

SDSoC プロファイリング および最適化ガイド

UG1235 (v2019.1) 2019 年 5 月 22 日

この資料は表記のバージョンの英語版を翻訳したもので、内容に相違が生じる場合には原文を優先します。資料によっては英語版の更新に対応していないものがあります。日本語版は参考用としてご使用の上、最新情報につきましては、必ず最新英語版をご参照ください。

改訂履歴

次の表に、この文書の改訂履歴を示します。

セクション	改訂内容
2019 年 5 月 22 日 バージョン 2019.1	
資料全体	編集上のアップデートのみ。技術内容の変更なし。

目次

改訂履歴.....	2
第 1 章: 概要.....	5
SDSoC を使用したソフトウェア アクセラレーション.....	5
SDSoC アプリケーションの実行モデル.....	6
SDSoC のビルド プロセス.....	8
第 2 章: パフォーマンスの見積もり.....	10
パフォーマンスを計測するためのコードのプロファイリングおよび計測用関数呼び出しの追加.....	10
SDSCC/SDS++ パフォーマンス見積もりフロー オプション.....	11
第 3 章: システム パフォーマンスの向上.....	14
ハードウェア関数の並列機能の改善.....	14
システムでの並列処理および同時処理の増加.....	16
SDSoC でのデータ モーション ネットワークの生成.....	18
第 4 章: ハードウェア関数の最適化.....	25
ハードウェア関数の最適化手法.....	25
ハードウェア関数のベースライン.....	27
メトリクスの最適化.....	28
パフォーマンスのためのパイプライン処理.....	28
パフォーマンスのための構造最適化.....	32
レイテンシの削減.....	34
エリアの削減.....	35
デザイン最適化のワークフロー.....	36
最適化ガイドライン.....	37
ハードウェア関数のインターフェイス.....	45
第 5 章: メモリ アクセスの最適化.....	47
データ モーションの最適化.....	47
データ アクセス パターン.....	50
第 6 章: AXI Performance Monitor を使用したパフォーマンス計測.....	65
スタンドアロン プロジェクトの作成と APM のインプリメント.....	65
スタンドアロン システムの監視.....	66
Linux プロジェクトの作成と APM のインプリメント.....	67
Linux システムの監視.....	68
パフォーマンスの解析.....	70

第 7 章: 実際の例.....	71
トップダウン: オプティカル フローのアルゴリズム.....	71
ボトムアップ: ステレオ ビジョン アルゴリズム.....	80
付録 A: その他のリソースおよび法的通知.....	88
ザイリンクス リソース.....	88
Documentation Navigator およびデザイン ハブ.....	88
参考資料.....	88
トレーニング リソース.....	89
お読みください: 重要な法的通知.....	89

概要

Zynq[®]-7000 SoC プロセッサおよび Zynq[®] UltraScale+[™] MPSoC プロセッサは、Arm[®] ベースのプロセッサのソフトウェア プログラマビリティと FPGA のハードウェア プログラマビリティを統合し、1つのデバイス上に CPU、DSP、ASSP、および混合信号機能を統合して、解析およびハードウェア アクセラレーションを可能にします。SDSoC[™] 環境は、Zynq プラットフォームを使用してヘテロジニアス エンベデッド システムをインプリメントするための Eclipse ベースの統合設計環境 (IDE) です。

SDSoC 環境には、C/C++ プログラムを選択した関数がプログラマブル ロジックにコンパイルされた完全なハードウェア/ソフトウェア システムに変換するシステム コンパイラが含まれ、選択した関数のハードウェア アクセラレーションを可能にします。このガイドでは、ソフトウェア プログラム向けに、アクセラレーションに使用されるハードウェアについて、パフォーマンスを制限または向上する要素や、最高のシステム パフォーマンスを達成するための設計手法などを説明します。

SDSoC 環境を使用するのにハードウェアの詳細な知識は必要ありませんが、デバイス上で使用可能なハードウェア リソースについてと、ハードウェア関数で並列化を増加することにより高パフォーマンスを達成する方法を理解しておく、パフォーマンス要件を満たすためにコンパイラ最適化指示子を選択するのに役立ちます。

設計手法は、次のとおりです。

1. ハードウェアでアクセラレーションする関数を検出します。ソフトウェアのプロファイルをすると、計算負荷の高い領域を検出しやすくなります。
2. Vivado[®] HLS コーディング ガイドラインを使用してハードウェア関数コードを最適化して、パフォーマンス ターゲットを達成します。
3. CPU プロセッサ システム (PS) とハードウェア関数のプログラマブル ロジック (PL) 間のデータ転送を最適化します。この段階では、データ アクセスをバーストしやすく再構築し、データムーバーを選択します。

このガイドでは、ハードウェア アクセラレーションを達成する方法を説明した後、ユーザーが SDSoC ベースのアプリケーションで利用できる設計手法を使用した実際の例を示します。

SDSoC を使用したソフトウェア アクセラレーション

ザイリンクス デバイスのプログラマブル ロジック (PL) では、プロセッサ アーキテクチャよりも、アプリケーションの実行をより高い割合で並列処理可能です。アクセラレータのハードウェア関数用に `sds++/sdsc` (`sds++` と表記) システム コンパイラで生成されるカスタム プロセッシング アーキテクチャは、CPU とは実行の枠組みが異なるので、パフォーマンスを大幅に向上できます。既存のエンベデッド プロセッサ アプリケーションを PL でアクセラレーションするよう変更することはできますが、[ザイリンクス xfOpenCV ライブラリ](#)などの既存のハードウェア関数のソースコード ライブラリを使用してアプリケーションを記述したり、PL デバイスのアーキテクチャがうまく使用されるようにコードを変更したりすることで、パフォーマンスを大幅に向上して、消費電力を削減できます。

CPU のリソースは固定されており、タスクまたは演算を並列処理する機会は限られます。プロセッサは、そのタイプにかかわらず、プログラムをプロセッサのコンパイラ ツールで生成された命令順に実行します。コンパイラ ツールは、C/C++ で記述されたアルゴリズムをターゲット プロセッサにネイティブのアセンブリ言語の構文に変換します。2つの値の乗算のような単純な演算ですら、複数のアセンブリ命令となり、複数のクロック サイクルで実行されます。

FPGA は本質的に並列処理デバイス ファブリックであり、プロセッサ上で実行可能な関数をどれでもインプリメントできます。サイリンクス デバイスにはプログラムおよびコンフィギュレーション可能なリソースが大量に含まれており、カスタム アーキテクチャをインプリメントしてどんなレベルの並列処理でも達成できます。FPGA のプログラム ロジックは、すべての計算が同じ ALU を共有するプロセッサとは異なり、アクセラレーション関数を定義してインプリメント可能な空白のキャンバスのようなものです。FPGA コンパイラは、ALU 全体ではなくニューラル ネットの積和ハードウェアのみをインプリメントするなど、各アプリケーションまたはアルゴリズム用に最適化された独自の回路を作成します。

sds++ システム コンパイラを `-c` オプションを指定して実行すると、必要な関数定義に対して Vivado 高位合成 (HLS) ツールが起動されてファイルがハードウェア IP にコンパイルされます。HLS ツールを呼び出す前に、sds++ コンパイラで `#pragma SDS` が HLS ツールで解釈可能なプラグマに変換されます。HLS ツールは、同時処理を増やすように、スケジューリング、パイプライン、データデータフローを含むハードウェア指向の変換と最適化を実行します。

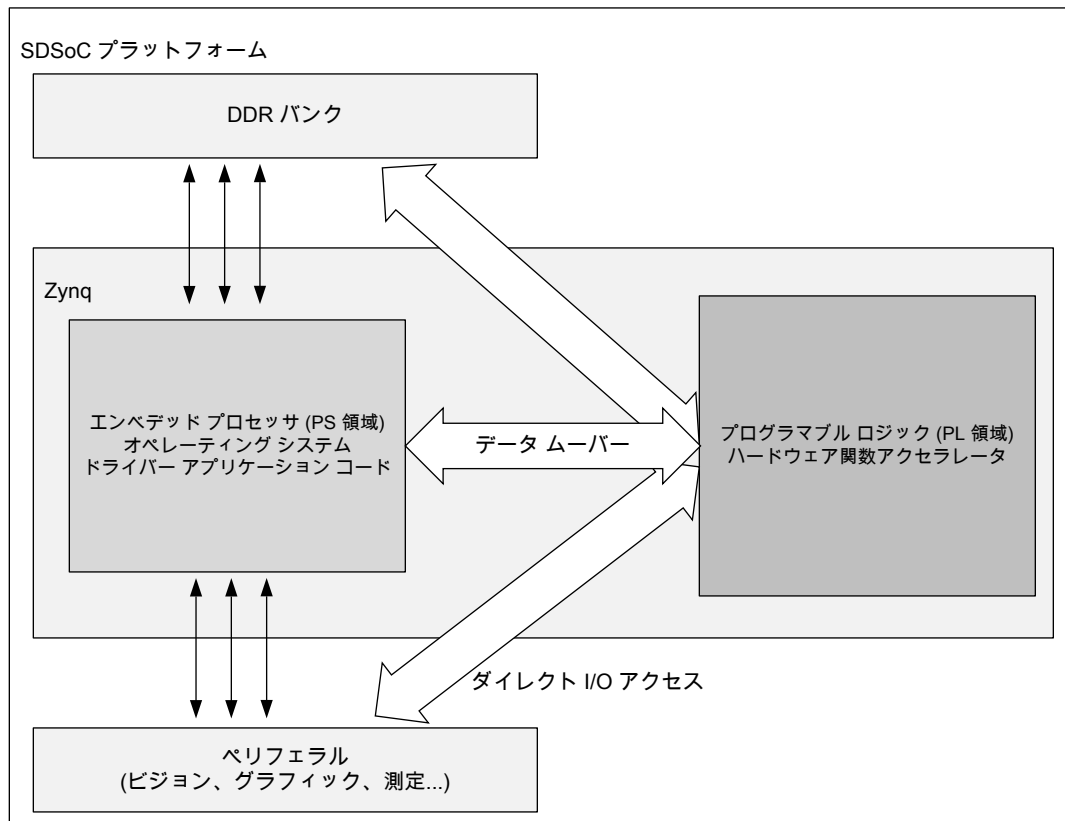
sds++ リンカーは、プログラム データフローを解析し、ハードウェア関数内への呼び出しおよびハードウェア関数間の呼び出しを含め、ハードウェアのデータ モーション ネットワークおよびソフトウェア制御コード (スタブと呼ぶ) をシステムにマップして、データ ムーバーを介したアクセラレータとデータ転送を調整します。次のセクションで説明するように、sds++ リンカーは、データ転送をスケジューリングして共有可能な演算を特定し、待機バリア API 呼び出しをスタブに挿入して、プログラム セマンティックが保持されるようにします。

SDSoC アプリケーションの実行モデル

SDSoC 環境アプリケーションの実行モデルは、プラットフォームのブート後にターゲット CPU で実行される C++ プログラムの通常実行を考えると理解できます。C++ バイナリの実行ファイルがハードウェアとどのようにインターフェイスするかを理解しておくことが有益です。

プログラム内で宣言されたハードウェア関数はハードウェア アクセラレータにコンパイルされ、標準 C ランタイムでこれらの関数への呼び出しによりアクセスされます。各ハードウェア関数呼び出しがタスクとしてアクセラレータを起動し、関数への各引数が CPU とアクセラレータ間で転送されて、アクセラレータ タスクの完了後にプログラムでアクセスできるようになります。メモリとアクセラレータ間のデータ転送には、DMA エンジンなどのデータ ムーバーが使用され、`zero_copy` などのユーザー データ ムーバー プラグマを考慮して、sds++ システム コンパイラにより自動的に挿入されます。

図 1: SDSoC システムのアーキテクチャ



X21358-082418

プログラムが正しく機能するようにするため、システムコンパイラはハードウェア関数への各呼び出しをインターセプトし、シグネチャが同じで派生した名前が付けられたスタブ関数への呼び出しに置き換えます。スタブ関数はすべてのデータ移動とアクセラレータの演算を制御し、ハードウェア関数呼び出しの終了時にソフトウェアとアクセラレータハードウェアを同期化します。スタブ内では、すべてのアクセラレータとデータムーバーは `sds_lib` ライブラリに含まれる送信および受信 API により制御されます。

ハードウェア関数呼び出し間のプログラムデータフローに、プログラム内で関数呼び出し (デストラクターまたは `free()` 呼び出し以外) が実行された後にアクセスされない配列引数が関係する場合、およびハードウェアアクセラレータをストリームを使用して接続できる場合、メモリへの往復がインプリメントされるのではなく、1つのハードウェアアクセラレータから直接ハードウェアストリーム接続を介して次のハードウェアアクセラレータにデータが転送されます。この最適化により、パフォーマンスが大幅に向上し、ハードウェアリソースを削減できます。

SDSoC プログラム実行モデルの手順は次のとおりです。

1. プログラムのコンストラクターが `main()` に入る前に `sds_lib` ライブラリが初期化されます。
2. プログラム内で、ハードウェア関数への各呼び出しが、元の関数と関数シグネチャが同じ (名前は異なる) のスタブ関数への関数呼び出しによりインターセプトされます。スタブ関数内では、次の手順が実行されます。
 - a. 同期アクセラレータタスク制御コマンドがハードウェアに送信されます。
 - b. ハードウェア関数への各引数に対して、非同期データ転送要求が関連付けられた `wait()` ハンドルと共に適切なデータムーバーに送信されます。void 以外の戻り値は、出力スカラー引数として扱われます。

- c. 各転送要求に対して、バリア `wait()` が発行されます。アクセラレータ間のデータ転送がダイレクトハードウェアストリームとしてインプリメントされる場合、この転送のバリア `wait()` は、この引数のアクセラレータ関数チェーンの最後のアクセラレータ関数に対するスタブ関数で発生します。
3. プログラムのデストラクターが `main()` を終了するときに、`sds_lib` ライブラリがクリーンアップされます。



ヒント: 手順 2a ~ 2c は、アクセラレータ パイプラインの開始および終了時にプログラムを正しく保ち、それと同時にパイプライン内での同時実行を可能にします。

場合によっては、システムコンパイラで自動的に推論できないアクセラレータタスクの同時実行の可能性を、プログラムが認識していることもあります。`sds++` システムコンパイラでは `#pragma SDS async(ID)` プラグマがサポートされており、このような場合にハードウェア関数への呼び出しの直前に挿入できます。このプラグマを指定すると、データ転送に対してバリア `wait()` 呼び出しなしでスタブ関数が生成されます。その結果、すべての転送要求を発行した後、制御がプログラムに戻り、アクセラレータの実行中にプログラムの同時実行が可能になります。この場合、プログラム内の適切な同期ポイントに `#pragma SDS wait(ID)` を挿入してください。このプラグマは `sds_wait(ID)` API 呼び出しとなり、ハードウェアアクセラレータ、その暗示的なデータムーバー、および CPU が正しく同期化されます。



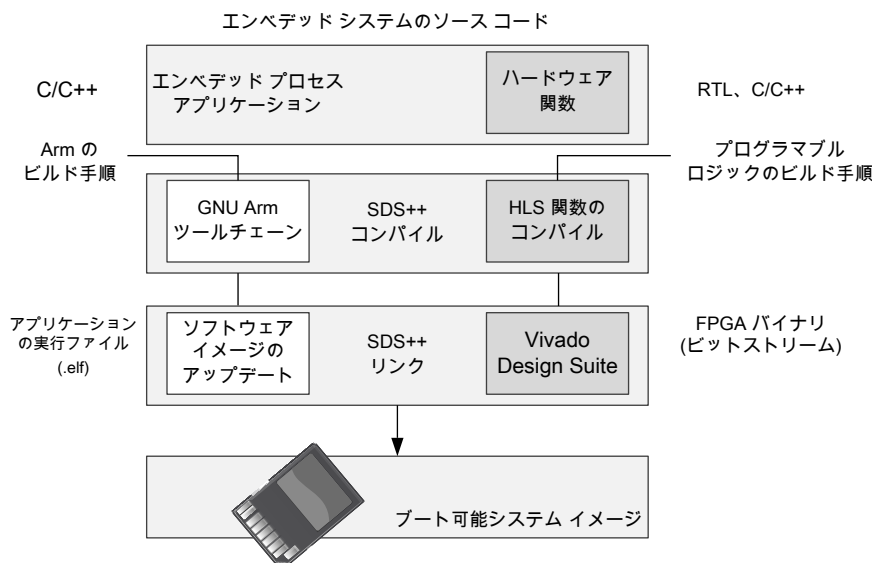
重要: 各 `async(ID)` プラグマには、対応する `wait(ID)` プラグマが必要です。

SDSoC のビルド プロセス

SDSoC のビルド プロセスでは、標準のコンパイルおよびリンク プロセスが使用されます。`g++` と同様に、`sds++` システムコンパイラはサブプロセスを呼び出してコンパイルおよびリンクを実行します。

次の図に示すように、コンパイル段階では、CPU で実行されるオブジェクトコードのコンパイルだけでなく、Vivado 高位合成 (HLS) ツールを使用したハードウェア関数の IP ブロックへのコンパイルおよびリンク、およびターゲット CPU ツールチェーンを使用した標準オブジェクトファイル (`.o`) の作成も実行されます。システムのリンク段階では、すべてのハードウェア関数における呼び出し元/呼び出し先の関係のプログラム解析、各ハードウェア関数呼び出しをインプリメントするアプリケーション特定のハードウェア/ソフトウェアネットワークの生成が実行されます。`sds++` システムコンパイラは、Vivado HLS (関数コンパイラ)、生成されたハードウェアシステムをインプリメントする Vivado Design Suite、CPU で実行されるアプリケーションバリアを作成する Arm コンパイラおよび `sds++` リンカーなど必要なすべてのツールを呼び出して、SD カード用の完全なブート可能イメージを出力することにより、各ハードウェア関数用のアクセラレータ (スタブ) を呼び出します。

図 2: SDSoC のビルド プロセス



X21126-041119

コンパイル プロセスには、次のタスクが含まれます。

1. コードが解析され、Arm コアの main アプリケーションがコンパイルされ、各ハードウェア アクセラレータに対して個別にコンパイルが実行されます。
2. 標準 GNU Arm コンパイル ツールでアプリケーションがコンパイルされ、最終出力としてオブジェクト ファイル (.o) が生成されます。
3. ハードウェアでアクセラレーションされる関数が HLS ツールで実行され、カスタム ハードウェア作成プロセスが開始し、出力としてオブジェクト ファイル (.o) が生成されます。

コンパイル後のリンク プロセスには、次のタスクが含まれます。

1. デザイン内のデータ移動が解析され、ハードウェア プラットフォームがアクセラレータを受け入れるよう変更されます。
2. Vivado Design Suite で合成およびインプリメンテーションが実行されてハードウェア アクセラレータがプログラマブル ロジック (PL) 領域にインプリメントされ、デバイス用のビットストリームが生成されます。
3. エンベデッド プロセッサ アプリケーションからハードウェア関数を呼び出すため、ソフトウェア イメージがハードウェア アクセス API でアップデートされます。
4. ボードを ELF (Executable and Linkable Format) ファイルのアプリケーションでブートする統合 SD カード イメージが生成されます。

パフォーマンスの見積もり

パフォーマンスを計測するためのコードのプロファイリングおよび計測用関数呼び出しの追加

コードのプロファイリングおよびインストルメント化の最初の主なタスクは、ハードウェアにインプリメントするのに適したアプリケーション コードの部分を特定して、ハードウェアで実行した場合に全体的なパフォーマンスが大幅に向上するようにすることです。コードの計算負荷の高い部分はハードウェア アクセラレーションに適しており、ハードウェアと CPU およびメモリ間でデータをストリーミングして計算と通信をオーバーラップさせることができる場合に特に適しています。ソフトウェア プロファイリングは、プログラムの CPU の負荷が高い部分を特定する標準的な方法です。アクセラレーションには向いていないのは、たとえば計算よりもアクセラレータのデータ転送に時間がかかる関数です。SDSoC™ 環境には、`gprof`、非介入 TCF (Target Communication Framework) プロファイラー、Eclipse のパフォーマンス解析パースペクティブなどのザイリンクス SDK ツールに含まれるパフォーマンスおよびプロファイリング機能がすべて含まれています。

スタンドアロンアプリケーションに対して TCF プロファイラーを実行するには、次の手順に従います。

1. [Project Explorer] ビューでプロジェクトを右クリックし、[Build Configurations] → [Set Active] → [Debug] をクリックして、アクティブなビルド コンフィギュレーションを [Debug] に設定します。
2. [Project Explorer] ビューでプロジェクト名を右クリックし、[Debug As] → [Launch on hardware (SDx Application Debugger)] をクリックしてデバッガーを起動します。

注記: ボードをコンピューターに接続して電源をオンにしておく必要があります。アプリケーションが `main()` に入る地点でブレークします。

3. [Window] → [Show View] → [Other] をクリックして TCF プロファイラーを起動します。表示されたウィンドウで [Debug] を展開し、[TCF profiler] を選択します。
4. [TCF Profiler] タブの上部にある緑色の [Start] ボタンをクリックして TCF プロファイラーを開始します。
5. [Profiler Configuration] ダイアログ ボックスで [Aggregate per function] をイネーブルにします。
6. プロファイリングを開始するには、[Resume] ボタンをクリックするか F8 キーを押します。プログラムが実行を完了し、`exit()` 関数でブレークします。
7. [TCF Profiler] タブで結果を確認します。

プロファイリングは、CPU プログラム カウンターのサンプリングおよび実行中のプログラムへの相関に基づいてコードのよく使用される領域を検索するための統計的な手法です。プログラムのパフォーマンスを計測するもう 1 つの方法として、実行中のプログラムの異なる部分に実際にかかる時間を判断するため、アプリケーションに計測用の関数呼び出しを追加する方法があります。

[TCF Profiler] タブには、Standalone または Linux OS アプリケーションのいずれかに関連する詳細な情報が含まれます。前の手順に示すように、プロファイラーを使用するために追加のコンパイル オプションは必要ありません。

注記: このタイプのハードウェアのプロファイルには、JTAG 接続が必要です。

SDSoC 環境の `sds_lib` ライブラリには、次の例に示すように、アプリケーションパフォーマンスの計測に使用可能な単純なソースコードアノテーションベースの、タイムスタンプAPIが含まれます。

```
/*
 * @return value of free-running 64-bit Zynq(TM) global counter
 */
unsigned long long sds_clock_counter(void);
```

このAPIを使用してタイムスタンプを収集してそれらの差を調べることで、プログラムの主要な部分の時間を判断できます。たとえば、次のコード例に示すように、データ転送やハードウェア関数の全体的な実行時間を計測できます。

```
class perf_counter
{
public:
    uint64_t tot, cnt, calls;
    perf_counter() : tot(0), cnt(0), calls(0) {};
    inline void reset() { tot = cnt = calls = 0; }
    inline void start() { cnt = sds_clock_counter(); calls++; }
    inline void stop() { tot += (sds_clock_counter() - cnt); }
    inline uint64_t avg_cpu_cycles() { return (tot / calls); }
};

extern void f();
void measure_f_runtime()
{
    perf_counter f_ctr;
    f_ctr.start();
    f();
    f_ctr.stop();
    std::cout << "Cpu cycles f(): " << f_ctr.avg_cpu_cycles()
                << std::endl;
}
```

SDSoC 環境内のパフォーマンス見積もり機能はこのAPIを使用し、ハードウェアインプリメンテーションに選択された関数をインストルメント化し、ターゲット上でアプリケーションを実行して実際の実行時間を計測して、ハードウェア関数の見積もりの実行時間と比較します。

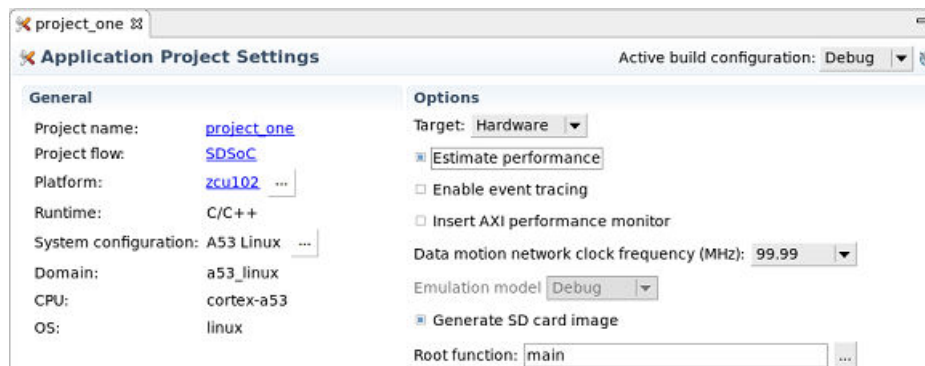
注記: アプリケーションの分割にはCPUの負荷が高い関数をプログラマブルロジックに移行するのが経験的に最も信頼性の高い方法ですが、システムパフォーマンスが向上するとはかぎらないので、アルゴリズムを変更してメモリアクセスを最適化することが必要な場合があります。CPUの外部メモリへのランダムアクセス速度は、マルチレベルキャッシュおよび高速クロック(通常プログラマブルロジックよりも2~8倍高速)により、プログラマブルロジックで達成できる速度よりもかなり速くなります。大型のインデックスセットのインデックスを並べ替えるソートルーチンなどの広いアドレス範囲に対するポインター変数の処理はCPUには適していますが、関数をプログラマブルロジックに移動するとマイナスになることがあります。これはそのような計算関数がハードウェアに移動する良い候補ではないということではなく、コードまたはアルゴリズムを再構成することが必要な場合があるということです。これは、DSPおよびGPUコプロセッサの既知の問題です。

SDSCC/SDS++ パフォーマンス見積もりフロー オプション

完全なビットストリームのコンパイルには、ソフトウェアのコンパイルよりもかなり時間がかかります。そのため、`sds++/sdsc`(`sds++`と呼ぶ)には、ハードウェア関数呼び出しの実行時間の改善を見積もるパフォーマンス見積もりオプションが含まれています。

[Application Project Settings] で [Estimate Performance] チェック ボックスをオンにすると、現在のビルド コンフィギュレーションでパフォーマンス見積もりがイネーブルになり、プロジェクトがビルドされます。

図 3: [Application Project Settings] でのパフォーマンス見積もりの設定



スピードアップの見積もりは、2 段階のプロセスです。

1. SDSoC 環境でハードウェア関数をコンパイルしてシステムを生成します。システムをビットストリームに合成する代わりに、sds++ でハードウェア関数のレイテンシ見積もりとハードウェア関数呼び出し元のデータ転送見積もり時間に基づいて、パフォーマンスが見積もられます。
2. パフォーマンス ベースラインおよびパフォーマンス見積もりを確認するには、生成されたパフォーマンス レポートで [Click Here] をクリックしてインストール化されたバージョンのソフトウェアをターゲットで実行します。

パフォーマンス レポートの使用方法に関するチュートリアルは、『SDSoC 環境チュートリアル: 概要』 (UG1028) を参照してください。

コマンド ラインからパフォーマンス見積もりを生成することもできます。ソフトウェア ランタイムに関するデータを収集するため、まず `-perf-funcs` オプションを使用してプロファイルする関数を指定し、`-perf-root` オプションを使用してプロファイルされる関数への呼び出しを含むルート関数を指定します。

これにより、sds++ システム コンパイラでこれらの関数がインストール化され、ボードでアプリケーションを実行したときに自動的にランタイム データが収集されます。インストール化されたアプリケーションをターゲットで実行すると、プログラムにより SD カードに `swdata.xml` という実行のランタイム パフォーマンス データを含むファイルが作成されます。

`swdata.xml` をホストにコピーし、ハードウェア関数呼び出しごとおよび `-perf-root` で指定した最上位関数でのパフォーマンス向上を見積もるビルドを実行します。`-perf-est` オプションを使用して、`swdata.xml` をこのビルドの入力データとして指定します。

次の表に、アプリケーションをビルドするのに通常使用される sds++ システム コンパイラ オプションを示します。

表 1: よく使用される sds++ オプション

オプション	説明
<code>-perf-funcs function_name_list</code>	インストール化されたソフトウェア アプリケーションでプロファイリングするすべての関数をカンマで区切って指定します。
<code>-perf-root function_name</code>	プロファイリングする関数へのすべての呼び出しを含むルート関数を指定します。デフォルトは main 関数です。

表 1: よく使用される sds++ オプション (続き)

オプション	説明
<code>-perf-est data_file</code>	インストール化されたソフトウェア アプリケーションをターゲット上で実行したときに生成されたランタイム データを含むファイルを指定します。ハードウェア アクセラレーションされた関数のパフォーマンス向上を見積もります。このファイルのデフォルト名は、 <code>swdata.xml</code> です。
<code>-perf-est-hw-only</code>	ソフトウェア実行データを収集せずに見積もりフローを実行します。このオプションを使用すると、ベースラインとの比較は含まれませんが、ハードウェア レイテンシおよびリソース使用率を確認できます。



注意: プロファイル データを収集するためにボード上の `sd_card` イメージを実行したら、`cd /; sync; umount /mnt;` と入力します。これにより、`swdata.xml` ファイルが SD カードに書き込まれます。

システム パフォーマンスの向上

この章では、プログラマが全体的なシステム パフォーマンスを向上しやすくするために、SDSoC™ システム コンパイラの基本的な原則と推論規則について説明します。

- ハードウェア関数での並列処理の増加。
- システムでの並列処理および同時処理の増加。
- プログラマブル ロジックから外部メモリへのアクセスを向上。
- データ モーション ネットワークの理解 (デフォルト動作とユーザー仕様)。

全体的なシステム パフォーマンスに影響する要素は多数あります。適切に設計されたシステムでは、すべてのハードウェア コンポーネントが有益な処理を実行するように、計算と通信のバランスが取られます。

- 計算負荷の高いアプリケーションでは、ハードウェア アクセラレータのスループットを最大にし、レイテンシを最低限に抑えるようにしてください。
- メモリ バウンド アプリケーションでは、ハードウェアの一時的なおよび空間的な局所性を増加するようアルゴリズムを再構築することが必要な場合があります。たとえば、外部メモリへのランダム配列アクセスではなく、copy-loop や memcpy を追加してデータ ブロックをハードウェアに戻すなどです。

コード内にプラグマを使用すると、さまざまな最適化の機能を制御できます。使用可能なすべてのプラグマの詳細は、『SDx プラグマ リファレンス ガイド』 ([UG1253](#)) を参照してください。

関連情報

[ハードウェア関数の並列機能の改善](#)

[システムでの並列処理および同時処理の増加](#)

[メモリ アクセスの最適化](#)

[SDSoC でのデータ モーション ネットワークの生成](#)

ハードウェア関数の並列機能の改善

このセクションでは、プログラマブル ロジックにクロスコンパイル可能な効率的なコードを記述するための概要を示します。

SDSoC 環境では、Vivado® HLS (高位合成) ツールをプログラマブル ロジックのクロス コンパイラとして使用して、C/C++ 関数をハードウェアに変換します。

このセクションで説明される原則に従うと、合成済み関数のパフォーマンスを大幅に改善でき、アプリケーションの全体的なシステム パフォーマンスを大幅に向上できる可能性があります。

最上位ハードウェア関数のガイドライン

このセクションでは、Vivado HLS ツール ハードウェア関数で Arm® コア GNU ツールチェーンで生成されたオブジェクトコードと一貫したインターフェイスが使用されるようにするためのコーディング ガイドラインを示します。

最上位ハードウェア関数引数には標準 C99 データ型を使用

1. `bool` の配列は使用しないでください。 `bool` の配列のメモリ レイアウトは、GNU Arm クロス コンパイラと HLS ツールで異なります。
2. ハードウェア関数の最上位インターフェイスには `hls::stream` を使用しないでください。このデータ型は、HLS コンパイラがハードウェア関数内に効率的なロジックを合成するのに役立ちますが、アプリケーションソフトウェアには役立たないからです。

最上位ハードウェア関数の引数の HLS インターフェイス指示子を使用しない

HLS インターフェイス プラグマを含む最上位ハードウェア関数はサポートされていますが、通常は使用しないでください。 `sdcc/sds++` (`sds++` と呼ぶ) システム コンパイラでは、自動的に適切な HLS インターフェイス指示子が生成されます。

最上位ハードウェア関数に次の 2 つの SDSoC 環境プラグマを指定すると、 `sds++` システム コンパイラで必要な HLS インターフェイス指示子が生成されるようになります。

- `#pragma SDS data zero_copy()`: ハードウェアに AXI マスター インターフェイスとしてインプリメントされる共有メモリ インターフェイスを生成します。
- `#pragma SDS data access_pattern(argument:SEQUENTIAL)`: ハードウェアに FIFO インターフェイスとしてインプリメントされるストリーミングインターフェイスを生成します。

最上位関数の引数に対して `#pragma HLS interface` を使用してインターフェイスを指定すると、その引数に対する HLS インターフェイス指示子は SDSoC 環境では生成されないため、生成されたハードウェアインターフェイスがその他すべての関数引数のハードウェアインターフェイスと一貫したものになるようにしてください。



推奨: 互換性のない HLS インターフェイス タイプを使用した関数があると、意味不明な `sds++` システム コンパイラのエラーメッセージが表示されるので、必須ではありませんが、HLS インターフェイス プラグマを削除することをお勧めします。

Vivado Design Suite HLS ライブラリの使用

このセクションでは、SDSoC 環境で Vivado HLS ツールライブラリを使用する方法について説明します。

HLS ライブラリは、SDSoC 環境の HLS ツール インストールにソースコードとして含まれており、HLS ツールを使用してプログラマブル ロジック向けにクロス コンパイルする予定のほかのソースコード同様に使用できます。ソースコードは、『SDSoC 環境プログラマガイド』 (UG1278) の **ハードウェア関数の引数型** に説明されている規則に従っている必要があります。このとき、関数でソフトウェアインターフェイスがアプリケーションにエクスポートされるようにするため、C/C++ ラッパー関数を作成することが必要な場合があります。

SDSoC IDE では、すべての基本的なプラットフォームの有限インパルス応答 (FIR) サンプル テンプレートに、HLS ライブラリを使用する例が含まれています。 `samples/hls_lib` ディレクトリには、HLS math ライブラリを使用するコード例が数個含まれています。たとえば、 `samples/hls_lib/hls_math` には平方根関数をインプリメントして使用する例が含まれています。

my_sqrt.h ファイルには次が含まれています。

```
#ifndef _MY_SQRT_H
#define _MY_SQRT_H_

#ifdef __SDSVHLS__
#include "hls_math.h"
#else
// The hls_math.h file includes hdl_fpo.h which contains actual code and
// will cause linker error in the ARM compiler, hence we add the function
// prototypes here
static float sqrtf(float x);
#endif

void my_sqrt(float x, float *ret);

#endif // _SQRT_H_
```

my_sqrt.cpp ファイルには次が含まれています。

```
#include "my_sqrt.h"

void my_sqrt(float x, float *ret)
{
    *ret = sqrtf(x);
}
```

makefile には、これらのファイルをコンパイルするコマンドが含まれています。

```
sds++ -c -hw my_sqrt -sds-pf zc702 my_sqrt.cpp
sds++ -c my_sqrt_test.cpp
sds++ my_sqrt.o my_sqrt_test.o -o my_sqrt_test.elf
```

システムでの並列処理および同時処理の増加

同時処理のレベルを増加することは、システムの全体的なパフォーマンスを向上するための標準的な方法であり、並列処理のレベルを増加することは同時処理を増加させる標準的な方法です。プログラマブル ロジックは、同時実行されるアプリケーション特定のアクセラレータを含むアーキテクチャをインプリメントするのに適しており、特にデータ プロデューサーとコンシューマー間で同期化されるフロー制御ストリームを介した通信に適しています。

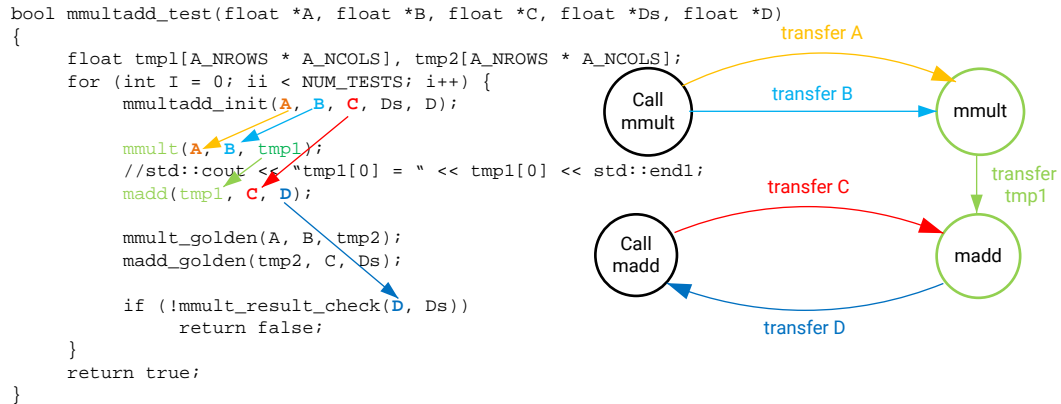
SDSoC 環境では、関数およびデータ ムーバー レベルでのマクロ アーキテクチャの並列処理、ハードウェア アクセラレータ内でのマクロ アーキテクチャの並列処理を制御できます。sds++/sdsc (sds++ と呼ぶ) でシステム接続とデータ ムーバーがどのように推論されるかを理解することにより、必要に応じてアプリケーション コードを構成してプラグマを適用して、アクセラレータとソフトウェア間のハードウェア接続、データ ムーバーの選択、ハードウェア関数のアクセラレータ インスタンス数、タスク レベルのソフトウェア制御を制御できます。

Vivado HLS ツールまたは C 呼び出し可能/リンク可能ライブラリとして組み込む IP 内で、マイクロ アーキテクチャの並列処理、同時処理、およびハードウェア関数のスループットを制御できます。

システム レベルでは、ハードウェア関数間のデータフローでプログラマブル ロジックとシステム メモリの間の引数転送が不要な場合は、sds++ コンパイラによりハードウェア関数がチェーン接続されます。

たとえば、mmult および madd 関数がハードウェアに選択されている次の図に示すコードがあるとします。

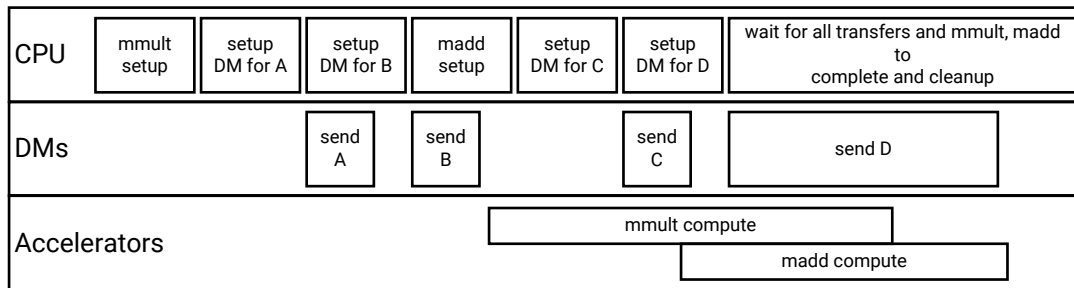
図 4: 直接接続を使用したハードウェア/ソフトウェアの接続



X14763-042519

2つのハードウェア関数間でデータを渡すには中間配列変数 `tmp1` のみを使用されるので、`sds++` コンパイラにより2つの関数が直接接続を使用してチェーン接続されます。

ハードウェアへの呼び出しのタイムラインを次の図に示すように考慮すると有益です。

図 5: `mmult/madd` 関数呼び出しのタイムライン


X14764-121417

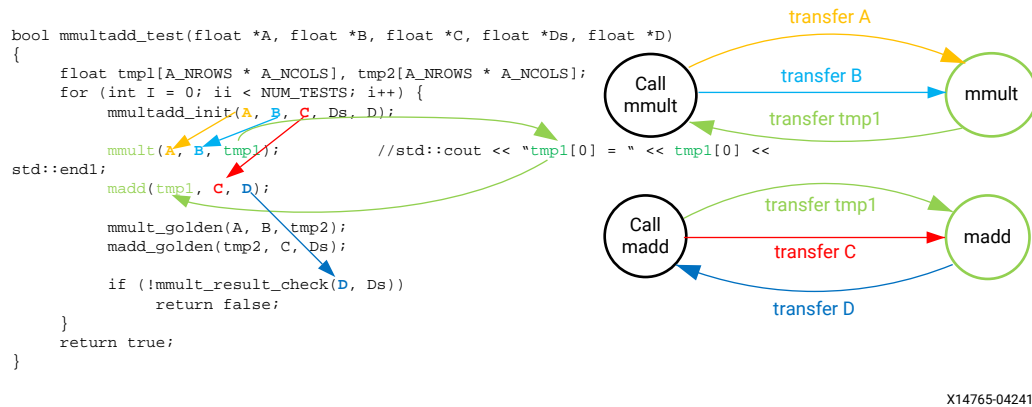
プログラムでは元のプログラムセマンティクスが保持されますが、標準 Arm コアのプロシージャ呼び出しシーケンスではなく、各ハードウェア関数呼び出しがデータムーバー (DM) とアクセラレータの両方に対してセットアップ、実行、およびクリーンアップを含む複数のフェーズに分割されます。CPU は各ハードウェア関数 (基になる IP 制御インターフェイス) と関数呼び出しのデータ転送をノンブロッキング API でセットアップし、すべての呼び出しと転送が完了するのを待ちます。

図に示す例では、`mmult` と `madd` 関数の入力を使用可能になると、これらの関数が同時に実行されます。プログラム、データムーバー、アクセラレータ構造に基づいて `sds++` システムコンパイラで自動的に生成された制御コードにより、コンパイルされたプログラムですべての関数呼び出しが調整されます。

通常、`sds++` システムコンパイラでアプリケーションコード内の関数呼び出しの悪影響を判断することはできないので (たとえば `sds++` でリンクされたライブラリ内の関数のソースコードにアクセスできないなど)、ハードウェア関数呼び出し間で変数の中間アクセスが発生する場合は、データをメモリに戻す必要があります。

たとえば、次の図に示すデバッグプリント文のコメントを不注意にせずしてしまつと、大幅に異なるデータ転送グラフとなり、その結果システムおよびアプリケーションのパフォーマンスがまったく異なるものになる可能性があります。

図 6: 直接接続が切断されたハードウェア/ソフトウェアの接続



プログラムでは、複数の呼び出しサイトから1つのハードウェア関数を呼び出すことができます。この場合、sds++システムコンパイラは次のように動作します。関数呼び出しのどれかが直接接続データフローとなった場合、sds++システムコンパイラにより同様の直接接続をサービスするハードウェア関数のインスタンスと、メモリ(ソフトウェア)とPL間の残りの呼び出しをサービスするハードウェア関数のインスタンスが作成されます。

PLで高パフォーマンスを達成するには、ハードウェア関数間を直接接続データフローを使用してアプリケーションコードを構成するのが最適な方法です。データストリームで接続されたアクセラレータの多段パイプラインを作成することにより、同時実行の可能性が高くなります。

sds++システムコンパイラを使用して並列処理と同時処理を増加させるには、もう1つ方法があります。ハードウェア関数を呼び出す直前に次のプラグマを挿入して、ハードウェア関数の複数のインスタンスが作成されるようになります。

```
#pragma SDS resource(<id>) // <id> a non-negative integer
```

このプラグマは、<id>で参照されているハードウェアインスタンスを作成します。

次に、ハードウェア関数mmultの2つのインスタンスを作成するコード例を示します。

```
{
#pragma SDS resource(1)
  mmult(A, B, C); // instance 1
#pragma SDS resource(2)
  mmult(D, E, F); // instance 2
}
```

アクセラレータのインスタンスを複数作成しない場合は、sds_asyncメカニズムにより、ハードウェアスレッドを明示的に処理して非常に高レベルの並列処理および同時処理を達成することもできますが、明示的なマルチスレッドプログラミングモデルでは同期に細心の注意を払い、非決定の動作やデッドロックを回避する必要があります。詳細は、『SDSoC環境プログラミングガイド』(UG1278)を参照してください。

SDSoCでのデータ モーション ネットワークの生成

このセクションでは、次について説明します。

- SDSoC 環境のデータ モーション ネットワークに使用されるコンポーネントのほか、SDSoC で生成されるデータ モーション ネットワークについて。
- 適切な SDSoC プラグマを使用してデータ モーション ネットワークを生成するためのガイドライン。

ソフトウェア プログラムとハードウェア関数の間の各転送には、データ ムーバーが必要です。データ ムーバーは、データを移動するハードウェア コンポーネントとオペレーティングシステム特定のライブラリ関数で構成されます。次の表に、サポートされるデータ ムーバーとそれぞれの特性を示します。

表 2: SDSoC でサポートされるデータ ムーバー

SDSoC データ ムーバー	Vivado® IP データ ムーバー	アクセラレータ IP のポート タイプ	転送サイズ	連続するメモリのみ
axi_dma_simple	axi_dma	bram、ap_fifo、axis	≤ 32 MB	○
axi_dma_sg	axi_dma	bram、ap_fifo、axis	なし	X (推奨されない)
axi_fifo	axi_fifo_mm_s	bram、ap_fifo、axis	≤ 300 B	X
zero_copy	accelerator IP	aximm master	なし	○

- 配列引数の場合は、転送サイズ、ハードウェア関数のポート マップ、および関数呼び出しサイト情報に基づいてデータ ムーバーが推論されます。データ ムーバーには、それぞれ次のようにパフォーマンスとリソースのトレードオフがあります。
 - `axi_dma_simple` データ ムーバーは、最も効率的なバルク転送エンジンですが、32 MB の転送までしかサポートされていないので、これより大きい転送には向いていません。
 - `axi_fifo` データ ムーバーには、DMA ほど多くのハードウェア リソースは必要ありませんが、転送レートが遅いので、最大 300 バイトのペイロードまでに使用することをお勧めします。
 - `axi_dma_sg` (スキッター ギャザー DMA) データ ムーバーは、DMA パフォーマンスが遅く、ハードウェア リソースの消費が多くなりますが、制限が少なく、プラグマ指示子がない場合に最適なデフォルト データ ムーバーとなることがよくあります。

プログラム ソースの関数宣言の直前に次のようなプラグマを挿入すると、別のデータ ムーバーを選択できます。

```
#pragma SDS data data_mover (A:AXI_DMA_SIMPLE)
```

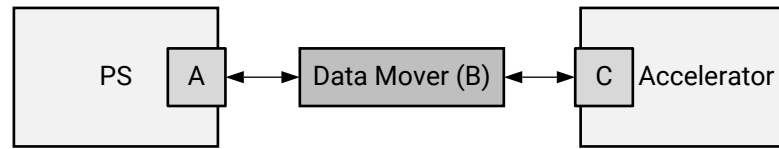
注記: `#pragma SDS` は常にヒントではなく規則として処理されるので、前の表のデータ ムーバーの要件に必ず従うようにしてください。

SDSoC 環境のデータ モーション ネットワークには、次の 3 つのコンポーネントが含まれます。

- PS のメモリ システム ポート (A)
- PS とアクセラレータ間およびアクセラレータ間のデータ ムーバー (B)
- アクセラレータのハードウェア インターフェイス (C)

次の図は、これらの 3 つのコンポーネントを示しています。

図 7: データ モーション ネットワーク コンポーネント



X22627-051019

SDS プラグマがない場合、SDSoC 環境ではソース コードの解析に基づいてデータ モーション ネットワークが自動的に生成されますが、SDSoC 環境にはデータ モーション ネットワーク生成をガイドするためのプラグマも提供されています。詳細は、『SDx プラグマ リファレンス ガイド』 ([UG1253](#)) を参照してください。

システム ポート

システム ポートは、データ ムーバーを PS に接続します。システム ポートとしては、Zynq®-7000 SoC または Zynq® UltraScale+™ MPSoC プロセッサのアクセプタンスフィルタ ID (AFI - ハイ パフォーマンス ポートに対応)、メモリー インターフェイス ジェネレーター (MIG - PL ベースの DDR メモリ コントローラー)、またはストリーム ポートを使用できます。

AFI ポートは、キャッシュ コヒーレンシ ポートではありません。キャッシュ フラッシュおよびキャッシュ無効化などのキャッシュ コヒーレンシは、必要に応じてソフトウェアにより維持されます。

AFI ポートは、転送データ、データのキャッシュ属性、およびデータ サイズ要件によって異なります。データが `sds_alloc_non_cacheable()` または `sds_register_dmabuf()` で割り当てられる場合は、キャッシュのフラッシュ/無効化を避けるために AFI ポートに接続することをお勧めします。

注記: これらの関数は、`sds_lib.h` に含まれており、『SDSoC 環境 プログラム ガイド』 ([UG1278](#)) の環境 API の部分で説明されています。

SDSoC システム コンパイラでは、アクセラレータとのデータ転送のためにこれらのメモリ属性が解析され、データ ムーバーが適切なシステム ポートに接続されるようになっています。

コンパイラの指定を上書きする場合や、コンパイラでこのような解析が実行できない場合は、次のプラグマを使用するとシステム ポートを指定できます。

```
#pragma SDS data sys_port(arg:ip_port)
```

たとえば、次の関数は直接 FIFO AXI インターフェイスに接続します。ip_port は AFI または MIG のいずれかにできます。

```
#pragma SDS data sys_port:(A:fifo_S_AXI) *
void foo(int* A, int* B, int* C);
```

この関数は、ストリーミング インターフェイスにも使用できます。

```
#pragma SDS data sys_port:(A:stream_fifo_S_AXIS) *
void foo(int* A, int* B, int* C)
```

注記: Vivado® 高位合成 (HLS) ツールでの AXI の機能については、『Vivado Design Suite ユーザー ガイド: 高位合成』 ([UG902](#)) を参照してください。

次の `sds++` システム コンパイラ コマンドを使用すると、プラットフォームのシステム ポートがリストされます。

```
sds++ -sds-pf-info <platform> -verbose
```

データムーバー

データムーバーは、PSとアクセラレータ間、およびアクセラレータどうしの間でデータを転送します。SDSoC環境では、転送されるデータの特性とサイズに基づいてさまざまなタイプのデータムーバーを生成できます。

- スカラー: スカラーデータは、常に `AXI_LITE` データムーバーで転送されます。
- 配列: `sds++` システムコンパイラでは、次のデータムーバーを生成します。
 - `AXI_DMA_SG`
 - `AXI_DMA_SIMPLE`
 - `AXI_FIFO`
 - `zero_copy` (アクセラレータマスターマスターのAXI4バス)
 - `AXI_LITE` (メモリ属性と配列のデータサイズによって異なる)

たとえば、配列が `malloc()` を使用して割り当てられている場合、メモリが物理的に隣接していない場合、SDSoC環境では通常スキャッターギャザーDMA (`AXI_DMA_SG`) が生成されますが、データサイズが300バイト未満の場合は `AXI_FIFO` が代わりに生成されます。これは、この方がデータ転送時間が `AXI_DMA_SG` よりも短く、PLリソースの使用量も少ないからです。

- 構造体またはクラス: `struct` のインプリメンテーションは、構造体 (`struct`) がハードウェアにどのように渡されるか (値渡し、参照渡し、または `structs` の配列として渡されるか) によって異なります。次の表に、さまざまなインプリメンテーションを示します。

表 3: 構造体のインプリメンテーション

構造体の渡し方	デフォルト (プラグマなし)	#pragma SDS data zero_copy (arg)	#pragma SDS data zero_copy (arg[0:SIZE])	#pragma SDS data copy (arg)	#pragma SDS data copy (arg[0:SIZE])
値渡し (<code>struct RGB arg</code>)	各フィールドがフラットになり、それぞれがスカラーまたは配列として渡されます。	これはサポートされておらず、エラーになります。	これはサポートされておらず、エラーになります。	<code>struct</code> が1つの幅のスカラーにパックされます。	各フィールドがフラットになり、それぞれがスカラーまたは配列として渡されます。 <code>SIZE</code> の値が無視されます。
ポインター渡し (<code>struct RGB *arg</code>) または参照渡し (<code>struct RGB &arg</code>)	各フィールドがフラットになり、それぞれがスカラーまたは配列として渡されます。	<code>struct</code> が1つの幅のスカラーにパックされ、1つの値として転送されます。データはAXI4バスを介してハードウェアアクセラレータに転送されます。	<code>struct</code> が1つの幅のスカラーにパックされます。AXI4バスを介してハードウェアアクセラレータに転送されるデータ値の数は、 <code>SIZE</code> の値で定義されます。	<code>struct</code> が1つの幅のスカラーにパックされます。	<code>struct</code> が1つの幅のスカラーにパックされます。 <code>AXI_DMA_SG</code> または <code>AXI_DMA_SIMPLE</code> を使用してハードウェアアクセラレータに転送されるデータ値の数は、 <code>SIZE</code> の値で定義されます。

表 3: 構造体のインプリメンテーション (続き)

構造体の渡し方	デフォルト (プラグマなし)	#pragma SDS data zero_copy (arg)	#pragma SDS data zero_copy (arg[0:SIZE])	#pragma SDS data copy (arg)	#pragma SDS data copy (arg[0:SIZE])
struct の配列 (struct RGB arg[1024])	配列の各 struct 要素が1つの幅のスカラーにパックされます。	配列の各 struct 要素が1つの幅のスカラーにパックされます。 データは AXI4 バスを使用してハードウェアアクセラレータに転送されます。	配列の各 struct 要素が1つの幅のスカラーにパックされます。 データは AXI4 バスを使用してハードウェアアクセラレータに転送されます。 SIZE の値が配列サイズよりも優先され、アクセラレータに転送されるデータ値の数が決まります。	配列の各 struct 要素が1つの幅のスカラーにパックされます。 データは AXI_DMA_SG または AXI_DMA_SIMPLE などのデータムーバーを使用してハードウェアアクセラレータに転送されます。	配列の各 struct 要素が1つの幅のスカラーにパックされます。 データは AXI_DMA_SG または AXI_DMA_SIMPLE などのデータムーバーを使用してハードウェアアクセラレータに転送されます。 SIZE の値が配列サイズよりも優先され、アクセラレータに転送されるデータ値の数が決まります。

配列を転送するのにどのデータムーバーを使用するかは、配列の2つの属性(データサイズと物理メモリの連続性)によって異なります。たとえば、メモリサイズが1MBで物理的に隣接していない(malloc() で割り当てられる)場合、AXI_DMA_SGを使用する必要があります。次の表に、これらのデータムーバーの適用基準を示します。

表 4: データムーバーの選択

データムーバー	物理メモリの連続性	データサイズ(バイト)
AXI_DMA_SG	連続または非連続	> 300
AXI_DMA_Simple	連続	< 32M
AXI_FIFO	非連続	< 300

通常、SDSoC クロス コンパイラではこれら2つの属性用にハードウェアアクセラレータへ転送される配列が解析され、適切なデータムーバーが選択されますが、このような解析ができないこともあります。この場合、SDSoC クロス コンパイラで SDS プラグマを使用してメモリ属性が指定できないことを示す警告メッセージが表示されます。

```
WARNING: [DMAAnalysis 83-4492] Unable to determine the memory attributes
passed to rgb_data_in of function img_process at
C:/simple_sobel/src/main_app.c:84
```

メモリ属性を指定するプラグマは、次のようになります。

```
#pragma SDS data mem_attribute(function_argument:contiguity)
```

contiguity は PHYSICAL_CONTIGUOUS または NON_PHYSICAL_CONTIGUOUS のいずれかにできます。次のプラグマを使用して、転送するデータのサイズを指定します。

```
#pragma SDS data copy(function_argument[offset:size])
```

size は、数値または任意の演算式にできます。

zero_copy データ ムーバー

zero_copy はアクセラレータ インターフェイスとデータ ムーバーの両方に使用できる固有のデータ ムーバーです。このプラグマの構文は、次のとおりです。

```
#pragma SDS data zero_copy(arg[offset:size])
```

[offset:size] はオプションで、配列のデータ転送サイズをコンパイル時に決定できない場合にのみ必要です。

デフォルトでは、SDSoC 環境は配列引数の copy を実行します。つまり、データはデータ ムーバーを介して PS からアクセラレータに明示的にコピーされますが、この ZERO_COPY プラグマを使用すると、SDSoC でアクセラレータの指定した引数に対して AXI-Master インターフェイスが生成され、アクセラレータ コードで指定したとおりに PS からデータが取得されます。

ZERO_COPY プラグマを使用するには、配列に該当するメモリは物理的に連続している (sds_alloc で割り当てる) 必要があります。

アクセラレータのインターフェイス

生成されるアクセラレータのインターフェイスは、引数のデータ型によって異なります。

- スカラー: スカラー引数の場合、アクセラレータの入力および出力を通すためにレジスタ インターフェイスが生成されます。
- 配列: 配列を転送するためのアクセラレータのハードウェア インターフェイスは、そのアクセラレータが配列内のデータにどのようにアクセスするかによって、RAM インターフェイスかストリーミング インターフェイスのいずれかにできます。

RAM インターフェイスでは、アクセラレータ内でデータにランダムにアクセスできますが、アクセラレータ内でメモリ アクセスを実行する前に、配列全体をアクセラレータに転送する必要があります。さらに、このインターフェイスを使用すると、配列を格納するため、アクセラレータ側にブロック RAM リソースが必要となります。

ストリーミング インターフェイスでは、配列全体を格納するためのメモリは必要ないので、配列要素の処理をパイプライン処理できます。たとえば、アクセラレータで前の配列要素を処理中に次の配列要素の処理を開始できます。ただし、ストリーミング インターフェイスを使用する場合、アクセラレータが厳密な順序で配列にアクセスする必要があり、転送されるデータ量はアクセラレータで予測されるデータ量と同じである必要があります。

SDSoC 環境では、デフォルトで配列に対して RAM インターフェイスが生成されるようになっています。ただし、SDSoC 環境ではストリーミング インターフェイスを生成するよう指定するプラグマが含まれています。

- 構造体またはクラス: struct のインプリメンテーションは、struct がハードウェアにどのように渡されるか (値渡し、参照渡し、または structs の配列として渡されるか) によって変わります。前の表は、さまざまなインプリメンテーションを示しています。

次の SDS プラグマは、アクセラレータのインターフェイス生成をガイドするために使用できます。

```
#pragma SDS data access_pattern(function_argument:pattern)
```

pattern は、RANDOM または SEQUENTIAL のいずれか、arg はアクセラレータ関数の配列引数名にできます。

配列引数のアクセス パターンを RANDOM に指定すると RAM インターフェイスが生成され、SEQUENTIAL に指定するとストリーミング インターフェイスが生成されます。

注記:

- 配列引数のデフォルトのアクセス パターンは RANDOM です。
- 指定したアクセス パターンは、アクセラレータ関数のビヘイビアと一貫している必要があります。
SEQUENTIAL アクセス パターンの場合、関数がすべての配列要素に厳密な順序でアクセスする必要があります。
- このプラグマは、`zero_copy` プラグマがない引数にのみ適用できます。

関連情報

[zero_copy データ ムーバー](#)

ハードウェア関数の最適化

SDSoC™ 環境ではヘテロジニアス クロス コンパイラが導入されており、Zynq®-7000 および Zynq® UltraScale+™ MPSoC プロセッサ CPU の Arm® CPU 用コンパイラと、ハードウェア関数用のプログラマブル ロジック (PL) クロス コンパイラとして Vivado® HLS (高位合成) ツールが含まれています。このセクションでは、デフォルトの動作と、HLS クロス コンパイラに関連する最適化指示子について説明します。

HLS ツールのデフォルトの動作では、ハードウェアが C/C++ コードを正確に反映するように、関数およびループが順次実行されます。最適化指示子を使用すると、パイプライン処理を使用することにより、ハードウェア関数のパフォーマンスが大幅に向上します。この章では、高パフォーマンスを達成するようデザインを最適化するための一般的な手法を示します。

HLS ツールを使用してデザインを最適化するには、さまざまな目標が考えられます。この設計手法では、クロック サイクルごとに新しい入力データ 1 サンプルを処理するパフォーマンスができるだけ高いデザインを作成することを目標としていると想定されるので、そのための最適化がレイテンシまたはリソースを削減する最適化の前に実行されます。

この章で説明する最適化に関する詳細な説明は、『Vivado Design Suite ユーザー ガイド: 高位合成』(UG902) を参照してください。



推奨: ザイリンクスでは、特定の最適化の詳細を確認する前に、設計手法を確認し、ハードウェア関数の最適化をグローバルな観点で理解することをお勧めします。

関連情報

[実際の例](#)

ハードウェア関数の最適化手法

ハードウェア関数は Vivado HLS ツール コンパイラにより PL に合成されます。このコンパイラでは、C/C++ コードが FPGA ハードウェア インプリメンテーションに自動的に変換されますが、ほかのコンパイラと同様、コンパイラ デフォルトが使用されます。

HLS ツールには、コンパイラ デフォルトだけでなく、コード内にプラグマを使用して C/C++ コードに適用できる最適化が多数あります。この章では、適用可能な最適化と、推奨される適用手法を説明します。

ハードウェア関数を最適化するには、次の 2 つのフローがあります。

- **トップダウン フロー:** SDSoC 環境内でプログラムがトップダウンでハードウェア関数に分解され、自動的にデータ フロー モードで動作する関数のパイプラインがシステムのクロスコンパイラで作成されるようにします。各ハードウェア関数のマイクロアーキテクチャは、HLS ツールを使用して最適化します。
- **ボトムアップ フロー:** Vivado Design Suite に含まれる HLS ツールのコンパイラを使用してハードウェア関数をシステムとは別に最適化します。ハードウェア関数を解析して、最適化指示子を使用してデフォルトとは異なるインプリメンテーションを作成し、その結果のハードウェア関数を SDSoC 環境に組み込みます。

ボトムアップフローは、ソフトウェアとハードウェアが別のチームによって最適化される場合や、ソフトウェアプログラマが自社またはパートナーからの既存のハードウェアインプリメンテーションを利用する場合などによく使用されます。どちらのフローもサポートされており、同じ最適化手法を使用できます。どちらのワークフローでも同じ高パフォーマンスのシステムを作成できます。この選択は各チームおよび組織によるワークフロー決定事項であり、ザイリンクスではどちらのフローを使用するかは特に推奨しません。

次の図は、ハードウェア関数の最適化手法を示しています。

図 8: ハードウェア関数の最適化手法

デザインのシミュレーション	- C 関数の検証
デザインの合成	- デザインのベースライン制約の作成
1: 初期最適化	- インターフェイス (およびデータ パック) の定義 - ループトリップ カウントの定義
2: パフォーマンスのためのパイプライン処理	- パイプライン処理およびデータフロー
3: パフォーマンスのための構造最適化	- メモリおよびポートの分割 - 偽依存性の削除
4: レイテンシの削減	- レイテンシ要件の指定 (オプション)
5: エリアの縮小	- 共有によるリソースの回復 (オプション)

X15638-110617

この図は、この手法のすべての手順を示しています。次のセクションで、最適化の詳細を説明します。

注記: コードを最適化する際は、-O 最適化のため、ハードウェア用にビルドするときに [Debug] ではなく [Release] を使用するようにしてください。[Debug] は関数およびシステムが正しく機能することを確認するために使用し、[Release] はパフォーマンス最適化を確認するために使用します。



重要: デザインは、手順 3 の後に最適なパフォーマンスになります。

- 手順 1: [メトリクスの最適化](#)。最適化前にこの章のトピックを確認してください。
- 手順 2: [パフォーマンスのためのパイプライン処理](#)。
- 手順 3: [パフォーマンスのための構造最適化](#)。
- 手順 4: [レイテンシの削減](#)。この手順はデザイン全体のレイテンシを最小限に抑えるか、特定の制御をするために使用し、レイテンシが問題となるようなアプリケーションでのみ必須です。

- 手順5: [エリアの削減](#)。このトピックでは、ハードウェアインプリメンテーションに必要なリソースを削減します。これは、通常大型のハードウェア関数が使用可能なリソースにインプリメントできなかった場合にのみ用います。FPGAのリソース数は決まっているので、パフォーマンス目標が達成されていれば、より小型のインプリメンテーションを作成する利点は通常ありません。

ハードウェア関数のベースライン

ハードウェア関数の最適化の前に、既存コードとコンパイラのデフォルトを使用して達成されるパフォーマンスを理解し、パフォーマンスがどのように測定されるか把握することが重要です。ハードウェアにインプリメントする関数を選択して、プロジェクトを構築します。

プロジェクトをビルドすると、レポートがIDE「Hardware Reports」セクションに表示されます。レポートは、
`<project name>/<build_config>/_sds/vhls/<hw_function>/solution/syn/report/
<hw_function>.rpt` にも含まれています。このレポートには、パフォーマンスの見積もりと使用率の見積もりが表示されます。

パフォーマンス見積もりでは、タイミング、間隔(ループの開始間隔など)、レイテンシの順番で重要になります。

- タイミングサマリには、ターゲットとクロック周期の見積もりが表示されます。クロック周波数の見積もりがターゲットよりも大きい場合、ハードウェアはこのクロック周期では機能しません。クロック周期は、[Project Settings] → [Data Motion Network Clock Frequency] オプションを使用すると削減できます。ただし、これはフローのこの段階ではあくまでも見積もりにすぎないので、見積もりがターゲットを20%しか超えていない場合などは、変更しなくても残りのフローを進めることが可能なこともあります。ビットストリームが生成されるときにさらに最適化が適用され、タイミング要件を満たす可能性はまだありますので、これはあくまでも、ハードウェア関数でタイミングが満たされない可能性があることを示しているだけです。
- 関数の開始間隔(II)は、関数で新しい入力を受信できるようになるまでのクロックサイクル数で、通常システムの最も重要なパフォーマンスメトリクスです。理想的なハードウェア関数では、ハードウェアでクロックサイクルごとに1サンプルのレートでデータが処理されます。ハードウェアに渡される最大容量のデータセットがサイズN(例: `my_array[<N>]`)の場合、最適なIIは $<N> + 1$ になります。つまり、ハードウェア関数では $<N>$ 個のデータサンプルが $<N>$ クロックサイクルで処理され、 $<N>$ 個のサンプルすべてが処理された1クロックサイクル後に新しいデータを受信できます。II $<N>$ のハードウェア関数を作成することはできますが、利点はほとんどなく、必要なプログラマブルロジック(PL)のリソースが増加します。このハードウェア関数は、残りのシステムよりも速いレートでデータを生成して消費するので理想的なものになります。
- ループ開始間隔とは、ループの次の繰り返しでデータの処理が開始するまでのクロックサイクル数です。このメトリクスは、詳細な解析でパフォーマンスのボトルネックを発見して削除する際に重要となります。
- レイテンシとは、関数がすべての出力値を計算するのに必要なクロックサイクル数で、データが適用されてから使用可能になるまでの時間です。ほとんどのアプリケーションでは、ハードウェア関数のレイテンシがソフトウェア関数またはシステム関数(DMAなど)のレイテンシを大きく上回る場合などは特に、これが問題となることはほとんどありませんが、アプリケーションで問題とならないことを確認しておく必要があります。
- ループ繰り返しレイテンシはループ1回分を終了するのにかかるクロックサイクル数であり、ループレイテンシはループのすべての繰り返しを実行するのにかかるサイクル数です。詳細は、[メトリクスの最適化](#)を参照してください。

レポートの「Area Estimates」セクションには、ハードウェア関数をインプリメントするのにPLで必要となるリソース数、および使用可能なリソース数が示されます。重要なメトリクスは使用率(Utilization(%))で、100%を超えないようにする必要があります。100%を超えている場合は、そのハードウェア関数をインプリメントするのに十分なリソースがないことを示しており、より大型のFPGAデバイスに移行することが必要な可能性があります。これは、タイミングと同様、フローのこの段階ではあくまでも見積もりです。使用率が100%を少しだけ超えている場合は、ビットストリーム作成中にハードウェアが最適化される可能性があります。

システムに必要なパフォーマンスおよびハードウェア関数に必要なメトリクスは既にわかっていたはずですが、クロックサイクルなどのハードウェアの概念を熟知していなくても、最高パフォーマンスのハードウェア関数の開始間隔は $II = \langle N \rangle + 1$ (N はその関数で処理される最大データセット) であることがわかるようになりました。現在のデザインパフォーマンスと基本的なパフォーマンスメトリクスについて理解したら、次はハードウェア関数に最適化指示子を適用します。

メトリクスの最適化

次の表に、デザインに追加するかどうかを最初に考慮する必要のある指示子をリストします。

表 5: 最適化ストラテジの手順 1: メトリクスのための最適化

指示子およびコンフィギュレーション	説明
LOOP_TRIPCOUNT	可変範囲のループに使用されます。ループの繰り返し回数の見積もりを指定します。これは合成には影響がなく、レポートにのみ影響します。

ハードウェア関数を最初にコンパイルすると、レポート ファイルにレイテンシと開始間隔 (II) が数値ではなくクエスチョンマーク (?) として表示されることがよくあります。デザインに可変範囲のループがある場合は、コンパイラでレイテンシまたは II を判断できず、この状況を示すためにクエスチョンマーク (?) が使用されます。可変範囲のループでは、ループの繰り返し回数の上限が可変の高さ、幅、深さのパラメーターなどのようにハードウェア関数への入力引数であるため、ループの繰り返し回数の上限をコンパイル時に決定できません。

この状況を解消するには、ハードウェア関数のレポートで数値が示されていない最下位ループを見つけ、LOOP_TRIPCOUNT 指示子を使用して tripcount の見積もり値を指定します。tripcount は、見積もられる繰り返し回数の最小値、平均値、最大値です。これにより、レイテンシと II の値がレポートされるようになり、さまざまな最適化を適用したインプリメンテーションを比較できるようになります。

LOOP_TRIPCOUNT 値はレポートにしか使用されず、結果のハードウェアインプリメンテーションには影響がないので、任意の値を使用できますが、より正確な値を使用した方が有益なレポートが得られます。

パフォーマンスのためのパイプライン処理

高パフォーマンス デザインを作成する次の段階では、関数、ループ、および演算をパイプライン処理します。パイプライン処理により、同時処理が最大限に実行されるようになり非常に高いパフォーマンスを得ることができます。次の表に、パイプライン処理のために使用する指示子を示します。

表 6: 最適化ストラテジの手順 2: パフォーマンスのためのパイプライン処理

指示子およびコンフィギュレーション	説明
PIPELINE	ループまたは関数内で演算を同時に実行できるようにして開始間隔を削減します。
DATAFLOW	タスクレベルのパイプライン処理を有効にし、関数およびループが同時に実行されるようにします。開始間隔を最小にするために使用します。
RESOURCE	変数 (配列、算術演算) をインプリメントするために使用されるハードウェアリソースのパイプライン処理を指定します。
Config Compile	ボトムアップフローを使用する場合に、繰り返し回数に基づいてループが自動的にパイプライン処理されるようにします。

最適化プロセスのこの段階では、できるだけ多くの同時処理演算が作成されます。PIPELINE 指示子は関数およびループに適用できます。DATAFLOW 指示子を関数およびループを含むレベルで使用すると、それらを並列実行できます。RESOURCE 指示子が必要なことはまれですが、最高レベルのパフォーマンスを達成できるようにするために使用可能です。

推奨されるのはボトムアップ方式で、次の点に注意する必要があります。

- 関数およびループの中には、サブ関数が含まれるものがあります。サブ関数がパイプライン処理されていないと、それより上位の関数がパイプライン処理されたときにあまり改善が見られないことがあります。これは、サブ関数がパイプライン処理されていないことが原因です。
- 関数およびループの中には、下位ループが含まれるものがあります。PIPELINE 指示子を使用すると、それより下の階層のループすべてが自動的に展開され、かなり多くのロジックが作成される可能性があります。このため、下位階層のループをパイプライン処理することを推奨します。
- 上位階層をパイプライン処理してその階層より下のループを展開した方が良い場合、可変範囲のループは展開できず、その上の階層のループおよび関数はパイプライン処理できません。この問題を回避するには、可変範囲のループをパイプライン処理し、DATAFLOW 最適化を使用してパイプライン処理されたループが同時に実行されるようにして、ループを含む関数のパフォーマンスを向上します。または、可変範囲を削除するようループを記述し直します。条件付きブレイクを使用して最大の上限を適用します。

最適化プロセスのこの段階での基本的なストラテジは、タスク (関数およびループ) をできるだけパイプライン処理することです。どの関数およびループをパイプライン処理するかについての詳細は、[ハードウェア関数のパイプラインストラテジ](#)を参照してください。

あまり一般的ではありませんが、演算子レベルでパイプライン処理を適用することもできます。たとえば、FPGA のワイヤ配線により予期しない大きな遅延が発生し、デザインに必要なクロック周波数でインプリメントすることが困難な場合があります。このような場合、RESOURCE 指示子を使用して乗算器、加算器、およびブロック RAM などの特定の演算をパイプライン処理してロジックレベルにパイプラインレジスタ段を追加し、ハードウェア関数でデータが再帰の必要なしでできるだけ高いパフォーマンスレベルで処理されるようにします。

注記: コンフィギュレーションコマンドを使用すると、最適化のデフォルト設定を変更できます。これらのコマンドは、ボトムアップフローを使用した場合に Vivado® HLS ツール内からのみ使用できます。詳細は、『Vivado Design Suite ユーザーガイド: 高位合成』([UG902](#))を参照してください。

ハードウェア関数のパイプラインストラテジ

高パフォーマンスのデザインを得るのに重要な最適化指示子は、PIPELINE および DATAFLOW 指示子です。このセクションでは、さまざまな C コードアーキテクチャにこれらの指示子を適用する方法を説明します。

C/C++ 関数には、フレームベースとサンプルベースの 2 種類のコード形式があります。どちらのコーディングスタイルを使用しても、ハードウェア関数は同じパフォーマンスでインプリメントできます。違いは、最適化指示子の適用方法のみです。

フレームベースの C コード

フレームベースのコーディングスタイルの主な特徴は、各トランザクションで関数により複数のデータサンプル (1 データフレーム) が処理されることです。データフレームは、ポインター演算を使用してアクセスされる、データを含む配列またはポインターとして供給されます。トランザクションは、C 関数の完全な実行 1 回と考えられます。このコーディングスタイルでは、データは通常連続したループまたは入れ子のループで処理されます。

次に、フレームベースのCコード例を示します。

```
void foo(
    data_t in1[HEIGHT][WIDTH],
    data_t in2[HEIGHT][WIDTH],
    data_t out[HEIGHT][WIDTH] {
    Loop1: for(int i = 0; i < HEIGHT; i++) {
        Loop2: for(int j = 0; j < WIDTH; j++) {
            out[i][j] = in1[i][j] * in2[i][j];
            Loop3: for(int k = 0; k < NUM_BITS; k++) {
                . . . . .
            }
        }
    }
}
```

C/C++ コードをパイプライン処理してハードウェアでのパフォーマンスを向上する場合は、データのサンプルが処理されるレベルに PIPELINE 最適化指示子を配置します。

上記の例は、画像またはビデオ フレームを処理するのに使用されるコードで、ハードウェア関数を効率的にパイプライン処理する方法を示しています。2つの入力セットがデータ フレームとして関数に供給され、出力もデータ フレームです。この関数をパイプライン処理できる箇所は、次のように複数あります。

- foo 関数のレベル。
- Loop1 ループのレベル。
- Loop2 ループのレベル。
- Loop3 ループのレベル。

PIPELINE 指示子をこれらの箇所に配置するには、利点と欠点があります。これらを理解しておく、PIPELINE 指示子をコードのどこに配置するのが最適化を判断するのに役立ちます。

- 関数レベル: 関数は、データ フレーム (in1 および in2) を入力として受信します。関数が $II = 1$ (各クロック サイクルごとに新しい入力セットを読み込み) でパイプライン処理されると、in1 と in2 のすべての $HEIGHT \times WIDTH$ 値が1つのクロック サイクルで読み込まれるようコンパイルされます。これは1サイクルで読み込むには大量のデータであり、このようなデザインは望ましくないことがほとんどです。

PIPELINE 指示子を関数 foo に適用する場合、この階層の下位にあるループすべてを展開する必要があります。これは、パイプライン内に順序ロジックを存在させることはできないというパイプライン処理の要件です。このため、ロジックのコピーが $HEIGHT \times WIDTH \times NUM_ELEMENT$ 個作成され、デザインが大きくなります。

データは順番にアクセスされるので、ハードウェア関数へのインターフェイスの配列は、次のような複数のハードウェア インターフェイスのタイプとしてインプリメントできます。

- ブロック RAM インターフェイス
- AXI4 インターフェイス
- AXI4-Lite インターフェイス
- AXI4-Stream インターフェイス
- FIFO インターフェイス

ブロック RAM インターフェイスは、クロックごとに2サンプルを供給可能なデュアルポート インターフェイスとしてインプリメントできます。その他のインターフェイス タイプでは、クロックごとに1サンプルしか供給できないので、これがボトルネックとなり、高度に並列化された大型ハードウェア デザインですべてのデータを並列処理できず、ハードウェア リソースが無駄に使用される結果となります。

- Loop1 レベル: Loop1 内のロジックでは、2 次元行列の行全体が処理されます。ここに PIPELINE 指示子を配置すると、クロックサイクルごとに 1 行を処理するデザインが作成されます。これにより、これより下のループは展開されるので、追加のロジックが作成されます。この追加ロジックを利用するには、クロックサイクルごとにデータの行全体 (各ワードが `WIDTH * <number of bits in data_t>` ビット幅になる `HEIGHT` データワードの配列) を転送します。

PS で実行されるホストコードがこのような大型データワードを処理できることはまれなので、この場合も高度に並列化されたハードウェアリソースで、帯域幅の制限のため並列実行できないという結果になる可能性があります。

- Loop2 レベル: Loop2 内のロジックでは、配列からの 1 サンプルが処理されます。画像アルゴリズムの場合は、これが 1 つのピクセルのレベルです。デザインでクロックサイクルごとに 1 サンプル処理される場合は、これがパイプライン処理を実行するレベルです。これは、インターフェイスが PS に入出力されるデータを消費して生成するレートでもあります。

これにより Loop3 が完全に展開されますが、クロックごとに 1 サンプル処理されるようになります。Loop3 の演算がすべて並列処理されることが要件です。典型的なデザインでは、Loop3 のロジックはシフトレジスタで、1 ワード内のビットを処理します。クロックごとに 1 サンプル処理されるようにするには、ループを展開してこれらの処理が並列実行されるようにしてください。Loop2 をパイプライン処理することにより作成されるハードウェア関数では、クロックごとに 1 データサンプル処理され、必要なデータスループットを達成するのに必要な場合のみ並列ロジックが作成されます。

- Loop3 レベル: 前述のように Loop2 が各データサンプルまたはピクセルを演算をする場合、Loop3 のロジックでは通常ビットレベルタスクまたはデータシフトタスクが実行されるので、このレベルではピクセルごとに複数の演算が実行されます。このレベルでパイプライン処理を実行すると、クロックごとにこのループの各演算が 1 回実行されるので、ピクセルごとに `NUM_BITS` クロックとなり、各ピクセルまたはデータサンプルが複数クロックのレートで処理されます。

たとえば、Loop3 にウィンドウリングまたはたたみ込みアルゴリズム用に前のピクセルを保持するシフトレジスタが含まれているとします。このレベルに PIPELINE 指示子を追加すると、各クロックサイクルでデータ値が 1 つシフトされるようになります。デザインは Loop2 のロジックに戻って `NUM_BITS` 回の繰り返し後に次の入力を読み込むので、データ処理レートはかなり遅くなります。

この例の場合、パイプライン処理する理想的な箇所は Loop2 です。

フレームベースのコードの場合、ループレベルでパイプライン処理し、通常はサンプルのレベルで演算を実行するループをパイプライン処理することをお勧めします。確信がない場合は、C コードに `print` コマンドを記述し、これが各クロックサイクルで実行するレベルであるかどうかを確認してください。

上記の例では入れ子のループのセットが 1 つしかありませんが、階層の同レベルに複数のループがある場合は、ループごとに PIPELINE 指示子を配置するのに最適な場所を決定してから、PIPELINE 指示子を関数に適用して、各ループが同時に実行されるようにします。

サンプルベースの C コード

次にサンプルベースの C コードの例を示します。このコーディングスタイルの主な特徴は、トランザクションごとに関数で 1 つのデータサンプルが処理されることです。

```
void foo (data_t *in, data_t *out) {
    static data_t acc;
    Loop1: for (int i=N-1;i>=0;i--) {
        acc+= ..some calculation..;
    }
    *out=acc>>N;
}
```

サンプルベースのコーディングスタイルでは、関数にスタティック変数が含まれることがよくあります。スタティック変数の値は、アキュムレータやサンプルカウンタなど、関数呼び出し間で保持される必要があります。

サンプルベースのコードを使用すると、PIPELINE 指示子の位置が明確になり、 $II = 1$ を達成して、クロックサイクルごとに 1 つのデータ値が処理されるようになります。このためには、関数のパイプライン処理が必要です。

パイプライン処理により関数内のループがすべて展開され、追加ロジックが作成されますが、これを回避する方法はありません。Loop1 がパイプライン処理されない場合、完了するのに最低でも N クロックサイクルがかかります。この後にのみ、関数は次の x 入力値を読み込むことができます。

サンプルレベルで動作する C コードを使用する際は、常に関数をパイプライン処理するようにします。

このタイプのコーディングスタイルでは、ループは通常配列に対して実行され、シフトレジスタまたはラインバッファ関数が実行されます。[パフォーマンスのための構造最適化](#)に示すように、これらの配列を個別の要素に分割し、すべてのサンプルが 1 クロックサイクルでシフトされるようにするのが一般的な方法です。配列がブロック RAM にインプリメントされる場合、各クロックサイクルで読み込みまたは書き出しできるのは最大 2 サンプルだけなので、これがデータ処理のボトルネックとなります。

この例でのソリューションは、関数 `foo` をパイプライン処理することです。これにより、クロックごとに 1 サンプル処理するデザインが得られます。

パフォーマンスのための構造最適化

C コードに、必要なパフォーマンスを達成するための関数またはループのパイプライン処理を妨げるような記述が含まれていることがあります。これは通常、C コードの構造または PL をインプリメントするのに使用されるデフォルトのロジック構造からわかります。この場合、コードを変更する必要があることもありますが、ほとんどの場合は追加の最適化指示子を使用することによりこれらの問題を解決できます。

次に、最適化指示子を使用してインプリメンテーションの構造およびパイプライン処理のパフォーマンスを改善する例を示します。最初の例では、ループに PIPELINE 指示子を追加して、ループのパフォーマンスを改善しています。このコード例は、関数内で使用されているループを示しています。

```
#include "bottleneck.h"
dout_t bottleneck(...) {
    ...
    SUM_LOOP: for(i=3;i<N;i=i+4) {
#pragma HLS PIPELINE
        sum += mem[i] + mem[i-1] + mem[i-2] + mem[i-3];
    }
    ...
}
```

上記のコードがハードウェアにコンパイルされると、次のメッセージが表示されます。

```
INFO: [SCHED 61] Pipelining loop 'SUM_LOOP'.
WARNING: [SCHED 69] Unable to schedule 'load' operation ('mem_load_2',
bottleneck.c:62) on array 'mem' due to limited memory ports.
INFO: [SCHED 61] Pipelining result: Target II: 1, Final II: 2, Depth: 3.
I
```

この例の問題は、配列が PL の効率的なブロック RAM リソースを使用してインプリメントされていることです。これにより、小型で、コストパフォーマンスの高い、高速デザインが得られます。ただし、ブロック RAM では、DDR や SRAM などのメモリと同様、データポート数が通常 2 までに制限されます。

上記のコードでは、sum の値を計算するのに mem から4つのデータ値が必要です。mem は配列であり、データポート数が2つのみのブロック RAM にインプリメントされるので、各クロックサイクルで読み出しまたは書き込みできるのは2つの値のみです。この構成では、sum の値を1クロックサイクルで計算することは不可能であり、データの消費または生成の開始間隔 (II) は1です (クロックごとに1つのデータサンプルを処理)。

メモリポートの制限による問題は、mem 配列に ARRAY_PARTITION 指示子を使用すると解決できます。この指示子を使用すると、配列が複数の小型の配列に分割され、データポート数が増加し、高パフォーマンスのパイプライン処理が可能になります。

次に示すように指示子を追加すると、配列 mem が2つのデュアルポートメモリに分割され、4つの読み出しすべてを1クロックサイクルで実行できるようになります。配列を分割するには、複数のオプションがあります。この例では、係数2でのサイクリック分割により、最初のパーティションに元の配列からの要素0、2、4およびなどが含まれ、2つ目のパーティションに要素1、3、5などが含まれます。分割することにより2つのデュアルポートブロック RAM (合計4つのデータポート) が使用されるようになるので、要素0、1、2、3を1つのクロックサイクルで読み出すことができます。

注記: アクセラレータとして選択された関数の引数である配列に対しては、ARRAY_PARTITION 指示子は使用できません。

```
#include "bottleneck.h"
dout_t bottleneck(...) {
#pragma HLS ARRAY_PARTITION variable=mem cyclic factor=2 dim=1
    ...
    SUM_LOOP: for(i=3;i<N;i=i+4) {
#pragma HLS PIPELINE
        sum += mem[i] + mem[i-1] + mem[i-2] + mem[i-3];
    }
    ...
}
```

ループおよび関数をパイプライン処理するには、ほかにも問題が発生する可能性があります。次の表に、これらの問題に対処するのに有益な、データ構造のボトルネックを削減する指示子をリストします。

表 7: 最適化ストラテジの手順 3: パフォーマンスのための構造最適化

指示子およびコンフィギュレーション	説明
ARRAY_PARTITION	大型の配列を複数の配列または個別のレジスタに分割し、データへのアクセスを改善してブロック RAM のボトルネックを削除します。
DEPENDENCE	ループキャリー依存性を克服し、ループをパイプライン処理できるようにする (またはより短い間隔でパイプラインできるようにする) 追加情報を提供します。
INLINE	関数をインライン展開し、関数の階層をすべて削除します。関数の境界を超えたロジック最適化をイネーブルにし、関数呼び出しのオーバーヘッドを削減することによりレイテンシ/間隔を改善します。
UNROLL	for ループを展開し、複数の演算を1つにまとめたものではなく、複数の個別の演算を作成して、ハードウェアの並列化を向上します。これにより、ループの部分展開が可能になります。
Config Array Partition	グローバル配列を含めた配列の分割方法と、分割が配列ポートに影響するかどうかを指定します。
Config Compile	自動ループパイプラインおよび浮動小数点の math 最適化など、合成特有の最適化を制御します。
Config Schedule	合成のスケジューリング段階で使用するエフォートレベル、出力メッセージの詳細度、およびタイミングを満たすためにパイプライン処理されたタスクの開始間隔 (II) を緩和するかどうかを指定します。
Config Unroll	指定したループ繰り返し数以下のすべてのループを展開します。

配列の自動分割には、ARRAY_PARTITION 指示子だけでなく、配列分割のコンフィギュレーション (Config Array Partition) も使用できます。

ループをパイプライン処理する際に暗示される依存性を削除するため、DEPENDENCE 指示子が必要な場合があります。このような依存性は、SCHED-68 メッセージでレポートされます。

```
@W [SCHED-68] Target II not met due to carried dependence(s)
```

INLINE 指示子を使用すると、関数の境界が削除されます。これは、ロジックまたはループを 1 レベル上の階層に移動するために使用できます。ロジックをその上の関数に含め、ループを上階層に統合すると、上階層では中間サブ関数呼び出しのオーバーヘッドなしですべてのループを同時実行する DATAFLOW 最適化を使用できるので、関数内のロジックをより効率的にパイプライン処理できるようになる場合があります。これにより、デザインのパフォーマンスを向上できる可能性があります。

ループに必要な開始間隔 (II) でパイプライン処理できない場合は、UNROLL 指示子が必要である可能性があります。ループをパイプライン処理しても II = 4 しか達成できない場合、システム内のその他のループおよび関数も II = 4 に制約されます。ループを展開するとさらにロジックが作成されますが、ボトルネックは削除されるので、場合によってはループを展開または部分展開すると有益です。ループをパイプライン処理しても II = 4 しか達成できない場合、係数 4 を使用してループを展開すると、ループの 4 つの繰り返しを並列処理するロジックが作成され、II = 1 を達成できます。

コンフィギュレーション コマンドを使用すると、最適化のデフォルト設定を変更できます。これらのコマンドは、ボトムアップフローを使用した場合に Vivado HLS ツール内からのみ使用できます。詳細は、『Vivado Design Suite ユーザー ガイド: 高位合成』(UG902) を参照してください。開始間隔 (II) を改善するために最適化指示子を使用できない場合、コードの変更が必要となる可能性があります。具体例については、同じガイドを参照してください。

レイテンシの削減

コンパイラで開始間隔 (II) を最小限に抑える処理が終了すると、レイテンシを最小限に抑えるための処理が実行されます。次の表にリストされる最適化指示子を使用すると、特定のレイテンシを指定したり、生成されたレイテンシよりも短いレイテンシを達成するように指定して、結果の II が大きくなってもレイテンシ指示子を満たすようにされるようになります。これにより、デザインのパフォーマンスが低下する可能性があります。

ほとんどのアプリケーションにはスループット要件はありますが、レイテンシ要件はないので、レイテンシ指示子は通常必要ありません。ハードウェア関数がプロセッサと統合される場合、プロセッサのレイテンシが通常システムの制限要因となります。

ループおよび関数がパイプライン処理されない場合、現在のタスクが完了するまで次の入力セットを読み込むことはできないので、スループットがレイテンシにより制限されます。

表 8: 最適化ストラテジの手順 4: レイテンシの削減

指示子	説明
LATENCY	最小および最大レイテンシ制約を指定します。
LOOP_FLATTEN	入れ子のループを 1 つのループに展開して、ループ遷移のオーバーヘッドを削減し、レイテンシを改善します。入れ子のループは、PIPELINE 指示子を適用すると、自動的にフラットになります。
LOOP_MERGE	連続するループを結合して、全体的なレイテンシを削減し、ロジック リソースの共有を増やして最適化を向上します。

ループ最適化指示子は、ループ階層をフラットにしたり、連続するループを結合するために使用できます。レイテンシを向上できるのは、通常ループで作成されたロジックに入ってから出るまでに制御ロジックで 1 クロック サイクル費やされるからです。ループ間の遷移数が少ないほど、デザインが完了するまでにかかるクロック数も少なくなります。

エリアの削減

ハードウェアでは、ロジック関数をインプリメントするのに必要なリソース数はデザイン エリアと呼ばれます。デザイン エリアは、決まったサイズの PL ファブリックでどれだけのリソースが使用されるかも意味します。エリアは、ハードウェアがターゲット デバイスにインプリメントするには大きすぎる場合、およびハードウェア関数が使用可能なエリアをかなり高い割合 (> 90%) で消費している場合などに重要となります。この場合、ハードウェア ロジックどうしをワイヤ接続しようとする、ワイヤ自体がリソースを必要とするので、接続が困難になります。

必要なパフォーマンス ターゲットまたは開始間隔 (II) が満たされたら、次は同じパフォーマンスを維持しながらエリアを削減します。ただし、ハードウェア関数が必要なパフォーマンスで動作していて、残りの PL のスペースにインプリメントされるハードウェア関数がほかにない場合は、エリアを削減しても意味がないので、この段階はオプションです。

最もよく使用されるエリア最適化は、データフロー メモリ チャンネルの最適化で、ハードウェア関数をインプリメントするのに必要なブロック RAM リソース数を削減できます。各デバイスのブロック RAM リソースの数には制限があります。

DATAFLOW 最適化を使用している場合に、デザインのタスクがストリーミング データであるかどうかをコンパイラで判断できない場合は、ピンポンバッファを使用してデータフロー タスク間にメモリ チャンネルがインプリメントされます。これには、それぞれサイズ <N> (<N> はタスク間を転送されるサンプル数で、通常はタスク間で渡される配列のサイズ) の 2 つのブロックが必要です。デザインがパイプライン処理されて、データが 1 つのタスクから次のタスクにストリーミングされて値がシーケンシャルに生成および消費される場合は、STREAM 指示子を使用して配列がストリーミング方式 (深さを指定可能な単純な FIFO を使用) でインプリメントされるように指定すると、エリアを大幅に削減できます。レジスタを使用して浅い FIFO がインプリメントされます。PL ファブリックには、多数のレジスタが含まれています。

ほとんどのアプリケーションでは、深さを 1 に指定すると、メモリ チャンネルが単純なレジスタとしてインプリメントされます。アルゴリズムでデータ圧縮または外挿がインプリメントされ、生成されるよりも多くのデータが消費されたり、消費されるよりも多くのデータが生成されるようなタスクがある場合は、一部の配列の深さを大きくする必要があります。

- 同じレートでデータを生成および消費するタスクでは、それらの間に配列を指定して深さ 1 でストリーミングされるようにします。
- データ レートを X:1 で削減するタスクでは、タスクの入力に配列を指定して、深さ X でストリーミングされるようにします。関数内のこれより前の配列の深さもすべて X にして、FIFO がフルになったためにハードウェア関数が停止することがないようにします。
- データ レートを 1:Y で増加するタスクでは、タスクの出力に配列を指定して、深さ Y でストリーミングされるようにします。関数内のこれより後の配列の深さもすべて Y にして、FIFO がフルになったためにハードウェア関数が停止することがないようにします。

注記: 設定した深さが小さすぎると、FIFO がフルになって残りのシステムが待機状態になるため、ハードウェア エミュレーション中にハードウェア関数が停止してパフォーマンスが低下したり、場合によってはデッドロック状態になることがあります。

次の表に、デザインをインプリメントするために使用されるリソースを最小限に抑える場合に考慮すべきその他の指示子およびコンフィギュレーションを示します。

表 9: 最適化ストラテジの手順 5: エリアの削減

指示子およびコンフィギュレーション	説明
ALLOCATION	使用される演算、ハードウェア リソース、または関数の数を制限します。これによりハードウェア リソースが強制的に共有されますが、レイテンシは増加する可能性があります。
ARRAY_MAP	複数の小型の配列を 1 つの大型の配列に結合し、ブロック RAM リソース数を削減します。
ARRAY_RESHAPE	配列を多数の要素を含むものからワード幅の広いものに変更します。ブロック RAM 数を増やさずにブロック RAM アクセスを向上するのに有益です。
DATA_PACK	内部構造体のデータ フィールドをワード幅の広い 1 つのスカラーにパックして、1 つの制御信号ですべてのフィールドを制御できるようにします。
LOOP_MERGE	連続するループを結合して、全体的なレイテンシを削減し、共有を増やして最適化を向上します。
OCCURRENCE	関数またはループをパイプライン処理する際に、あるロケーションのコードがそれを含む関数またはループのコードよりも低速で実行されるように指定します。
RESOURCE	変数 (配列、算術演算) をインプリメントするために使用される特定のハードウェア リソース (コア) を指定します。
STREAM	特定のメモリ チャネルを FIFO (オプションで深さを指定) としてインプリメントするよう指定します。
Config Bind	合成のバインド段階で使用するエフォート レベルを指定します。使用される演算数をグローバルに最小限に抑えるために使用します。
Config Dataflow	DATAFLOW 最適化でのデフォルトのメモリ チャネルと FIFO の深さを指定します。

演算数を制限し、演算をインプリメントするのに使用するコア (ハードウェア リソース) を選択するには、ALLOCATION と RESOURCE 指示子を使用します。たとえば、関数またはループに乗算器が 1 つだけ使用されるように制限し、パイプライン乗算器を使用してインプリメントされるよう指定できます。

開始間隔を向上するために ARRAY_PARTITION 指示子を使用する場合は、その代わりに ARRAY_RESHAPE 指示子を使用することも考慮してみてください。ARRAY_RESHAPE 最適化では、配列の分割と同様のタスクが実行されますが、分割により作成された要素がより幅の広いデータ ポートを持つ 1 つのブロック RAM に再結合され、必要な RAM リソース数が増加しないようにすることができ、可能性があります。

C コードに類似のインデックスを持つ一連のループが含まれる場合、LOOP_MERGE 指示子を使用してループを結合することにより実行できるようになる最適化もあります。最後に、パイプライン領域のコードの一部が、領域の残りの部分よりも小さい開始間隔で動作する場合は、OCCURRENCE 指示子を使用して、このロジックが低いレートで実行されるよう最適化できます。

注記: コンフィギュレーション コマンドを使用すると、最適化のデフォルト設定を変更できます。これらのコマンドは、ボトムアップフローを使用した場合に Vivado HLS ツール内からのみ使用できます。詳細は、『Vivado Design Suite ユーザー ガイド: 高位合成』(UG902) を参照してください。

デザイン最適化のワークフロー

最適化を実行する前に、プロジェクト内に新しいビルド コンフィギュレーションを作成することをお勧めします。別のビルド コンフィギュレーションを使用すると、結果のセットを異なる結果のセットと比較できます。[Project Settings] → [Manage Build Configurations for the Project] ツールバー ボタンを使用すると、標準の Debug および Release コンフィギュレーションに加えて、よりわかりやすい名前 (Opt_ver1、UnOpt_ver など) のカスタム コンフィギュレーションをウィンドウ内に作成できます。

異なるビルド コンフィギュレーションを使用すると、結果だけでなく、ログ ファイルや FPGA のインプリメントに使用される出力 RTL ファイルも比較できます (RTL ファイルはハードウェア デザインを熟知したユーザーにのみ推奨)。

高パフォーマンス デザインを得るための基本的な最適化戦略は、次のとおりです。

- 初期またはベースライン デザインを作成します。
- ループおよび関数をパイプライン処理します。DATAFLOW 最適化を適用してループおよび関数が同時に実行されるようにします。
- 配列のボトルネックやループの依存性など、パイプラインを制限する問題を解決します (ARRAY_PARTITION および DEPENDENCE 指示子を使用)。
- 特定のレイテンシを指定するか、データフロー メモリ チャンネルのサイズを削減し、ALLOCATION および RESOURCES 指示子を使用してさらにエリアを削減します。

注記: パフォーマンスを満たすため、必要に応じてコードを変更します。

エリアを削減するよりも前に、まずパフォーマンスを満たすようにします。できるだけ少ないリソースでデザインを作成することが戦略である場合は、パフォーマンスを改善する手順を実行しないようにします。ただし、ベースラインの結果は最小のデザインに近いものとなります。

最適化プロセス中は、コンパイル後にコンソールへの出力 (またはログ ファイル) を確認することをお勧めします。コンパイラで指定した最適化のパフォーマンス目標を達成できなかった場合、目標が自動的に緩和され (クロック周波数は例外)、達成できる目標でデザインが作成されます。このため、コンパイルのログ ファイルとレポートを確認して、どのような最適化が実行されたのか理解することが重要です。

最適化適用の詳細は、『Vivado Design Suite ユーザー ガイド: 高位合成』 ([UG902](#)) を参照してください。

最適化ガイドライン

このセクションでは、Vivado HLS ツールを使用してハードウェア関数のパフォーマンスを向上させる基本的な最適化手法について説明します。これらの手法は、関数のインライン展開、ループおよび関数のパイプライン処理、ループ展開、ローカル メモリの帯域幅の増加、およびループと関数間のデータフローのストリーミングです。

関連情報

[関数のインライン展開](#)

[ループのパイプライン処理とループ展開](#)

[ローカル メモリ帯域幅の増加](#)

[データフローのパイプライン処理](#)

関数のインライン展開

ソフトウェア関数のインライン展開と同様、ハードウェア関数のインライン展開にも利点があります。

関数をインライン展開すると、実際の引数と仮引数が解決された後に、関数呼び出しが関数本体のコピーに置き換えられます。インライン展開された関数は、別の階層として表示されなくなります。関数のインライン展開では、インライン関数内の演算が周辺の演算と一緒に効率的に最適化されるので、ループの全体的なレイテンシまたは開始間隔を向上できます。

関数をインライン展開するには、インライン展開する関数の本体の最初に「#pragma HLS inline」と入力します。次のコードでは、mmult_kernel 関数をインライン展開するように Vivado HLS ツールに指示しています。

```
void mmult_kernel(float in_A[A_NROWS][A_NCOLS],
                 float in_B[A_NCOLS][B_NCOLS],
                 float out_C[A_NROWS][B_NCOLS])
{
#pragma HLS INLINE
    int index_a, index_b, index_d;
    // rest of code body omitted
}
```

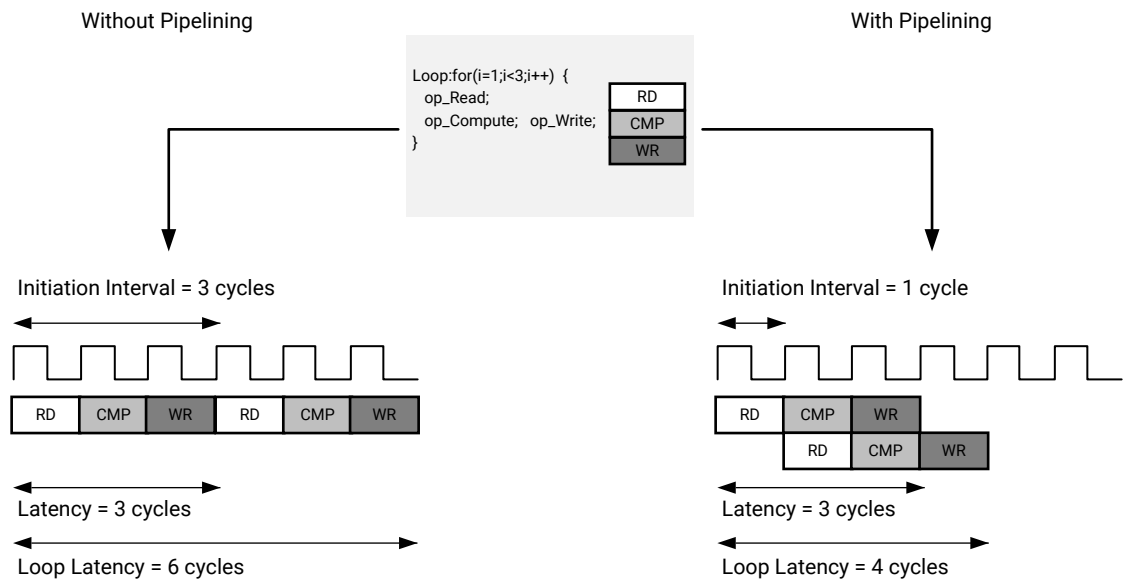
ループのパイプライン処理とループ展開

ループのパイプライン処理とループ展開は、どちらもループの繰り返し間の並列処理を可能にすることで、ハードウェア関数のパフォーマンスを改善する手法です。ここでは、ループのパイプライン処理とループ展開の基本的な概念とこれらの手法を使用するコード例を示し、これらの手法を使用して最適なパフォーマンスを達成する際に制限となる要因について説明します。

ループのパイプライン処理

C/C++ のような逐次言語の場合、ループの演算は順番に実行され、ループの次の繰り返しは、現在の繰り返しの最後の演算が終了してから開始されます。ループのパイプライン処理を使用すると、次の図に示すようにループ内の演算が並列方式でインプリメントできるようになります。

図 9: ループのパイプライン処理



X14770-042319

前の図に示すように、パイプライン処理しない場合、2つの RD 演算間に 3 クロック サイクルあるので、ループ全体が終了するのに 6 クロック サイクル必要となります。パイプライン処理を使用すると、2つの RD 演算間は 1 クロック サイクルなので、ループ全体が終了するのに 4 クロック サイクルしか必要となりません。ループの次の繰り返しは現在の繰り返しが終了する前に開始できます。

開始間隔 (II) はループのパイプライン処理における重要な用語で、ループの連続する繰り返しの開始時間の差をクロックサイクル数で示します。上記の図では、連続するループの繰り返しの開始時間の差は1クロックサイクルしかないので、開始間隔 (II) は1です。

ループをパイプライン処理するには、次に示すように、ループ本体の開始部分に `#pragma HLS pipeline` と記述します。Vivado HLS ツールで、最小限の開始間隔でループのパイプライン処理が試みられます。

```
for (index_a = 0; index_a < A_NROWS; index_a++) {
    for (index_b = 0; index_b < B_NCOLS; index_b++) {
#pragma HLS PIPELINE II=1
        float result = 0;
        for (index_d = 0; index_d < A_NCOLS; index_d++) {
            float product_term = in_A[index_a][index_d] * in_B[index_d]
[index_b];
            result += product_term;
        }
        out_C[index_a * B_NCOLS + index_b] = result;
    }
}
```

ループ展開

ループ展開は、ループの繰り返し間を並列処理するための別の手法で、ループ本体の複数コピーを作成して、ループの繰り返しカウンタをそれに合わせて調整します。次のコードは、展開されていないループを示しています。

```
int sum = 0;
for(int i = 0; i < 10; i++) {
    sum += a[i];
}
```

ループを係数2で展開すると、次のようになります。

```
int sum = 0;
for(int i = 0; i < 10; i+=2) {
    sum += a[i];
    sum += a[i+1];
}
```

係数 $<N>$ でループを展開すると、ループ本体の $<N>$ 個のコピーが作成され、各コピーで参照されるループ変数 (前述の例の場合は `a[i+1]`) がそれに合わせてアップデートされ、ループの繰り返しカウンタ (前述の例の場合は `i+=2`) もそれに合わせてアップデートされます。

ループ展開では、ループの各繰り返しにより多くの演算が作成されるので、Vivado HLS ツールでこれらの演算を並列処理できるようになります。並列処理が増えると、スループットが増加し、システムパフォーマンスも向上します。

- 係数 $<N>$ がループの繰り返しの合計 (前述の例の場合は10) よりも少ない場合、「部分展開」と呼ばれます。
- 係数 $<N>$ がループの繰り返し数と同じ場合は、「全展開」と呼ばれます。全展開の場合、コンパイル時にループ範囲がわかっている必要がありますが、並列処理は最大限に実行されます。

ループを展開するには、そのループの開始部分に `#pragma HLS unroll [factor=N]` を挿入します。オプションの `factor=N` を指定しない場合、ループは全展開されます。

```
int sum = 0;
for(int i = 0; i < 10; i++) {
    #pragma HLS unroll factor=2
    sum += a[i];
}
```

ループのパイプライン処理とループ展開で達成される並列処理を制限する要因

ループのパイプライン処理とループ展開は、どちらもループの繰り返し間の並列処理を可能にしますが、ループの反復間の並列処理は、次の2つの主要な要因により制限されます。

- ループの反復間のデータ依存性。
- 使用可能なハードウェアリソース数。

連続する繰り返しにおける1つの繰り返しの演算から次の繰り返しの演算へのデータ依存性は「ループキャリー依存性」と呼ばれ、現在の繰り返しの演算が終了して次の繰り返しの演算用のデータ入力計算されるまで、次の繰り返しの演算を開始できないことを意味します。ループキャリー依存性があると、ループのパイプライン処理を使用して達成可能な開始間隔とループ展開を使用して実行可能な並列処理が制限されます。

次の例は、変数 `a` と `b` を出力する演算と入力として使用する演算間でのループキャリー依存性を示しています。

```
while (a != b) {
    if (a > b)
        a -= b;
    else
        b -= a;
}
```

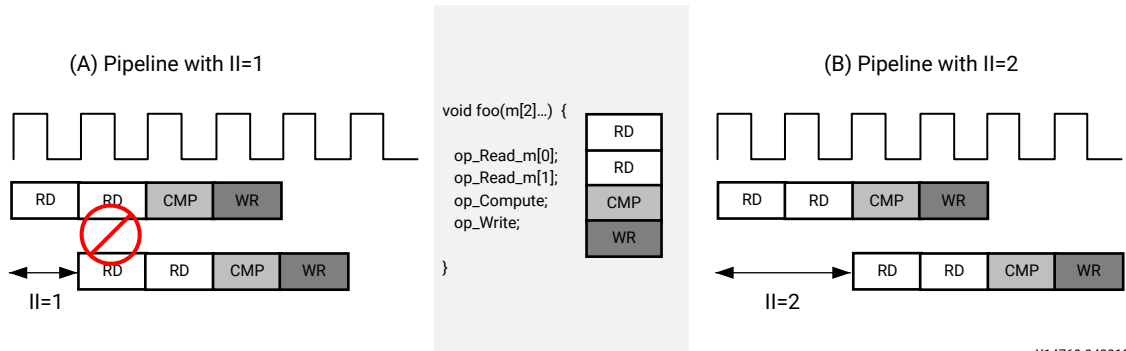
演算このループの次の反復は、次に示すように、現在の反復が計算されて、`a` および `b` の値がアップデートされるまで開始されません。次の例に示すように、ループキャリー依存性は配列アクセスによって発生することがよくあります。

```
for (i = 1; i < N; i++)
    mem[i] = mem[i-1] + i;
```

この例の場合、現在の繰り返しが配列の内容をアップデートするまでループの次の繰り返しの開始できません。ループのパイプライン処理の場合、最小の開始間隔はメモリ読み出し、加算、メモリ書き込みに必要な合計クロックサイクル数です。

使用可能なハードウェアリソース数もループのパイプライン処理およびループ展開のパフォーマンスを制限する要因です。次の図は、リソースの制限により発生する問題の例を示しています。この場合、ループを開始間隔1でパイプライン処理することはできません。

図 10: リソースの競合



X14768-042319

この例の場合、ループが開始間隔 1 でパイプライン処理されると、2つの読み出しが実行されることになります。メモリにシングルポートしかない場合、この2つの読み出しは同時に実行できず、2サイクルで実行する必要があります。このため、最小の開始間隔は図の (B) に示すように 2 になります。同じことは、その他のハードウェアリソースでも発生します。たとえば、`op_compute` が DSP コアを使用してインプリメントされ、それが各サイクルごとに新しい入力を受信できず、このような DSP コアが 1つしかない場合、`op_compute` はサイクルごとに DSP に出力できないので、開始間隔 1 は使用できません。

ローカル メモリ帯域幅の増加

このセクションでは、Vivado HLS ツールで提供されているローカル メモリ帯域幅を増加するいくつかの方法を示します。これらの方法をループのパイプライン処理およびループ展開と共に使用して、システムパフォーマンスを向上できます。

C/C++ プログラムでは、配列は理解しやすく便利なコンストラクトです。これにより、アルゴリズムを簡単にキャプチャして理解できます。HLS ツールでは、各配列はデフォルトでは 1つのポート メモリ リソースを使用してインプリメントされますが、このようなメモリ インプリメンテーションは、パフォーマンス指向のプログラムにとっては最適なメモリ アーキテクチャでないことがあります。制限されたメモリ ポートにより発生するリソース競合の例は、[ループのパイプライン処理とループ展開](#)を参照してください。

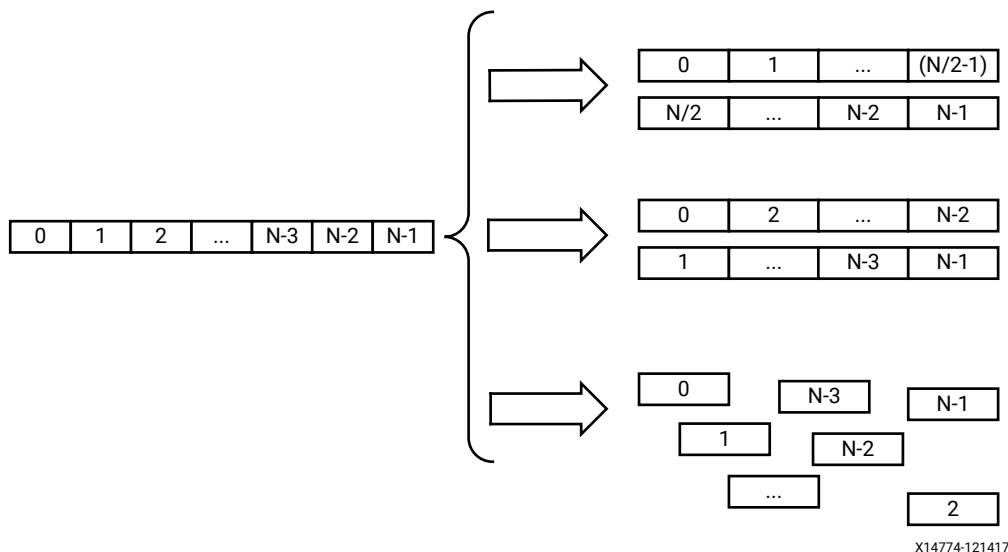
配列の分割

配列は、より小型の配列に分割できます。メモリの物理的なインプリメンテーションでは、読み出しポートと書き込みポートの数に制限があり、ロード/ストアが多用されるアルゴリズムではスループットが制限されます。元の配列 (1つのメモリ リソースとしてインプリメント) を複数の小型の配列 (複数のメモリとしてインプリメント) に分割してロード/ストアポートの有効数を増加させることにより、メモリ帯域幅を向上できる場合があります。

Vivado HLS ツールでは、次の図に示すように、3つの配列分割方法があります。

- **block**: 元の配列の連続した要素が同じサイズのに分割されます。
- **cyclic**: 元の配列の要素がインターリーブされて同じサイズのブロックに分割されます。
- **complete**: デフォルトでは配列が個別の要素に分割されます。これは、配列をメモリとしてではなく複数のレジスタとしてインプリメントすることに対応します。

図 11: 多次元配列の分割



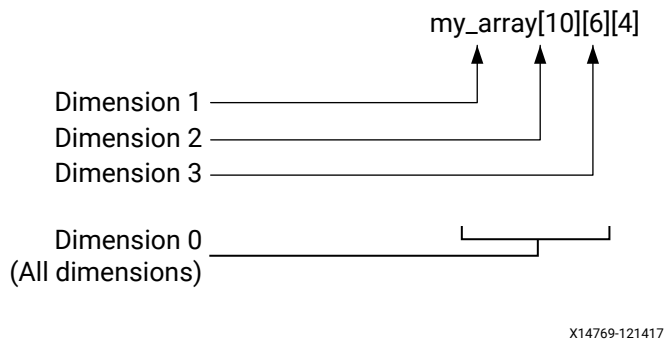
HLS ツールで配列を分割するには、これをハードウェア関数ソースコードに挿入します。

```
#pragma HLS array_partition variable=<variable> <block, cyclic, complete>
factor=<int> dim=<int>
```

block および cyclic 分割では、`factor` オプションを使用して作成する配列の数を指定できます。上記の図では、係数として 2 が使用され、2 つの配列に分割されています。配列に含まれる要素の数が指定された係数の整数倍でない場合、最後の配列に含まれる要素の数はほかの配列よりも少なくなります。

多次元配列を分割する場合は、`dim` オプションを使用してどの次元を分割するかを指定できます。次の図に、多次元配列の異なる次元を分割した例を示します。

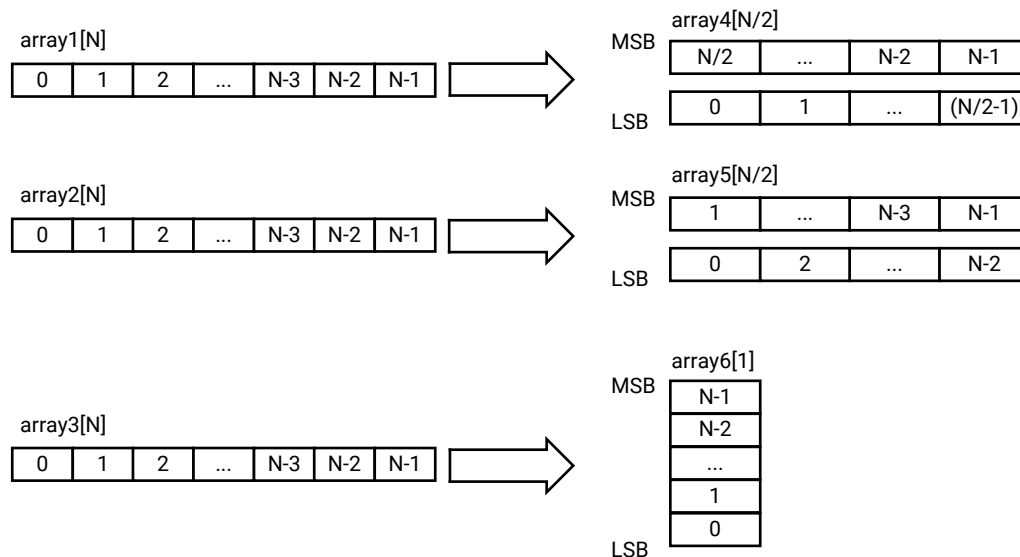
図 12: 多次元配列の分割



配列の再形成

配列を再形成して、メモリ帯域幅を増加できます。再形成では、元の配列の 1 つの次元から異なる要素を取り出して、1 つの幅の広い要素に結合します。配列の再形成は配列の分割に似ていますが、複数の配列に分割するのではなく、配列の要素の幅を広くします。次の図に、配列の再形成の概念を示します。

図 13: 配列の再形成



X14773-121417

Vivado HLS ツールで配列の形状を変更するには、これをハードウェア関数ソースコードに挿入します。

```
#pragma HLS array_reshape variable=<variable> <block, cyclic, complete>
factor=<int> dim=<int>
```

オプションは、配列分割プラグマと同じです。

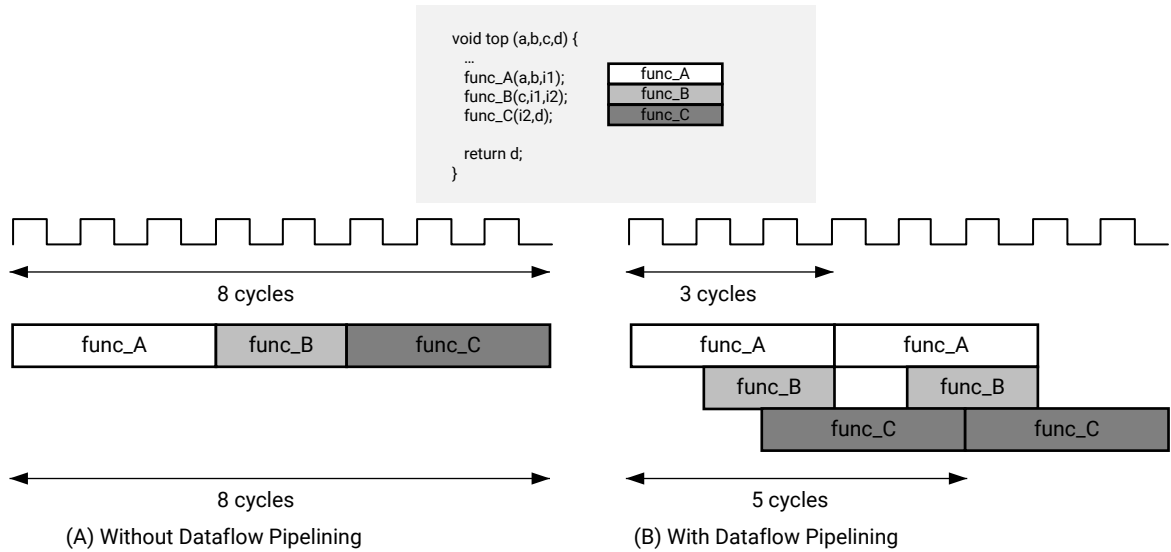
データフローのパイプライン処理

これまでに説明した最適化手法はすべて、乗算、加算、メモリのロード/ストアなどの演算子レベルでの細粒度の並列処理最適化でした。これらの最適化では、これらの演算子間が並列処理されます。一方データフローのパイプライン処理では、関数およびループのレベルで粗粒度の並列処理が実行されます。データフローパイプライン処理により、関数およびループの同時処理が増加します。

関数のデータフローのパイプライン処理

Vivado HLS ツールの一連の関数呼び出しは、デフォルトでは1つの関数が完了してから次の関数が開始します。次の図の (A) は、関数のデータフローパイプライン処理を実行しない場合のレイテンシを示しています。3つの関数に8クロックサイクルかかると想定した場合、このコードでは `func_A` で新しい入力を処理できるようになるまでに8サイクルかかり、`func_C` で出力が書き込まれる (`func_C` の最後に出力が書き込まれると想定) までに8サイクルかかります。

図 14: 関数のデータフローのパイプライン処理



X14772-121417

上記の図の (B) は、データフローパイプライン処理を使用した例を示しています。func_A の実行に 3 サイクルかかるとすると、func_A はこの 3 つの関数すべてが完了するまで待たずに、3 クロックサイクルごとに新しい入力の処理を開始できるので、スループットが増加します。最終的な値は 5 サイクルで出力されるようになり、全体的なレイテンシが短くなります。

HLS ツールでは、関数のデータフローパイプライン処理は関数間にチャンネルを挿入することにより実行されます。これらのチャンネルは、データのプロデューサーおよびコンシューマーのアクセスパターンによって、ピンポンバッファまたは FIFO としてインプリメントされます。

- 関数パラメーター (プロデューサーまたはコンシューマー) が配列の場合は、該当するチャンネルがマルチバッファとして標準メモリアクセス (関連のアドレスおよび制御信号を使用) を使用してインプリメントされます。
- スカラー、ポインター、参照パラメーター、および関数の戻り値の場合は、チャンネルは FIFO としてインプリメンテーションされます。この場合、アドレス生成は不要なので使用されるハードウェアリソースは少なくなります。データに順次アクセスする必要があります。

関数のデータフローパイプライン処理を使用するには、データフロー最適化が必要な部分に #pragma HLS dataflow を挿入します。次に、コード例を示します。

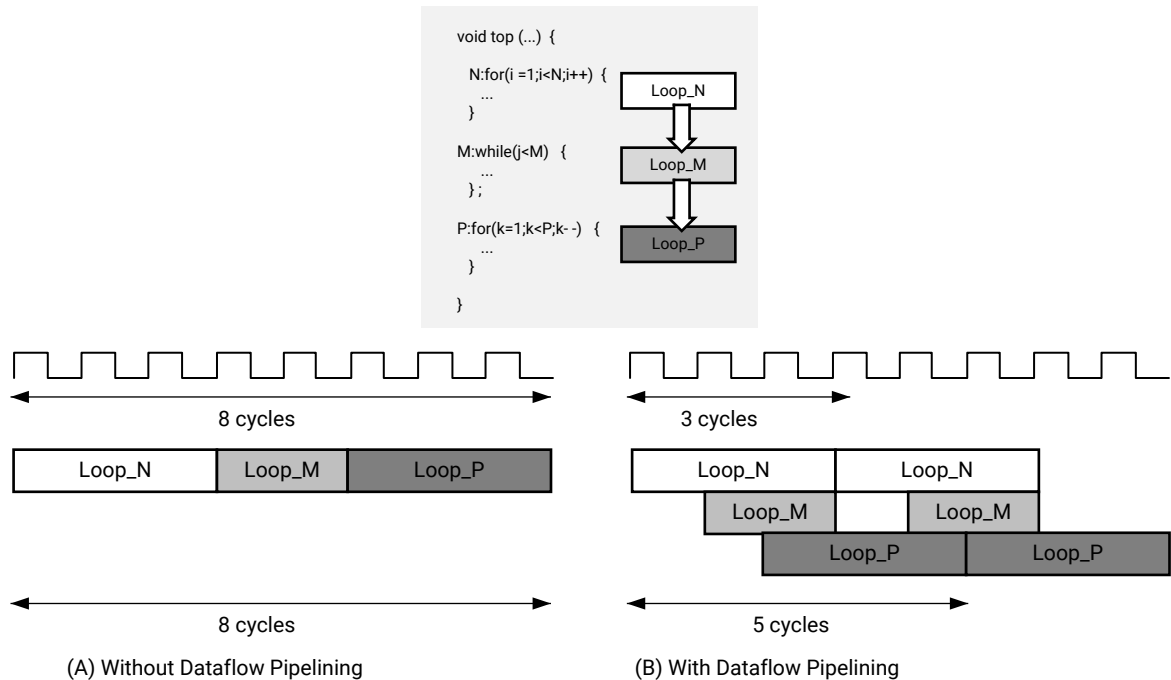
```
void top(a, b, c, d) {
#pragma HLS dataflow
    func_A(a, b, i1);
    func_B(c, i1, i2);
    func_C(i2, d);
}
```

ループのデータフローパイプライン処理

データフローパイプライン処理は、関数に適用するのと同様の方法でループにも適用できます。これにより、ループのシーケンス (通常は順次処理) がイネーブルになり、同時処理されるようになります。データフローパイプライン処理は、関数、ループ、またはすべて関数かすべてループを含む領域に適用する必要があります。ループと関数が混合したスコープに適用しないでください。

データフローパイプライン処理をループに適用した場合の利点については、次の図を参照してください。データフローパイプライン処理を実行しない場合、ループ_Mを開始する前にループ_Nのすべての繰り返しを実行し、完了する必要があります。ループ_Mとループ_Pにも同様の関係があります。この例では、ループ_Nで新しい値を処理できるようになるまでに8サイクルかかり、出力が書き込まれる(ループ_Pが終了したときに出力が書き込まれると想定)までに8サイクルかかります。

図 15: ループのデータフローのパイプライン処理



X14771-051419

データフローパイプラインを使用すると、これらのループが同時に処理されるようになります。上記の図の(B)は、データフローパイプライン処理を使用した例を示しています。ループ_Mの実行に3サイクルかかるとすると、このコードでは3サイクルごとに新しい入力を受信できます。同様に、同じハードウェアリソースを使用して5サイクルごとに出力値を生成できます。Vivado HLS ツールではループ間に自動的にチャンネルが挿入され、データが1つのループから次のループに非同期に流れるようになります。データフローパイプラインを使用した場合と同様、ループ間のチャンネルはマルチバッファかFIFOのいずれかとしてインプリメントされます。

ループのデータフローパイプライン処理を使用するには、データフロー最適な部分に `#pragma HLS dataflow` を挿入します。

ハードウェア関数のインターフェイス

アクセラレーションに必要な関数を定義したら、コンパイルを有効化するための注意点がいくつかあります。Vivado HLS ツールのデータ型 (`ap_int`、`ap_uint`、`ap_fixed` など) は、アプリケーション呼び出しの関数パラメータリストに含めることができません。これらのデータ型はHLS独自のものなので、関連するツールやコンパイラ以外には影響がありません。

たとえば、次の関数がHLSで記述された場合、パラメーターリストを調整して、関数本体でHLSからのデータがさらに汎用のデータ型に移行されるようにしておく必要があります。

```
void foo(ap_int *a, ap_int *b, ap_int *c) { /* Function body */ }
```

ローカル変数を使用する場合は、これを次のように変更する必要があります。

```
void foo(int *a, int *b, int *c) {  
    ap_int *local_a = a;  
    ap_int *local_b = b;  
    ap_int *local_c = c;  
    // Remaining function body  
}
```



重要: 入力データを使用してローカル変数を初期化すると、アクセラレータ内のメモリがかなり使用されてしまうことがあります。このため、入力データ型を適切なHLSデータ型に変更するようにしてください。

メモリ アクセスの最適化

効率的なメモリ アクセスは、FPGA 上で実行されるハードウェア関数のパフォーマンスに重要です。ハードウェア関数に適用できる最適化の最初のカテゴリは、PS と PL 間のメモリ アクセスと転送レートを向上することです。

通常データは外部メモリに格納されますが、オフチップにアクセスするには時間がかかり、システム パフォーマンスが低下します。高パフォーマンス システムでメモリ アクセス時間を削減して実行時間を短縮するため、データをローカル キャッシュに格納できます。

FPGA には、これらのメモリに加え、データ ブロックを格納して効率的にアクセスするための小型から中型のローカル メモリがあります。FPGA を使用してパフォーマンスを向上するシステムのメモリ アーキテクチャは、最も頻繁なメモリ アクセスに最もローカルのメモリを使用する CPU+GPU または CPU+DSP システムのメモリ アーキテクチャに似ています。

次の手法を使用すると、ローカル メモリを多用してデータのアクセスおよび格納のレイテンシの影響を最小限に抑えることにより、ハードウェア関数のデザインを向上できます。

- データ モーションの最適化: PS と PL コンポーネントの間のデータ転送を含みます。SDSoC™ 環境では、PL に選択された関数の引数に基づいてデフォルトのデータ モーション アーキテクチャがインプリメントされますが、データを連続したメモリに格納するために最適化指示子を使用できます。これにより、データ転送の効率が向上します。非常に大型のデータ セットをより効率的に転送するため、スキャッター ギャザー 転送も使用できます。
- データ アクセス パターン: FPGA では、データを同時に高速処理することが可能です。不適切なデータ アクセス パターンを使用すると、データフローが途切れ、FPGA が計算機能を発揮できません。適切なデータ アクセス パターンを使用すると、データの再読み出しを最小限に抑え、条件分岐を使用して 1 つのサンプルをさまざまな方法で処理できます。
- オンチップ メモリ: オンチップ メモリ (プログラマブル ロジック RAM (PLRAM)) は、FPGA のブロック RAM を使用し、物理的に計算の近くに配置されます。1 サイクルの読み出しおよび書き込みが可能であり、メモリ アクセス パフォーマンスを大幅に向上します。最適なデータ アクセス パターンを使用した DDR からこれらのローカル メモリへのデータのコピーは、バースト トランザクションを使用することにより効率的に実行でき、パフォーマンスを向上できます。バースト トランザクションなどの最適なデータ アクセス パターンを使用してデータを DDR からこれらのローカル メモリに効率的にコピーすることにより、パフォーマンスを向上できます。

データ モーションの最適化

データ モーション ネットワークは、PS で実行されるプログラムを PL ファブリックにインプリメントされたハードウェア関数に接続するために使用されるハードウェアです。SDSoC 環境では、データ型に基づいてデータ モーション ネットワークが自動的に作成されますが、指示子を使用すると、速度およびハードウェア リソースの両方を目的としてインプリメンテーションを最適化できます。各データ型に対して作成されるデフォルトのデータ モーション ネットワークは、次のとおりです。

- スカラー: スカラー データ型は、常に AXI LITE データ ムーバーで転送されます。これは、インプリメントするのに必要なハードウェア リソースが非常に少ないメモリ マップド インターフェイスで、ハンドシェイク付きの 1 ワード読み込み/書き出し転送を実行します。このため、スカラー データ型は、起動ごとに 1 回しか転送されない関数引数に理想的です。システムを最適化する場合、これらのデータ型をアドレス指定する必要はほとんどありません。
- 配列: 配列には複数のデータ値が含まれ、高速 DMA 転送を使用すると、より効率的に転送されます。SDSoC 環境には、ハードウェアの効率とハードウェアをインプリメントするのに使用されるリソースを制御するためのオプションが多数あり、特定のデータ ムーバーを選択することもできます。
- ポインター: プラグマを使用しない場合、C/C++ で 1 次元配列型が示されている場合でも、ポインター引数はデフォルトではスカラーとして処理されます。ポインターの書き込みまたは読み出しが複数回実行される場合は、プラグマを使用してデータが効率的に転送されるようにする必要があります。
- 構造体: 1 つの struct はフラット化され、使用されるデータ ムーバーはスカラーか配列かによってデータ メンバーごとに異なります。struct の配列の場合、データ モーション ネットワークは前述の array (配列) と同じです。

SDSoC 環境のデフォルトの動作を理解しておく、要件に基づいて最適なデータ モーション ネットワークを選択することが可能です。

メモリ割り当て

sds++/sdsc (sds++ と表記) コンパイラは、プログラムを解析し、ペイロード サイズ、アクセラレータのハードウェア インターフェイス、および関数引数のプロパティに基づいて、ソフトウェアとハードウェア間の各ハードウェア関数呼び出しの要件を満たすデータ ムーバーを選択します。コンパイラで配列引数を物理的に隣接したメモリに確実に配置できる場合、最も効率の高いデータ ムーバーを使用できます。次の `sds_lib` ライブラリ関数を使用して配列を割り当てまたはメモリ マップすると、メモリが物理的に隣接していることをコンパイラに通知できます。

```
// guarantees physically contiguous memory
sds_alloc(size_t size);

// paddr must point to contiguous memory
sds_mmap(void *paddr, size_t size, void *vaddr);

// assumes physically contiguous memory
sds_register_dmabuf(void *vaddr, int fd);
```

プログラム構造が原因で sds++ コンパイラでメモリが隣接していることを推測できない場合は、次のような警告メッセージが表示されます。

```
WARNING: [DMAAnalysis 83-4492] Unable to determine the memory attributes
passed to foo_arg_A of function foo at foo.cpp:102
```

次のプラグマを関数宣言の直前に挿入すると、データが物理的に隣接するメモリに割り当てられていることをコンパイラに通知できます。

注記: このプラグマは、物理的に隣接したメモリに割り当てることを指定するものではありません。このようなメモリを割り当てるには、`sds_alloc` を使用する必要があります。

```
#pragma SDS data mem attribute (A:PHYSICAL_CONTIGUOUS) // default is
NON_PHYSICAL_CONTIGUOUS
```

データがどのように使用されるかを理解することは、使用するデータ ムーバーを決定するのに役立つので重要です。これは、連続割り当てが必要なものとそうでないものがあるからです。たとえば、データがランダムに使用される場合、スキャッター ギャザー DMA が使用され、次のタイプのオーバーヘッドが発生します。

- ページテーブルのウォーク
- ページのピン留め
- ディスクリプターの作成
- アクセラレータが完了したときのクリーンアップ

コピーおよび共有メモリ セマンティクス

デフォルトでは、ハードウェア関数呼び出しには関数引数のコピー インおよびコピー アウト セマンティクスが関係します。ハードウェア関数引数の共有メモリ モデルを強制することもできますが、バースト転送のスループットが良い一方で、プログラマブル ロジックから外部 DDR へのレイテンシが CPU と比較して大幅に長くなることに注意する必要があります。変数転送で共有メモリ セマンティクスを使用することを宣言するには、次のプラグマを関数宣言の直前に挿入します。

```
#pragma SDS data zero_copy(A[0:<array_size>]) // array_size = number of
elements
```

合成可能なハードウェア関数内では、共有メモリから1語を読み書き (ZERO_COPY プラグマを使用) するのは通常非効率です。パイプライン処理された for-loop を使用してメモリからデータをバーストで読み書きし、ローカル メモリに格納する方が効率的です。

copy および zero_copy メモリ セマンティクスを使用するのも、プログラマブル ロジックと外部 DDR の間でデータをストリーミングしてメモリ効率を最大化できるので効率的です。これにより、変数に対して非シーケンシャル アクセスおよび繰り返しアクセスを実行する必要がある場合に、ハードウェア関数内のローカル メモリにデータが格納されます。たとえば、ビデオ アプリケーションでは通常データがピクセル ストリームとして入力され、FPGA メモリにラインバッファがインプリメントされてピクセル ストリーム データへの複数アクセスがサポートされます。

ハードウェア関数で配列データ転送にストリーミング アクセスが許容される (各エレメントがインデックス順に1回だけアクセスされる) ことを `sds++/sdsc` (`sds++` と表記) コンパイラ コマンドに対して宣言するには、次のプラグマを関数プロトタイプの前直前に挿入します。

```
#pragma SDS data access_pattern(A:SEQUENTIAL) // access pattern = SEQUENTIAL
| RANDOM
```

ポインター型引数としてハードウェア関数に渡された配列では、コンパイラが転送サイズを推論できる場合もありますが、できない場合は次のようなメッセージが表示されます。

```
WARNING: [DMAAnalysis 83-4439] Cannot determine data size for argument p of
function foo
```

次を使用して転送するデータのサイズを指定します。

```
#pragma SDS data copy(p[0:<array_size>]) // for example, int *p
```

データ転送サイズは関数呼び出しごとに変更可能で、プラグマ定義で `<array_size>` を関数呼び出しの範囲内で設定することにより (サイズ設定のすべての変数はその関数へのスカラー引数)、ハードウェア関数に不要なデータ転送を回避できます。

```
#pragma SDS data copy(A[0:L+2*T/3]) // scalar arguments L, T to same function
```

sds++ コンパイラは、特定の同期化コード (`cf_wait()` など) を含むスタブ関数を作成して関数呼び出しインターフェイスをインプリメントし、プログラムの動作を維持して一貫性を保ちます。

関連情報

SDSoC アプリケーションの実行モデル

データ キャッシュ コヒーレンシ

sds++/sdsc (sds++ と表記) コンパイラでは、システムに必要なデータ ムーバーに対して自動的にソフトウェア コンフィギュレーション コードが生成されます。このコードには、必要に応じて下位デバイス ドライバーへのインターフェイスも含まれます。デフォルトでは、システム コンパイラで、CPU とハードウェア関数の間で転送される配列に割り当てられたメモリのキャッシュ コヒーレンシが保持されると想定されます。このため、ハードウェア関数にデータを転送する前にキャッシュをフラッシュし、ハードウェア関数からメモリにデータを転送する前にキャッシュを無効にするコードがコンパイラで生成される場合があります。どちらの処理も正確性を保つために必要ですが、パフォーマンスに影響します。Zynq® デバイスの HP ポートなどを使用する場合は、CPU がメモリにアクセスせず、アプリケーションの正確性がキャッシュ コヒーレンシに依存しないことが確実であれば、デフォルトを変更できます。不要なキャッシュのフラッシュのオーバーヘッドを回避するには、次の API を使用してメモリを割り当てます。

```
void *sds_alloc_non_cacheable(size_t size)
```

キャッシュ不可能なメモリの典型的なユースケースとしては、一部のフレームバッファがプログラマブルロジックからアクセスされるが CPU からはアクセスされないビデオアプリケーションが挙げられます。

データ アクセス パターン

FPGA では大量の並列アーキテクチャによりプロセッサのシーケンシャル演算よりもかなり高速に演算を実行できるので、そのパフォーマンスを活用するため、C コードのインプリメントに FPGA が選択されます。

ここでは、C コードのアクセスパターンが結果にどう影響するかについて説明します。最も注意が必要なアクセスパターンはハードウェア関数への入力および出力のパターンですが、ハードウェア関数内にボトルネックがあると、関数への入力および出力の転送レートに悪影響を及ぼすので、関数内のアクセスパターンを考慮することをお勧めします。

このセクションでは画像のたたみ込みアルゴリズムについて説明し、データアクセスパターンがパフォーマンスに悪影響を与え、適切なパターンを使用することにより FPGA の並列化およびパフォーマンス機能を活用できることを示します。

- まずアルゴリズムを確認し、FPGA でのパフォーマンスの制限となるデータアクセスについて説明します。
- 次に、最高のパフォーマンスを達成するためのアルゴリズムの記述方法について説明します。

不適切なデータ アクセス パターンを使用したアルゴリズム

ここでは画像に適用される標準的なたたみ込み関数を使用して、FPGA で可能なパフォーマンスが C コードによりどのように劣化するかを説明します。この例では、データに対してまず水平たたみ込みが実行された後、垂直たたみ込みが実行されます。画像のエッジのデータはたたみ込み範囲外にあるので、最後に境界周辺のデータが処理されます。

アルゴリズムでの処理は、次の順で実行されます。

- 水平たたみ込み。
- 垂直たたみ込み。

- 境界ピクセル処理。

```
static void convolution_orig(
    int width,
    int height,
    const T *src,
    T *dst,
    const T *hcoeff,
    const T *vcoeff) {
    T local[MAX_IMG_ROWS*MAX_IMG_COLS];

    // Horizontal convolution
    HconvH:for(int row = 0; row < height; row++){
        HconvWfor(int col = border_width; col < width - border_width; col++){
            Hconv:for(int i = - border_width; i <= border_width; i++){
                ...
            }
        }
    }

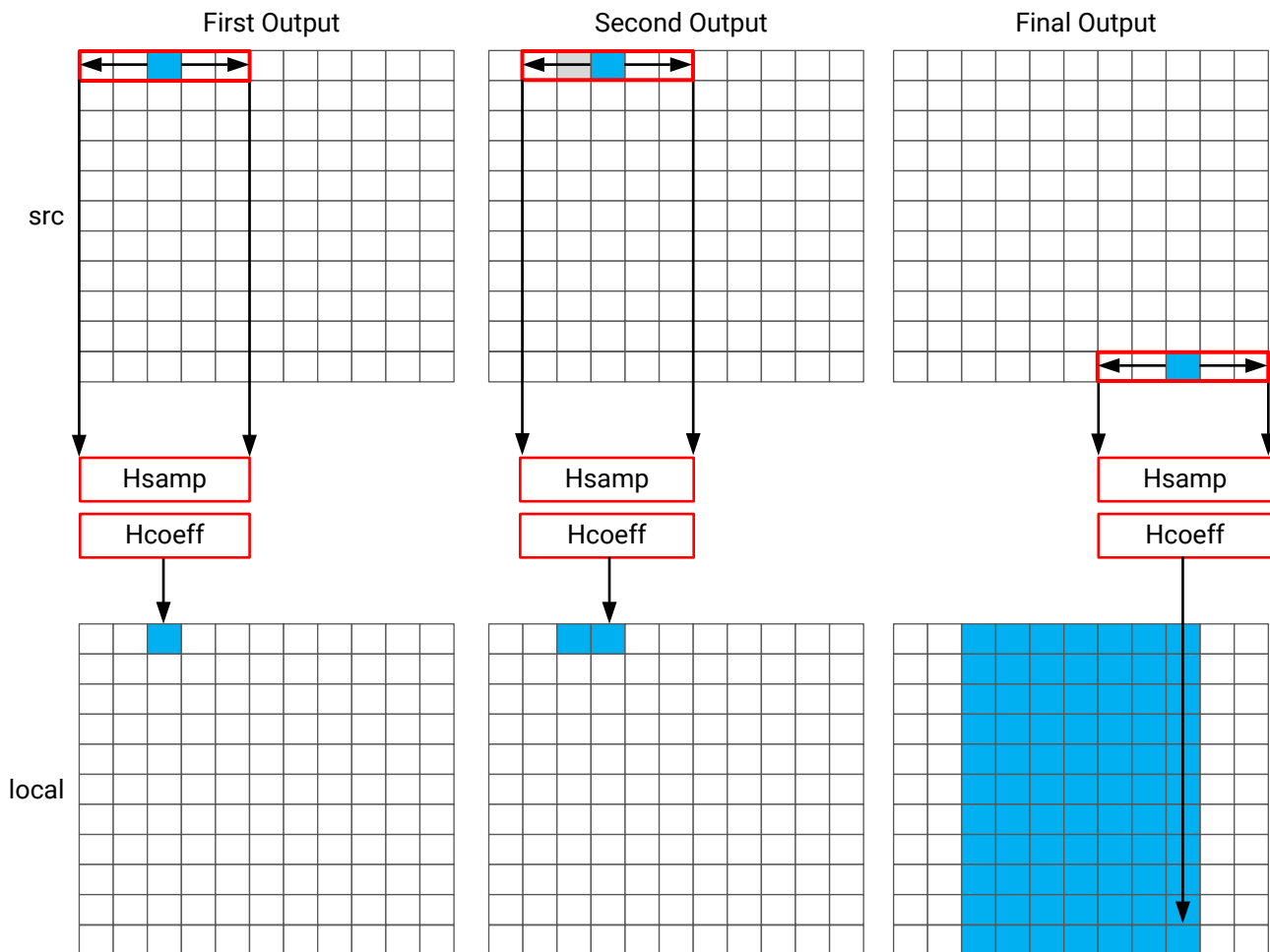
    // Vertical convolution
    VconvH:for(int row = border_width; row < height - border_width; row++){
        VconvW:for(int col = 0; col < width; col++){
            Vconv:for(int i = - border_width; i <= border_width; i++){
            }
        }
    }

    // Border pixels
    Top_Border:for(int col = 0; col < border_width; col++){
    }
    Side_Border:for(int row = border_width; row < height - border_width; row++){
    }
    Bottom_Border:for(int row = height - border_width; row < height; row++){
    }
}
```

標準水平たたみ込み

まず、次の図に示すように水平方向のたたみ込みを実行します。

図 16: 水平方向のたたみ込み



X14296-121417

たたみ込みは、K個のデータサンプルとK個のたたみ込み係数を使用して実行されます。上の図ではKの値は5ですが、この値はコードで定義されます。たたみ込みを実行するには、K個以上のデータサンプルが必要です。たたみ込みウィンドウは、画像外にあるピクセルを含む必要があるため、最初のピクセルでは開始できません。

対称たたみ込みを実行すると、src入力からの最初のK個のデータサンプルが水平係数でたたみ込まれ、最初の出力が計算されます。2つ目の出力を計算するには、次のK個のデータサンプルが使用されます。この計算は、最後の出力が書き込まれるまで各行に対して実行されます。

この操作を実行するCコードは次のとおりです。

```
const int conv_size = K;
const int border_width = int(conv_size / 2);

#ifdef __SYNTHESIS__
T * const local = new T[MAX_IMG_ROWS*MAX_IMG_COLS];
```

```
#else // Static storage allocation for HLS, dynamic otherwise
T local[MAX_IMG_ROWS*MAX_IMG_COLS];
#endif

Clear_Local:for(int i = 0; i < height * width; i++){
    local[i]=0;
}

// Horizontal convolution
HconvH:for(int col = 0; col < height; col++){
    HconvWfor(int row = border_width; row < width - border_width; row++){
        int pixel = col * width + row;
        Hconv:for(int i = - border_width; i <= border_width; i++){
            local[pixel] += src[pixel + i] * hcoeff[i + border_width];
        }
    }
}
```

コードは簡単ですが、ハードウェアの結果の質に悪影響を及ぼす問題がいくつかあります。

1つ目の問題は、C コンパイル中に大型ストレージが必要であるということです。アルゴリズムの中間結果は内部 local 配列に格納されます。HEIGHT*WIDTH の配列が必要で、1920*1080 の標準ビデオ画像では 2,073,600 の値が保持されます。

- Zynq®-7000 SoC または Zynq® UltraScale+™ MPSoC デバイスをターゲットとするコンパイラおよび多くのホストシステムでは、この量のローカルストレージのためランタイムでスタックオーバーフローが発生します (ターゲットデバイス上での実行、Vivado® HLS (高位合成) ツール内での協調シミュレーションの実行など)。local 配列のデータはスタックに配置され、OS で管理されるヒープには含まれません。arm-linux-gnueabi-hf-g++ でクロスコンパイルする際は、-Wl, "-z stacksize=4194304" リンカー オプションを使用して十分なスタック空間を割り当てます。

注記: このオプションの構文は、リンカーによって異なります。

- 関数がハードウェアでのみ実行される場合は、__SYNTHESIS__ マクロを使用するとこのような問題を回避できます。このマクロは、ハードウェア関数がハードウェアに合成されるときにシステムコンパイラにより自動的に定義されます。上記のコードでは、C シミュレーション中にダイナミックメモリ割り当てを使用してコンパイルの問題を回避しており、合成中はスタティックストレージのみが使用されます。このマクロを使用する短所は、C シミュレーションで検証されたコードが合成されるコードとは異なるものになることです。この例の場合はコードは複雑ではないので、動作は同じになります。
- この local 配列の主な問題は、FPGA インプリメンテーションの質です。これは配列なので、内部 FPGA ブロック RAM を使用してインプリメントされます。これは、FPGA 内にインプリメントするにはかなり大きいメモリであり、より大型でコストのかかる FPGA デバイスが必要となる可能性があります。DATAFLOW 最適化を使用して、小型で効率的な FIFO を介してデータをストリーミングすると、ブロック RAM の使用を最小限に抑えることはできますが、データがストリーミングでシーケンシャルに使用されるようにする必要があります。現在のところ、そのような要件はありません。

2つ目の問題は、パフォーマンスと local 配列の初期化に関するものです。Clear_Local ループは local 配列の値を 0 に設定するために使用されます。高パフォーマンスで実行するためにこのループをハードウェアでパイプライン処理しても、この操作をインプリメントするのに約 2 百万クロック サイクル (HEIGHT*WIDTH) が必要です。このメモリの初期化中、システムで画像処理を実行することはできません。同じデータの初期化は、HConv ループ内の一時的な変数を使用して、書き出し前に累積を初期化することにより実行できます。

最後の問題は、データのスループット、つまりシステム パフォーマンスがデータ アクセス パターンにより制限されることです。

- 最初のたたみ込み出力を作成するため、最初の K 個の値が入力から読み出されます。
- 2 つ目の出力の計算には、新しい値が読み出され、その後同じ K-1 個の値が再度読み出されます。

高パフォーマンスの FPGA にするには、PS へのアクセスを最小限に抑えることが重要です。前に既にフェッチされたデータに再度アクセスすることは、システムのパフォーマンスに悪影響を与えます。FPGA では多数の計算を同時に実行して高パフォーマンスを達成することが可能ですが、データのフローが値を再読み出しするために頻繁に中断されると高パフォーマンスは得られません。

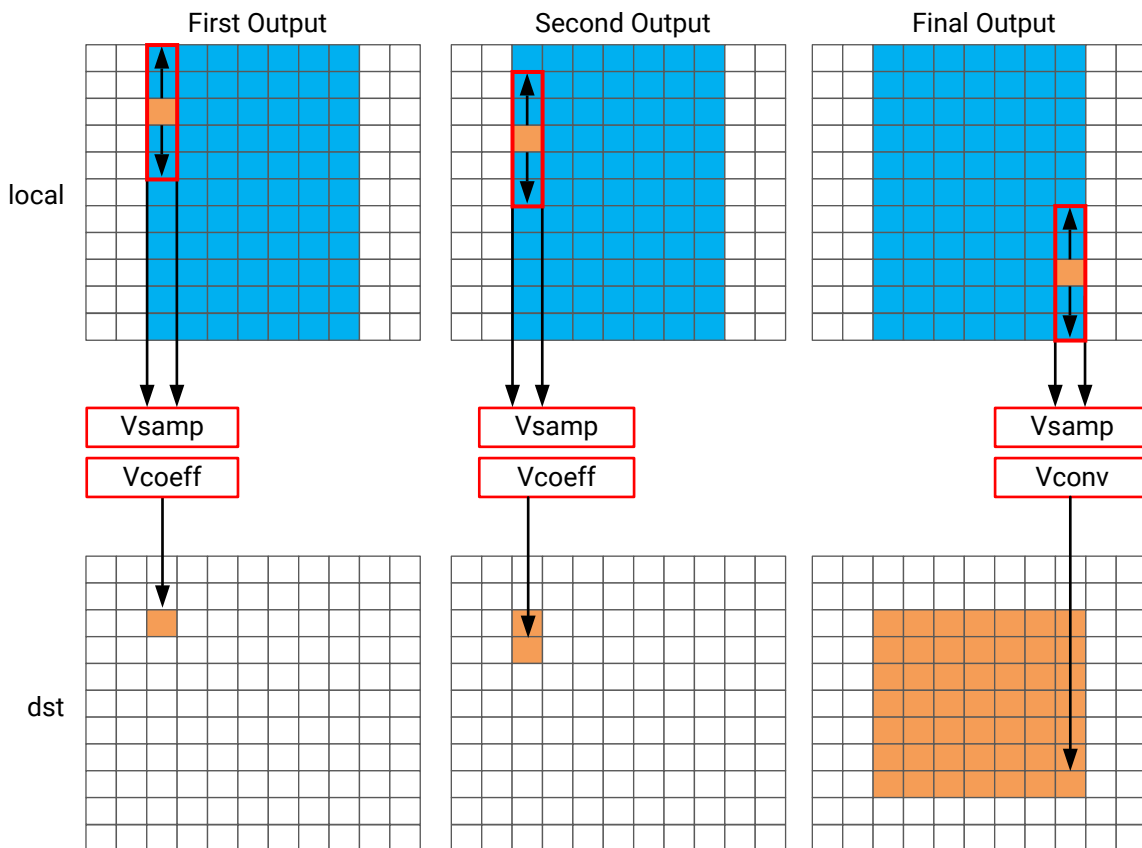
注記: 高パフォーマンスを達成するには、PS からのデータにアクセスするのは 1 回のみにし、小型のローカル ストレージ (小型から中型のサイズの配列) に格納して再利用する必要があります。

上記のコードではデータを何度も読み出す必要があるので、操作を使用してプロセッサから直接ストリーミングできません。

標準垂直たたみ込み

次の段階では、次の図に示す垂直たたみ込みを実行します。

図 17: 垂直たたみ込み



X14299-011519

垂直たたみ込みのプロセスは、水平たたみ込みと似ています。たたみ込み係数(この場合は Vcoeff)を使用したたたみ込みには、K 個のデータサンプルが必要です。垂直方向の最初の K 個のサンプルを使用して最初の出力が作成された後、次の K 個の値を使用して 2 つ目の出力が作成されます。この処理は、最後の出力が作成されるまで各列に対して実行されます。

水平および垂直の境界効果により、垂直たたみ込み後の画像はソース画像 src よりも小さくなります。

これらを実行するコードは次のとおりです。

```
Clear Dst:for(int i = 0; i < height * width; i++){
    dst[i]=0;
}
// Vertical convolution
VconvH:for(int col = border_width; col < height - border_width; col++){
    VconvW:for(int row = 0; row < width; row++){
        int pixel = col * width + row;
        Vconv:for(int i = - border_width; i <= border_width; i++){
            int offset = i * width;
            dst[pixel] += local[pixel + offset] * vcoeff[i + border_width];
        }
    }
}
```

このコードには、水平たたみ込みコードを使用して既に説明した問題と同様の問題があります。

- 出力画像 `dst` の値を 0 に設定するのに、多数のクロック サイクルが費やされます。この場合、1920*1080 画像サイズに対してさらに約 2 百万サイクル必要です。
- `local` 配列に格納されたデータを再度読み出すために、各ピクセルが複数回アクセスされます。
- 出力配列/ポート `dst` に対しても、各ピクセルが複数回書き込まれます。

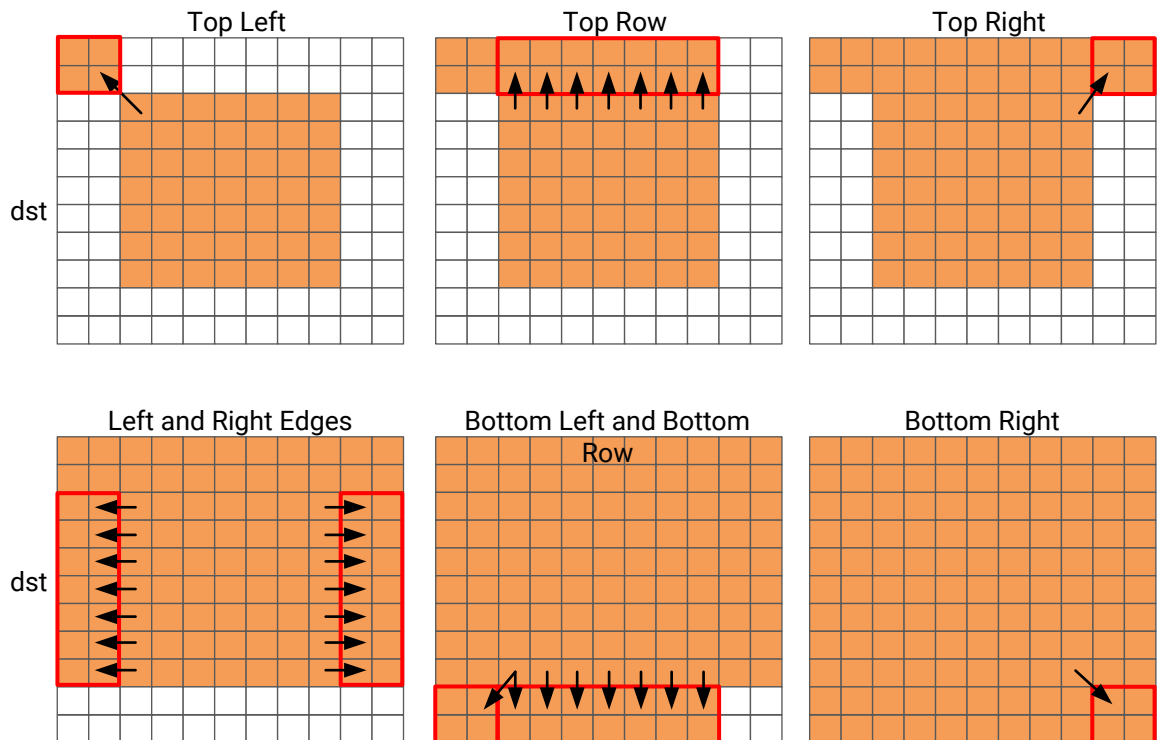
上記のコードのアクセス パターンでは、大型の `local` 配列が必要となります。アルゴリズムでは、最初の計算を実行するために行 `K` のデータが使用可能になっていることが必要です。次の列に進む前に各行のデータを処理するため、画像全体がローカルに格納されている必要があります。そのためすべての値が格納されるので、FPGA に大型のローカルストレージが作成されます。

さらに、`local` 配列からデータがストリーミング出力されないで、コンパイラ指示子を使用してハードウェア関数のパフォーマンスを最適化する段階に達したときに、水平および垂直ループ間のデータのフローを FIFO (高パフォーマンスで低リソースのユニット) を使用して制御することができません。FIFO は、シーケンシャル アクセス パターンでのみ使用可能です。このコードでは任意/ランダム アクセスが必要なため、パフォーマンスを向上するにはピンポンブロック RAM が必要です。ローカル配列のインプリメンテーションに必要なメモリが 2 倍の約 4 百万個のデータ サンプルになり、1 つの FPGA には大きすぎます。

標準境界ピクセルたたみ込み

たたみ込みの最後の段階では、境界周辺のデータを作成します。これらのピクセルは、たたみ込み出力の最も近いピクセルを再利用することにより作成されます。次の図に、これをどのように達成するかを示します。

図 18: 標準境界ピクセルたたみ込み



X14294-011519

境界領域は、最も近い有効な値を使用して作成されます。前の図に示す操作は、次のコードで実行されます。

```
int border_width_offset = border_width * width;
int border_height_offset = (height - border_width - 1) * width;

// Border pixels

Top_Border:for(int col = 0; col < border_width; col++){
    int offset = col * width;
    for(int row = 0; row < border_width; row++){
        int pixel = offset + row;
        dst[pixel] = dst[border_width_offset + border_width];
    }
    for(int row = border_width; row < width - border_width; row++){
        int pixel = offset + row;
        dst[pixel] = dst[border_width_offset + row];
    }
    for(int row = width - border_width; row < width; row++){
        int pixel = offset + row;
        dst[pixel] = dst[border_width_offset + width - border_width - 1];
    }
}

Side_Border:for(int col = border_width; col < height - border_width; col++){
    int offset = col * width;
    for(int row = 0; row < border_width; row++){
        int pixel = offset + row;
        dst[pixel] = dst[offset + border_width];
    }
    for(int row = width - border_width; row < width; row++){
        int pixel = offset + row;
        dst[pixel] = dst[offset + width - border_width - 1];
    }
}

Bottom_Border:for(int col = height - border_width; col < height; col++){
    int offset = col * width;
    for(int row = 0; row < border_width; row++){
        int pixel = offset + row;
        dst[pixel] = dst[border_height_offset + border_width];
    }
    for(int row = border_width; row < width - border_width; row++){
        int pixel = offset + row;
        dst[pixel] = dst[border_height_offset + row];
    }
    for(int row = width - border_width; row < width; row++){
        int pixel = offset + row;
        dst[pixel] = dst[border_height_offset + width - border_width - 1];
    }
}
```

このコードには、データに繰り返しアクセスするという同じ問題があります。FPGA 外の dst 配列に格納されたデータは、入力データとして複数回読み出されることが可能になっている必要があります。最初のループでも、dst[border_width_offset + border_width] が複数回読み出されますが、border_width_offset および border_width の値は変更されません。

このコードは、読み出しと書き込みのどちらも非常にわかりやすいものです。SDSoC 環境でインプリメントすると約 120M クロックサイクルであり、CPU のパフォーマンスよりも少し長くなります。ただし、次のセクションで示すように最適なデータ アクセス パターンを使用すると、同じアルゴリズムをクロックサイクルごとに 1 ピクセルのレート (約 2M クロックサイクル) で FPGA にインプリメントできます。

次のような不適切なデータ アクセス パターンを使用すると、パフォーマンスが低下し、FPGA インプリメンテーションのサイズが大きくなります。

- データを繰り返し読み出すために複数回アクセスします。可能な場合は、ローカル ストレージを使用してください。
- データに任意またはランダムにアクセスします。データをローカルに配列として格納する必要があり、リソースが多く必要になります。
- 配列のデフォルト値を設定すると、クロックサイクル数が多くなり、パフォーマンスが低下します。

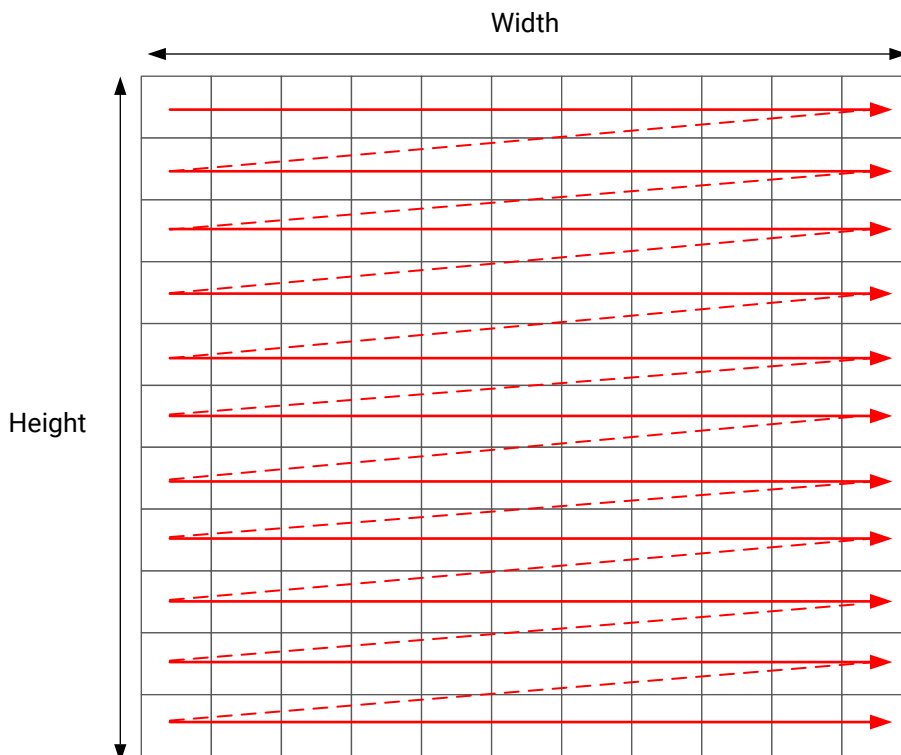
最適なデータ アクセス パターンを使用したアルゴリズム

前のセクションで説明したたたみ込みの例 (リソース使用量が最小の高パフォーマンス デザイン) をインプリメントするには、次を実行します。

- システム中のデータフローを最大限にします。データフローが途切れてしまうようなコーディング手法やアルゴリズム処理は避けてください。
- データの再利用を最大限にします。ローカル キャッシュを使用して、同じデータを何回も読み出す必要がないようにし、入力データのフローが途切れないようにします。
- 条件分岐を使用します。これは CPU、GPU、または DSP ではコストがかかりますが、FPGA では最適な方法です。

最初の段階では、システムから FPGA、FPGA からシステムにデータがどのように流れるかを理解します。たたみ込みアルゴリズムは、画像に対して実行されます。画像からのデータは、次の図に示すような標準のラスタ走査順序で転送されます。

図 19: ラスタ走査



X14298-121417

データがFPGAにストリーミング方式で転送された場合、FPGAではそれをストリーミング方式で処理し、同じ方式で転送し戻す必要があります。

次に示すたたみ込みアルゴリズムでは、このコード形式を使用しています。この抽象レベルでは、コードが簡潔に示されますが、各ループ間に中間バッファ `hconv` および `vconv` があります。これらはストリーミング方式でアクセスされるので、最終的なインプリメンテーションでは1つのレジスタに最適化されます。

```
template<typename T, int K>
static void convolution_strm(
    int width,
    int height,
    T src[TEST_IMG_ROWS][TEST_IMG_COLS],
    T dst[TEST_IMG_ROWS][TEST_IMG_COLS],
    const T *hcoeff,
    const T *vcoeff)
{
    T hconv_buffer[MAX_IMG_COLS*MAX_IMG_ROWS];
    T vconv_buffer[MAX_IMG_COLS*MAX_IMG_ROWS];
    T *phconv, *pvconv;

    // These assertions let HLS know the upper bounds of loops
    assert(height < MAX_IMG_ROWS);
    assert(width < MAX_IMG_COLS);
    assert(vconv_xlim < MAX_IMG_COLS - (K - 1));
    // Horizontal convolution
    HConvH:for(int col = 0; col < height; col++) {
        HConvW:for(int row = 0; row < width; row++) {
            HConv:for(int i = 0; i < K; i++) {
            }
        }
    }
    // Vertical convolution
    VConvH:for(int col = 0; col < height; col++) {
        VConvW:for(int row = 0; row < vconv_xlim; row++) {
            VConv:for(int i = 0; i < K; i++) {
            }
        }
    }
    Border:for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
        }
    }
}
```

これで3つの処理ループに条件分岐が含まれ、データが連続処理されるようになります。

最適な水平たたみ込み

前のK個のサンプルを使用してたたみ込み結果を計算する必要があるので、サンプルが一時キャッシュ `hwin` にコピーされます。このようにローカルストレージを使用すると、PSから値を読み込み直す必要はなく、データのフローは途切れません。最初の計算では、`hwin` に結果を計算するのに十分な値が含まれていないので、条件により出力値は書き出されません。FPGAインプリメンテーションで効率的な方法で計算を実行するため、水平たたみ込みが計算されます。

アルゴリズムは入力サンプルを読み込み続け、`hwin` キャッシュに格納します。新しいサンプルが読み込まれるたびに、不要なサンプルが `hwin` から排出されます。K番目の入力を読み込まれると、最初の出力値を書き込むことができるようになります。このように、最後のサンプルが読み込まれるまで行ごとに処理されます。この段階で `hwin` に格納されているのは最後のK個のサンプルだけであり、そのすべてがたたみ込み計算に必要となります。

次に示すように、この操作を実行するコードでは、ローカルストレージを使用することによりPLからの再読み込みを回避して最終インプリメンテーションでローカルストレージからの読み出しを並列に実行できるようにし、さらに条件分岐を多用して各新しいデータサンプルを異なる方法で処理できるようにしています。

```
// Horizontal convolution
phconv=hconv_buffer; // set / reset pointer to start of buffer

// These assertions let HLS know the upper bounds of loops
assert(height < MAX_IMG_ROWS);
assert(width < MAX_IMG_COLS);
assert(vconv_xlim < MAX_IMG_COLS - (K - 1));
HConvH:for(int col = 0; col < height; col++) {
    HConvW:for(int row = 0; row < width; row++) {
        #pragma HLS PIPELINE
        T in_val = *src++;
        // Reset pixel value on-the-fly - eliminates an O(height*width) loop
        T out_val = 0;
        HConv:for(int i = 0; i < K; i++) {
            hwin[i] = i < K - 1 ? hwin[i + 1] : in_val;
            out_val += hwin[i] * hcoeff[i];
        }
        if (row >= K - 1) {
            *phconv++=out_val;
        }
    }
}
```

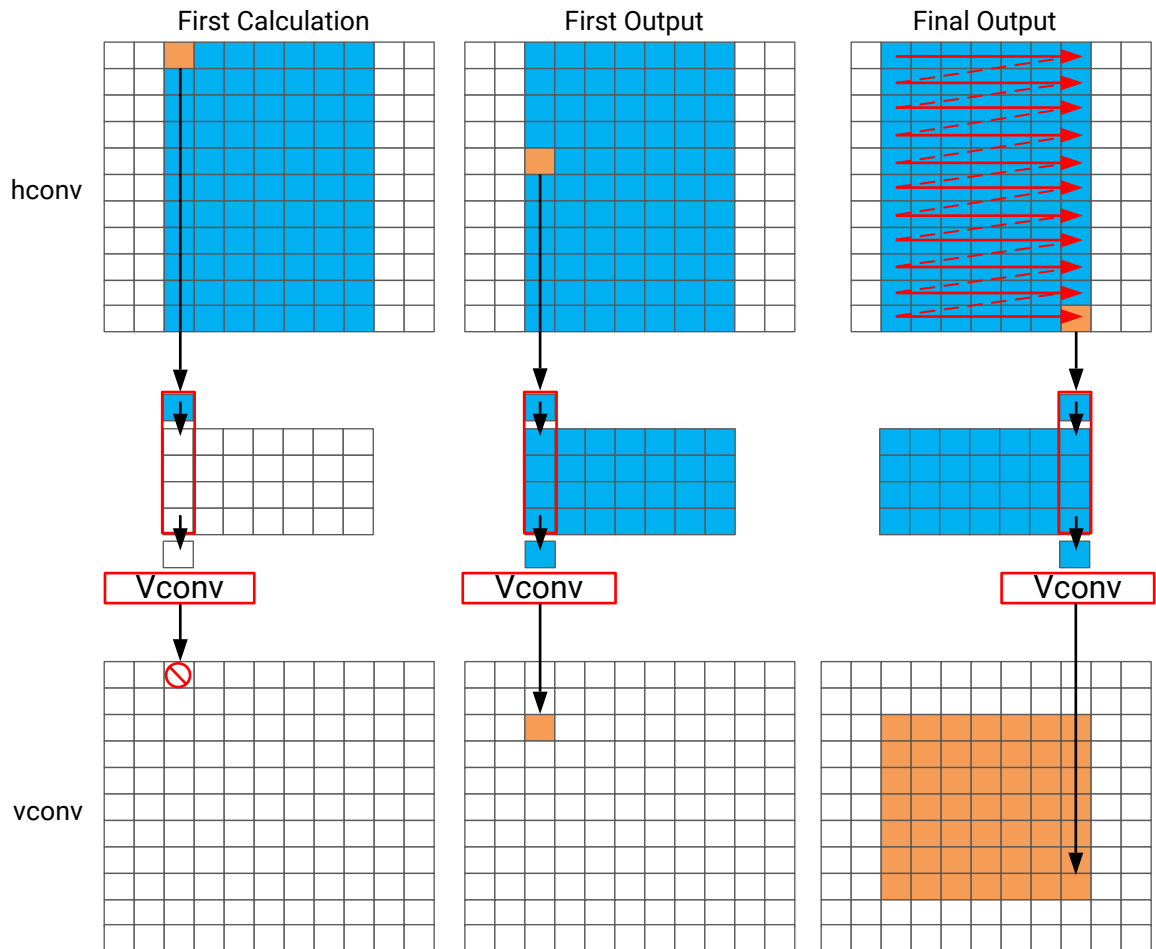
上記のコードでは、一時変数 `out_val` を使用してたたみ込み計算が実行されています。この変数は、計算の実行前に0に設定されるので、前の例で示したように、値をリセットするために2百万クロックサイクルを費やす必要はありません。

プロセス全体を通して、`src` 入力のサンプルはラスターストリーミング方式で処理されます。すべてのサンプルが順番に読み込まれます。タスクからの出力は破棄または使用されますが、タスクは常に計算を実行し続けます。この点が、CPUで実行するために記述されたコードと異なります。

最適な垂直たたみ込み

垂直たたみ込みでは、FPGA向けのストリーミングデータモデルを記述するのが困難です。データには列ごとにアクセスする必要がありますが、画像全体を格納するのは望ましくありません。ソリューションは、次の図に示すようにラインバッファを使用することです。

図 20: ラインバッファ



X14300-011519

先ほどと同様、サンプルはストリーミング方式で読み込まれますが、この場合はローカルバッファの `hconv` から読み込まれます。このアルゴリズムでは、最初のサンプルを処理するのに少なくとも $K-1$ 行のデータが必要です。この前に実行される計算はすべて、条件を使用することにより破棄します。

ラインバッファには、 $K-1$ 行のデータを格納できます。新しいサンプルが読み込まれるたびに、別のサンプルがラインバッファから排出されます。つまり、最新のサンプルが計算に使用されると、そのサンプルがラインバッファに格納され、古いサンプルが排出されます。これにより、 $K-1$ 行のみをキャッシュすればよく、ローカルストレージの使用が最小限に抑えられます。ラインバッファにはローカルで格納するために複数行が必要ですが、たたみ込みのカーネルサイズ K はフルビデオ画像の 1080 行よりもかなり小さくなります。

最初の計算は、 K 行目にある最初のサンプルが読み込まれると実行されます。その後、最後のピクセルが読み込まれるまで値が出力されます。

```
// Vertical convolution
phconv=hconv_buffer; // set/reset pointer to start of buffer
pvconv=vconv_buffer; // set/reset pointer to start of buffer
VConvH:for(int col = 0; col < height; col++) {
    VConvW:for(int row = 0; row < vconv_xlim; row++) {
        #pragma HLS DEPENDENCE variable=linebuf inter false
        #pragma HLS PIPELINE
        T in_val = *phconv++;
```



```
// Reset pixel value on-the-fly - eliminates an O(height*width) loop
T out_val = 0;
VConv:for(int i = 0; i < K; i++) {
    T vwin_val = i < K - 1 ? linebuf[i][row] : in_val;
    out_val += vwin_val * vcoeff[i];
    if (i > 0)
        linebuf[i - 1][row] = vwin_val;
}
if (col >= K - 1) {
    *pvconv++ = out_val;
}
}
```

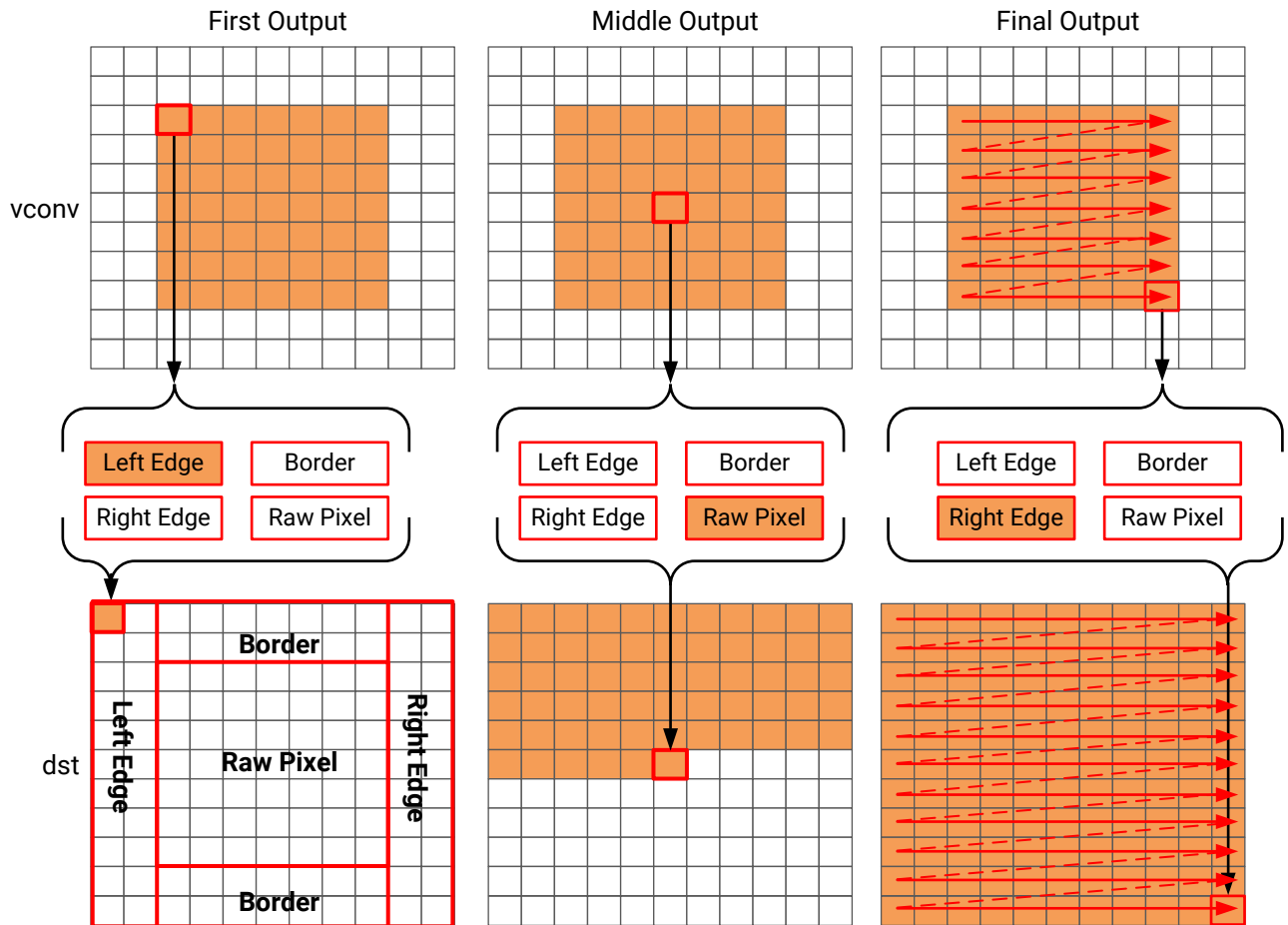
このコードでは、デザインのサンプルがすべてストリーミング方式で処理されます。タスクは、継続して実行されます。再読み出し (または再書き込み) の回数を最小限に抑えるコーディングスタイルに従う場合、データをローカルでキャッシュする必要があります。これは、FPGA をターゲットにする場合の理想的な戦略です。

最適な境界ピクセルたたみ込み

このアルゴリズムの最後には、エッジピクセルを境界領域に複製します。一定したデータフローおよびデータ再利用を実現するため、ローカル キャッシュが使用されます。次の図に、境界サンプルがどのように画像に組み込まれるかを示します。

- 各サンプルが垂直たたみ込みからの `vconv` 出力から読み込まれます。
- サンプルが4つのピクセルタイプのいずれかとしてキャッシュに格納されます。
- サンプルが出力ストリームに書き出されます。

図 21: 境界サンプルの組み込み



X14295-110617

次に、境界ピクセルの位置を決定するコードを示します。

```
// Border pixels
pvconv=vconv buffer; // set/reset pointer to start of buffer
Border:for (int i = 0; i < height; i++) {
    for (int j = 0; j < width; j++) {
        T pix_in, l_edge_pix, r_edge_pix, pix_out;
#pragma HLS PIPELINE
        if (i == 0 || (i > border_width && i < height - border_width)) {
            // read a pixel out of the video stream and cache it for
            // immediate use and later replication purposes
            if (j < width - (K - 1)) {
                pix_in = *pvconv++;
                borderbuf[j] = pix_in;
            }
            if (j == 0) {
                l_edge_pix = pix_in;
            }
            if (j == width - K) {
                r_edge_pix = pix_in;
            }
        }
        // Select output value from the appropriate cache resource
    }
}
```

```

    if (j <= border_width) {
        pix_out = l_edge_pix;
    } else if (j >= width - border_width - 1) {
        pix_out = r_edge_pix;
    } else {
        pix_out = borderbuf[j - border_width];
    }
    *dst++=pix_out;
}
}

```

このコードの明らかな違いは、タスク内に条件文が多く使用されている点です。これにより、タスクがパイプライン処理された後、データが継続的に処理されるようになります。条件文の結果はパイプラインの実行には影響しません。結果は出力値に影響しますが、入力サンプルが使用可能である限り、パイプラインは継続的に処理されます。

最適なデータ アクセス パターン

FPGA で最適なパフォーマンスを得るためのデータ アクセス パターンは、次のとおりです。

- データ入力の読み込みを最小限にします。データがブロックに読み込まれると、多くの並列パスに簡単に供給できますが、ハードウェア関数への入力がパフォーマンスのボトルネックになることがあります。データを読み込んだ後、再利用する必要がある場合は、ローカル キャッシュを使用します。
- 配列、特に大型の配列へのアクセスを最小限に抑えます。配列はブロック RAM にインプリメントされますが、ブロック RAM では I/O ポートと同様ポート数が限られるので、パフォーマンスのボトルネックになることがあります。配列は小型の配列および個別のレジスタに分割できますが、大型の配列を分割すると使用されるレジスタ数が多くなります。小型のローカル キャッシュを使用して累積などの結果を保持してから、最終結果を配列に書き出すようにします。
- パイプライン処理されたタスクであっても、タスクを条件で実行するのではなく、パイプライン処理されたタスク内で条件分岐を実行するようにします。条件文は、パイプラインで個別のパスとしてインプリメントされます。データを1つのタスクから次のタスクに流し、そのタスク内で条件が実行されるようにすると、システムのパフォーマンスが向上します。
- 入力の読み込みと同様にポートはボトルネックになるので、出力の書き出しを最小限にします。追加のアクセスを複製すると、問題がシステム内に先送りされるだけです。

データをストリーミング方式で処理する C コードでは、関数引数に対する読み出し/書き込みを一度のみにすると、FPGA に効率的にインプリメントできるようになります。FPGA が必要なパフォーマンスで動作しない理由をデバッグするよりも、高パフォーマンスの FPGA インプリメンテーションが得られる C のアルゴリズムを設計する方が生産的です。

AXI Performance Monitor を使用したパフォーマンス計測

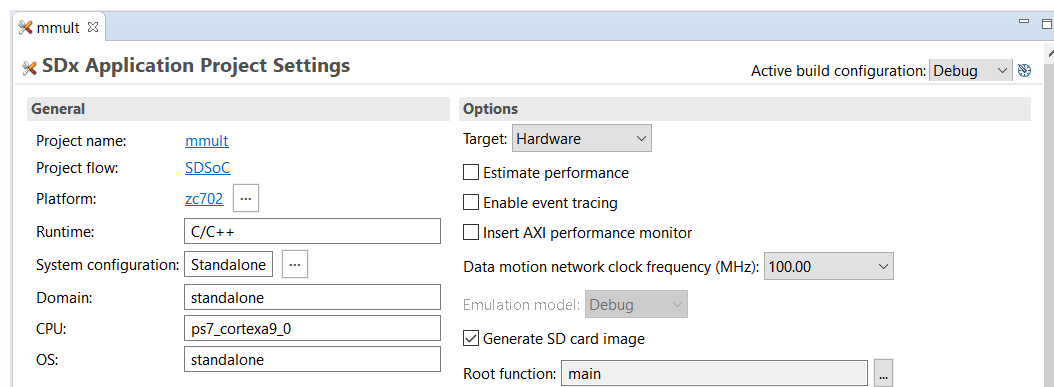
AXI Performance Monitor (APM) モジュールは、PS 内の Arm® コアと PL 内のハードウェアの間のデータ転送に関する基本的な情報を監視するために使用します。読み出し/書き込みトランザクション数、システムのバス上の AXI トランザクションレイテンシなどの統計を収集します。

このセクションでは、システムへの APM コアの挿入方法、計測用に設定されたシステムの監視方法、および生成されたパフォーマンスデータの表示方法を示します。

スタンドアロン プロジェクトの作成と APM のインプリメント

1. SDSoC™ 環境を開き、任意のプラットフォームまたはオペレーティングシステムを使用して新しい SDSoC プロジェクトを作成します。[Matrix Multiplication and Addition] テンプレートを選択します。
2. [Application Project Settings] で [Insert AXI Performance Monitor] をオンにします。

このオプションをオンにしてプロジェクトを作成すると、ハードウェアシステムに APM IP コアが追加されます。APM IP は、プログラマブル ロジックの少量のリソースを使用します。SDSoC 環境により、APM がハードウェア/ソフトウェアインターフェイスポートである汎用 (GP) ポートおよびハイ パフォーマンス (HP) ポートに接続されます。



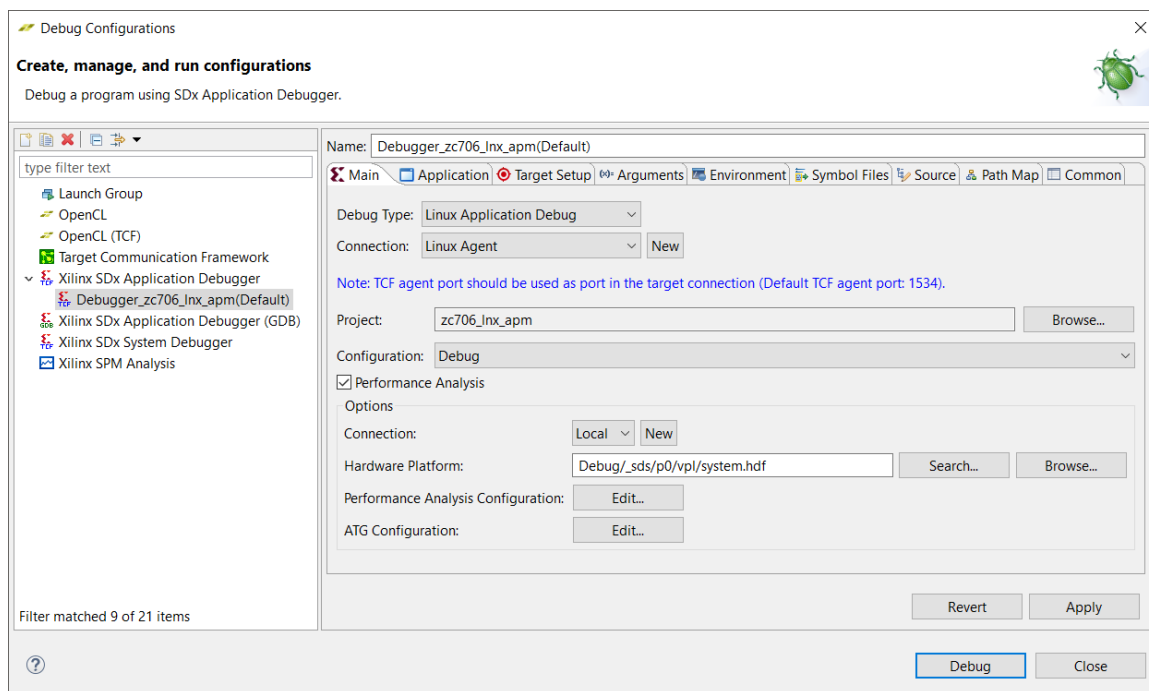
3. `mmult` および `madd` 関数をハードウェアでインプリメントされるように選択します。デバッグ コンフィギュレーション (デフォルト) を使用して、プロジェクトをクリーンアップおよびビルドします。

スタンドアロン システムの監視

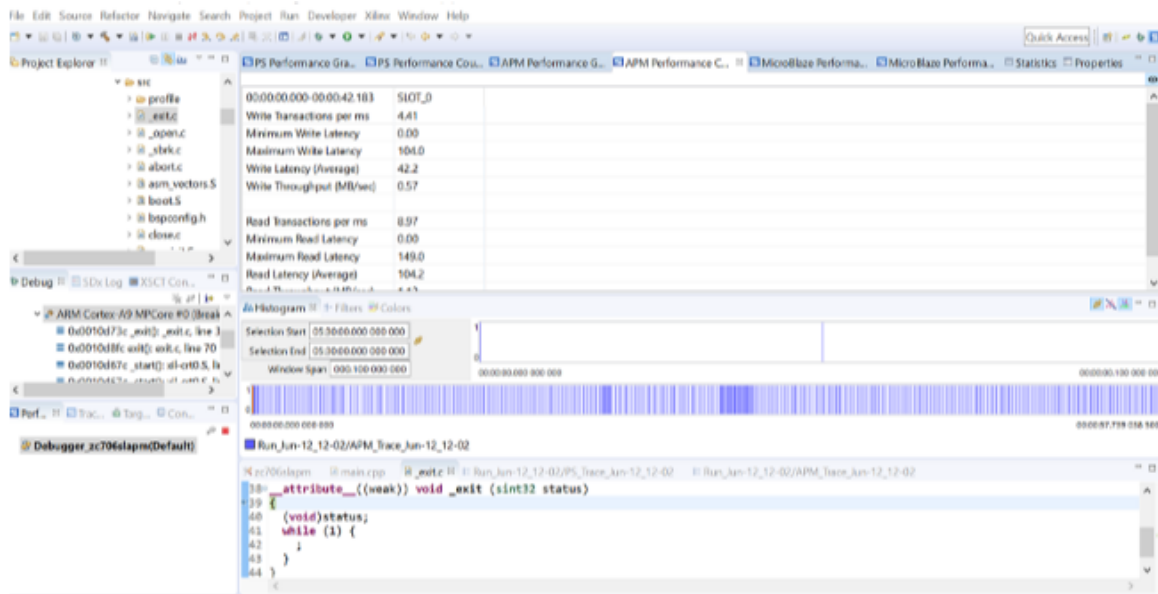
システムを監視するには、次の手順に従います。

1. ビルドが完了したら、ボードをコンピューターに接続してボードの電源を入れます。
2. [Run] → [Debug Configurations] をクリックして、ウィンドウを開きます。
3. デバッグ コンフィギュレーション ツリーで [Xilinx SDx Application Debugger] をクリックします。
4. [New Launch Configuration] をクリックし、新しい SDx 環境アプリケーション デバッガー コンフィギュレーションを作成します。
5. [Debug Type] を [Standalone Application Debug] に設定します。
6. [Connection] を [Local] に設定します。
7. [Main] タブで [Performance Analysis] チェック ボックスをオンにします。

[Performance Analysis] をオンにすると、[Main] タブでパフォーマンス解析オプションが自動的に設定されます。



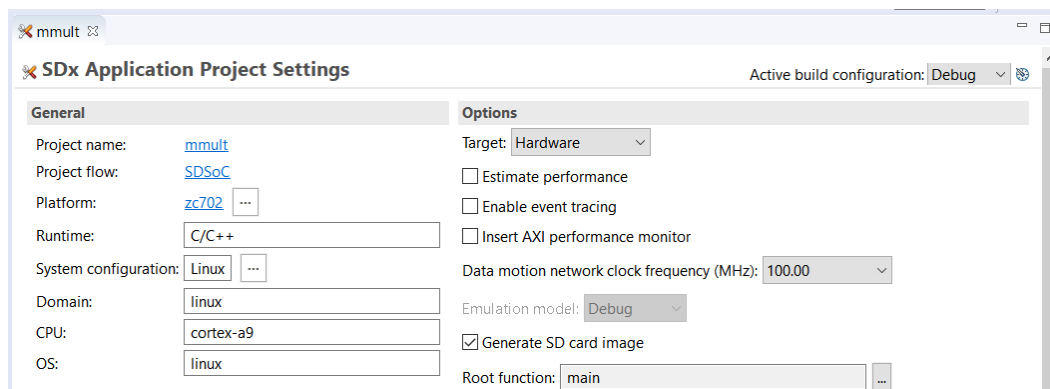
8. [Apply] をクリックし、[Debug] をクリックします。
パースペクティブを切り替えることを尋ねるメッセージが表示されたら、[Yes] をクリックします。
9. [Debug] パースペクティブが開いたら、[Window] → [Perspective] をクリックし、[Open Perspective] ダイアログ ボックスで [Performance Analysis] をオンにして、[OK] をクリックします。
10. アプリケーションの実行を再開するには、[Debug] タブで [Resume] をクリックします。
パースペクティブを切り替えるかどうか尋ねるメッセージが表示されたら、[No] をクリックして [Performance Analysis] パースペクティブにとどまります。



Linux プロジェクトの作成と APM のインプリメント

1. SDSoC 環境を開き、任意のプラットフォームまたはオペレーティングシステムを使用して新しいプロジェクト/ワークスペースを作成します。[Matrix Multiplication and Addition Template] を選択します。
2. [SDx Application Project Settings] で [Insert AXI Performance Monitor] をオンにします。

このオプションをオンにしてプロジェクトを作成すると、ハードウェアシステムに APM IP コアが追加されます。APM IP は、プログラマブル ロジックの少量のリソースを使用します。SDSoC 環境により、APM がハードウェア/ソフトウェアインターフェイスポートである GP ポートおよび HP ポートに接続されます。



3. mmult および madd 関数をハードウェアでインプリメントされるように選択します。
4. デバッグ コンフィギュレーション (デフォルト) を使用してプロジェクトをクリーンアップおよびビルドします。

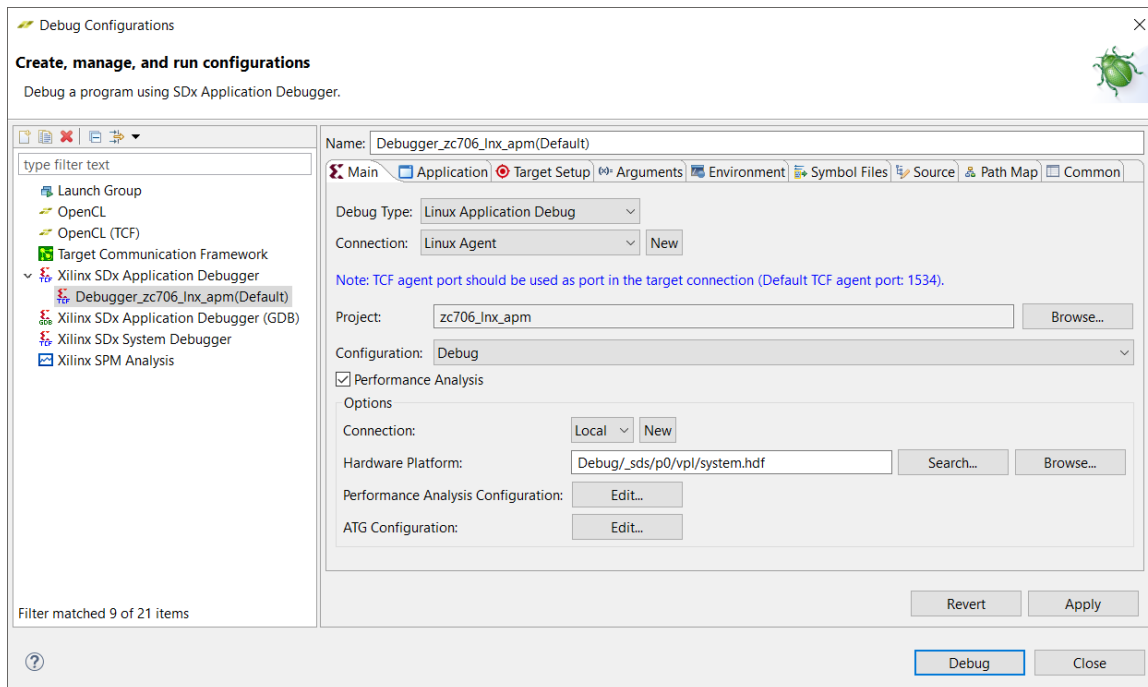
次の図は、APM カウンターを示しています。

00:00:00.000-00:09:59.937		SLOT_0
Write Transactions per ms		9.30
Minimum Write Latency		0.00
Maximum Write Latency		122.0
Write Latency (Average)		52.6
Write Throughput (MB/sec)		1.19
Read Transactions per ms		27.9
Minimum Read Latency		0.00
Maximum Read Latency		149.0
Read Latency (Average)		77.1
Read Throughput (MB/sec)		3.57

Linux システムの監視

1. ビルドが終了したら、sd_card ディレクトリの内容を SD カードにコピーし、ボードで Linux を起動します。
2. UART と JTAG ケーブルの両方を使用してボードをコンピューターに接続します。
3. ボードの IP アドレスを使用して Linux TCF エージェント ターゲット接続を設定します。TCF の詳細は、[SDK ヘルプ](#)を参照してください。
4. [Run] → [Debug Configuration] をクリックし、[Debug Configurations] ダイアログ ボックスを開きます。
5. デバッグ コンフィギュレーション ツリーで [Xilinx SDx Application Debugger] をクリックします。
6. [New Launch Configuration] をクリックし、新しいアプリケーション デバッガー コンフィギュレーションを作成します。
7. [Debug Type] を [Linux Application Debug] に設定します。
8. [Connection] を [Local] に設定します。
9. [Performance Analysis] チェック ボックスをオンにします。

[Performance Analysis] をオンにすると、[Main] タブにパフォーマンス解析オプションが自動的に設定されます。



10. [Apply] をクリックします。

11. [Debug] をクリックします。

パースペクティブを [Debug] に切り替えることを尋ねるメッセージが表示されたら、[Yes] をクリックします。

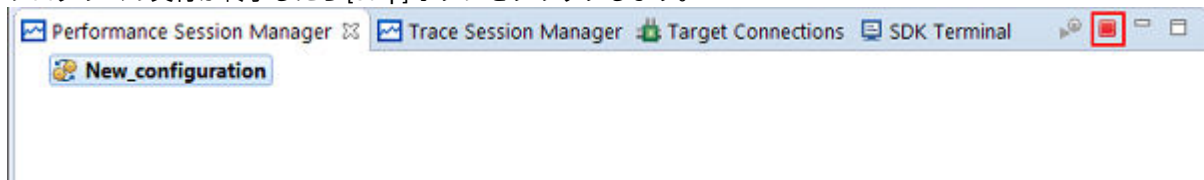
12. [Debug] パースペクティブが表示されたら、[Window] → [Perspective] をクリックし、[Open Perspective] ダイアログボックスで [Performance Analysis] をオンにします。

13. [OK] をクリックします。

14. [Debug] タブをクリックし、[Resume] をクリックしてアプリケーションを再開します。

パースペクティブを切り替えるかどうか尋ねるメッセージが表示されたら、[No] をクリックして [Performance Analysis] パースペクティブにとどまります。

15. プログラムの実行が終了したら [Stop] ボタンをクリックします。



[Confirm Perspective Switch] ダイアログボックスに [Performance Analysis] パースペクティブにとどまるかどうかを確認するメッセージが表示されたら、[No] をクリックします。

16. パースペクティブの下部で解析プロットをスクロールし、異なるパフォーマンス統計を確認します。

17. プロットエリアをクリックすると、パースペクティブの中央に拡大表示されます。

オレンジ色のボックスにより、データの特定の時間範囲に焦点を置くことができます。

パフォーマンスの解析

このシステムでは、APM は PS と PL の間で使用されているポートの 1 つと GP ポートに接続されます。

図 22: [APM Performance Count] の出力

00:00:00.000-00:09:59.937		SLOT_0
Write Transactions per ms		9.30
Minimum Write Latency		0.00
Maximum Write Latency		122.0
Write Latency (Average)		52.6
Write Throughput (MB/sec)		1.19
Read Transactions per ms		27.9
Minimum Read Latency		0.00
Maximum Read Latency		149.0
Read Latency (Average)		77.1
Read Throughput (MB/sec)		3.57

乗算器および加算器アクセラレータ コアはどちらも、データの入出力用にアクセラレータ コヒーレンシ ポート (ACP) に接続されます。

GP ポートは、制御コマンドの発行およびアクセラレータ コアのステータスを取得するためにのみ使用され、データ転送には使用されません。青色のスロット 0 は GP ポートに接続され、緑色のスロット 1 は ACP に接続されています。

注記: ACP ポートは、SDSoC 環境の Zynq UltraScale+ MPSoC デバイスではサポートされません。

APM は、ACP および GP ポートにそれぞれ 1 つずつ、2 つの監視スロットを使用して、プロファイル モードに設定されます。プロファイル モードでは、各スロットにイベント カウント機能が含まれます。読み出しおよび書き込みに対して APM で算出される統計のタイプには、次のものが含まれます。

- トランザクション数: バス上で発生する要求の総数。
- バイト数: 送信されたバイトの総数 (書き込みスループットの算出に使用)。
- レイテンシ: アドレス発行の開始から最後の要素が送信されるまでの時間。

レイテンシおよびバイト カウンターの統計は、スループット (MB/s) を自動的に算出するために APM で使用されます。表示されるレイテンシおよびスループット値は、50 ミリ秒 (ms) 間隔です。

レイテンシおよびスループットの最小、最大、平均も表示されます。

実際の例

この章では、実際の例を使用して次について説明します。

- トップダウン フローおよびボトムアップ フローの両方を使用して最適化する方法
 - トップダウン フローは、Lucas-Kanade (LK) オプティカル フロー アルゴリズムを使用して説明します。
 - ボトムアップ フローは、ステレオ ビジョン ブロック マッチング アルゴリズムを使用して説明します。
- 適用される最適化指示子
- これらの指示子を使用する理由

トップダウン: オプティカル フローのアルゴリズム

Lucas-Kanade (LK) 法は、オプティカル フローの見積もりや、2つの関連画像間におけるピクセル移動の見積もりに広く使用される差分手法です。このシステム例では、関連画像は 1つのビデオ ストリームの現在の画像と前の画像です。LK 法は計算負荷の高いアルゴリズムで、隣接するピクセル領域に適用され、最小二乗の差を使用して一致するピクセルを見つけます。

次のコード例は、このアルゴリズムをインプリメントする方法を示しています。2つの入力ファイルが読み込まれ、`fpga_optflow` 関数で処理され、結果が出力ファイルに書き出されています。

```
int main()
{
    FILE *f;
    pix_t *inY1 = (pix_t *)sds_alloc(HEIGHT*WIDTH);
    yuv_t *inCY1 = (yuv_t *)sds_alloc(HEIGHT*WIDTH*2);
    pix_t *inY2 = (pix_t *)sds_alloc(HEIGHT*WIDTH);
    yuv_t *inCY2 = (yuv_t *)sds_alloc(HEIGHT*WIDTH*2);
    yuv_t *outCY = (yuv_t *)sds_alloc(HEIGHT*WIDTH*2);
    printf("allocated buffers\n");

    f = fopen(FILENAME, "rb");
    if (f == NULL) {
        printf("failed to open file %s\n", FILENAME);
        return -1;
    }
    printf("opened file %s\n", FILENAME);

    read_yuv_frame(inY1, WIDTH, WIDTH, HEIGHT, f);
    printf("read 1st %dx%d frame\n", WIDTH, HEIGHT);
    read_yuv_frame(inY2, WIDTH, WIDTH, HEIGHT, f);
    printf("read 2nd %dx%d frame\n", WIDTH, HEIGHT);

    fclose(f);
    printf("closed file %s\n", FILENAME);
}
```

```

convert_Y8toCY16(inY1, inCY1, HEIGHT*WIDTH);
printf("converted 1st frame to 16bit\n");
convert_Y8toCY16(inY2, inCY2, HEIGHT*WIDTH);
printf("converted 2nd frame to 16bit\n");

fpga_optflow(inCY1, inCY2, outCY, HEIGHT, WIDTH, WIDTH, 10.0);
printf("computed optical flow\n");

// write optical flow data image to disk
write_yuv_file(outCY, WIDTH, WIDTH, HEIGHT, ONAME);

sds_free(inY1);
sds_free(inCY1);
sds_free(inY2);
sds_free(inCY2);
sds_free(outCY);
printf("freed buffers\n");

return 0;
}

```

これは、標準 C/C++ データ型を使用した典型的なトップダウンのデザイン フローです。

次のコード例に示す `fpa_optflow` 関数には、次のサブ関数が含まれています。

- `readMatRows`
- `computeSum`
- `computeFlow`
- `getOutPix`
- `writeMatRows`

```

int fpga_optflow (yuv_t *frame0, yuv_t *frame1, yuv_t *framef, int height,
int width, int stride, float clip_flowmag)
{
#ifdef COMPILEFORSW
    int img_pix_count = height*width;
#else
    int img_pix_count = 10;
#endif

    if (f0Stream == NULL) f0Stream = (pix_t *) malloc(sizeof(pix_t) *
img_pix_count);
    if (f1Stream == NULL) f1Stream = (pix_t *) malloc(sizeof(pix_t) *
img_pix_count);
    if (ffStream == NULL) ffStream = (yuv_t *) malloc(sizeof(yuv_t) *
img_pix_count);

    if (ixix == NULL) ixix = (int *) malloc(sizeof(int) * img_pix_count);
    if (ixiy == NULL) ixiy = (int *) malloc(sizeof(int) * img_pix_count);
    if (iyiy == NULL) iyiy = (int *) malloc(sizeof(int) * img_pix_count);
    if (dix == NULL) dix = (int *) malloc(sizeof(int) * img_pix_count);
    if (diy == NULL) diy = (int *) malloc(sizeof(int) * img_pix_count);

    if (fx == NULL) fx = (float *) malloc(sizeof(float) * img_pix_count);
    if (fy == NULL) fy = (float *) malloc(sizeof(float) * img_pix_count);

    readMatRows (frame0, f0Stream, height, width, stride);
    readMatRows (frame1, f1Stream, height, width, stride);

```

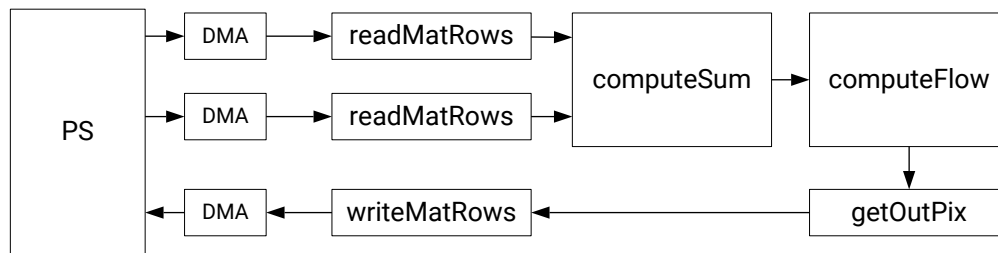
```
computeSum (f0Stream, f1Stream, ixix, ixiy, iyiy, dix, diy, height, width);
computeFlow (ixix, ixiy, iyiy, dix, diy, fx, fy, height, width);
getOutPix (fx, fy, ffStream, height, width, clip_flowmag);

writeMatRows (ffStream, framef, height, width, stride);

return 0;
}
```

この例では、fpga_optflow 内のすべての関数がライブ ビデオ データを処理するので、ハードウェア アクセラレーションを実行し、PS とのデータ転送に DMA を使用すると有益です。5 つの関数すべてをハードウェア関数に指定すると場合、システムのトポロジは次の図のようになります。

図 23: システム トポロジ



X22628-051019

システムは、ハードウェアと、パフォーマンスを詳細に解析するためのイベント トレースにコンパイルできます。

ここでの問題は、終了するまでに時間がかかり過ぎることです (1 フレームに 15 秒)。HD ビデオを処理するには、毎秒 60 フレームまたは 16.7 ms ごとに 1 フレーム処理される必要があります。次に示す最適化指示子を使用すると、システムがターゲット パフォーマンスを満たすようになります。

オプティカル フローのメモリ アクセス最適化

設計手法の最初のタスクはデータ転送の最適化です。この場合、システムはストリーミング ビデオを処理するので (各サンプルを連続処理)、メモリ転送最適化では、すべてのアクセスがシーケンシャルに実行されることが SDSoC™ 環境で認識されるようにします。

これには、関数シングネチャの前に SDSoC プラグマを追加して、すべての関数に適用されるようにします。

```
#pragma SDS data access_pattern(matB:SEQUENTIAL, pixStream:SEQUENTIAL)
#pragma SDS data mem_attribute(matB:PHYSICAL_CONTIGUOUS)
#pragma SDS data copy(matB[0:stride*height])
void readMatRows (yuv_t *matB, pix_t* pixStream,
                  int height, int width, int stride);

#pragma SDS data access_pattern(pixStream:SEQUENTIAL, dst:SEQUENTIAL)
#pragma SDS data mem_attribute(dst:PHYSICAL_CONTIGUOUS)
#pragma SDS data copy(dst[0:stride*height])
void writeMatRows (yuv_t* pixStream, yuv_t *dst,
                  int height, int width, int stride);

#pragma SDS data access_pattern(f0Stream:SEQUENTIAL, f1Stream:SEQUENTIAL)
#pragma SDS data access_pattern(ixix_out:SEQUENTIAL, ixiy_out:SEQUENTIAL,
                                iyiy_out:SEQUENTIAL)
#pragma SDS data access_pattern(dix_out:SEQUENTIAL, diy_out:SEQUENTIAL)
void computeSum(pix_t* f0Stream, pix_t* f1Stream,
                int* ixix_out, int* ixiy_out, int* iyiy_out,
```

```
int* dix_out, int* diy_out,
int height, int width);

#pragma SDS data access_pattern(ixix:SEQUENTIAL, ixiy:SEQUENTIAL,
iyiy:SEQUENTIAL)
#pragma SDS data access_pattern(dix:SEQUENTIAL, diy:SEQUENTIAL)
#pragma SDS data access_pattern(fx_out:SEQUENTIAL, fy_out:SEQUENTIAL)
void computeFlow(int* ixix, int* ixiy, int* iyiy,
int* dix, int* diy,
float* fx_out, float* fy_out,
int height, int width);

#pragma SDS data access_pattern(fx:SEQUENTIAL, fy:SEQUENTIAL,
out_pix:SEQUENTIAL)
void getOutPix (float* fx, float* fy, yuv_t* out_pix,
int height, int width, float clip_flowmag);
```

プロセッサへのインターフェイスである `readMatRows` および `writeMatRows` 関数引数では、メモリ転送は物理的に隣接したメモリからシーケンシャルにアクセスされるように指定され、データは単にアクセラレータからアクセスされるのではなく、ハードウェア関数にコピーおよびハードウェア関数からコピーされます。これにより、データが効率的にコピーされます。次のオプションが使用できます。

- **Sequential:** データは処理されるのと同じ順序でシーケンシャルに転送されます。このタイプの転送では、高速データ処理レートのハードウェア オーバーヘッドは最小限であり、エリア効率の高いデータムーバーが使用されます。
- **Contiguous:** データは隣接したメモリからアクセスされるので、データ転送レートにスキッター ギャザー オーバーヘッドはなく、効率的で高速のハードウェア データムーバーが使用されます。この指示子は `main()` 関数内の関連する `scs_alloc` ライブラリ呼び出しでサポートされており、これらの引数のデータが隣接するメモリに格納されます。
- **Copy:** データはアクセラレータにコピーおよびアクセラレータからコピーされるので、CPU または DDR メモリへのデータアクセスは必要ありません。ポインターが使用されるので、コピーするデータのサイズが指定されます。

残りのハードウェア関数では、データ転送はシーケンシャルと指定されるので、プログラマブル ロジック (PL) ファブリックの関数が最も効率的なハードウェアを使用して接続されます。

オプティカル フローのハードウェア関数最適化

最高レベルのパフォーマンスで実行するため、ハードウェア関数にも最適化指示子が必要です。これらは既にデザインサンプルに含まれています。詳細は、[ハードウェア関数の最適化手法](#)を参照してください。このデザイン例のハードウェア関数のほとんどは、`getOutPix` 関数と同様に主に PIPELINE 指示子を使用して最適化されています。

`getOutPix` 関数の特徴は次のとおりです。

- サブ関数に **INLINE** 最適化が適用され、これらの関数のロジックが上位の関数に統合されます。この統合は小型の関数では自動的に実行されますが、この指示子を使用するとサブ関数が常にインライン展開されるので、サブ関数をパイプライン処理する必要はありません。
- `getOutPix` 関数内のループは、データを各ピクセル レベルで処理するループであり、クロックごとに 1 ピクセル処理するように PIPELINE 指示子で最適化されます。

```
pix_t getLuma (float fx, float fy, float clip_flowmag)
{
#pragma HLS inline
float rad = sqrtf (fx*fx + fy*fy);

if (rad > clip_flowmag) rad = clip_flowmag; // clamp to MAX
rad /= clip_flowmag; // convert 0..MAX to 0.0..1.0
```

```

    pix_t pix = (pix_t) (255.0f * rad);

    return pix;
}

pix_t getChroma (float f, float clip_flowmag)
{
#pragma HLS inline
    if (f > clip_flowmag) f = clip_flowmag; // clamp big positive f to MAX
    if (f < (-clip_flowmag)) f = -clip_flowmag; // clamp big negative f to -MAX
    f /= clip_flowmag; // convert -MAX..MAX to -1.0..1.0
    pix_t pix = (pix_t) (127.0f * f + 128.0f); // convert -1.0..1.0 to
    -127..127 to 1..255

    return pix;
}

void getOutPix (float* fx,
               float* fy,
               yuv_t* out_pix,
               int height, int width, float clip_flowmag)
{
    int pix_index = 0;
    for (int r = 0; r < height; r++) {
        for (int c = 0; c < width; c++) {
            #pragma HLS PIPELINE
            float fx_ = fx[pix_index];
            float fy_ = fy[pix_index];

            pix_t outLuma = getLuma (fx_, fy_, clip_flowmag);
            pix_t outChroma = (c&1)? getChroma (fy_, clip_flowmag) : getChroma
(fx_, clip_flowmag);
            yuv_t yuvpix;

            yuvpix = ((yuv_t)outChroma << 8) | outLuma;

            out_pix[pix_index++] = yuvpix;
        }
    }
}

```

ARRAY_PARTITION および DEPENDENCE 指示子の例は、computeSum 関数を参照してください。この関数では、img1Win 配列に ARRAY_PARTITION 指示子が使用されています。img1Win は配列なので、次のコードのサマリに示すように、デフォルトで最大 2 つのポートを持つブロック RAM にインプリメントされます。

- img1Win: クロックサイクルごとに 1 サンプルを処理するようにパイプライン処理された for ループで使用されます。
- img1Win: for ループ内で 8 + (KMEDP1-1) + (KMEDP1-1) 回読み出されます。
- img1Win: for ループ内で (KMEDP1-1) + (KMEDP1-1) 回書き込まれます。

```

void computeSum(pix_t* f0Stream,
               pix_t* f1Stream,
               int* ixix_out,
               int* ixiy_out,
               int* iyiy_out,
               int* dix_out,
               int* diy_out)
{
    static pix_t img1Win [2 * KMEDP1], img2Win [1 * KMEDP1];

```



```
#pragma HLS ARRAY_PARTITION variable=img1Win complete dim=0
...
for (int r = 0; r < MAX_HEIGHT; r++) {
    for (int c = 0; c < MAX_WIDTH; c++) {
        #pragma HLS PIPELINE
        ...
        int cIxTopR = (img1Col_ [wrt] - img1Win [wrt*2 + 2-2]) /2 ;
        int cIyTopR = (img1Win [ (wrt+1)*2 + 2-1] - img1Win [ (wrt-1)*2
+ 2-1]) /2;
        int delTopR = img1Win [wrt*2 + 2-1] - img2Win [wrt*1 + 1-1];
        ...
        int cIxBotR = (img1Col_ [wrb] - img1Win [wrb*2 + 2-2]) /2 ;
        int cIyBotR = (img1Win [ (wrb+1)*2 + 2-1] - img1Win [ (wrb-1)*2
+ 2-1]) /2;
        int delBotR = img1Win [wrb*2 + 2-1] - img2Win [wrb*1 + 1-1];
        ...
        // shift windows
        for (int i = 0; i < KMEDP1; i++) {
            img1Win [i * 2] = img1Win [i * 2 + 1];
        }
        for (int i=0; i < KMEDP1; ++i) {
            img1Win [i*2 + 1] = img1Col_ [i];
            ...
        }
        ...
    } // for c
} // for r
...
}
```

ブロック RAM ではクロック サイクルごとに 2 アクセスまでしかサポートされないの、これらのアクセスすべてを 1 クロックで完了することはできません。前述のように、ARRAY_PARTITION 指示子を使用して配列を小型のブロックに分割します。この例では、complete オプションを使用して個別の要素に分割しています。これにより、配列のすべての要素に同時に並列アクセスでき、for ループでクロック サイクルごとにデータが処理されるようになります。

最後に確認する指示子は、DEPENDENCE です。csIxix 配列に、DEPENDENCE 指示子が適用されています。次のコード例では、この配列が読み出された後別のインデックスを使用して書き込まれ、パイプライン処理されたループ内でこれらの読み出しおよび書き込みが実行されます。

```
void computeSum(pix_t* f0Stream,
                pix_t* f1Stream,
                int* ixix_out,
                int* ixiy_out,
                int* iyiy_out,
                int* dix_out,
                int* diy_out)
{
    ...
    static int csIxix [MAX_WIDTH], csIxiy [MAX_WIDTH], csIyiy [MAX_WIDTH],
    csDix [MAX_WIDTH], csDiy [MAX_WIDTH];
    ...
    #pragma HLS DEPENDENCE variable=csIxix inter WAR false
    ...
    int zIdx= - (KMED-2);
    int nIdx = zIdx + KMED-2;

    for (int r = 0; r < MAX_HEIGHT; r++) {
        for (int c = 0; c < MAX_WIDTH; c++) {
            #pragma HLS PIPELINE
            ...
            if (zIdx >= 0) {
                csIxixL = csIxix [zIdx];
            }
        }
    }
}
```

```

    ...
}
...
csIxix [nIdx] = csIxixR;
...
zIdx++;
if (zIdx == MAX_WIDTH) zIdx = 0;
nIdx++;
if (nIdx == MAX_WIDTH) nIdx = 0;
...
} // for c
} // for r
...
}

```

ループがハードウェアでパイプライン処理されると、配列へのアクセスは時間的にオーバーラップします。コンパイラで配列へのすべてのアクセスが解析され、 N 回目の書き込みにより $N + K$ 回目のデータが上書きされて値が変わってしまうような条件があると、警告メッセージが表示されます。この警告により、 $II = 1$ のパイプラインはインプリメントされません。

次に、インデックス 0～9 の配列に対して読み出しと書き込みをループで複数回実行する例を示します。上記のコードと同様に、読み出しと書き込みのアドレス カウンターが異なり、ループのすべての繰り返しを終了する前に 0 に戻る場合があります。これらの演算は、パイプライン処理されたインプリメンテーションと同様、時間的にオーバーラップします。

```

R4-----W8
R5-----W9
R6-----W0
R7-----W1
R8-----W2
R9-----W3
R0-----W4
R1-----W5
R2-----W6

```

各繰り返しが次の繰り返しの開始前に終了するシーケンシャル C コードでは、読み出しと書き込みの順序は明確ですが、同時実行されるハードウェアパイプラインではアクセスが重なり、異なる順序で実行されます。上記に示すように、インデックス 8 (R8) からの読み出しが、R8 の前に実行されるはずであったインデックス 8 (W8) への書き込みよりも前に実行される可能性があります。

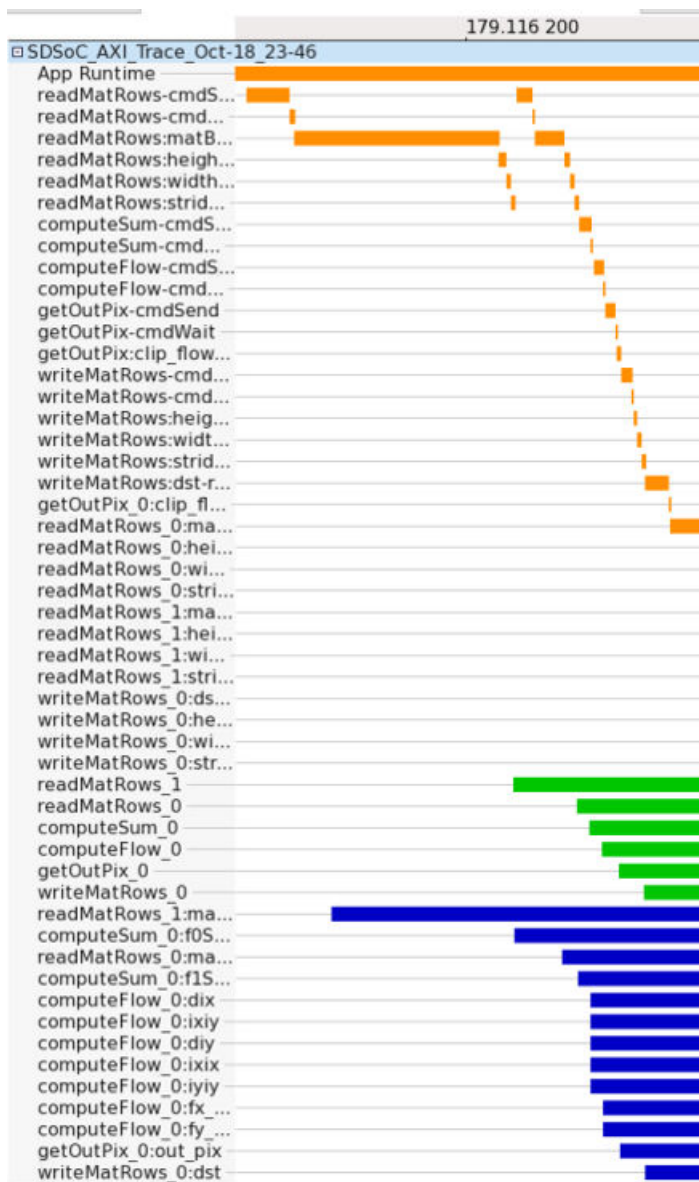
コンパイラでこの状況に関する警告メッセージが表示され、DEPENDENCE 指示子が `false` に設定されていて WAR (Write-After-Read) に依存性はないことが示されているので、 $II=1$ で実行されるパイプライン処理ハードウェアが作成されます。

DEPENDENCE 指示子は通常、コードのスタティック解析では認識されない関数外部のアルゴリズム動作と条件をコンパイラに通知するために使用します。DEPENDENCE 指示子が正しく設定されていないと、ハードウェアの結果がソフトウェアでの結果と異なる場合に、ハードウェアエミュレーションで問題が発生します。

オプティカル フローの結果

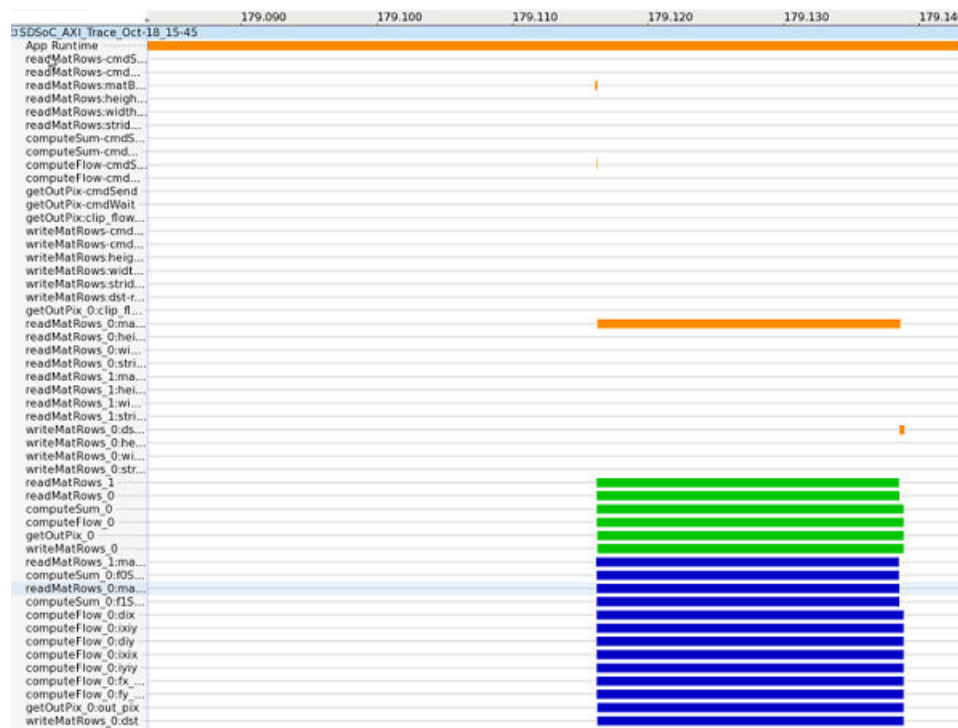
データ転送とハードウェア関数の両方を最適化したら、ハードウェア関数をコンパイルし直して、イベントトレースを使用してパフォーマンスを解析します。次の図に、イベントトレースの開始を示します。パイプライン処理されたハードウェア関数は前の関数が終了するまで実行されません。各ハードウェア関数はデータが使用可能になるとデータを処理し始めます。

図 24: トレース結果



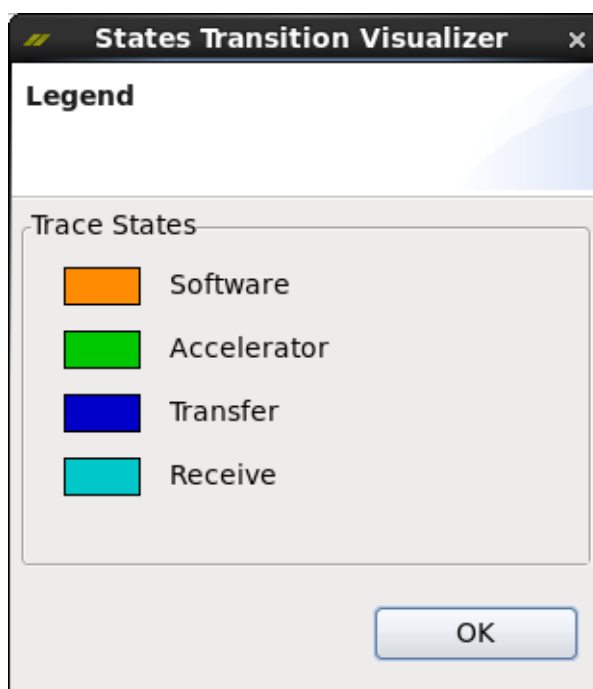
次に示すイベントトレースの全体図では、システムのパフォーマンスができるだけ高くなるように、ハードウェア関数とデータ転送のすべてが並列実行されていることが示されています。

図 25: イベントトレース



レーンの 1 つにカーソルを置くと、アクセラレータのランタイムの期間がポップアップで表示されます。実行時間は 15.5 ms 未満であり、毎秒 60 フレームを達成するための要件である 16.8 ms を満たしています。[AXI State View] ビューのトレースは、次のように色分けされます。

図 26: [AXI State View] ビューの凡例



- Software: Arm® プロセッサ コアで実行。
- アクセラレータ: アクセラレータで実行。
- Transfer: Arm コアから送信されるデータ。
- Receive: Arm プロセッサ コアで受信されるデータ。

ボトムアップ: ステレオ ビジョン アルゴリズム

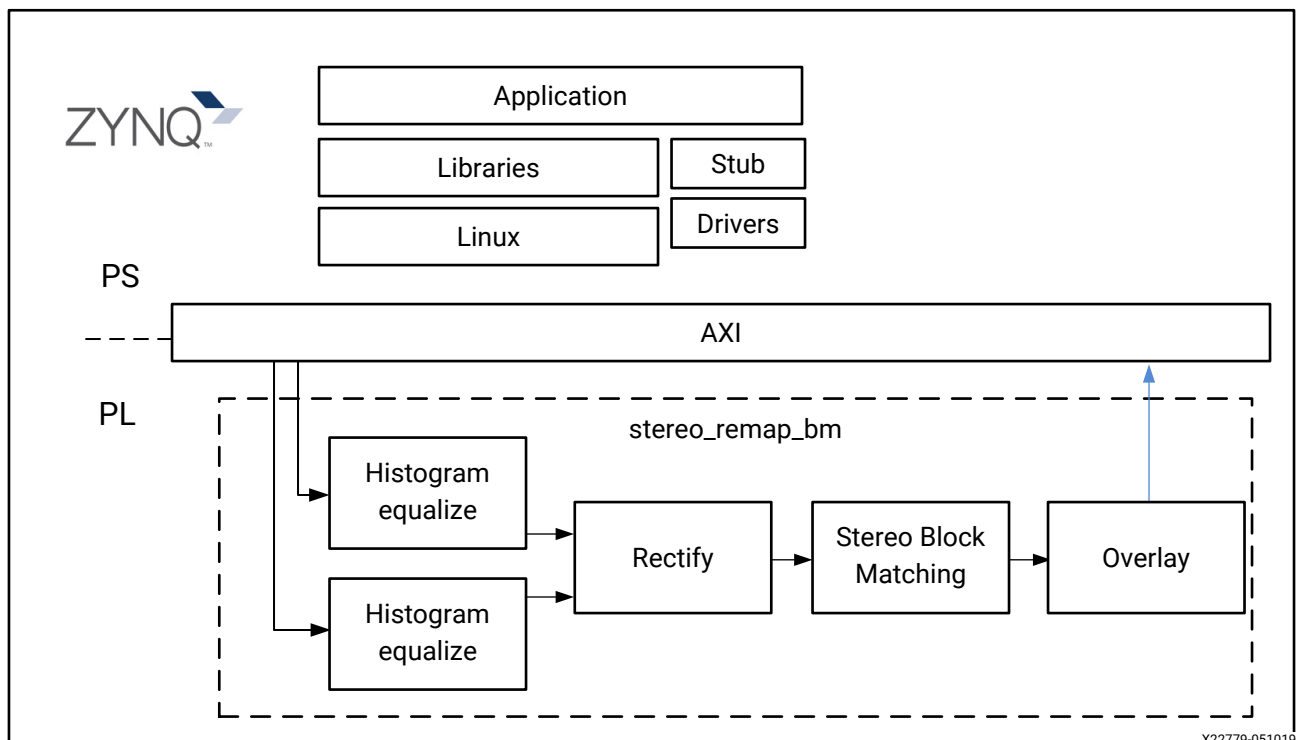
ステレオ ビジョン アルゴリズムは、水平方向にずらして配置された 2 つのカメラからの画像を使用して、人間の視覚と同様、2 つの視点からの 2 つの映像を表示します。映像からの相対的な深さの情報は、2 つの画像を比較して視差マップを作成することで取得できます。視差マップでは、水平座標のオブジェクトどうしの相対的な位置は値が対応するピクセル位置の映像の深さに反比例するようエンコードされます。

ボトムアップ手法では、Vivado® HLS (高位合成) ツールで既に合成済みの完全に最適化されたデザインから開始し、最適化済みのハードウェア関数を SDSoC 環境でソフトウェアと統合します。

このフローを使用すると、HLS ツールを熟知したハードウェア設計者が高度な HLS 機能を使用してハードウェア関数全体をビルドして最適化した後、それをソフトウェア プログラマが利用できます。

次のセクションでは、ステレオ ビジョンのデザイン例を使用して、HLS ツールで最適化されたハードウェア関数から開始し、SDSoC 環境を使用して、ボードで実行されるハードウェアとソフトウェアを完全なシステムに統合するアプリケーションをビルドする手順を説明します。次の図に、既存のハードウェア関数 `stereo_remap_bm` を SDSoC 環境に組み込んだ最終的なシステムを示します。

図 27: システムのブロック図



ボトムアップフローでは、この資料で説明する SDSoC 環境の一般的な最適化手法を逆にします。つまり、最適化済みのハードウェア関数から始めて、それを SDSoC 環境に組み込んだ後、データ転送を最適化します。

ステレオ ビジョンのハードウェア関数の最適化

次のコード例に、既存のハードウェア関数 `stereo_remap_bm` と最適化プラグマを示します。最適化指示子を確認する前に、この関数の詳細について説明します。

- `readLRinput`、`writeDispOut`、`writeDispOut` というサブ関数が含まれ、これらも最適化されています。
- Vivado HLS ツールのビデオ ライブラリ `hls_video.h` から `namespace hls` という接頭辞の付いた最適化済みの関数が使用されています。これらのサブ関数では、独自のデータ型 `MAT` が使用されています。

```
#include "hls_video.h"
#include "top.h"
#include "transform.h"

void readLRinput (yuv_t *inLR,
                 hls::Mat<IMG_HEIGHT, IMG_WIDTH, HLS_8UC1>& img_l,
                 hls::Mat<IMG_HEIGHT, IMG_WIDTH, HLS_8UC1>& img_r,
                 int height, int dual_width, int width, int stride)
{
    for (int i=0; i < height; ++i) {
        #pragma HLS loop_tripcount min=1080 max=1080 avg=1080
        for (int j=0; j < stride; ++j) {
            #pragma HLS loop_tripcount min=1920 max=1920 avg=1920
            #pragma HLS PIPELINE
            yuv_t tmpData = inLR [i*stride + j];          // from yuv_t array: consume
height*stride
            if (j < width)
                img_l.write (tmpData & 0x00FF);          // to HLS_8UC1 stream
            else if (j < dual_width)
                img_r.write (tmpData & 0x00FF);          // to HLS_8UC1 stream
        }
    }
}

void writeDispOut (hls::Mat<IMG_HEIGHT, IMG_WIDTH, HLS_8UC1>& img_d,
                  yuv_t *dst,
                  int height, int width, int stride)
{
    pix_t tmpOut;
    yuv_t outData;

    for (int i=0; i < height; ++i) {
        #pragma HLS loop_tripcount min=1080 max=1080 avg=1080
        for (int j=0; j < stride; ++j) {
            #pragma HLS loop_tripcount min=960 max=960 avg=960
            #pragma HLS PIPELINE
            if (j < width) {
                tmpOut = img_d.read().val[0];
                outData = ((yuv_t) 0x8000) | ((yuv_t) tmpOut);
                dst [i*stride + j] = outData;
            }
            else {
                outData = (yuv_t) 0x8000;
                dst [i*stride + j] = outData;
            }
        }
    }
}
```

```

}

namespace hls {
void SaveAsGray(
    Mat<IMG_HEIGHT, IMG_WIDTH, HLS_16SC1>& src,
    Mat<IMG_HEIGHT, IMG_WIDTH, HLS_8UC1>& dst)
{
    int height = src.rows;
    int width = src.cols;
    for (int i = 0; i < height; i++) {
#pragma HLS loop_tripcount min=1080 max=1080 avg=1080
        for (int j = 0; j < width; j++) {
#pragma HLS loop_tripcount min=960 max=960 avg=960
#pragma HLS pipeline II=1
            Scalar<1, short> s;
            Scalar<1, unsigned char> d;
            src >> s;

            short uval = (short) (abs ((int)s.val[0]));

            // Scale to avoid overflow. The right scaling here for a
            // good picture depends on the NDISP parameter during
            // block matching.
            d.val[0] = (unsigned char)(uval >> 1);
            //d.val[0] = (unsigned char)(s.val[0] >> 1);
            dst << d;
        }
    }
} // namespace hls

int stereo_remap_bm_new(
    yuv_t *img_data_lr,
    yuv_t *img_data_disp,
    hls::Window<3, 3, param_T > &lcameraMA_l,
    hls::Window<3, 3, param_T > &lcameraMA_r,
    hls::Window<3, 3, param_T > &lirA_l,
    hls::Window<3, 3, param_T > &lirA_r,
    param_T (&ldistC_l)[5],
    param_T (&ldistC_r)[5],
    int height, // 1080
    int dual_width, // 1920 (two 960x1080 images side by side)
    int stride_in, // 1920 (two 960x1080 images side by side)
    int stride_out) // 960
{
    int width = dual_width/2; // 960

#pragma HLS DATAFLOW

    hls::Mat<IMG_HEIGHT, IMG_WIDTH, HLS_8UC1> img_l(height, width);
    hls::Mat<IMG_HEIGHT, IMG_WIDTH, HLS_8UC1> img_r(height, width);

    hls::Mat<IMG_HEIGHT, IMG_WIDTH, HLS_8UC1> img_l_remap(height,
width); // remapped left image
    hls::Mat<IMG_HEIGHT, IMG_WIDTH, HLS_8UC1> img_r_remap(height,
width); // remapped left image
    hls::Mat<IMG_HEIGHT, IMG_WIDTH, HLS_8UC1> img_d(height, width);

    hls::Mat<IMG_HEIGHT, IMG_WIDTH, HLS_16SC2> map1_l(height, width);
    hls::Mat<IMG_HEIGHT, IMG_WIDTH, HLS_16SC2> map1_r(height, width);
    hls::Mat<IMG_HEIGHT, IMG_WIDTH, HLS_16UC2> map2_l(height, width);
    hls::Mat<IMG_HEIGHT, IMG_WIDTH, HLS_16UC2> map2_r(height, width);

```



```

hls::Mat<IMG_HEIGHT, IMG_WIDTH, HLS_16SC1> img_disp(height, width);
hls::StereoBMState<15, 32, 32> state;

// ddr -> kernel streams: extract luma from left and right yuv images
// store it in single channel HLS_8UC1 left and right Mat's
readLInput (img_data_lr, img_l, img_r, height, dual_width, width,
stride_in);

////////// remap left and right images, all types are
HLS_8UC1 //////////
hls::InitUndistortRectifyMapInverse(lcameraMA_l, ldistC_l, lirA_l,
map1_l, map2_l);
hls::Remap<8>(img_l, img_l_remap, map1_l, map2_l, HLS_INTER_LINEAR);
hls::InitUndistortRectifyMapInverse(lcameraMA_r, ldistC_r, lirA_r,
map1_r, map2_r);
hls::Remap<8>(img_r, img_r_remap, map1_r, map2_r, HLS_INTER_LINEAR);

////////// find disparity of remapped images //////////
hls::FindStereoCorrespondenceBM(img_l_remap, img_r_remap, img_disp,
state);
hls::SaveAsGray(img_disp, img_d);

// kernel stream -> ddr : output single wide
writeDispOut (img_d, img_data_disp, height, width, stride_out);

return 0;
}

int stereo_remap_bm(
    yuv_t *img_data_lr,
    yuv_t *img_data_disp,
    int height,          // 1080
    int dual_width,      // 1920 (two 960x1080 images side by side)
    int stride_in,       // 1920 (two 960x1080 images side by side)
    int stride_out)      // 960
{
//1920*1080
//#pragma HLS interface m_axi port=img_data_lr depth=2073600
//#pragma HLS interface m_axi port=img_data_disp depth=2073600

hls::Window<3, 3, param_T > lcameraMA_l;
hls::Window<3, 3, param_T > lcameraMA_r;
hls::Window<3, 3, param_T > lirA_l;
hls::Window<3, 3, param_T > lirA_r;
param_T ldistC_l[5];
param_T ldistC_r[5];

for (int i=0; i<3; i++) {
    for (int j=0; j<3; j++) {
        lcameraMA_l.val[i][j]=cameraMA_l[i*3+j];
        lcameraMA_r.val[i][j]=cameraMA_r[i*3+j];
        lirA_l.val[i][j]=lirA_l[i*3+j];
        lirA_r.val[i][j]=lirA_r[i*3+j];
    }
}
for (int i=0; i<5; i++) {
    ldistC_l[i] = distC_l[i];
    ldistC_r[i] = distC_r[i];
}

int ret = stereo_remap_bm_new(img_data_lr,
                               img_data_disp,
                               lcameraMA_l,

```

```

        lcameraMA_r,
        lirA_l,
        lirA_r,
        ldistC_l,
        ldistC_r,
        height,
        dual_width,
        stride_in,
        stride_out);

    return ret;
}

```

ハードウェア関数の最適化手法で説明したように、主に使用される最適化指示子は PIPELINE および DATAFLOW です。このほか、LOOP_TRIPCOUNT 指示子も使用されます。

データフレームを処理するハードウェア関数の最適化に関する推奨事項に基づいて、PIPELINE 指示子はサンプルレベル(この場合、ピクセルレベル)でデータを処理する for ループすべてに適用されています。これにより、ハードウェアパイプライン処理が使用され、最高パフォーマンスのデザインが達成されます。

LOOP_TRIPCOUNT 指示子が for ループに使用されています。この for ループではループインデックスの上限が変数で定義されているので、正確な値はコンパイル時には不明です。トリップカウントまたはループ繰り返し数に見積もり値を使用すると、HLS で生成されるレポートに、レイテンシおよび開始間隔 (II) の見積もり値が含まれるようになります。この指示子は、作成されるハードウェアには影響せず、レポートにのみ影響します。

最上位関数 `stereo_remap_bm` には、最適化されたサブ関数と HLS ビデオ ライブラリ (`hls_video.h`) からの多くの関数が含まれます。HLS のライブラリ関数の詳細は、『Vivado Design Suite ユーザー ガイド: 高位合成』(UG902) を参照してください。

HLS ビデオ ライブラリの関数は既に最適化済みであり、できるだけ高パフォーマンスでインプリメントされるようにするためのすべての最適化指示子を含みます。最上位関数には最適化されたサブ関数が含まれているので、DATAFLOW 指示子を使用して、各サブ関数がデータが使用可能になった直後にハードウェアで実行を開始する必要があるだけです。

```

int stereo_remap_bm(..) {

#pragma HLS DATAFLOW
    readLRinput (img_data_lr, img_l, img_r, height, dual_width, width, stride
        hls::InitUndistortRectifyMapInverse(lcameraMA_l, ldistC_l, lirA_l, map1_l,
        map2_l);
    hls::Remap<8>(img_l, img_l_remap, map1_l, map2_l, HLS_INTER_LINEAR);
    hls::InitUndistortRectifyMapInverse(lcameraMA_r, ldistC_r, lirA_r, map1_r,
        map2_r);
    hls::Remap<8>(img_r, img_r_remap, map1_r, map2_r, HLS_INTER_LINEAR);
    hls::Duplicate(img_l_remap, img_l_remap_bm, img_l_remap_pt);
    hls::FindStereoCorrespondenceBM(img_l_remap_bm, img_r_remap, img_disp,
        state);
    hls::SaveAsGray(img_disp, img_d);
    writeDispOut (img_l_remap_pt, img_d, img_data_disp, height, dual_width,
        width, stride);

}

```

SDSoC™ 環境では、データが使用可能になるとすぐに自動的に 1 つのハードウェア関数から次の関数に渡されるので、通常 DATAFLOW 最適化は必要ありません。ただし、この例では `stereo_remap_bm` 内の関数で HLS のデータ型 `hls::stream` が使用されています。このデータ型は、Arm® プロセッサでコンパイルできず、SDSoC 環境のハードウェア関数インターフェイスでは使用できません。そのため、最上位ハードウェア関数は `stereo_remap_bm` である必要があり、サブ関数間で高パフォーマンスの転送が達成できるように DATAFLOW 指示子が使用されています。このようになっていない場合は、DATAFLOW を削除して、`stereo_remap_bm` 内の各サブ関数をハードウェア関数として指定できます。

この例のハードウェア関数では、HLS データ型 `hls::stream` に基づいたデータ型 `Mat` が使用されています。
`hls::stream` データ型は、シーケンシャルにのみアクセスできるので、データは入力されると出力されます。

- ソフトウェアシミュレーションでは、`hls::stream` データ型のサイズは無限です。
- ハードウェアでは、`hls::stream` データ型が1つのレジスタとしてインプリメントされ、ストリーミングデータは前のデータが上書きされる前に消費されると想定されるので、一度に1つのデータ値しか格納できません。

最上位関数 `stereo_remap_bm` をハードウェア関数として指定すると、ソフトウェア環境でこれらのハードウェア型の影響を無視できます。ただし、これらの関数を SDSoC 環境に組み込むと、Arm プロセッサでコンパイルできないので、システムはハードウェアエミュレーションおよびターゲットプラットフォーム上での実行によってのみ検証可能です。



重要: HLS ハードウェア データ型を含むハードウェア関数を SDSoC 環境に組み込む場合は、HLS ツール環境内で C コンパイルおよびハードウェアシミュレーションで完全に検証された状態にしてください。



重要: `hls::stream` データ型は HLS ツール内で使用するよう設計されており、エンベデッド CPU でソフトウェアを実行するのには不向きなので、このデータ型は最上位関数インターフェイスには含めないようにしてください。

ハードウェア関数のいずれかの引数で HLS ツール特有のデータ型が使用されている場合、関数引数リストにネイティブ C/C++ 型のみを含む最上位 C/C++ ラッパー関数にその関数を含める必要があります。

データ モーション ネットワークの最適化

最適化済みのハードウェア関数を SDSoC 環境のプロジェクトにインポートしたら、まずインターフェイス最適化をすべて削除します。PS とハードウェア関数は、ハードウェア関数とデータ アクセスのデータ型に基づいて管理され、自動的に最適化されます。詳細は、[データ モーションの最適化](#)を参照してください。

- ハードウェア関数の INTERFACE 指示子を削除します。
- ハードウェア関数引数リストの変数を参照する DATA_PACK 指示子を削除します。
- 関数引数にネイティブ C/C++ データ型のみを使用するラッパーに最上位関数を含めて、Vivado HLS ツールのハードウェアデータ型を削除します。

この例では、アクセラレーションされる関数は1つの最上位ハードウェア関数 `stereo_remap_bm` に含まれていません。

```
int main() {

    unsigned char *inY = (unsigned char *)sds_alloc(HEIGHT*DUALWIDTH);
    unsigned short *inCY = (unsigned short *)sds_alloc(HEIGHT*DUALWIDTH*2);
    unsigned short *outCY = (unsigned short *)sds_alloc(HEIGHT*DUALWIDTH*2);
    unsigned char *outY = (unsigned char *)sds_alloc(HEIGHT*DUALWIDTH);

    // read double wide image from disk
    if (read_yuv_file(inY, DUALWIDTH, DUALWIDTH, HEIGHT, FILENAME) != 0)
        return -1;

    convert_Y8toCY16(inY, inCY, HEIGHT*DUALWIDTH);

    stereo_remap_bm(inCY, outCY, HEIGHT, DUALWIDTH, DUALWIDTH);

    // write single wide image to disk
    convert_CY16toY8(outCY, outY, HEIGHT*DUALWIDTH);
    write_yuv_file(outY, DUALWIDTH, DUALWIDTH, HEIGHT, ONAME);

    // write single wide image to disk
```

```
sds_free(inY);
sds_free(inCY);
sds_free(outCY);
sds_free(outY);
return 0;
}
```

ハードウェアへのメモリアクセスを最適化する際は、ハードウェア関数に渡されるデータ型を確認することが重要です。関数シグネチャを確認すると、最適化する主要な変数は入力データストリーム `img_data_lr` と出力データストリーム `img_data_disp` であることがわかります。

```
int stereo_remap_bm(
    yuv_t *img_data_lr,
    yuv_t *img_data_disp,
    int height,
    int dual_width,
    int stride);
```

データはシーケンシャルに転送されるので、まず両方の引数のアクセスパターンを `SEQUENTIAL` と定義します。次の最適化では、`memory_attribute` を `PHYSICAL_CONTIGUOUS|NON_CACHEABLE` に指定して、データ転送がスキャッターギャザーDMAの動作により中断されないようにします。これにはメモリを `sds_lib` からの `sds_alloc` を使用して割り当てるようにする必要があります。

```
#include "sds_lib.h"
int main() {

    unsigned char *inY = (unsigned char *)sds_alloc(HEIGHT*DUALWIDTH);
    unsigned short *inCY = (unsigned short *)sds_alloc(HEIGHT*DUALWIDTH*2);
    unsigned short *outCY = (unsigned short *)sds_alloc(HEIGHT*DUALWIDTH*2);
    unsigned char *outY = (unsigned char *)sds_alloc(HEIGHT*DUALWIDTH);

}
```

最後に、`copy` 指示子でデータがアクセラレータにコピーされるようにし、共有メモリからアクセスされないようにします。

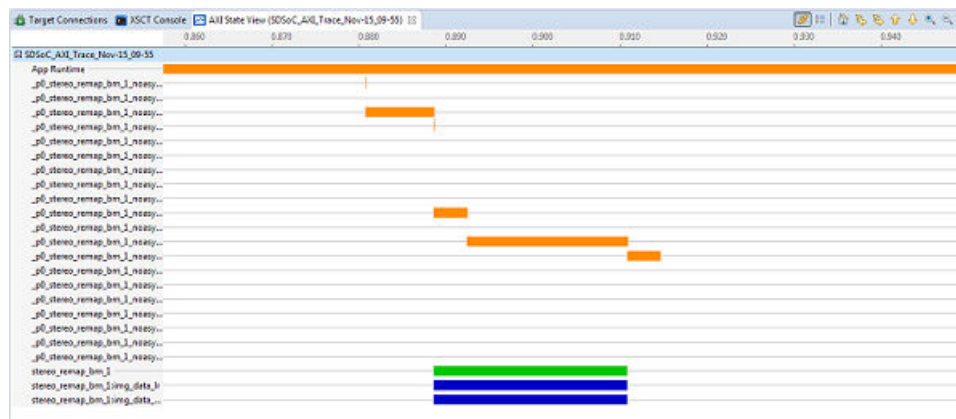
```
#pragma SDS data access_pattern(img_data_lr:SEQUENTIAL)
#pragma SDS data mem_attribute(img_data_lr:PHYSICAL_CONTIGUOUS|NON_CACHEABLE)
#pragma SDS data copy(img_data_lr[0:stride*height])
#pragma SDS data access_pattern(img_data_disp:SEQUENTIAL)
#pragma SDS data mem_attribute(img_data_disp:PHYSICAL_CONTIGUOUS|
NON_CACHEABLE)
#pragma SDS data copy(img_data_disp[0:stride*height])
int stereo_remap_bm(
    yuv_t *img_data_lr,
    yuv_t *img_data_disp,
    int height,
    int dual_width,
    int stride);
```

これらの最適化指示子を使用すると、最も効率的な転送になるように PS と PL 間のメモリアクセスが最適化されます。

Vivado HLS ツールで最適化されたハードウェア関数をラップして HLS のハードウェアデータ型がハードウェア関数のインターフェイスに露出しないようにし、インターフェイス指示子を削除してデータ転送を最適化し、ハードウェア関数をコンパイルし直したら、イベントトレースを使用してパフォーマンスを解析します。

次に示すイベントトレースの全体図では、システムのパフォーマンスができるだけ高くなるように、ハードウェア関数とデータ転送のすべてが並列実行されていることが示されています。

図 28: イベントトレース



レーンの1つにカーソルを置くと、アクセラレータのランタイムの期間がポップアップで表示されます。実行時間は15.86 msであり、ライブビデオの毎秒60フレームを達成するために必要な条件の16.8 msを満たしています。

その他のリソースおよび法的通知

ザイリンクス リソース

アンサー、資料、ダウンロード、フォーラムなどのサポートリソースは、[ザイリンクス サポート](#) サイトを参照してください。

Documentation Navigator およびデザイン ハブ

ザイリンクス Documentation Navigator (DocNav) では、ザイリンクスの資料、ビデオ、サポート リソースにアクセスでき、特定の情報を取得するためにフィルター機能や検索機能を利用できます。DocNav を開くには、次のいずれかを実行します。

- Vivado® IDE で [Help] → [Documentation and Tutorials] をクリックします。
- Windows で [スタート] → [すべてのプログラム] → [Xilinx Design Tools] → [DocNav] をクリックします。
- Linux コマンド プロンプトに「docnav」と入力します。

ザイリンクス デザイン ハブには、資料やビデオへのリンクがデザイン タスクおよびトピックごとにまとめられており、これらを参照することでキー コンセプトを学び、よくある質問 (FAQ) を参考に問題を解決できます。デザイン ハブにアクセスするには、次のいずれかを実行します。

- DocNav で [Design Hub View] タブをクリックします。
- ザイリンクス ウェブサイトで[デザイン ハブ](#) ページを参照します。

注記: DocNav の詳細は、ザイリンクス ウェブサイトの [Documentation Navigator](#) ページを参照してください。



注意: DocNav からは、日本語版は参照できません。ウェブサイトのデザイン ハブ ページをご利用ください。

参考資料

このガイドの補足情報は、次の資料を参照してください。

日本語版のバージョンは、英語版より古い場合があります。

1. 『SDSoC 環境リリース ノート、インストール、およびライセンス ガイド』 ([UG1294](#))

2. 『SDSoC 環境ユーザー ガイド』 ([UG1027](#))
3. 『SDSoC 環境チュートリアル: 概要』 ([UG1028](#))
4. 『SDSoC 環境チュートリアル: プラットフォームの作成』 ([UG1236](#))
5. 『SDSoC 環境プラットフォーム開発ガイド』 ([UG1146](#))
6. 『SDSoC 環境プロファイリングおよび最適化ガイド』 ([UG1235](#))
7. 『SDx コマンドおよびユーティリティ リファレンス ガイド』 ([UG1279](#))
8. 『SDSoC 環境プログラマ ガイド』 ([UG1278](#))
9. 『SDSoC 環境デバッグ ガイド』 ([UG1282](#))
10. 『SDx プラグマ リファレンス ガイド』 ([UG1253](#))
11. 『UltraFast エンベデッド デザイン設計手法ガイド』 (UG1046: [英語版](#)、[日本語版](#))
12. 『Zynq-7000 SoC ソフトウェア開発者向けガイド』 (UG821: [英語版](#)、[日本語版](#))
13. 『Zynq UltraScale+ MPSoC: ソフトウェア開発者向けガイド』 (UG1137: [英語版](#)、[日本語版](#))
14. 『Zynq-7000 XC7Z020 SoC 用 ZC702 評価ボード ユーザー ガイド』 ([UG850](#))
15. 『ZCU102 評価ボード ユーザー ガイド』 ([UG1182](#))
16. 『PetaLinux ツール資料: リファレンス ガイド』 ([UG1144](#))
17. 『Vivado Design Suite ユーザー ガイド: 高位合成』 ([UG902](#))
18. 『Vivado Design Suite ユーザー ガイド: カスタム IP の作成とパッケージ』 ([UG1118](#))
19. [SDSoC 開発環境ウェブ ページ](#)
20. [Vivado® Design Suite 資料](#)

トレーニング リソース

1. [トレーニング コース: SDSoC 開発環境および設計手法](#)
2. [トレーニング コース: アドバンス SDSoC 開発環境および設計手法](#)

お読みください: 重要な法的通知

本通知に基づいて貴殿または貴社 (本通知の被通知者が個人の場合には「貴殿」、法人その他の団体の場合には「貴社」。以下同じ) に開示される情報 (以下「本情報」といいます) は、サイリンクスの製品を選択および使用することのためにのみ提供されます。適用される法律が許容する最大限の範囲で、(1) 本情報は「現状有姿」、およびすべて受領者の責任で (with all faults) という状態で提供され、サイリンクスは、本通知をもって、明示、黙示、法定を問わず (商品性、非侵害、特定目的適合性の保証を含みますがこれらに限られません)、すべての保証および条件を負わない (否認する) ものとします。また、(2) サイリンクスは、本情報 (貴殿または貴社による本情報の使用を含む) に関係し、起因し、関連する、いかなる種類・性質の損失または損害についても、責任を負わない (契約上、不法行為上 (過失の場合を含む)、その他のいかなる責任の法理によるかを問わない) ものとし、当該損失または損害には、直接、間接、特別、付随的、結果的な損失または損害 (第三者が起こした行為の結果被った、データ、利益、業務上の信用の損失、その他あらゆる種類の損失や損害を含みます) が含まれるものとし、それは、たとえ当該損害や損失が合理的に予見可能であっ

たり、サイリンクスがそれらの可能性について助言を受けていた場合であったとしても同様です。サイリンクスは、本情報に含まれるいかなる誤りも訂正する義務を負わず、本情報または製品仕様のアップデートを貴殿または貴社に知らせる義務も負いません。事前の書面による同意のない限り、貴殿または貴社は本情報を再生産、変更、頒布、または公に展示してはなりません。一定の製品は、サイリンクスの限定的保証の諸条件に従うこととなるので、<https://japan.xilinx.com/legal.htm#tos> で見られるサイリンクスの販売条件を参照してください。IP コアは、サイリンクスが貴殿または貴社に付与したライセンスに含まれる保証と補助的条件に従うことになります。サイリンクスの製品は、フェイルセーフとして、または、フェイルセーフの動作を要求するアプリケーションに使用するために、設計されたり意図されたりしていません。そのような重大なアプリケーションにサイリンクスの製品を使用する場合のリスクと責任は、貴殿または貴社が単独で負うものです。<https://japan.xilinx.com/legal.htm#tos> で見られるサイリンクスの販売条件を参照してください。

自動車用のアプリケーションの免責条項

オートモティブ製品 (製品番号に「XA」が含まれる) は、ISO 26262 自動車用機能安全規格に従った安全コンセプトまたは余剰性の機能 (「セーフティ設計」) がない限り、エアバッグの展開における使用または車両の制御に影響するアプリケーション (「セーフティ アプリケーション」) における使用は保証されていません。顧客は、製品を組み込むすべてのシステムについて、その使用前または提供前に安全を目的として十分なテストを行うものとします。セーフティ設計なしにセーフティ アプリケーションで製品を使用するリスクはすべて顧客が負い、製品責任の制限を規定する適用法令および規則にのみ従うものとします。

商標

© Copyright 2016-2019 Xilinx, Inc. Xilinx、Xilinx のロゴ、Alveo、Artix、Kintex、Spartan、Versal、Virtex、Vivado、Zynq、およびこの文書に含まれるその他の指定されたブランドは、米国およびその他の各国のサイリンクス社の商標です。OpenCL および OpenCL のロゴは Apple Inc. の商標であり、Khronos による許可を受けて使用されています。HDMI、HDMI のロゴ、および High-Definition Multimedia Interface は、HDMI Licensing LLC の商標です。AMBA、AMBA Designer、Arm、ARM1176JZ-S、CoreSight、Cortex、PrimeCell、Mali、および MPCore は、EU およびその他の各国の Arm Limited の商標です。すべてのその他の商標は、それぞれの所有者に帰属します。

この資料に関するフィードバックおよびリンクなどの問題につきましては、jpn_trans_feedback@xilinx.com まで、または各ページの右下にある [フィードバック送信] ボタンをクリックすると表示されるフォームからお知らせください。フィードバックは日本語で入力可能です。いただきましたご意見を参考に早急に対応させていただきます。なお、このメール アドレスへのお問い合わせは受け付けておりません。あらかじめご了承ください。