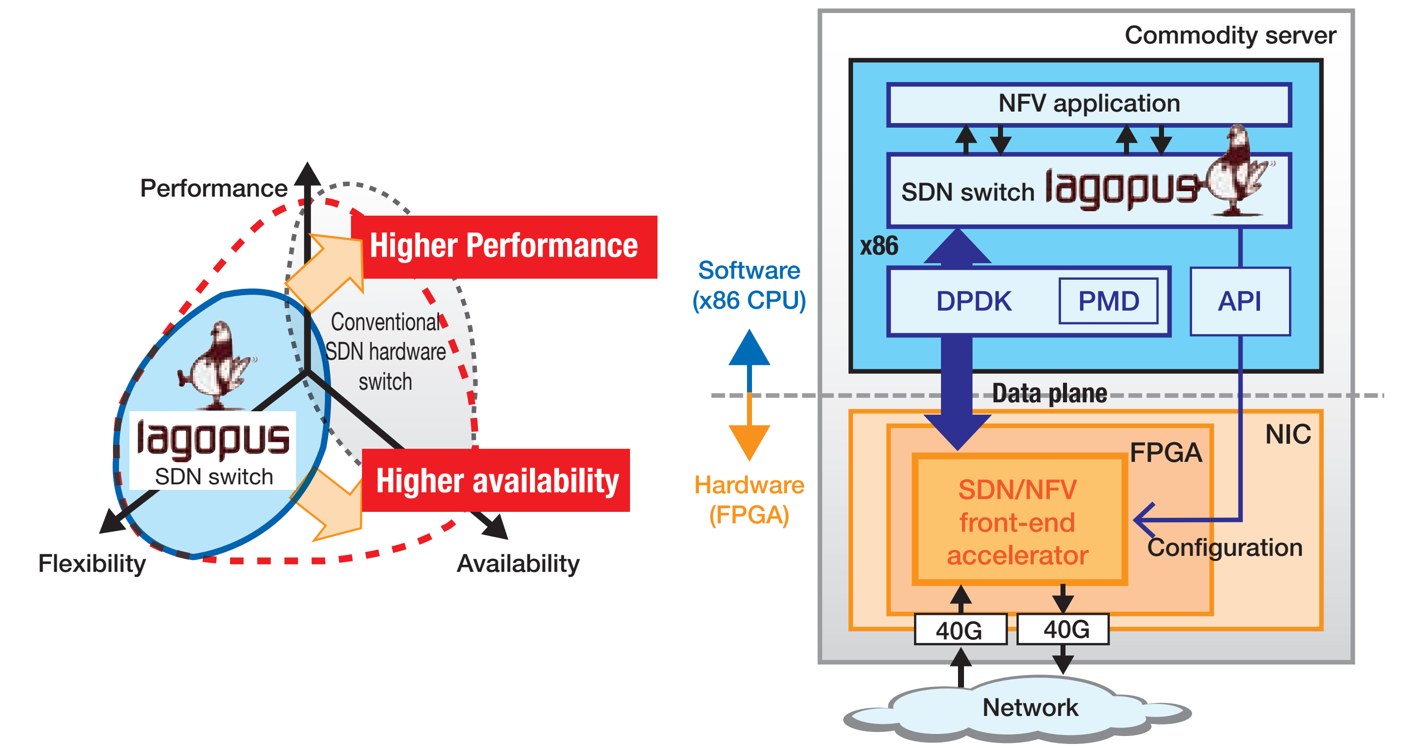


図 1 — Lagopus FPGA のコンセプトとアーキテクチャ

フロー制御を実現する試みであり、Lagopus はこのプロジェクトの主要な成果です。Lagopus の主な利点として、コモディティ サーバー上での 10Gbps を超える高性能ソフトウェア パケット処理、最大 100 万エントリに対応する柔軟なネットワーク フロー制御、virtual Provider Edge (vPE)、virtual Customer Premises Equipment (vCPE)、virtual Evolved Packet Core (vEPC) フレームワークなどの NFV アプリケーションに対応するスケーラブルなフロー振分け機能が挙げられます。

他方、Lagopus FPGA プロジェクトの目的は、コモディティ サーバーで動作する FPGA 上でソフトウェア機能とハードウェア アクセラレーション機能を分割することにより、40/100Gbps に対応する高性能なパケット処理を実現することです。図 1 に、Lagopus FPGA のコンセプトとアーキテクチャを示します。この包括的なアーキテクチャにより、Lagopus スイッチを純粋なソフトウェアインプリメンテーションとして使用した場合の 10Gbps のライン レートは、FPGA アクセラレーションによって 40Gbps のライン レートまで高速化されます。この性能向上に必要な電力は、x86 CPU の消費電力の 10% 未満です。このアーキテクチャにより、本格的な仮想化ネットワークに必要不可欠なネットワーク トラブルシューティング機能も大幅に強化されます。

筆者らは現在、最先端の FPGA と設計ツールを使用して、Lagopus 用の先進のソフトウェア プログラマブル データ プレーンとネットワーク通信事業者用のオリジナル ハードウェア知的設計資産 (IP コア) の協調設計に取り組んでいます。これにより、システム性能の向上だけでなく、消費電力とコストの削減も期待されます。筆者らは、ザイリンクス社のチームと協力して、ザイリンクス Virtex®-7 All Programmable FPGA ベースの 80Gbps NIC 試作ボード上で Lagopus と IP コアの統合に成功しました。

2015 年 2 月に東京で開催された「NTT R&D フォーラム 2015」では、Lagopus FPGA の初めてのデモを実施しました。2015 年 8 月にカリフォルニア州クパチーノで開催された「Hot Chips 27 」でも研究成果を発表しています [2]。

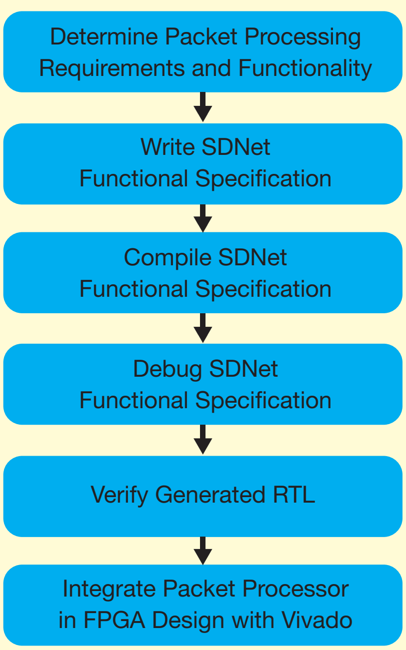
Lagopus FPGA システムの開発には、SDNet 開発環境を活用しました。動的に再プログラム可能な新しいデータ プレーン パケット処理ツール チェーンを用いて、パケットの分類、編集、検索、ロード バランシング、統計情報の測定など (マルチギガビット イーサネット ライン レート (10/40/100GbE) で実現可能な) 負荷の大きいデータ プレーン処理を、FPGA NIC にオフロードすることにより、Lagopus および NFV アプリケーションを高速化できました。このプロジェクトで SDN/NFV テクノロジの主要なコンポーネントである分類/検索機能を実現するには、最善のソリューションであると考えられます。SDNet 環境の迅速で再構成可能なパケット パイプライン機能により、ネットワーキング プロトコルと機能を素早く簡単に更新できます。

SDNet 環境は、クラウド コンピューティング データ センターと広域ネットワークの両方で広範囲にわたるユースケースをカバーし、Lagopus FPGA の用途を拡大します。これらを支える NTT 研究所のソフトウェア/ハードウェア協調設計技術により、差別化したネットワーク サービスの機敏な導入が可能となります。

SDNet 環境を使用したデザインの基本

SDN/NFV 市場における競合テクノロジの増加に直面して、Lagopus FPGA プロジェクトの設計課題の 1 つは、適切なタイミングで展開とプロモーションが行えるように、厳しい開発期間の枠内で作業を進めることでした。筆者らは 2014 年 10 月に Lagopus FPGA システムの設計に着手し、わずか 3 ヵ月後の 2015 年 1 月に最初の統合を完了しました。

システム設計の複雑さを考えると、これは非常に大きな成果と言えます。図 1 に、Lagopus FPGA システムのトップ レベルのアーキテクチャを示します。このアーキテクチャ




```
class:: Tuple[out] {
  struct{
    DST_MAC  : 48,
    SRC_MAC  : 48,
    VLAN_ID  : 16,
    ETH_TYPE : 16
  }
}L2KEY;
```

```
classVLAN :: Section[2] {
  ...
  method update ={
    L2KEY.VLAN_ID      =vlan,
  ...
  }
  method increment_offset = size[];
}
```

```
$ ppp -pxFile class_sdnet.px -bus lbus -dw 256 optimizedRTL
Target optimizedRTL completed.
```

図 2 — SDNet 環境の設計フロー

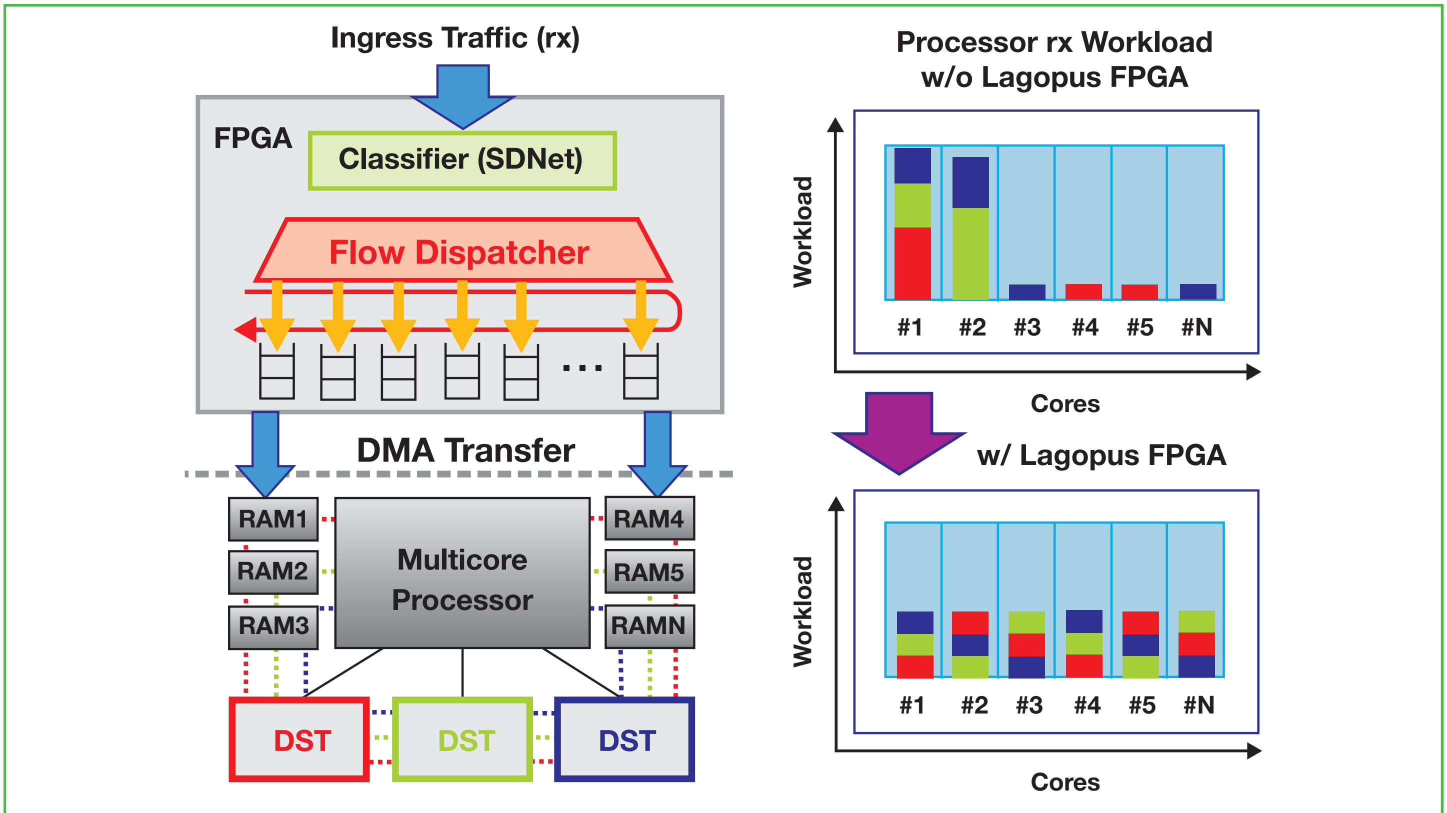


図 3 — FPGA によるフローの分類と振分け

は、ソフト FPGA IP バンドルを含む、次の 4 つの技術ソフトウェア レイヤーで構成されます。(1) NFV アプリケーション。(2) Lagopus ソフトウェア スイッチ。(3) アプリケーション プログラミング インターフェイス (API) およびインテルの データ プレーン開発キット (DPDK) (x86 高速パケット処理用の一連のライブラリおよびドライバー) などのハードウェア抽象化レイヤー。(4) FPGA NIC IP コア スイート。このように複数の技術レイヤーで構成されるため、パケットの取りこぼしや性能低下などの発生源の追跡が困難になり、デバッグしてただちに障害を切り分けることができなくなるおそれがあります。実際、これはすべての SDN/NFV アーキテクチャにおいて主要な課題となります。そこで、これらの問題を解決するために、筆者らは SDNet 環境とザイリンクスの Vivado® Design Suite を利用しました。

Lagopus FPGA のデザインでは、まず始めにパケット処理機能の要件を決め、開発フローを策定しました。図 2 に、開発フローの概要と SDNet 仕様のサンプル コードを示します。筆者らは、VLAN からの主要な情報を使用する完全一致フィルターを作成することに決めました。これにより、ハードウェア分類機能を FPGA NIC にオフロードすることで、x86 上の Lagopus のソフトウェア データ プレーンを高速化できます。OpenFlow プロトコルを使って SDN コントローラから Lagopus にフローエントリを挿入し、DPDK のフロー ディレクタ API によってフィルターのエントリを設定できます。

この手法をインプリメントするために、対応する SDNet 機能記述を作成しました (図 2 のコードを参照)。次に、バスのタイプ、バス幅、生成される RTL のタイプなどのオプションを指定して、SDNet コンパイラにコード生成を指示しました。コンパイルは数秒で完了しました。SDNet 機能記述の実際のコード サイズは約 250 行でした。対照的に、同等の RTL コードは数万行に達します。厳しい作業

スケジュールを考えると、SDNet 仕様の簡潔さは非常に高く評価できます。開発期間の厳しい制約の下でこれほど複雑なモジュールをゼロから設計、検証することは、RTL ではまず不可能でした。

次の手順では、生成された RTL と他のペリフェラル IP コアを、Vivado Design Suite 上で Tcl (Tool Command Language) シェルを使って統合しました。図 3 に、ザイリンクス Virtex-7 XC7VX690T FPGA のプログラミングをターゲットとする、統合された SDNet 分類機能とカスタマイズしたフロー振分け機能を示します。

分類されたパケット フロー (32 個の受信 [rx] DMA キューをターゲットとする) は、x86 マルチコア CPU 上で Lagopus のソフトウェア データ プレーンによって効率的に振り分けられるため、統合された FPGA デザインにより、システムは Lagopus の OpenFlow ワーカー スレッドの CPU サイクルを削減できるだけでなく、各コアのワークロードを平滑化できます (図 3)。その結果、Lagopus FPGA を使用した高性能 40Gbps ワイヤスピード ソフトウェア パケット処理を実現できました。そのためのコストは、x86 CPU の消費電力の 10% 未満に抑えられました (図 4 を参照)。

SDNet 環境と Vivado Design Suite により、プロジェクトの立ち上げが容易になり、Lagopus FPGA システムの機能セットの拡大、性能の最適化、消費電力の削減が可能になりました。SDN/NFV 分野における NTT の研究開発部門のリーダーシップとザイリンクスの SDNet 開発環境を活用すれば、通信およびクラウド インフラストラクチャの革命的な進化を実現できそうです。この目的に向かって、筆者らは、再プログラム可能なソフトウェア定義の SDNet ロード モジュールを利用して、設計手法の改善を続けていきます。API を含む SDNet 仕様が状況に応じて迅速に改訂されれば、将来のプラットフォームの定義をさらに充実させることができそうです。■

参考資料

[1] http://lagopus.github.io/

[2] K. Yamazaki, Y. Nakajima, T. Hatano and A. Miyazaki, “Lagopus FPGA: A reprogrammable data plane for high-performance software SDN switches.” Presented at Hot Chips 27, August 2015.

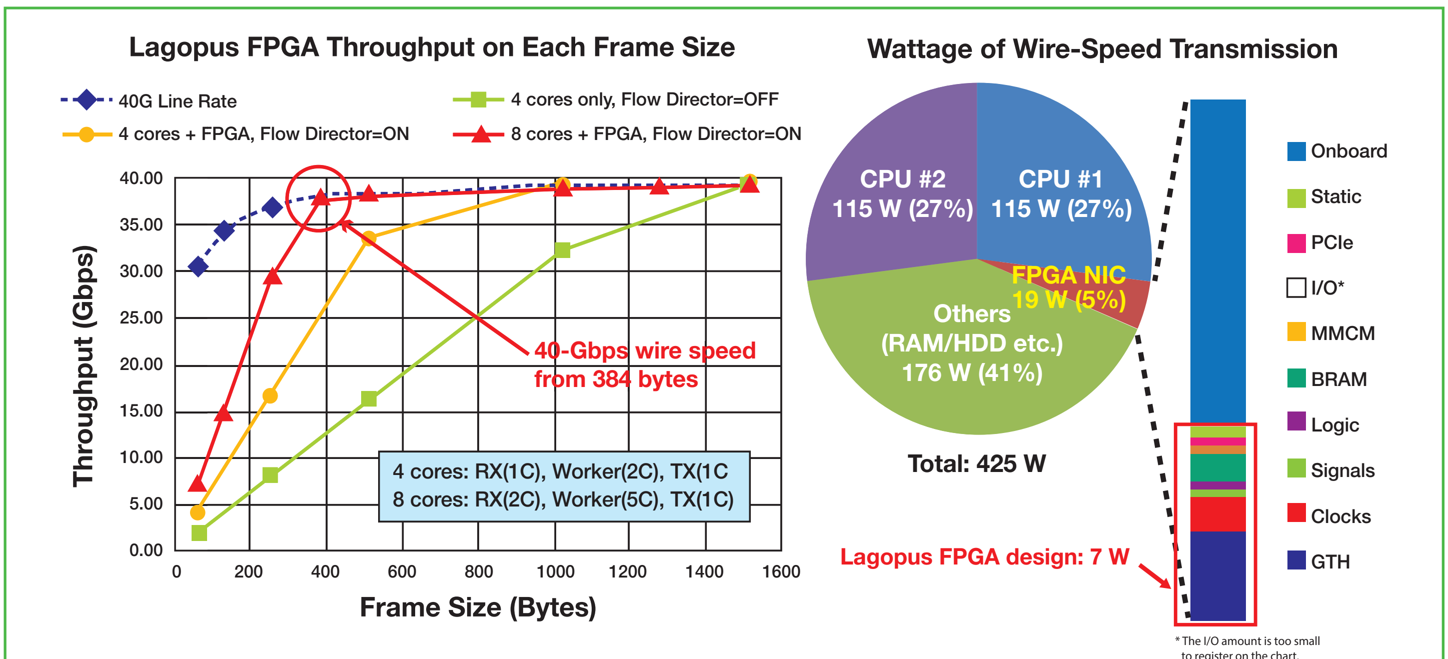


図 4 — 性能と消費電力の関係

Delivering FPGA Vision to the Masses

FPGA を活用した ビジョン システムの普及

NI 社の Vision Development Module と Vision Assistant は、マシン ビジョンの構想からプロトタイプ作成、アプリケーション導入に至るプロセスを支援します。

Kiran Nagaraj
Senior Software Engineer
National Instruments Corp.
kiran.nagaraj@ni.com

Christophe Caltagirone
Senior Software Engineer
National Instruments Corp.
christophe.caltagirone@ni.com

Dinesh Nair
Chief Architect
National Instruments Corp.
dinesh.nair@ni.com

　ビジョン システムはメインストリームへと移行しています。ビジョン技術は、コスト/利益分析結果の向上と利用可能なアプリケーションの広がりとともに、今や自動運転車から家電製品の品質検査システムまで、あらゆる機器に設計段階から組み込まれるようになっています。ビジョン システムの広汎な普及は、研究室にとどまらず、エンベデッド システムや工場にまで及んでいます。導入されるシステムは、多くの場合、I/O、広範囲に分散したカメラ、または制御ループ内のビジョンとの高度な同期を必要とします。プロセスとアプリケーションの複雑化に伴い、ビジョン システムには、より高速で高度な処理や、より厳密なタイミングと同期が求められています。

　これらの要件に応じて、ビジョン システム設計者は、専門的なタスク、I/O 要件、処理性能のニーズに対応できる、リアルタイム プロセッサと FPGA、GPU、または DSP プロセッシング ユニットを組み合わせたヘテロジニアス プロセッシング プラットフォームの利用が増えています。スマート カメラ、フレーム グラバー、ビジョン システムは、ヘテロジニアス アーキテクチャを利用してアプリケーションの要件に対応しています。

　ザイリンクスの All Programmable FPGA 製品シリーズをはじめとする FPGA の並列処理機能は、多くの画像処理アルゴリズムをインプリメントする用途に無理なく適合します。FPGA は大量データ処理と高速センサー測定の両方に使用できます。また、FPGA は驚くほど低レイテンシで動作します。システムが画像データに基づいて決定を下すまでの時間はデバイスのレイテンシの影響を受けるため、ビジョン アプリケーションにはこの条件が重要です。さらに、FPGA はジッターを回避できるため、確定性の高いプロセッシング ユニットとして機能します。

　しかし、FPGA を組み込んだヘテロジニアス システムを構築する場合、システム設計者はプログラミング上の重大な課題に直面します。それは、Time-to-Market 短縮への圧力が増すにつれて、ビジョン システム設計者には、複雑な機能を装備したソリューションのプロトタイプを迅速に作成するためのツールが必要になります。ヘテロジニアス システムのプログラミングを支援するツールは、ドメイン専門家が複数のプラットフォーム上で知的設計資産 (IP) 機能を設計でき、対象のハードウェア上でビジョン アルゴリズムをコンパイルして実行する前にアルゴリズムをテストできるもので

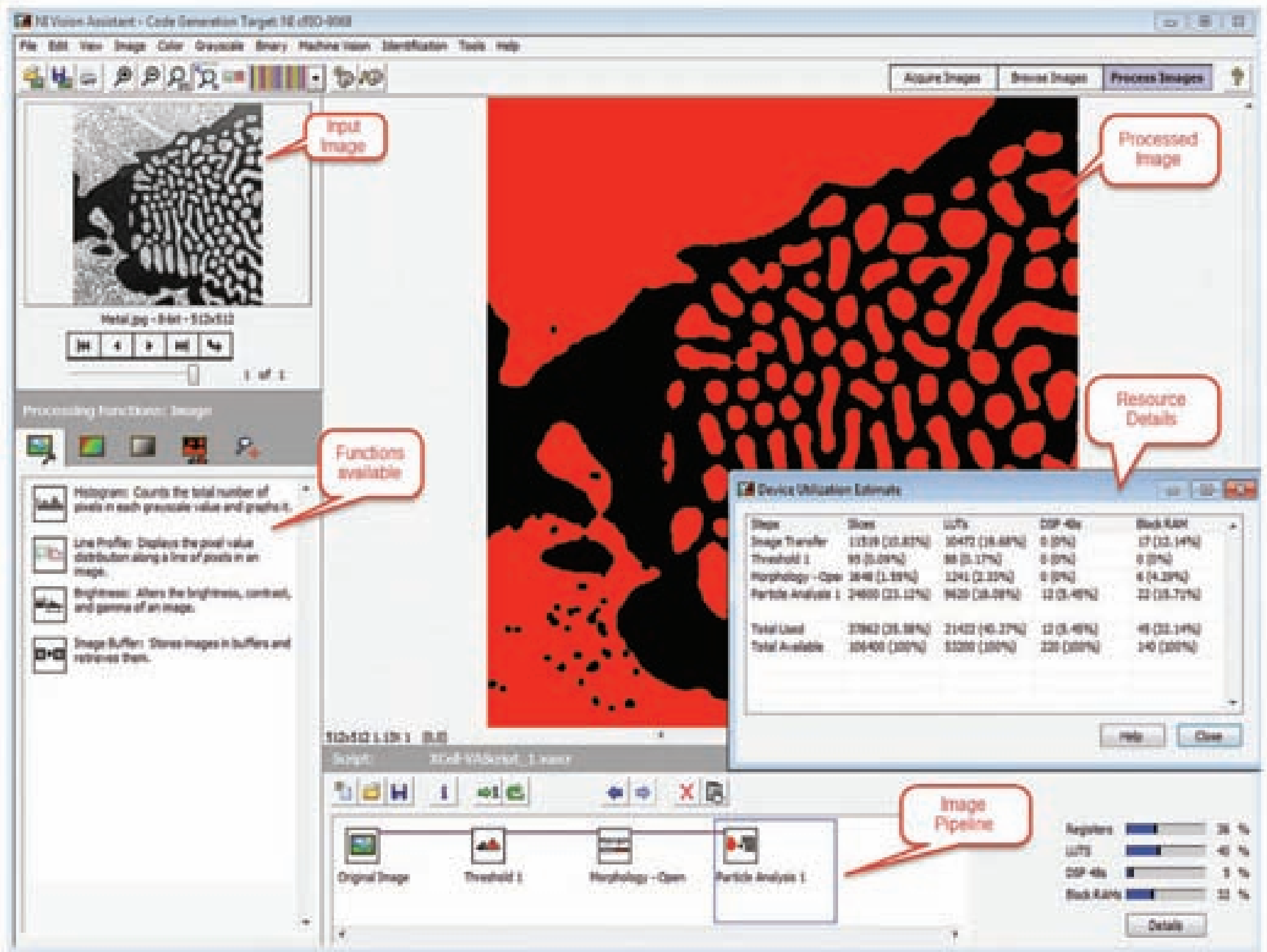


図 1 — Vision Assistant とデバイス使用率の推定値

なければなりません。このツールにより、プロトタイピングのプロセス全体を通してスループットとリソース使用率の情報に簡単にアクセスできる必要があります。

NI (National Instruments) 社ではこのプロセスをアルゴリズム エンジニアリングと呼んでいます。このプロセスにより、ドメイン専門家は、基盤となるハードウェア テクノロジに気をとられずに、目前の問題の解決に集中できます。NI 社の Vision Development Module (VDM) と付属の Vision Assistant は、設計者にこの機能を提供します。

VDM と付属の Vision Assistant は、高速プロトタイピングおよびコード生成、FPGA リソースの推定、自動コード並列化、(レイテンシ平衡化などのタスクにおける) 並列ストリームの同期を可能にします。VDM には、50 種類以上の FPGA 画像処理機能と、プロセッサと FPGA 間で画像を効率的に転送する機能が含まれています。VDM 内で Vision Assistant を使用して、FPGA ビジョン アプリケーションの迅速なプロトタイピングと開発が可能です。

コンフィギュレーション ベースのプロトタイピング

Vision Assistant はコンフィギュレーション ベースのプロトタイピング ツールであり、画像処理アルゴリズムのパラメーターを変えて繰り返し実行し、パラメーターの変更が画像にどのような影響を与えるかを確認できます。Vision Assistant は、画像パイプライン内の各ビジョン ブロック後の出力 (処理済みの画像) を可視化できます (図 1)。このツールを使用すれば、IP コアをコンパイルせずに、さまざまな画像セット上で異なったアルゴリズムとパラメーターをテストできるため、ビジョン アルゴリズムの設計にかかる時間が大幅に短縮されます。

NI 社は、FPGA プログラマの要件に添えるように、このツールをカスタマイズしました。FPGA 上にアルゴリズムを構築する際の主な課題として、FPGAファブリック上のリソース消費量、パイプラインのレイテンシ、特定のファブリックでアルゴリズムが達成できる最大周波数が挙げられます。Vision Assistant は、画像パイプライン内の各ブロックで消費されるリソースの推定値を提供して、アルゴリズムのテストを支援します。このツールにより、プロトタイプ環境および導入されたコード内でのアルゴリズムの実行結果をテストし、ユーザーのインプリメンテーションで同じ結果が得られるようにできます。

検討すべき 1 つの点は、画像フィルタリング処理にどのカーネル サイズを使用するかということです。カーネル サイズの選択は、パイプライン内のリソース使用率とレイテンシに影響を与えます。通常はカーネル サイズが大きいほど多くのリソースが必要になります。

アプリケーションに合ったカーネル サイズを選択するには、Vision Assistant を使用して、最小限のリソース使用で最大限の性能が得られるように、最良の動作性能を達成

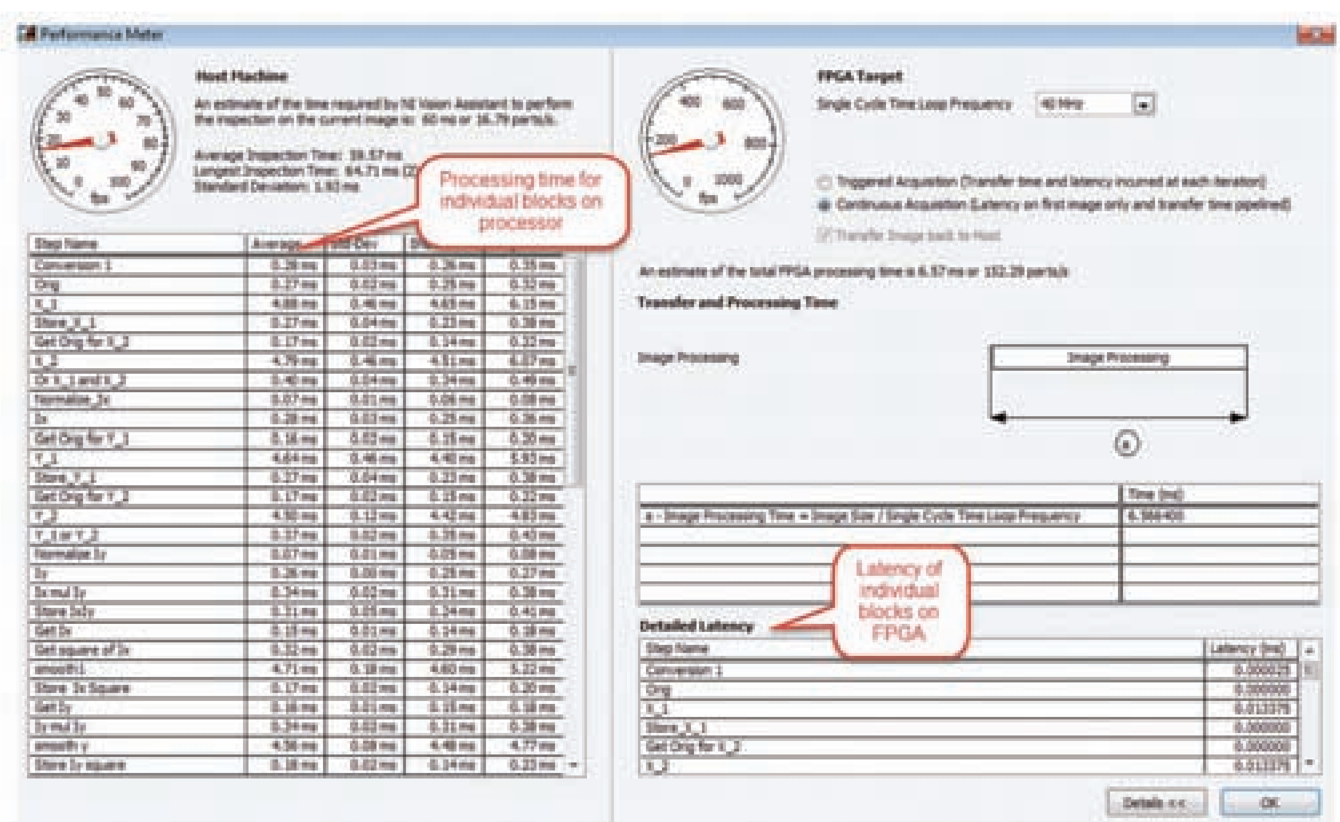


図 2 — 性能メーター ユーティリティ

するまで実験します。ビジョン IP 機能のリアルタイム推定 (図 1 を参照) は、プロトタイピング中に使用すると便利な機能です。

一般的な要件として、複数の画像パイプラインを並列で実行する場合があります。このようなシナリオでは、複数のパイプラインが 1 本のパイプラインに合流する時点で、並列パイプラインのレイテンシが平衡化されている必要があります。NI 社は、ビジョン FPGA IP ツールセットの一部として同期バッファを用意しています。Vision Assistant は、パイプライン内のレイテンシを自動的に計算して、並列パイプラインが合流する時点でレイテンシが平衡化されるように、ユーザーの代わりに同期バッファを設定します。これにより、パイプラインの最大レイテンシに基づいて、同期バッファ内の FIFO が十分な深さを持つことが保証されます。

Vision Assistant の性能メーター ユーティリティは、各フレームの処理にかかる最大時間を推定し (図 2)、パイプライン内のすべてのブロックの集合的レイテンシを表示します。NI 社のハードウェア製品ラインナップに含まれるプロセッサの大半は、リアルタイム オペレーティング システムが動作します。したがって、Vision Assistant を使って、ビジョン機能の実行に必要な時間を簡単に推定できます。

LabVIEW の経験がない皆様には、次のことを説明しておく必要があります。Vision Assistant を使えば、トランスファー バーチャル インスツルメント (VI) や DMA FIFO などのすべての依存関係と画像集録ロジックを含む、フルに機能するプロジェクトの作成が可能です。VI は、他のプログラミング言語の関数またはサブルーチンによく似ています。トランスファー VI は、ホスト/集録ロジックと FPGA の間の画像データの転送に必要になります。DMA FIFO はホスト プロセッサを利用しません。したがって、DMA FIFO は、FPGA ターゲットとホスト間の大量のデータ転送に利用できる最も高速な方法と言えます。集録ロジックは、ビジョン システムがインライン プロセッシングとコプロセッシングのどちらに基づいているかによって異なります。また、Vision Assistant は、プロセッサ上で動作するホスト VI などの他の VI とFPGA VI の作成にも役立ちます。作成した FPGA VI は、ザイリンクス Vivado® ツールでコンパイルすると、FPGA に導入するためのビットストリームを生成できます。

画像処理パイプラインを搭載したシステムは、集録ロジックが置かれる位置に応じて、インライン プロセッシングまたはコプロセッシングに大きく分類されることに注意してください。インライン プロセッシングでは、集録ロジックは FPGA 上に置かれます。カメラは集録ロジックを使用して設定され、画像は FPGA 上で処理されます。結果と処理済みの画像は、評価とさらなる分析のためにホストに送信されます。コプロセッシングでは、カメラ用の集録ロジックはプロセッサ上に置かれます。プロセッサから FPGA に画像を転送し、処理済みの画像を FPGA からプロセッサに戻すには、一定の長さの時間が必要です。画像パイプラインの処理を、プロセッサと FPGA の間で分割することも可能です。

FPGA を利用したビジョン システムの開発者は、FPGA が達成可能なスループットに注意する必要があります。スループットの情報とリアルタイムのリソース推定値に基づいて、何個の機能 (IP ブロック) を FPGA に導入できるかを決めることができます。コプロセッシングのシナリオでは、プロセッサの性能によって最終的なスループットが決まります。NI 社の Vision Development Module に付属する FPGA IP 機能を使用する場合、このことが当てはまります。これらの機能は完全にパイプライン化され、ほとんどのプロセッサよりも高い性能を発揮するからです。

プロトタイプから導入へ

Vision Development Module のビジョン FPGA IP コアにより、開発者は、大量並列処理とザイリンクス Vivado 高位合成 (HLS) ツールにより、アーキテクチャに合わせて最適化され、完全にパイプライン化された低レイテンシのビジョン IP コアを FPGA 上にインプリメントできます。NI 社のビジョン FPGA IP コアは、現在、Kintex®-7、Virtex®-5、Spartan®-6 の 3 種のザイリンクス FPGA ファミリとザイリンクス Zynq®-7000 All Programmable SoC を対象にしています。

画像は 2 次元配列と見なすことができるので、画像の操作の大半は行列に基づいています。FPGA は基本的な並列性処理によって高速性能を実現します。また、ループを使って画像の行列演算を実行できます。ループをアンロールし、アンロール後にFPGA の並列処理機能を利用していくつかのタスクを実行できます。開発者はLabVIEW

FPGA および LabVIEW FPGA IP Builder ツールにより、FPGA 上でビジョン IP コアを作成できます。

ビジョン FPGA IP 機能はシングル ピクセル処理なので、ピクセル ストリームから 1 ピクセルを受け入れ、1 ピクセルを出力します。これらの IP 機能は、イネーブル信号ベースのハンドシェイク プロトコル (4 線式ハンドシェイク プロトコル) で相互にやり取りします。このプロトコルをインプリメントする主な理由は、画像パイプライン内の機能が増えるにつれて制御パスが複雑になり、機能間でデータをシームレスに受け渡す必要があることです。4 線式プロトコルは、Single-Cycle Timed Loop (SCTL) 内に配置されたビジョン FPGA IP 機能間の損失のないデータ転送を保証します。SCTL を使用すれば、そのループ内のモジュールはユーザーが指定した周波数でクロッキングします。

図 3 に示した FPGA VI は、4 線式プロトコルとピクセル結合用の同期バッファを示しています。この 4 線式プロトコルは並列実行アルゴリズム用に設計され、プロデューサー /コンシューマー アーキテクチャでデータを処理してスループットを向上させます。さらに、4 線式ハンドシェイクが消費する FPGA 上のリソースは最小限に抑えられます。ハンドシェイク プロトコルは基盤となるビジョン機能のオーバーヘッドに影響を与えるため、このことは非常に重要です。

また、ビジョン FPGA IP コアは、パイプライン内にカスタム コードを柔軟に追加できるオープンな環境を提供します。カスタム コードには、4 線式ハンドシェイクがインプリメントされたラッパー VI が必要です。ここで画像パイプラインにカスタム コードを挿入できます。カスタム コードは完全にパイプライン化されている必要があります。そうでないと、パイプラインのインテグリティに影響を与えるおそれがあります。LabVIEW によりカスタム コードをインプリメントすることも、Lab-VIEW FPGA 内の HDL インテグレーション ノードによって VHDL 内の既存のコードを使用することもできます。

ビジョン FPGA IP ツールセットは、エッジ検出フィルター、畳み込みフィルター、ローパス フィルター、グレー モルフォロジー、バイナリ モルフォロジー、しきい値などのプリプロセッシング機能を提供します。このツールセットには、算術演算および論理演算を実行する ビジョン IP 機能や、重心などの結果を出力する機能も含まれています。ライン上のエッジを検出する機能 (Simple Edge ツール) は、

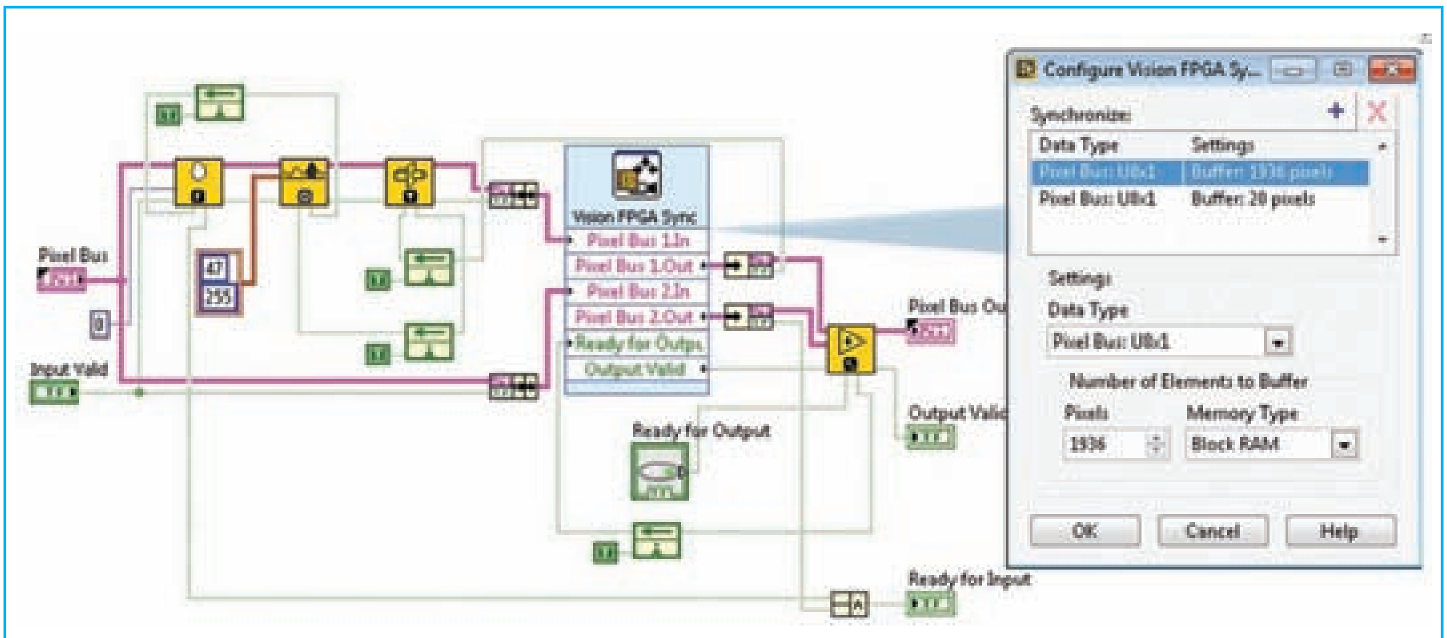


図 3 — FPGA VI の内部と同期ノード

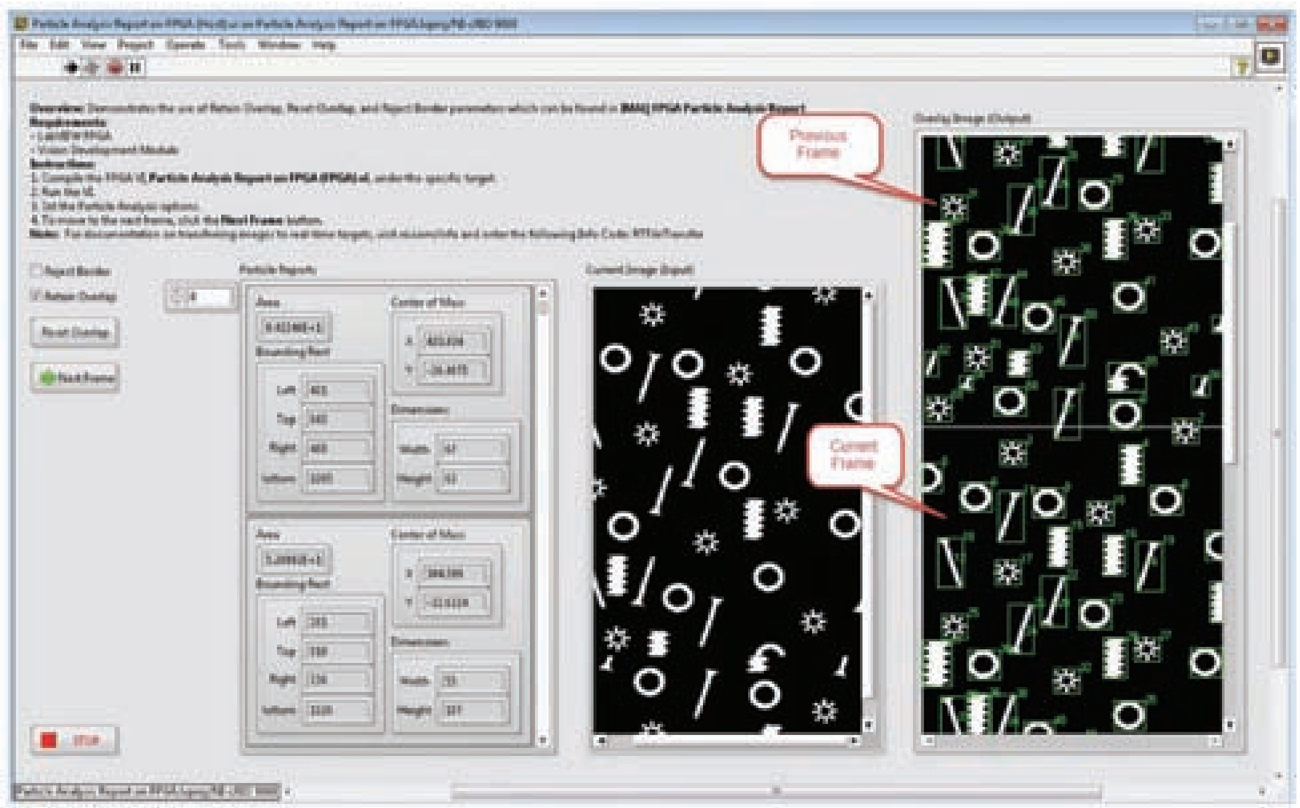


図 4 — 粒子解析レポートの例 (オーバーラップ維持有効)

キャリパー アプリケーションに便利です。Quantify 機能は、マスクされた画像と処理対象の画像ストリームを受け入れ、マスクされた画像によって定義される領域の面積、平均、標準偏差の情報を含むレポートを返します。Linear Average は、画像全体または一部の平均ピクセル強度 (平均ライン プロファイル) を計算します。

NI 社は最新のビジョン FPGA IP コアとして粒子解析レポートを追加しました。設計者は、粒子解析 (ブロブ解析) を実行して、画像内のピクセルの接続された領域またはグループを検出し、これらの領域の測定を選択して実行できます。この情報に基づいて、シリコン ウェハーの傷、電子ボード上のはんだ付け不良、モーション コントロール アプリケーションでの物体の位置などを検出できます。

この IP コアの独自の機能は、粒子情報が 2 つのフレームに分かれている場合でも粒子を検出できることです。VDM には粒子解析レポートの例が付属しています。図 4 に、画像表示機能を備えたホスト VI を示します。この機能は、検査対象の物体が 1 フレームで取り込まれるとは限らない検査システムで必要になります。

NI 社のビジョン FPGA IP 機能のほぼ 70% は、LabVIEW FPGA のユーティリティである IP Builder を使用して開発されました。IP Builder を使用すると、LabVIEW を使ってグラフィカル コードでコーディングを行い、Vivado HLS を使って RTL コードを出力できます。この手法の大きな利点は、グラフィカル コーディングをよく理解しているユーザーが、周波数およびレイテンシの要件を記述した指示子ファイルとともにアプリケーションを開発できることです。LabVIEW IP Builder と Vivado HLS を使用して、適切な VHDL コードを生成できます。設計者は画像上で配列ベースの演算を使用することができ、

Vivado HLS は、指示子セットに基づいて、VI が必要な動作周波数と最小レイテンシを達成することを保証します。

Vivado HLS は、IP Builder の生成された C コードからアルゴリズムの記述とデータ型の仕様 (整数型、固定小数点型) を抽象できるため、ビジョン開発に最適と言えます。また Vivado HLS は、機能の早期テストに必要なシミュレーション モデルを生成します。アーキテクチャに合わせて生成される VHDL コードにより、高品質で再現性の高い結果が得られます。

NI 社は、オープンで柔軟なシステムと、その活用のための適切なソフトウェア ツールを提供する取り組みを進めています。開発者は、ヘテロジニアス アーキテクチャに基づくビジョン システムを、ますます広範囲にわたるアプリケーションに設計段階から組み込んでいます。こうしたヘテロジニアス システムのソフトウェア デザインの次なるフロンティアは、アルゴリズムのコンポーネントをどこに導入するかをインテリジェントに決定する (各種のシステム コンポーネント (CPU、GPU、FPGA) の機能とリソースに基づいてそれを判断する) コンパイラまたはアプリケーション開発エンジンにありそうです。

先進の製品とプロセスにより、これまで以上に優れたビジョン システムが実現されるにつれて、アプリケーション開発者には、ビジョン機能のための効果的なプロトタイピング/アルゴリズム開発環境が必要になります。開発者およびドメイン専門家に適切なツールが提供されることで、ビジョン システム デザインの普及に向けた次世代のイノベーションが促されます。

NI 社のビジョン FPGA IP コアを試してみたい方は、LabVIEW FPGA と VDM をインストールする必要があります。まず http://www.ni.com/vision/ から 30 日間有効の評価版をインストールし、その後に評価期限を延長またはライセンスを購入できます。■

A Novel Approach to Software-Defined FPGA Computing

ソフトウェア定義の FPGA コンピューティングの新たな手法

Stephane Monboisset
Director, QuickPlay Marketing
and Business Development
PLDA
smo@quickplay.io

QuickPlay の高レベル ワークフローにより、ソフトウェア開発者は効率的な FPGA ベースのアプリケーションをただちに構築できます。

モノのインターネット (IoT) とビッグデータ処理の普及につれて、データ転送と処理能力のニーズも急激に増大し、CPU だけではもはや対応が困難になっています。アプリケーションを実行するプロセッサや仮想マシンの数を増やしても、アプリケーションごとに複数の CPU で並列化できる処理は限られているため、十分な効果は余り期待できません。他方、フィールド プログラマブル ゲート アレイ (FPGA) は、純粋な処理の面だけでなく (同じくらい重要な) 消費電力の面から見ても、必要な I/O 帯域幅と処理能力を提供します。データ センター機器メーカーは、以前から FPGA の採用を魅力的な構想であると考えていました。先頃インテルが業界第 2 位の FPGA ベンダーを買収したことは、CPU のみのソリューションではもはや十分ではないと考えたあかしと言えるでしょう。

FPGA が広く普及するに際して、インプリメンテーションの複雑さが主な障害の1つとなっていました。従来、FPGA ベースのプラットフォーム上でアプリケーションを開発するには、低レベルのハードウェア インプリメンテーションの扱いが必要でした。このため、多くの潜在的な顧客 (ソフトウェア開発者) が FPGA から遠ざけられる一方で、従来の FPGA 設計者の作業はますます複雑化を増してきました。

FPGA デザインの最近の手法は、高位合成 (HLS) ツールと OpenCL™、C、C++ などのソフトウェア プログラミング言語の利用を中心とした、ソフトウェア開発者が多くのアプリケーションで FPGA ベースのハードウェア アクセラレーション開発の恩恵を受けられる手段を提供しています。しかし、これらの手法には、必要不可欠な要素が 1 つ欠けています。それは、ソフトウェア開発者がアプリケーションに最適なハードウェア インフラストラクチャを自分で定義し、コンフィギュレーションできる機能です。そこで業界は、特に FPGA の専門知識がなくても FPGA ベースのプラットフォーム上でアプリケーションをインプリメントできるように、高レベル ワークフローの実現という目標を追求してきました。

過去 5 年間、PLDA 社はこのようなワークフローの開発に取り組んできました。PLDA 社のワークフローは QuickPlay と呼ばれ、インプリメンテーションの複雑性の問題に効率的に対処し、FPGA 開発のさまざまな利用モデルを実現します。その中核となる真価の 1 つは、ソフトウェア開発者が、CPU 向けに作成したアプリケーションの一部または全体を FPGA ハードウェア上にインプリメントできることです。QuickPlay は、すべての FPGA リソースを利用して、高性能ではあるが複雑性の高い FPGA のハードウェアを、ソフトウェアで定義されたプラットフォームに変換します。これにより、ハードウェア デザインに労力を費やさなくても、FPGA のメリットを享受できるようになります。

2 つの関数に分解されるソフトウェア アルゴリズムの場合を考えます。1つ目の関数で処理されたデータは、次の処理のために 2つ目の関数に送られます。ソフトウェアの観点から見ると、このインプリメンテーションは、処理されるデータの位置を指すポインタを使用して、Function1() の呼び出しに続いて別の Function2() の呼び出しを配置した簡単な構造です。

適切なハードウェア抽象化ツール フローを使用せずに、このようなアルゴリズムを FPGA ベースのハードウェア プラットフォーム上にインプリメントする場合、ソフトウェア開発者は図 1 のようなハードウェア デザインを作成する必要があります (カーネル 1 とカーネル 2 はそれぞれ関数 1 と関数 2 のハードウェア インプリメンテーション)。このハードウェア デザインは、制御プレーンとデータ プレーンの 2 つの要素を含んでいなければなりません。

制御プレーンは、クロックとリセットの生成、システム起動の管理、データ プレーン処理のオーケストレーション、すべてのハウスキーピング機能を実行する実行エンジンです。データ プレーンは、プロセッシング エレメント (カーネル 1 およびカーネル 2) と (データの読み込みおよび処理済みデータの書き出しに必要な) I/O インターフェイスをインスタンシエートし、接続します。この例では、使用するインターフェイスはイーサネットと PCI Express (PCIe) です (図 1 を参照)。アプリケーションの要件によっては、異なる I/O インターフェイスが必要になります。

ソフトウェア開発者は、特にハードウェアの専門知識がなくても、(通常は C または C++ で記述される) ソフトウェア関数 Function1() および Function2() をコンパイルして VHDL または Verilog の FPGA ハードウェア記述に変換する HLS ツールを使用して、カーネル 1 およびカーネル 2 を簡単に生成できます。ただし、アルゴリズムの性質を持たないデザイン内の他の要素 (インターフェイス、

コントロール、クロック、リセット) は、HLS ツールでは生成できません。これらの要素は、ハードウェア設計者がカスタム ハードウェア記述言語の機能または IP コアとして設計する必要があります。これらの要素の入手や接続の作業も問題となります。一部の要素がすぐには利用可能でない場合や、異なるインターフェイス (タイプおよびサイズ)、クロッキング要件、特定の起動シーケンスなどを必要とする場合があるからです。

デザイン作業の先には、(同じくらい難しい) インプリメンテーション作業が待っています。ここでは、選択した FPGA プラットフォームのリソースにデザインをマッピングし、適切な制約を生成し、ロジック合成および FPGA ハードウェアへのインプリメント後にこれらの制約が満たされることを確認する必要があります。新たな FPGA ハードウェア上に正常に機能するデザインをインプリメントするには、経験豊富なハードウェア設計者でも数週間かかることがあります。

したがって、ソフトウェア開発者がカスタム ハードウェアでアプリケーションを強化するためのツールには、次の機能が必要です。

- 純粋なソフトウェア コードからハードウェア機能を作成する
- 必要に応じて既存のハードウェア IP ブロックを統合する
- すべてのサポート ハードウェア (インターフェイス、コントロール、クロックなど) の推論と作成
- 汎用市販ボードとカスタム プラットフォームの使用をサポートする
- 生成されたハードウェアが適切に構築されていることを保証し、ハードウェア デバッグを不要にする
- 標準的なソフトウェア デバッグ ツールのみを使用した機能ブロックのデバッグをサポートする

PLDA 社は、これらすべての要件を満たすように、QuickPlay をゼロから開発しました。したがって、純粋なソフトウェア開発者は、ソフトウェア アーキテクチャ内で最小限の労力で FPGA を指定、構築、統合できます。

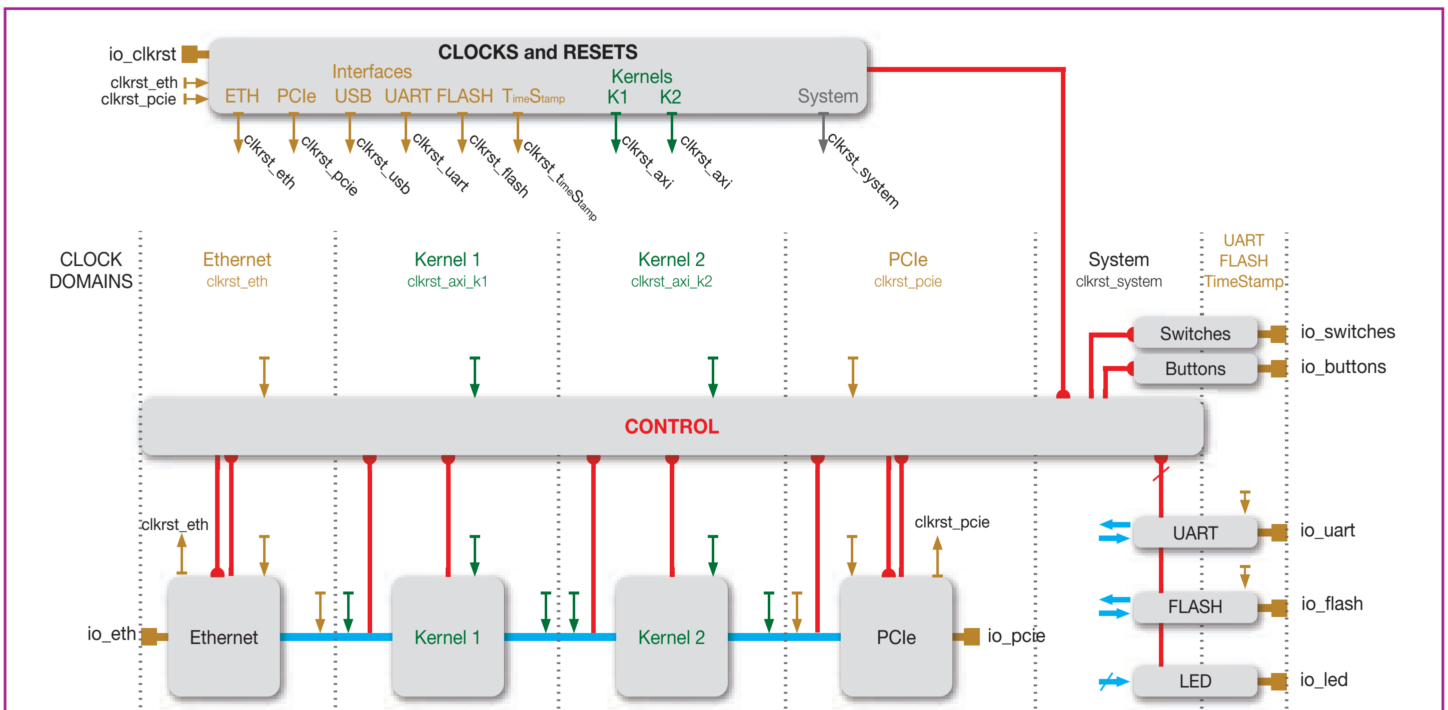


図 1 — 従来の FPGA ツールを使用した 2 機能アルゴリズムの詳細なハードウェア インプリメンテーション

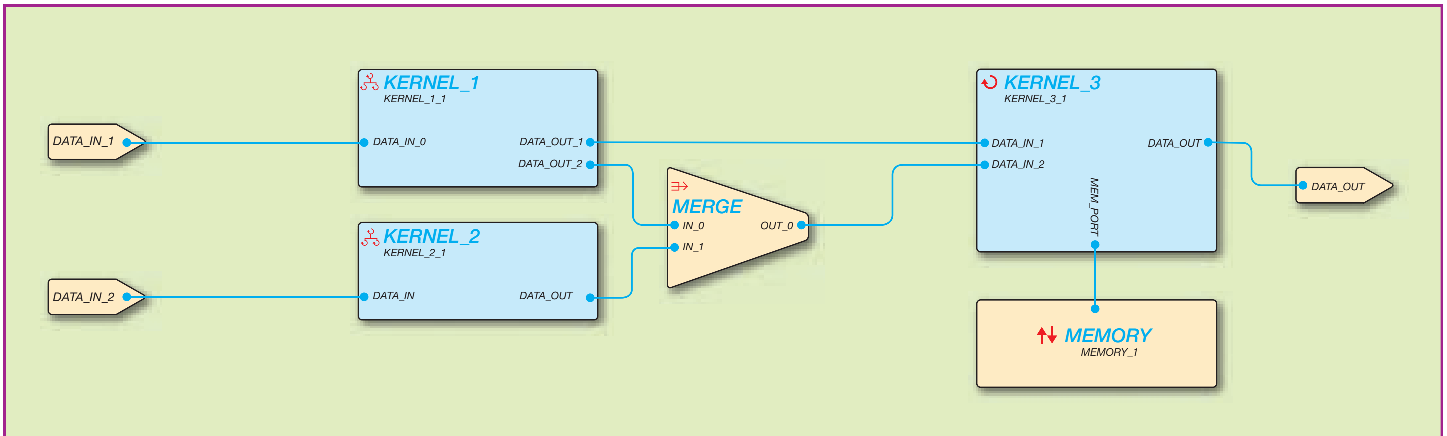


図 2 — QuickPlay によるデザインの例

ソフトウェア中心の手法

QuickPlay を使用してデザインをインプリメントする全体的なプロセスは、非常にシンプルです。

1. ハードウェア エンジンの C/C++ 機能モデルを作成します。
2. 標準的な C/C++ デバッグ ツールで機能モデルを検証します。
3. ターゲット FPGA プラットフォームと I/O インターフェイス (PCIe、イーサネット、DDR、QDR など) を指定します。
4. ハードウェア エンジンのコンパイルと構築を実行します。

このプロセスは簡単に思われますが、このプロセスがシームレスに機能するには、生成されるハードウェア エンジンが元のソフトウェア モデルと同じように機能することを保証する必要があります。つまり、ハードウェア インプリメンテーションがどれほど高速に動作する場合でも、ハードウェアの実行とソフトウェアの実行で全く同一の結果が得られるように、機能モデルは確定的でなければなりません。

残念なことに、ほとんどの並列システムには非確定的実行の問題が発生します。たとえば、マルチスレッド ソフトウェアの実行は、CPU、OS、および (同じホスト上で動作する) 相関性のないプロセスに依存します。同じマルチスレッド プログラムを複数回実行すると、異なる動作をすることがあります。ハードウェア内のこのような非確定性のために、ハードウェア エンジンそれ自体のデバッグが (電気波形レベルで) 必要になり、ソフトウェア開発者に対してハードウェアを抽象化するというツールの目的を達成できなくなるおそれもあります。

QuickPlay は、実行エンジンに関係なく確定的な実行を数学的に保証する、直感的なデータフロー モデルを使用します。このモデルは、ストリーミング チャネルと通信する、カーネルと呼ばれる同時機能で構成されます。したがって、このモデルには、ソフトウェア開発者が描くアプリケーションの設計図との高い相関性があります。確定的な動作を保証するには、カーネルが相互に通信する際に、競合状態やデッドロックなどのデータ ハザードが発生しないようにする必要があります。これは、(1) FIFO ベース、(2) ブロッキング読み出しおよびブロッキング書き込み、(3) ポイント ツー ポイントの 3 つの条件を満たすストリーミング チャネルによって実現します。

これらの条件は、PLDA 社の QuickPlay の基盤となる演算モデルであるカーン プロセス ネットワーク (KPN) の特性です。図 2 に示す QuickPlay のデザイン例は、KPN モデルの例を示しています。

カーネルの内容は、任意の C/C++ コード、サードパーティ製 IP コア、または (ハードウェア設計者用の) HDL コードです。QuickPlay は、わかりやすい設計フローを提供します (図 3)。

ここで、QuickPlay デザイン プロセスの各ステップについて詳しく説明します。

ステップ 1: 純粋なソフトウェア デザイン。このステージでは、プロセッシング カーネルを C で追加して接続し、ホスト ソフトウェアとの通信チャネルを指定することにより、FPGA デザインを作成します。QuickPlay の Eclipse ベースの統合開発環境 (IDE) は、カーネル、ストリーム、ストリーミング ポート、メモリ ポートの作成と、ストリーミング ポート

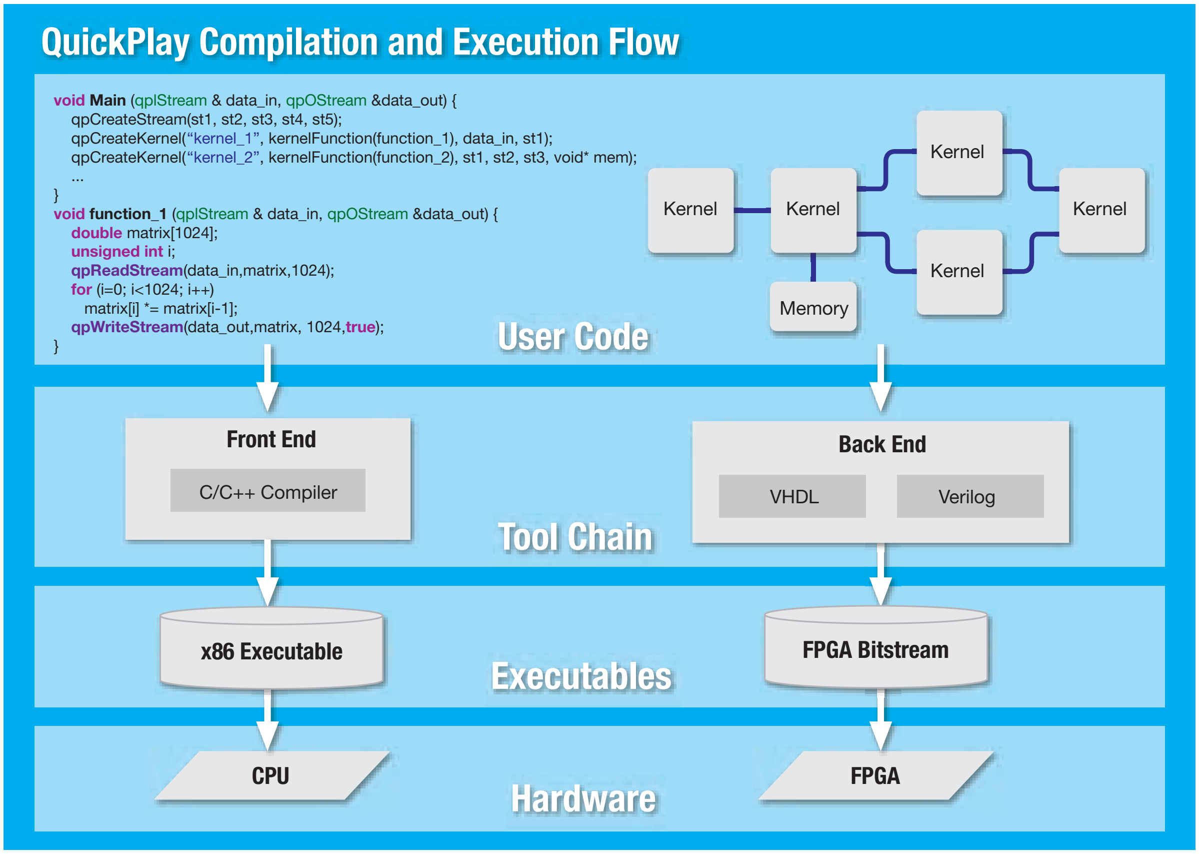


図 3 — QuickPlay は簡単な設計フローを実現します。

およびメモリ ポートに対する読み出しと書き込みに使用できる、簡単な API を備えた C/ C++ ライブラリを提供します。

また、QuickPlay IDE は、カーネルや他の設計要素をドラッグ アンド ドロップしてストリームを描画できる、直感的な操作が可能なグラフィカル エディターを備えています。

ステップ 2 : 機能検証。このステップでは、ステップ 1 で作成したソフトウェア モデルが正常に機能することを確認することに重点を置きます。そのために、デスクトップ上でソフトウェア モデルをコンパイルし、入力ポートにデータを送信するテスト プログラムを使ってソフトウェア モデルを実行し、出力の正しさを検証します。FPGA デザインのソフトウェア モデルは並列実行され、各カーネル別のスレッドが実際のハードウェア インプリメンテーションの並列性を模倣します。

次に、ブレークポイント、ウォッチポイント、ステップごとの実行、printf などの標準的なソフトウェア デバッグ手法およびツールを使用して、ソフトウェア モデルをデバッグします (モデルがハードウェアにインプリメントされたら、おそらくさらにテストを実行したくなります。これについては後で簡単に説明します)。設計フローの観点から見ると、すべての検証作業はこの段階で行われます。このデバッグ フェーズが完了し、機能の問題がすべて解決されたら、ハードウェア レベルでさらにデバッグを行う必要はありません。

この機能モデルには、ハードウェア インフラストラクチャの要素は全く含まれていないことに注意してください。上記の例では、簡単な 2 機能のモデルに焦点を合わせています。図 1 で追加されているシステムの要素 (通信コンポーネント、制御プレーン、クロック信号、リセット信号) は、このモデリング/検証フェーズでは影響を及ぼしません。

ステップ 3 : ハードウェアの生成。このステップでは、ソフトウェア モデルから FPGA ハードウェアを生成します。このステップは、次の 3 つの簡単な手順で構成されます。

1. QuickPlay GUI のドロップダウン メニューを使用して、デザインをインプリメントする FPGA ハードウェアを選択します。QuickPlay は、最先端のザイリンクス All Programmable FPGA、PCIe 3.0、10ギガビット

イーサネット、DDR3 SDRAM、QDR2+ SRAM などを搭載した既製ボードにデザインをインプリメントすることができます。この対応ボードの数は増え続けています。

2. 物理インターフェイスを選択し (結果的に通信プロトコルも) 、デザインの入力ポートと出力ポートにマッピングします。これもメニュー上で選択するだけの簡単な操作です。どのインターフェイスを選択するかは、PCIe、TCP/IP over 10-Gbit Ethernet、UDP over 10-Gbit Ethernet など、選択した FPGA ボード上で利用可能なインターフェイスによって異なります。通信プロトコルを選択すると、その接続をインプリメントするのに必要なハードウェア IP ブロックだけでなく、その上のソフトウェア スタック レイヤーも自動的に呼び出され、システム全体が作成されます。

3. 構築プロセスを開始します。プロセスを開始すると、(C コードからハードウェアを作成する) HLS エンジンが実行されて、必要なシステム ハードウェア機能 (筆者らの元の例では制御プレーン ロジック) が作成され、ボードが要求するハードウェア イメージの構築に必要なその他のツール (ザイリンクスの Vivado® 統合設計環境など) が実行されます。このプロセスは人の介入なしに自動的に完了します。

ステップ 4 : システムの実行。このステップは、ステップ 2 (機能検証) の機能モデルの実行によく似ていますが、ホスト アプリケーションはソフトウェア内で動作するのに対して、FPGA デザインは選択した FPGA ボード上で動作する点が異なります。つまり、FPGA ボードとの間で実際のデータをストリーミングできるため、機能の検証対象範囲が広がるメリットがあります。このテストは機能検証テストよりもはるかに高速に実行され、ライブ データ ソースを使用できるため、このステージでは、機能検証段階よりも多くのテストを実行できる可能性があります。

ステップ 5 : システム デバッグ。前のステップでは機能検証フェーズよりも多くのテストを実行するため、ステップ 2 では見つからなかった機能バグが見つかることもありそうです。これはどのようにデバッグすればいいでしょうか。

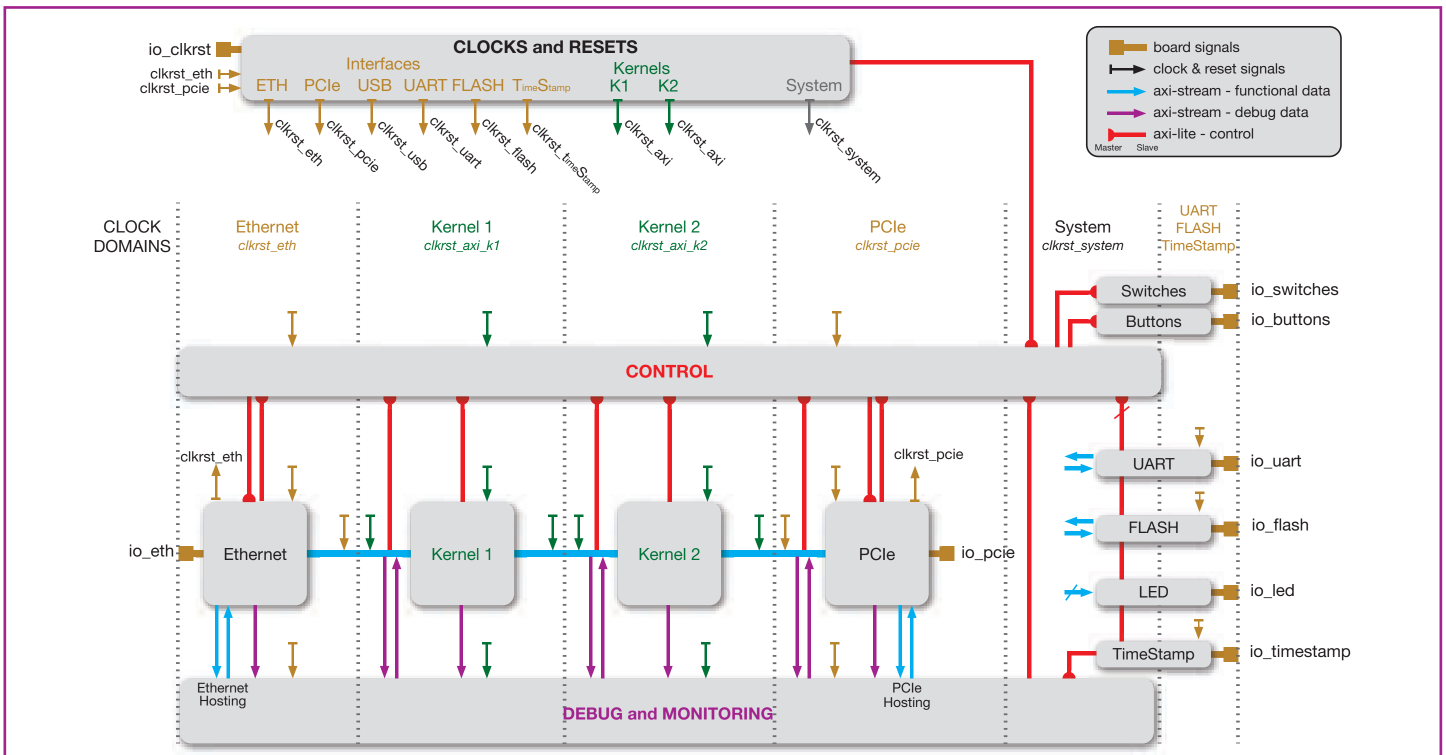


図 4 — デバッグ インフラストラクチャは自動的に作成されます。

既に述べたように、ハードウェア内で機能を実行した後にバグが見つかった場合でも、ハードウェア レベルでデバッグを行う必要はありません。Quick Play はソフトウェア モデルとハードウェア インプリメンテーションが機能的に同等であることを保証しているので、ハードウェア版のバグはソフトウェア版にも存在するはずです。したがって、ハードウェア内でデバッグを行う必要はなく、ソフトウェア ドメインに限定してデバッグを実行すれば十分です。

不合格になったテスト シーケンスがハードウェア内で特定されると、QuickPlay は、不正な動作を発生させた一連のイベントをデザインの入力側でキャプチャし、ソフトウェア環境内で再生できます。設計者はソフトウェア環境内でデバッグを行い、Eclipse デバッガを使ってバグの発生源を特定できます。

これを可能にするために、QuickPlay は、デザインのすべてのクリティカル ポイントを観察するためのハードウェア インフラストラクチャを自動的にプロビジョニングします。このインフラストラクチャを無効にすると、貴重なハードウェア リソースを解放できます。図 4 に、デバッグ回路が追加されたサンプル システムを示します。QuickPlay を使用しない場合は、何らかのデバッグ インフラストラクチャを手作業で挿入し、管理しなければなりません。QuickPlay を使用すれば、この処理はすべてソフトウェア開発者に対して透過的に、自動的に実行されます。

全体的なプロセスは、ソフトウェア内でのモデル化、システムの構築、ハードウェア内でのテストという順に進められます。バグが見つかった場合は、不合格になったテスト シーケンスをソフトウェア環境に再度インポートして、そこでデバッグを行い、ソース コードを修正し、このプロセスを繰り返します。この方法により、従来のフローに比べて生産性が飛躍的に向上します。

ステップ 6 (オプション) : システムの最適化。デバッグ フェーズが完了すれば、FPGA ボード上で正常に動作する機能デザインが得られます。ただし、ここで性能を多少最適化することができます。システムが正常に動作することは既にわかっているので、この段階で最適化を行うのが適切です。

最初に検討すべき最適化手法は、機能モデルの改善です。ここには並行処理を増やす機会がありそうです。たとえば、異なる方法で機能を分解またはリファクタリングしてみることができます。このレベルでは、最適化によって性能が目覚ましく向上する可能性があります。言うまでもなく、VHDL または Verilog デザインを使ってこのような最適化を行うには長時間が必要となりますが、C でデザインを変更するプロセスは迅速で簡単です。

第 2 に、より高速な FPGA を搭載した異なる FPGA ボードを試すことができます。機能モデルからボードへのマッピングは非常に容易なので、最適なボードを選ぶためにさまざまなボードを試すことは簡単です。

第 3 の最適化手法は、QuickPlay が高位合成によって作成するハードウェア カーネルに関係があります。得られたハードウェアは正常かつ効率的に機能することが保証されていますが、ハードウェア技術者が手作業で作成したハードウェアほど効率的に動作するとは限りません。このステージにはいくつかのオプションがあります。コードを最適化し、QuickPlay HLS の設定を調整して、生成されるハードウェアを改善できます。また、Vivado HLS を使用して、より効率的なハードウェアを生成することもできます。あるいは、最も重要なブロックはハードウェア設計者が HDL で作成することも可能です。

これらの最適化手順はどれも必須ではありませんが、より高性能なハードウェアが必要で、利用可能なハードウェア デザイン リソースが限られている場合にオプションで利用できます。これらの最適化手法については、ハードウェア技術者の支援を得ることもできるでしょう。これらの変更が完了したら、構築プロセスを繰り返します。

汎用ストリーミング関数

QuickPlay は、基盤となる物理的通信プロトコルを完全に抽象化する、汎用ストリーミング API を提供します。ストリーミング データは、ReadStream() 関数によって受信され、WriteStream() 関数を使って送信されます。これらの関数を使用して、カーネルとカーネルの間、エンベデッド メモリまたはボード レベル メモリとの間、あるいはエンベデッド CPU または外部ホスト CPU との間で、データの送信と受信ができます。これにより、ボードのアーキテクチャの柔軟性が得られます。開発者が基盤となる低レベル プロトコルを理解または管理する必要はありません。

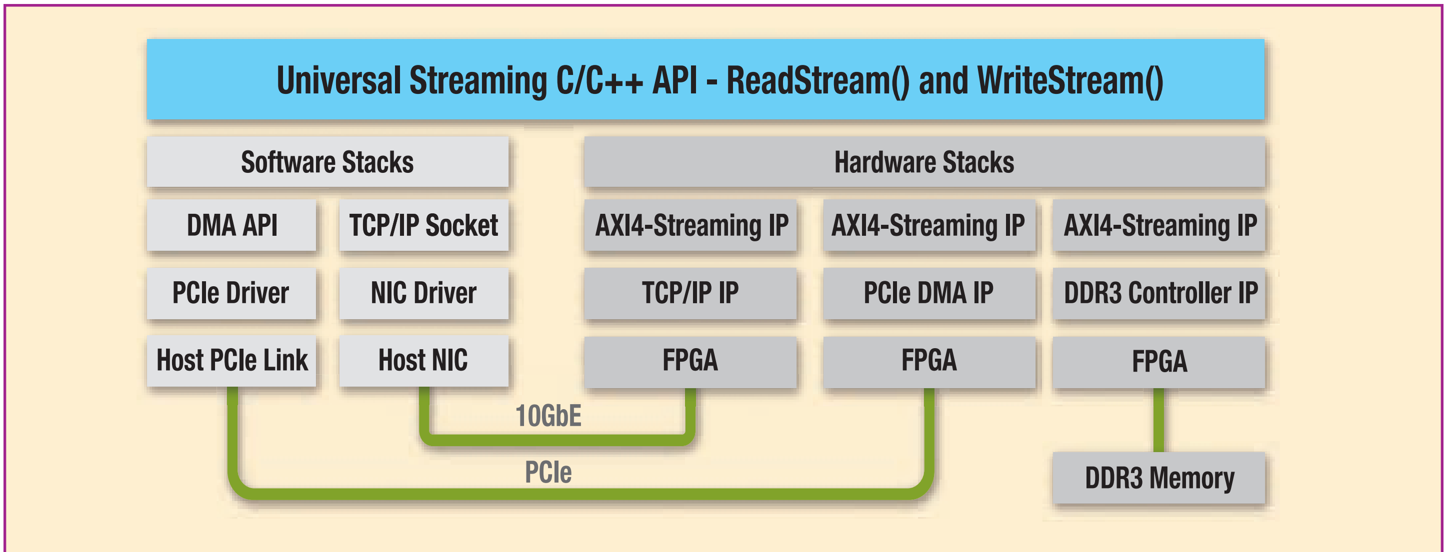


図 5 — 希望のプロトコルを選択すると、必要なハードウェアとソフトウェア スタックがセットアップされます。

データの送受信に使用されるハードウェアは、選択したプロトコルによって決まります。現在、QuickPlay は、ARM® AMBA® AXI4-Stream、DDR3、PCIe (DMA を使用)、TCP/IP をサポートしており、今後必要に応じてプロトコルが追加される予定です。希望のプロトコルを選択すると、そのプロトコルのインプリメントに必要なハードウェアだけでなく、上位のプロトコル レイヤーのサポートに必要なソフトウェア スタックもセットアップされます (図 5 を参照)。

QuickPlay は、これらの読み出しおよび書き込みの正確なインプリメンテーション (サイズ、アライメント、マーシャリングなど) を管理します。ReadStream() 文および WriteStream() 文の最も重要な特性は、ブロッキング文であることです。いずれかの文が検出されると、予想されるデータの読み出しまたは書き込みがすべて完了するまで、実行は次の文に渡されません。このことはアルゴリズムの確定性を実現するために重要です。

ReadStream() および WriteStream() ジェネリック文と、基盤となる実際のプロトコル ハードウェアは、QuickPlay ライブラリによって実行時にバインドされます。これにより、通信プロトコルの詳細によってソフトウェア プログラムが複雑になることを防げるだけでなく、モジュール性と移植性が実現します。実際のカーネル コードやホスト ソフトウェアを変更しなくても、通信プロトコルを簡単に変更できます。ReadStream() 文および WriteStream() 文は、プログラムのセマンティクスに影響を与えることなく、選択したプロトコルに自動的にバインドされます。

QuickPlay が提供する抽象化の結果、ソフトウェア アルゴリズムは純粋なままに保たれ、基盤となる通信プロトコルの詳細から完全に独立した形で、データ操作のみに集中できます。

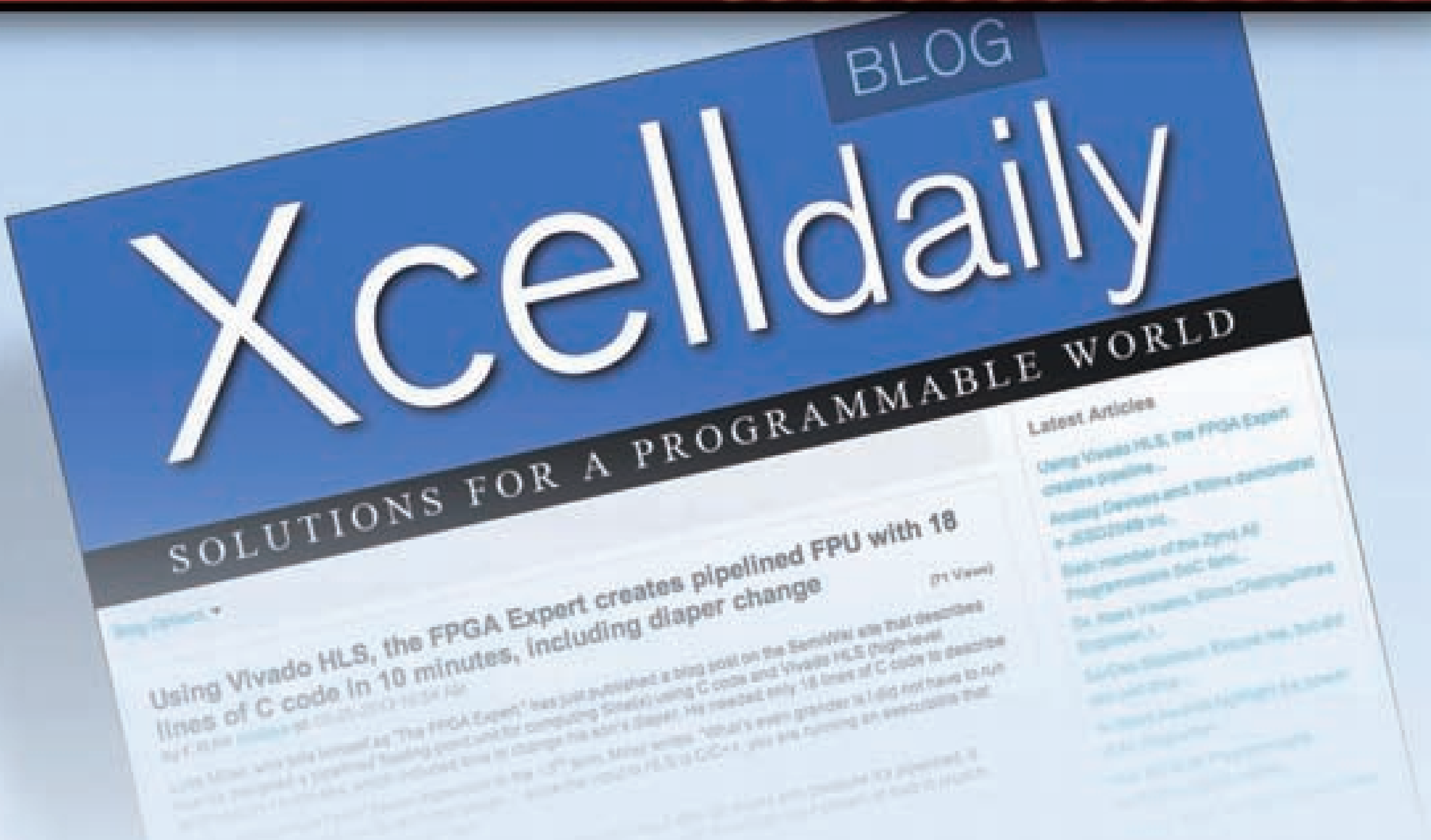
量産品質の結果

HLS ツールによっては、より効率的なハードウェアを生成できるコーディング スタイルの学習によって結果が改善される場合もありますが、この手法は必須ではありません。

QuickPlay 以外のツールを使用した場合、使用されるハードウェア プラットフォームは単なるプロトタイピングの手段と見なされがちですが、QuickPlay を使って作成されるシステムは量産品質に達しています。純粋なソフトウェア インプリメンテーションから、ハードウェアで支援されるインプリメンテーションまたはハードウェアのみのインプリメンテーションへの移行は、従来の方法では数カ月かかります。QuickPlay はこの期間を数日に短縮します。

QuickPlay 手法により、長らく追求してきた目標が達成され、ソフトウェア技術者がアプリケーションの全体または一部のハードウェア インプリメンテーションを作成できるようになります。ソフトウェア技術者は、使い慣れたドメインで作業しながら、必要に応じてカスタム ハードウェアを利用して、ハードウェアで強化されたアプリケーションを自動的に生成できます。このアプリケーションは、ソフトウェアのみのアプリケーションよりも効率的に動作し、手作業で作成されるデザインよりも数カ月早く量産体制に移行できます。■

Xcell Journal を拡充。新たに Daily Blog を追加



ザイリンクスは、数々の受賞歴がある Xcell Journal をさらに拡充し、エキサイティングな Xcell Daily Blog（英文）を始めました。このブログでは、コンテンツを頻繁に更新し、技術者の皆様がザイリンクスの製品とエコシステムの多岐にわたる機能が活用でき、All Programmable システムおよび Smarter System の開発に役立つ情報を提供します。

Recent（最近の記事）

- ステップごとの Zynq ベース チュートリアルにより、Echo Server プロジェクトで 1G イーサネット デザインを実現
- Electronic Design 社の William Wong 氏が 100Gbps Digilent NetFPGA-SUME ネットワーキング開発ボードを評価
- SDNet による再プログラム可能ネットワークのイノベーション
- SDAccel を使用した FPGA インプリメンテーションにより、NVMe over Fabrics サービスを 37.5 倍高速化
- Adam Taylor の MicroZed クロニクル、第 114 回 : Avnet Embedded Vision Kitを使用したエンベデッド ビジョン アプリケーションの開発

ブログ : www.forums.xilinx.com/t5/Xcell-Daily/bg-p/Xcell

XILINX
ALL PROGRAMMABLE.