

SDAccel プログラマ ガイド

UG1277 (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	
SDAccel のビルド プロセス	.xo ファイルにカーネルの説明を追加。
SDAccel 実行モデル	セクションをアップデート。
プログラム	コードと 3 番の説明をアップデート。
サブデバイス	セクションをアップデート。
FPGA デバイスでのコマンドの実行	説明をアップデート。
カーネルの設定	セクションをアップデート。セクションの順番を並べ替えて、実行のセクションを削除。
FPGA デバイスのバッファ転送	セクションをアップデート。
カーネル実行	新規セクション。
後処理および FPGA のクリーンアップ	セクションをアップデート。
ホストからカーネルのデータフローのイネーブル	セクションをアップデート。
まとめ	番号付きリストをアップデート。
カーネル ポートとグローバル メモリの接続	注記をアップデート。
付録 A: SDAccel ストリーミング プラットフォーム	新規付録。
2019 年 1 月 24 日 バージョン 2018.3	
資料全体	編集上のアップデート。
2018 年 12 月 5 日 バージョン 2018.3	
第 1 章: SDAccel のコンパイル フローおよび実行モデル	SDAccel アーキテクチャの図をアップデート。
デバイス	サブデバイスの記述を追加。
FPGA デバイスのバッファ転送	セクションをアップデート。
まとめ	説明の 3 番をアップデート。
第 3 章: C/C++ カーネルのプログラミング	説明を追加。
メモリのデータ入力および出力	セクション全体をアップデート。
カーネル ポートとグローバル メモリの接続	注記を追加。
2018 年 10 月 2 日 バージョン 2018.2.xdf	
	変更なし。
2018 年 7 月 2 日 バージョン 2018.2	
カーネル ポートとグローバル メモリの接続	--sp オプションの構文および要件をアップデート。
2018 年 6 月 6 日 バージョン 2018.2	
最初のサイリンクス リリース。	該当なし

目次

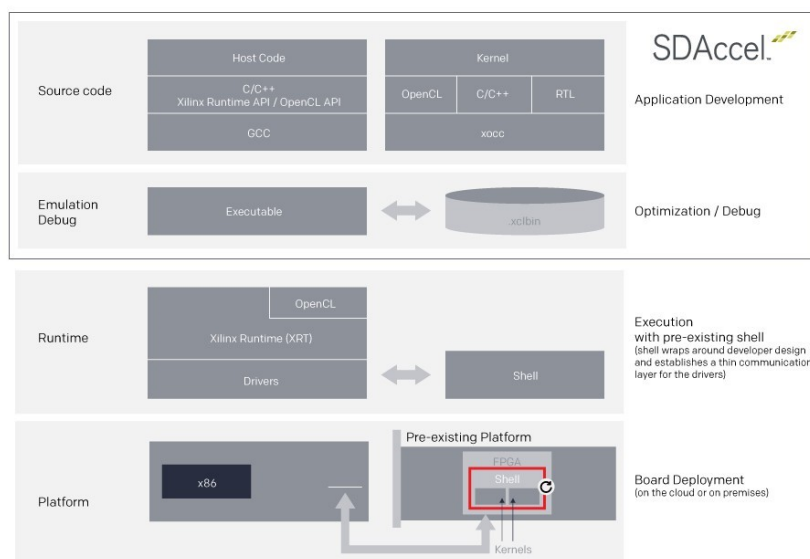
改訂履歴.....	2
第 1 章: SDAccel のコンパイル フローおよび実行モデル.....	5
プログラミング モデル.....	5
デバイス トポロジ.....	6
SDAccel のビルド プロセス.....	6
SDAccel 実行モデル.....	9
SDAccel のエミュレーション フロー.....	11
SDAccel サンプル デザイン.....	12
このガイドの構成.....	12
第 2 章: ホスト アプリケーションのプログラム.....	13
OpenCL 環境の設定.....	13
FPGA デバイスでのコマンドの実行.....	17
後処理および FPGA のクリーンアップ.....	27
まとめ.....	27
第 3 章: C/C++ カーネルのプログラミング.....	28
データ型.....	28
インターフェイス.....	30
ループ.....	33
データフロー最適化.....	38
配列コンフィギュレーション.....	40
関数のインライン展開.....	44
まとめ.....	44
第 4 章: システム アーキテクチャの設定.....	45
カーネルの複数のインスタンス.....	45
カーネル ポートとグローバル メモリの接続.....	46
まとめ.....	47
付録 A: SDAccel ストリーミング プラットフォーム.....	48
ホストとカーネル間のストリーミング データ転送.....	48
カーネル間のストリーミング データ転送.....	52
付録 B: OpenCL インストーラブル クライアント ドライバー ロードー.....	54
付録 C: その他のリソースおよび法的通知.....	55

ザイリンクス リソース.....	55
Documentation Navigator およびデザイン ハブ.....	55
参考資料.....	55
お読みください: 重要な法的通知.....	56

SDAccel のコンパイル フローおよび実行モデル

SDAccel™ 開発環境は、ザイリンクス FPGA デバイスを使用して計算負荷の高いタスクをアクセラレーションするためのヘテロジニアス システム アーキテクチャ プラットフォームです。SDAccel 環境には、次の図に示すように、PCIe® バスを介して 1 つまたは複数のザイリンクス FPGA デバイスに接続されているホスト x86 マシンが含まれます。

図 1: SDAccel アーキテクチャ



プログラミング モデル

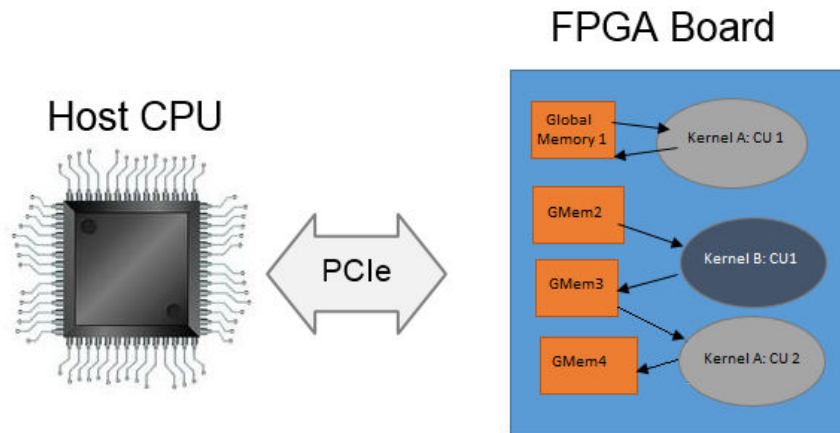
SDAccel 環境では、業界標準の OpenCL プロトコル (<https://www.khronos.org/opencl/>) を使用したヘテロジニアス計算がサポートされています。ホスト プログラムはホスト CPU で実行し、計算負荷の高いタスクは OpenCL のプログラミングの基本枠組みを使用してザイリンクス FPGA デバイス上で実行します。CPU (または GPU) とは異なり、FPGA はあらかじめ定義された命令セットまたは固定ワード サイズのないブランク キャンバスと考えることができ、同じまたは異なる命令を並列実行してアプリケーションのパフォーマンスを向上できます。

デバイス トポロジ

SDAccel 環境では、デバイスは PCIe バスを介してホスト x86 マシンに接続された 1 つまたは複数の FPGA です。FPGA には、カーネルをインプリメントして実行するプログラマブル領域が含まれます。FPGA プラットフォームには、1 つまたは複数のグローバル メモリ バンクが含まれます。ホスト マシンとカーネルの間のデータ転送は、これらのグローバル メモリ バンクを介して実行されます。FPGA 上で実行されるカーネルには、1 つまたは複数のメモリ インターフェイスを含めることができます。グローバル メモリ バンクからこれらのメモリ インターフェイスへの接続は柔軟にプログラム可能であり、カーネルのコンパイル オプションにより指定されます。

ザイリンクス デバイスのプログラマブル ロジックには、同時に複数のカーネルをインプリメントでき、タスクの並列実行を可能にします。1 つのカーネルを複数回インスタンス化することもできます。カーネルのインスタンス数はプログラム可能で、カーネルのコンパイル オプションにより指定されます。

図 2: SDAccel アーキテクチャ



上の図は、ホスト マシンからグローバル メモリ バンクを介した複数のカーネルへの柔軟な接続を示します。この図の FPGA ボード デバイスには、4 つの DDR メモリ バンクが含まれます。FPGA のプログラマブル ロジックでは、Kernel A および Kernel B という 2 つのカーネルが実行されます。各カーネルには 2 つのメモリ インターフェイス (読み出し用に 1 つ、書き込み用に 1 つ) があります。Kernel A のインスタンスは 2 つあり、FPGA 上には合計 3 つのカーネル インスタンスが含まれます。

図では、最初のインスタンスである Kernel A: CU1 は、読み出しと書き込みの両方に 1 つのメモリ インターフェイスを使用しています。Kernel B および Kernel A の 2 つ目のインスタンスである Kernel A: CU2 は、読み出しと書き込みに別のメモリ インターフェイスを使用します。Kernel B はグローバル メモリを介して Kernel A: CU2 に直接データを渡します。



推奨: 最大限のパフォーマンスを達成するため、グローバル メモリ バンクからカーネル インターフェイスへの接続は、カーネル ポートとグローバル メモリの接続に説明するように、注意して定義する必要があります。

SDAccel のビルド プロセス

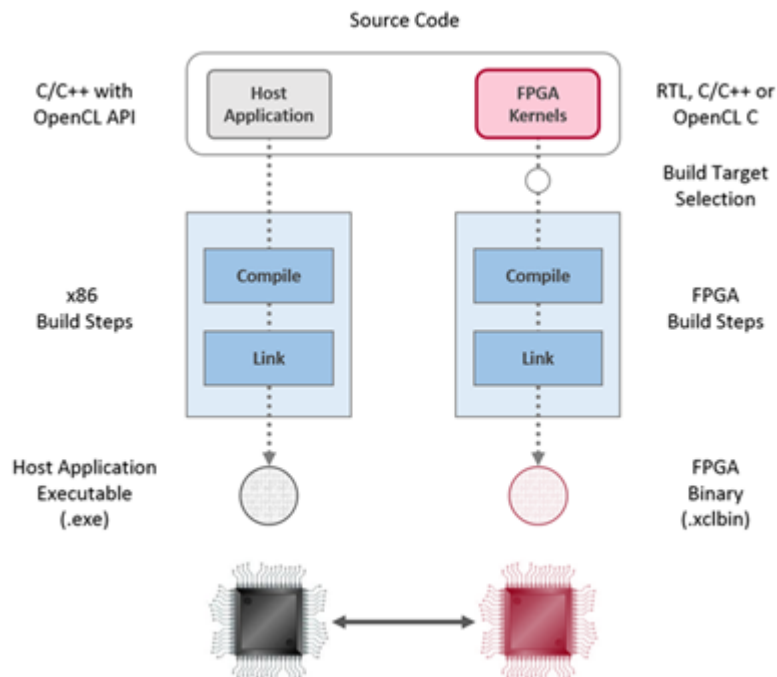
SDAccel 環境には、標準ソフトウェア開発環境の機能がすべて含まれています。

- ホスト アプリケーション用に最適化されたコンパイラ

- FPGA デのクロス コンパイラ
- コードの問題を見つけて解決しやすくするデバッグ環境
- ボトルネックを見つけてコードを最適化するパフォーマンス プロファイラー

この環境内のビルド プロセスでは、標準のコンパイルおよびリンク プロセスをプロジェクトのソフトウェア要素とハードウェア要素の両方に使用します。次の図に示すように、ホスト アプリケーションは標準 GCC コンパイラを使用した 1 つのプロセスでビルドされ、FPGA バイナリはザイリンクス `xocc` コンパイラを使用した別のプロセスでビルドされます。

図 3: ソフトウェア/ハードウェアのビルド プロセス

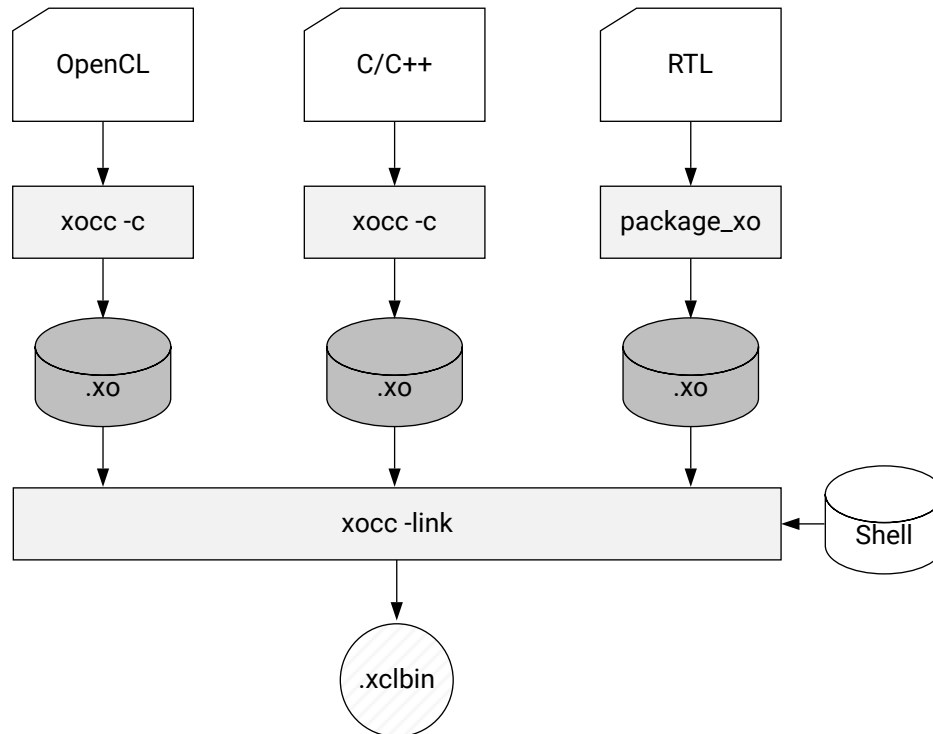


X22015-112618

1. GCC を使用したホスト アプリケーションのビルド プロセス:
 - ホスト アプリケーションのソース ファイルをそれぞれオブジェクト ファイル (.o) にコンパイルします。
 - オブジェクト ファイル (.o) をザイリンクス SDAccel ランタイム共有ライブラリとリンクし、実行ファイル (.exe) を作成します。
2. FPGA ビルド プロセスは、次の図でハイライトされています。
 - 各カーネルを個別にザイリンクス オブジェクト (.xo) ファイルにコンパイルします。
 - C/C++ および OpenCL C カーネルを `xocc` コンパイラを使用して FPGA にインプリメンテーションできるようにコンパイルします。この手順には、Vivado® HLS コンパイラが使用されます。Vivado HLS でサポートされるプラグマおよび属性を C/C++ および OpenCL C カーネル ソース コードで使用し、必要なカーネルのマイクロ アーキテクチャを指定して、コンパイル プロセスの結果を制御できます。
 - `package_xo` ユーティリティを使用して RTL カーネルをコンパイルします。SDAccel 環境の RTL カーネル ウィザードを使用すると、このプロセスを簡単に実行できます。
 - カーネル .xo ファイルがハードウェア プラットフォーム (シェル) にリンクされ、FPGA バイナリ (.xclbin) が作成されます。アーキテクチャの重要な点は、リンク段階で指定します。特に、カーネル ポートからグローバル メモリ バンクまでの接続を確立し、各カーネルのインスタンス数を指定します。

- ビルド ターゲットがソフトウェアまたはハードウェア エミュレーションの場合は、次に説明するように、`xocc` でデバイスの内容のシミュレーション モデルが生成されます。
- ビルド ターゲットがシステム (実際のハードウェア) の場合は、`xocc` で FPGA バイナリが生成され、デバイスが Vivado Design Suite を使用して合成およびインプリメンテーションできるようになります。

図 4: FPGA のビルド プロセス



X21155-111518

注記: `xocc` コンパイラでは Vivado HLS および Vivado Design Suite ツールが自動的に使用され、FPGA プラットフォームで実行するカーネルがビルドされます。この場合、ツールで良い QoR (結果の品質) が得られる定義済み設定が使用されます。SDAccel 環境および `xocc` コンパイラの使用には、これらのツールの知識は必要ありませんが、ハードウェアに精通していると、これらのツールで使用可能なすべての機能を活用してカーネルをインプリメントできます。

ビルド ターゲット

SDAccel ツールのビルド プロセスでは、ホスト アプリケーションの実行ファイル (`.exe`) と FPGA バイナリ (`.xclbin`) を生成します。SDAccel ビルド ターゲットは、ビルド プロセスで生成される FPGA バイナリの特徴を定義します。

SDAccel ツールには、デバッグおよび検証に使用する 2 つのエミュレーション ターゲット、および実際の FPGA バイナリを生成するのに使用されるデフォルトのハードウェア ターゲットの 3 つのビルド ターゲットがあります。

- ソフトウェア エミュレーション (`sw_emu`): ホスト アプリケーション コードとカーネル コードの両方が x86 プロセッサで実行できるようコンパイルされます。これにより、高速なビルドおよび実行ループを使用した反復アルゴリズムによる改善が可能になります。このターゲットは、構文エラーを特定し、アプリケーションと共に実行されるカーネル コード ソース レベルのデバッグを実行し、システムの動作を検証するのに便利です。

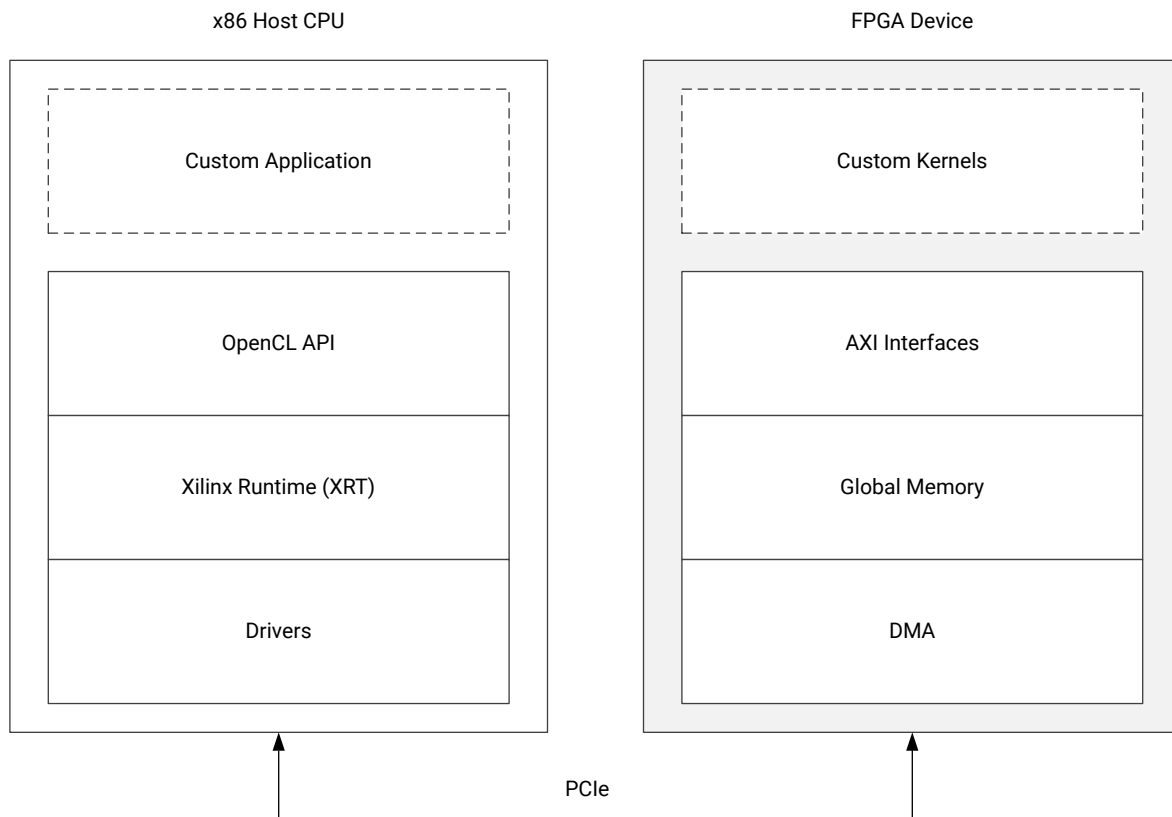
- ハードウェア エミュレーション (hw_emu): カーネル コードがハードウェア モデル (RTL) にコンパイルされ、専用シミュレータで実行されます。ビルドおよび実行ループにかかる時間は長くなりますが、詳細でサイクル精度のカーネル アクティビティが表示されます。このターゲットは、FPGA に含まれるロジックの機能をテストして、最初のパフォーマンス見積もりを得る場合に便利です。
- システム (hw): カーネル コードがハードウェア モデル (RTL) にコンパイルされた後 FPGA デバイスにインプリメントされて、実際の FPGA で実行されるバイナリが生成されます。

SDAccel 実行モデル

SDAccel フレームワークでは、アプリケーション プログラムがホスト アプリケーションとハードウェア アクセラレーションされたカーネル間で通信チャネルを使用して分割されます。C/C++ および OpenCL のような API 抽象化を使用して記述されたホスト アプリケーションは x86 サーバーで実行されますが、ハードウェア アクセラレーションされたカーネルはザイリンクス FPGA 内で実行されます。ハードウェア アクセラレータとの通信には、ザイリンクスランタイム (XRT) で管理される API 呼び出しが使用されます。制御およびデータ転送を含め、ホスト x86 マシンとアクセラレータ ボード間の通信は、PCIe バスで発生します。制御情報はハードウェアの特定のメモリ位置間で転送されますが、ホスト アプリケーションとカーネル間のデータ転送にはグローバル メモリが使用されます。グローバル メモリには、ホスト プロセッサとハードウェア アクセラレータの両方からアクセスできますが、ホスト メモリにはホスト アプリケーションからしかアクセスできません。

たとえば、典型的なアプリケーションの場合、ホストがまずカーネルで実行されるデータをホスト メモリからグローバル メモリに転送します。カーネルはデータを続けて処理し、結果をグローバル メモリに格納します。カーネルが終了したら、ホストが結果をホスト メモリに転送し戻します。ホストとグローバル メモリ間のデータ転送により、レイテンシが発生し、アクセラレーション全体に悪影響を及ぼすことがあります。実際のシステムでアクセラレーションを達成するには、ハードウェア アクセラレーション カーネルで達成される利点がこのデータ転送のレイテンシを上回るようにしてください。次の図は、このアクセラレーション プラットフォームの一般的な構造を示しています。

図 5: SDAccel アプリケーションのアーキテクチャ



X21835-103118

右側の FPGA ハードウェア プラットフォームにはハードウェアアクセラレーションされたカーネル、グローバル メモリとメモリ転送用の DMA が含まれます。カーネルは 1 つまたは複数のグローバル メモリ インターフェイスを含むことができ、プログラマブルです。SDAccel 実行モデルでは、次が実行されます。

1. ホスト アプリケーションは PCle を介して、カーネルで必要とされるデータを接続されたデバイスのグローバル メモリに書き込みます。
2. ホスト アプリケーションは、その入力パラメーターを使用してカーネルを設定します。
3. ホスト アプリケーションは FPGA のカーネル関数の実行をトリガーします。
4. カーネルは必要な計算をし、グローバル メモリからのデータの読み出しを必要に応じて実行します。
5. カーネルがグローバル メモリにデータを書き戻し、ホストにタスクが終了したことを通知します。
6. ホスト アプリケーションがグローバル メモリからホスト メモリにデータを読み出して、必要に応じて処理を続けます。

FPGA には一度に複数のカーネル インスタンス (別のカーネル タイプまたは同じカーネルの複数のインスタンス) を含めることができます。ホスト アプリケーションと FPGA のカーネル間の通信は、XRT で管理されます。カーネルのインスタンス数は、コンパイル オプションにより指定されます。

SDAccel のエミュレーション フロー

SDAccel 環境開発フローは、2 つの段階に分けることができます。1 つ目の段階では、ホスト コードおよびカーネル コードをコンパイルして、実行ファイルを生成します。2 つ目の段階では、ホスト CPU と SDAccel 環境アクセラレータ プラットフォームで構成されるヘテロジニアス システムで実行ファイルを実行します。ただし、カーネルのコンパイルは時間のかかるプロセスであり、カーネルのサイズとターゲットの FPGA のアーキテクチャによっては数時間かかることがあります。そのため、カーネルのコンパイル プロセスの前のデバッグ サイクル時間を短縮するために、SDAccel 環境にはテスト用にその他 2 つのビルド ターゲット (ソフトウェア エミュレーションおよびハードウェア エミュレーション) があります。これらのエミュレーション ターゲットのコンパイルと実行は非常に高速で、実際の FPGA ボードは必要はありません。これらのエミュレーション フローは、FPGA ボードとそのホスト マシンおよびソフトウェア モデルへの接続を抽象化して、ホストとカーネル コードを組み合わせたときの機能を確認し、設計プロセスの早期のパフォーマンス見積もりを取得します。このパフォーマンス見積もりはあくまでも見積もりですが、パフォーマンスのボトルネックを特定してデバッグするのに役立ちます。ソフトウェアおよびハードウェア エミュレーション フローを使用したデバッグの詳細は、『SDAccel 環境デバッグ ガイド』 ([UG1281](#)) を参照してください。

ソフトウェア エミュレーション フロー

ソフトウェア エミュレーション ターゲットのコンパイルは、最も高速です。ホスト コードとカーネル コードを一緒に実行したときに機能が正しいかどうかを確認するために主に使用されます。カーネル コードをホスト コードと共に実行するために最低限必要な変換が `xocc` コンパイラにより実行されるので、最終的なバイナリの作成の初期段階で機能が正しいかどうかを確認するのに役立ちます。ソフトウェア エミュレーション フローはアルゴリズムの調整および機能的な問題のデバッグに使用して、コードを向上するために反復作業をすばやく実行できます。

ハードウェア エミュレーション フロー

ハードウェア エミュレーション フローでは、`xocc` コンパイラによりカーネル モデルがハードウェア記述言語 (RTL Verilog) で生成されます。ハードウェア エミュレーション フローは、C、C++、または OpenCL カーネル コードから RTL の合成後に、最終的に作成されたバイナリが論理的に正しいかどうかをチェックするのに役立ちます。ハードウェア エミュレーション フローでは、システムが意図したとおりに機能していない場合に、波形ビューアーを使用してデバッグを実行できます。

表 1: ハードウェア実行を使用したエミュレーション フローの比較

ソフトウェア エミュレーション	ハードウェア エミュレーション	ハードウェア実行
ホスト アプリケーションをカーネルの C/C++ または OpenCL モデルを使用して実行。	ホスト アプリケーションをカーネルのシミュレーション済み RTL モデルを使用して実行。	ホスト アプリケーションをカーネルの実際のハードウェア インプリメンテーションを使用して実行。
システムの機能が論理的に正しいことを確認。	ホストとカーネルの統合をテスト、パフォーマンス見積もりを取得。	システムが正しく必要なパフォーマンスで実行されることを確認。
実行時間が最短。	最高のデバッグ機能、コンパイル時間は中程度。	最終的な FPGA インプリメンテーションおよび run で正確なパフォーマンス結果を提供、ビルド時間が長い。

SDAccel サンプル デザイン

GitHub 上の SDAccel サンプル デザイン

ザイリンクス では、[GitHub リポジトリ](#)で SDAccel 環境でのプログラムの例を多数提供しています。これらは、新規ユーザーはホストおよびカーネル コードのコーディング スタイルを理解するため、上級ユーザーはコード例のソースとして使用できます。すべての例には、ホスト コード、カーネル コード、コンパイル フローおよびランタイム フローに関連付けられた makefile が含まれています。次に示す例はその 1 つで、標準的なファイル構造を理解するのに役立ちます。

逆離散コサイン変換 (IDCT) の例

[IDCT の例](#)は、SDAccel 環境に必要な主要なコード構成を示します。

Readme.md ファイルには、この例を Makefile を使用してエミュレーション フローおよび FPGA フローの両方で実行する方法が詳細が示されています。

./src ディレクトリには、ホスト コード idct.cpp とカーネル コード krnl_idct.cpp があります。

この後の章に、SDAccel 環境用にホスト コードおよびカーネル コードをプログラムするために必要な基本的な情報を示します。このプロセス中、上記のデザインを例として参照できます。

このガイドの構成

このガイドには、次の章が含まれます。

- [第 2 章: ホスト アプリケーションのプログラム](#): ザイリンクス FPGA デバイスをターゲットとする OpenCL API を使用してホスト コードを記述する方法を説明します。この章は、OpenCL の知識があることを前提としており、ザイリンクス FPGA デバイスで実行されるアクセラレーション カーネルとインターフェイスする効率的なホスト アプリケーションを記述する際に従う必要のあるコード記述方法について説明します。
- [第 3 章: C/C++ カーネルのプログラミング](#): FPGA デバイスにインプリメントする高パフォーマンスの計算負荷の高いカーネル コードを記述する際の異なる要素を説明します。
- [第 4 章: システム アーキテクチャの設定](#): リンキング プロセス中にホスト アプリケーションを 1 つまたは複数のカーネル インスタンスに統合および接続する方法を示します。

ホスト アプリケーションのプログラム

SDAccel™ 環境では、ホスト コードは業界標準の OpenCL™ API を使用した C または C++ 言語で記述します。SDAccel 環境には、OpenCL 1.2 エンベデッド プロファイル準拠ランタイム API が含まれます。

注記: SDAccel 環境では、OpenCL インストーラブル クライアント ドライバー (ICD) 拡張 (cl_khr_icd) がサポートされます。これにより、OpenCL を複数インプリメンテーションして、同じシステム内に共存させることができます。詳細およびインストール方法については、[付録 B: OpenCL インストーラブル クライアント ドライバー ローダー](#)を参照してください。

SDAccel 環境には、ホスト x86 CPU とザイリンクス FPGA デバイスで実行される計算デバイスが含まれます。

通常、ホスト コードは次の 3 つのセクションに分けることができます。

1. 環境の設定。
2. 1 つまたは複数のカーネルの実行を含むコア コマンドの実行。
3. 後処理および FPGA の解放。

次のセクションでは、それぞれについて詳しく説明します。

注記: ホスト プログラムをマルチスレッドで実行する場合は、SDAccel 環境アプリケーションからの `fork()` システム呼び出しを呼び出す際に注意が必要です。`fork()` は、すべてのランタイム スレッドを複製するわけではありません。このため、子プロセスを完全な SDAccel 環境のアプリケーションとして実行することはできません。SDAccel 環境アプリケーションから別のプロセスを起動する場合は、`posix_spawn()` を使用することをお勧めします。

OpenCL 環境の設定

SDAccel 環境のホスト コードは、OpenCL プログラミングの基本枠組みに従います。環境を正しく設定するには、ホスト アプリケーションが標準 OpenCL モデルを特定する必要があります。これらは、つまりプラットフォーム、デバイス、コンテキスト、コマンド キュー、およびプログラムのことです。



ヒント: この資料で使用されるホスト コード例および API コマンドは、OpenCL C API に従っています。C API については、[SDAccel サンプル デザインの IDCT の例](#)でも説明されています。ただし、SDAccel ランタイム環境では OpenCL C++ ラッパー API もサポートされており、[GitHub リポジトリ](#)の多くの例が C++ API を使用して記述されています。この C++ ラッパー API の詳細は、<https://www.khronos.org/registry/OpenCL/specs/opencl-cplusplus-1.2.pdf> を参照してください。

プラットフォーム

ホスト コードの冒頭で、1 つまたは複数のザイリンクス FPGA で構成されるプラットフォームを特定する必要があります。次に、ザイリンクス デバイス ベースのプラットフォームを特定する標準ホスト コードを示します。

```
cl_platform_id platform_id;          // platform id

err = clGetPlatformIDs(16, platforms, &platform_count);

// Find Xilinx Platform
for (unsigned int iplat=0; iplat<platform_count; iplat++) {
    err = clGetPlatformInfo(platforms[iplat],
        CL_PLATFORM_VENDOR,
        1000,
        (void *)cl_platform_vendor,
        NULL);

    if (strcmp(cl_platform_vendor, "Xilinx") == 0) {
        // Xilinx Platform found
        platform_id = platforms[iplat];
    }
}
```

OpenCL API 呼び出し `clGetPlatformIDs` を使用して、システムで使用可能な OpenCL プラットフォームを検索します。その後、`clGetPlatformInfo` を使用して、`cl_platform_vendor` が文字列 "Xilinx" であるザイリンクス デバイス ベースのプラットフォームを取得します。



推奨: 上記のコードまたはこの章で使用されているほかのホスト コード例でははっきりと示されていませんが、各 OpenCL API 呼び出しの後にエラー チェックを使用することをお勧めします。そのようにするとデバッグしやすくなり、エミュレーション フローまたはハードウェア実行中にホスト コードおよびカーネル コードをデバッグする場合に生産性が向上します。次に、`clGetPlatformIDs` コマンドのエラー チェック のコード例を示します。

```
err = clGetPlatformIDs(16, platforms, &platform_count);
if (err != CL_SUCCESS) {
    printf("Error: Failed to find an OpenCL platform!\n");
    printf("Test failed\n");
    exit(1);
}
```

デバイス

プラットフォームが検出されると、プラットフォームに搭載されているザイリンクス FPGA デバイス特定されます。SDAccel 環境では、ザイリンクス FPGA デバイスがサポートされています。

次のコードでは、API `clGetDeviceIDs` を使用してすべてのザイリンクス デバイスを検索して、その名前を表示しています。

```
cl_device_id devices[16]; // compute device id
char cl_device_name[1001];

err = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_ACCELERATOR,
    16, devices, &num_devices);

printf("INFO: Found %d devices\n", num_devices);

//iterate all devices to select the target device.
```

```
for (uint i=0; i<num_devices; i++) {
    err = clGetDeviceInfo(devices[i], CL_DEVICE_NAME, 1024, cl_device_name,
0);
    printf("CL_DEVICE_NAME %s\n", cl_device_name);
}
```



重要: `clGetDeviceIDs` API は `device_type` および `CL_DEVICE_TYPE_ACCELERATOR` を指定して呼び出されており、使用可能なすべてのザイリンクス デバイスが返されます。

サブデバイス

SDAccel 環境では、デバイスに複数の同じカーネルまたは異なるカーネルのインスタンスが含まれることがあります。`clCreateSubDevices` OpenCL API を使用すると、ホスト コードでデバイスを複数のサブデバイスに分割できます (サブデバイスごとにカーネル インスタンスを 1 つずつ含む)。現在のところ、SDAccel 環境ではカーネル インスタンスを 1 つのみ含む等しいサブデバイスに分割できるようになっています。

この後の例では、次が実行されます。

1. サブデバイスは、サブデバイスごとに 1 つのカーネル インスタンスを実行する同等分割で作成されます。
2. 別のコンテキストおよびコマンド キューを使用してサブデバイス リストを反復実行し、それぞれでカーネルを実行します。
3. 単純にするため、カーネル実行に関する API (および対応するバッファに関する) コードは示していませんが、関数 `run_cu` 内に記述してください。

```
cl_uint num_devices = 0;
cl_device_partition_property props[3] = {CL_DEVICE_PARTITION_EQUALLY, 1, 0};

// Get the number of sub-devices
clCreateSubDevices(device, props, 0, nullptr, &num_devices);

// Container to hold the sub-devices
std::vector<cl_device_id> devices(num_devices);

// Second call of clCreateSubDevices
// We get sub-device handles in devices.data()
clCreateSubDevices(device, props, num_devices, devices.data(), nullptr);

// Iterating over sub-devices
std::for_each(devices.begin(), devices.end(), [kernel](cl_device_id sdev) {

    // Context for sub-device
    auto context = clCreateContext(0, 1, &sdev, nullptr, nullptr, &err);

    // Command-queue for sub-device
    auto queue = clCreateCommandQueue(context, sdev,
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &err);

    // Execute the kernel on the sub-device using local context and
    queue run_cu(context, queue, kernel); // Function not shown
});
```



重要: 上記の例に示すように、サブデバイスごとに別のコンテキストを作成してください。OpenCL では、複数のデバイスおよびサブデバイスを保持するコンテキストが作成できるようにはなっていますが、ザイリンクス ランタイム用に、デバイスおよびサブデバイスを別々のコンテキストに分けることをお勧めします。

コンテキスト

OpenCL コンテキストの作成プロセスはシンプルです。 [clCreateContext](#) API を使用して、ホスト マシンと通信する 1 つまたは複数のザイリンクス デバイスを含むコンテキストを作成します。

```
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
```

上記のコード例では、 [clCreateContext](#) API を使用して、1 つのザイリンクス デバイスを含むコンテキストを作成しています。ホスト プログラムから 1 つのデバイス用に作成できるコンテキストは 1 つのみです。サブデバイスが使用されている場合は、ホスト プログラムで複数のコンテキストを使用する必要があります (サブデバイスごとに 1 つのコンテキスト)。

コマンド キュー

各デバイスには、 [clCreateCommandQueue](#) API を使用して 1 つまたは複数のコマンド キューが作成されます。FPGA デバイスには、複数のカーネルを含めることができます。ホスト アプリケーションを開発する場合、デバイスでカーネルを実行するためのプログラミング方法が主に 2 つあります。

1. 1 つの順不同コマンド キュー: 複数のカーネル実行を同じコマンド キューで要求できます。SDAccel ランタイム環境によりこれらのカーネルができるだけ早く任意の順序で実行され、FPGA 上でのカーネルの同時実行を可能にします。
2. 複数の順序どおりのコマンド キュー: 各カーネル実行は異なる順序どおりのコマンド キューから要求されます。この場合、SDAccel ランタイム環境で、FPGA 上での同時実行のパフォーマンスが向上するように、任意のコマンド キューからのカーネルを実行できます。

次に、順不同および順序どおりのコマンド キューを作成する標準 API 呼び出しの例を示します。

```
// Out-of-order Command queue
commands = clCreateCommandQueue(context, device_id,
    CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &err);

// In-order Command Queue
commands = clCreateCommandQueue(context, device_id, 0, &err);
```

プログラム

[SDAccel のビルド プロセス](#)で説明するように、ホストおよびカーネル コードは別々にコンパイルされ、ホスト アプリケーション (.exe) および FPGA バイナリ (.xclbin) などの実行ファイルが別々に作成されます。ホスト アプリケーションが実行されると、 [clCreateProgramWithBinary](#) API を使用して .xclbin を読み込む必要があります。

次のコード例は、標準の OpenCL API を使用して .xclbin からプログラムをビルドする方法を示しています。

```
unsigned char *kernelbinary;
char *xclbin = argv[1];

printf("INFO: loading xclbin %s\n", xclbin);

int size=load_file_to_memory(xclbin, (char **) &kernelbinary);
size_t size_var = size;

cl_program program = clCreateProgramWithBinary(context, 1, &device_id,
    &size_var, (const unsigned char **) &kernelbinary,
    &status, &err);

// Function
```

```
int load_file_to_memory(const char *filename, char **result)
{
    uint size = 0;
    FILE *f = fopen(filename, "rb");
    if (f == NULL) {
        *result = NULL;
        return -1; // -1 means file opening fail
    }
    fseek(f, 0, SEEK_END);
    size = ftell(f);
    fseek(f, 0, SEEK_SET);
    *result = (char *)malloc(size+1);
    if (size != fread(*result, sizeof(char), size, f)) {
        free(*result);
        return -2; // -2 means file reading fail
    }
    fclose(f);
    (*result)[size] = 0;
    return size;
}
```

上記のコード例では、次が実行されます。

1. カーネルのバイナリ ファイル `.xclbin` をコマンド ライン引数 `argv[1]` から渡します。



ヒント: この例では、コマンド ライン引数を使用して `.xclbin` を渡しています。カーネル バイナリ ファイルはアプリケーションでハードコードすることもできます。環境変数を使用するか、カスタム初期化ファイルから読み込むか、その他の適切なメカニズムを使用してください。

2. `load_file_to_memory` 関数を使用して、ファイルの内容をホスト マシンのメモリ空間に読み込みます。
3. `clCreateProgramWithBinary` API を使用してプログラム作成プロセスを完了します。

FPGA デバイスでのコマンドの実行

OpenCL 環境が初期化されたら、ホスト アプリケーションがデバイスに対してコマンドを発行し、カーネルと対話できるようになります。このようなコマンドには、次が含まれます。

1. カーネルの設定。
2. FPGA デバイスのバッファ転送。
3. FPGA 上でのカーネルの実行。
4. イベントの同期化。

カーネルの設定

コンテキスト、コマンド キュー、およびプログラムなどのすべての準備の初期設定が終了したら、ホスト アプリケーションがデバイスで実行する必要のあるカーネルを見つけて、引数を設定するようにします。

カーネルの識別

最初に `clCreateKernel` API を使用して `.xclbin` ファイル内にあるカーネルにアクセスする必要があります。カーネル オブジェクトを示すカーネル ハンドル (`cl_kernel` 型) は、残りのホスト プログラムで使用できるようになっています。

```
kernel1 = (program, "<kernel_name_1>", &err);
kernel2 = clCreateKernel(program, "<kernel_name_2>", &err); // etc
```

カーネル引数の設定

SDAccel 環境フレームワークでは、次の2つのタイプのカーネル引数を設定できます。

1. スカラー引数は、定数またはコンフィギュレーション タイプのデータなどの小型のデータ転送に使用します。これらは、書き込み専用引数です。
2. バッファ引数は、大型のデータ転送に使用します。

カーネル引数は、次に示すように `clSetKernelArg` コマンドを使用して設定します。次の例では、2つのスカラーと2つのバッファ引数のカーネル引数を設定しています。

```
cl_mem dev_buf1 = clCreateBuffer(context, CL_MEM_WRITE, size,
&host_mem_ptr1, NULL);
cl_mem dev_buf2 = clCreateBuffer(context, CL_MEM_READ, size,
&host_mem_ptr2, NULL);

int err = 0;
// Setting up scalar arguments
cl_uint scalar_arg_image_width = 3840;
err |= clSetKernelArg(kernel, 0, sizeof(cl_uint), &scalar_arg_image_width);
cl_uint scalar_arg_image_height = 2160;
err |= clSetKernelArg(kernel, 1, sizeof(cl_uint),
&scalar_arg_image_height);

// Setting up buffer arguments
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &dev_buf1);
err |= clSetKernelArg(kernel, 3, sizeof(cl_mem), &dev_buf2);
```



重要: OpenCL ではカーネルのエンキュー前いつでもカーネル引数を設定できますが、ザイリンクスではできるだけ早期に設定することをお勧めしています。こうすることで、ザイリンクス ランタイムがデバイスのバッファ位置を決めやすくなります。このため、カーネル引数はエンキュー (たとえば `clEnqueueMigrateMemObjects`) を実行する前にバッファに設定するようにしてください。

FPGA デバイスのバッファ転送

ホスト アプリケーションとカーネル間の送受信方法は、デバイス内のグローバル メモリを転送されるデータによって異なります。FPGA とデータの送受信を実行するには、`clCreateBuffer`、`clEnqueueWriteBuffer` および `clEnqueueReadBuffer` コマンドを使用します。



推奨: ザイリンクスでは、`clEnqueueReadBuffer` および `clEnqueueWriteBuffer` ではなく、`clEnqueueMigrateMemObjects` を使用することをお勧めしています。

次にこのコード例を示します。

```
int host_mem_ptr[MAX_LENGTH]; // host memory for input vector
// Fill the memory input
for(int i=0; i<MAX_LENGTH; i++) {
    host_mem_ptr[i] = <... >
}
cl_mem dev_mem_ptr = clCreateBuffer(context, CL_MEM_READ_WRITE,
    sizeof(int) * number_of_words, NULL, NULL);

err = clEnqueueWriteBuffer(commands, dev_mem_ptr, CL_TRUE, 0,
    sizeof(int) * number_of_words, host_mem_ptr, 0, NULL, NULL);
```



重要: 1つのバッファが4 GBを超えないようにします。

注記: ホストからグローバルメモリのスループットを最大にする場合、かなり小さいバッファを送信するのはあまり効率的ではありません。ザイリンクスでは、バッファ サイズをできれば少なくとも2 MBに抑えることをお勧めしています。

単純なアプリケーションでは、ホストからデバイスのメモリにデータを転送するのに上記のコードで十分ですが、パフォーマンスおよび詳細な制御を最大にするために従う必要のあるコーディングプラクティスも多くあります。

clEnqueueMigrateMemObjects の使用

ザイリンクスでは、パフォーマンスを向上するために、`clEnqueueWriteBuffer` または `clEnqueueReadBuffer` の代わりに `clEnqueueMigrateMemObjects` を使用することをお勧めしています。`cl_mem` オブジェクトには、ホスト側ポインタとデバイス側ポインタの主に2つのポインタがあります。デバイス側ポインタは、カーネルが動作を開始する前に、デバイス側のメモリに暗示的に割り当てられる(たとえば、デバイスのグローバルメモリ内の特定位置に割り当てられる)ので、バッファがデバイスに含まれるようになりますが、

`clEnqueueMigrateMemObjects` を使用すると、この割り当てとデータ転送が前もって、カーネル実行よりもかなり前に発生するようになります。こうすることで、カーネルがまだ前のデータセットを処理している間に次のトランザクションのデータ転送を実行して、連続するカーネル実行のデータ転送レイテンシを隠すことができるので、ホストが同じカーネルを何度も実行する場合に、「ソフトウェアパイプライン」をイネーブルにするのに特に役立ちます。

次のコード例では、`clEnqueueMigrateMemObjects` を使用するように変更されています。

```
int host_mem_ptr[MAX_LENGTH]; // host memory for input vector

// Fill the memory input
for(int i=0; i<MAX_LENGTH; i++) {
    host_mem_ptr[i] = <... >
}

cl_mem dev_mem_ptr = clCreateBuffer(context,
    CL_MEM_READ_WRITE | CL_MEM_USE_HOST_PTR,
    sizeof(int) * number_of_words, host_mem_ptr, NULL);

clSetKernelArg(kernel, 0, sizeof(cl_mem), &dev_mem_ptr);

err = clEnqueueMigrateMemObjects(commands, 1, dev_mem_ptr, 0, 0,
    NULL, NULL);
```

ページ アライメントされたホスト メモリの割り当て

ザイリンクス ランタイムは、4K 境界のメモリ空間を割り当てて、内部メモリを管理します。ホスト メモリのポインターが ページ境界に揃っていない場合、ザイリンクス ランタイムが揃うように `memcpy` を追加で実行します。このため、ホスト メモリ ポインターを 4K 境界に揃えて、メモリ コピー操作が余分に実行されないようにしてください。

次に、ホストのメモリ空間に `malloc` ではなく `posix_memalign` を使用する例を示します。

```
int *host_mem_ptr; // = (int*) malloc(MAX_LENGTH*sizeof(int));
// Aligning memory in 4K boundary
posix_memalign(&host_mem_ptr, 4096, MAX_LENGTH*sizeof(int));

// Fill the memory input
for(int i=0; i<MAX_LENGTH; i++) {
    host_mem_ptr[i] = <... >
}

cl_mem dev_mem_ptr = clCreateBuffer(context,
                                    CL_MEM_READ_WRITE ,
                                    sizeof(int) * number_of_words, host_mem_ptr, NULL);

err = clEnqueueMigrateMemObjects(commands, 1, dev_mem_ptr, 0, 0,
                                NULL, NULL);
```

clEnqueueMapBuffer の使用

バッファを作成して管理するには、`clEnqueueMapBuffer` を使用する方法もあります。この方法を使用する場合、4K 境界に揃えられたホスト空間ポインターを作成する必要はありません。`clEnqueueMapBuffer` API は指定したバッファをマップして、ザイリンクス ランタイムで作成されたポインターをこのマップした領域に戻します。この後、ホスト側のポインターをユーザーのデータで埋めて、`clEnqueueMigrateMemObject` でデータをデバイスに送信したり受信したりします。次は、このスタイルを使用した例です。`CL_MEM_USE_HOST_PTR` が `clCreateBuffer` に使用されていないことに注目してください。

```
// Two cl_mem buffer, for read and write by kernel
cl_mem dev_mem_read_ptr = clCreateBuffer(context,
                                          CL_MEM_READ_ONLY,
                                          sizeof(int) * number_of_words, NULL, NULL);

cl_mem dev_mem_write_ptr = clCreateBuffer(context,
                                           CL_MEM_WRITE_ONLY,
                                           sizeof(int) * number_of_words, NULL, NULL);

// Setting arguments
clSetKernelArg(kernel, 0, sizeof(cl_mem), &dev_mem_read_ptr);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &dev_mem_write_ptr);

// Get Host side pointer of the cl_mem buffer object
auto host_write_ptr =
clEnqueueMapBuffer(queue, dev_mem_read_ptr, true, CL_MAP_WRITE, 0, bytes, 0, nullptr, nullptr, &err);
auto host_read_ptr =
clEnqueueMapBuffer(queue, dev_mem_write_ptr, true, CL_MAP_READ, 0, bytes, 0, nullptr, nullptr, &err);

// Fill up the host_write_ptr to send the data to the FPGA

for(int i=0; i< MAX; i++) {
```

```

    host_write_ptr[i] = <.... >
}

// Migrate
cl_mem mems[2] = {host_write_ptr, host_read_ptr};
clEnqueueMigrateMemObjects(queue, 2, mems, 0, 0, nullptr, &migrate_event));

// Schedule the kernel
clEnqueueTask(queue, kernel, 1, &migrate_event, &enqueue_event);

// Migrate data back to host
clEnqueueMigrateMemObjects(queue, 1, &dev_mem_write_ptr,
                           CL_MIGRATE_MEM_OBJECT_HOST, 1, &enqueue_event,
                           &data_read_event);

clWaitForEvents(1, &data_read_event);

// Now use the data from the host_read_ptr

```

デバイスでのバッファの割り当て

デフォルトでは、カーネルがリンクされる場合、すべてのカーネルからのメモリ インターフェイスが 1 つのデフォルトのグローバル メモリ バンクに接続されます。そのため、グローバル メモリ バンクとデータを転送できるのは一度に 1 つの計算ユニット (CU) のみになり、アプリケーションの全体的なパフォーマンスが制限されます。FPGA にグローバル メモリ バンクが 1 つしか含まれていない場合は、これしかオプションがありません。デバイスに複数のグローバル メモリ バンクが含まれる場合は、そのグローバル メモリ バンクの接続をカスタマイズして、デフォルトの接続を変更できます。詳細は、[カーネルポートとグローバルメモリの接続](#)を参照してください。複数のカーネル メモリ インターフェイスをイネーブルにして、別々のグローバル メモリ バンクからデータの読み出しおよび書き込みが同時にできるようにすることで、全体的なパフォーマンスが改善されます。

SDAccel 環境では、ホスト コードとカーネル コードは別々にコンパイルされます。ザイリンクス ランタイムは、カーネルのメモリ接続を検出して、データをホスト コードから正しいメモリ位置に送信する必要があります。最新の 2019.1 ザイリンクス ランタイムでは、`clSetKernelArgs` がバッファのエンキュー操作 (たとえば `clEnqueueMigrateMemObject`) よりも前に使用されると、自動的にカーネル バイナリ ファイルからバッファ位置を検出するようになっています。

2019.1 リリースより前のリリースの場合、OpenCL ホスト コードにザイリンクス拡張子 (`cl_mem_ext_ptr`) を使用して、デバイスのバッファ位置を指定する必要がありました。この方法はまだサポートはされていますが、もう必要ではないので、このガイドのバージョンでは説明しません。`cl_mem_ext_ptr` を使用してバッファ位置を指定する方法については、このガイドの以前のバージョンを参照してください。

サブバッファ

まれにですが、サブバッファを使用すると、特殊な状況で役に立つことがあります。次のセクションでは、サブバッファを使用すると利点がある例を示します。

デバイス バッファから特定部分を読み込む方法

カーネルへの入力によって、出力されるデータ量が異なることがあります。単純な例としては、入力データ パターンによって、出力サイズが変更する圧縮エンジンがあります。ホストは `clEnqueueMigrateMemObjects` を使用して出力バッファ全体を読み出すことはできますが、必要以上のメモリ転送が発生してしまうので、最適な方法ではありません。理想的なのは、ホストがカーネルの書き込んだのと同じ量のデータだけを読み出す方法です。このような場合にカーネルで使用可能な方法の 1 つに、出力データの開始時点で出力データのサイズ情報を書き込む方法があります。 `clEnqueueReadBuffer` を使用すると、ホスト コードが `clEnqueueReadBuffer` を 2 回使用できます (1 回目はデータのサイズを読み出して、2 回目で最初の読み出しからのサイズ情報を使用して同じ量のデータを読み出します)。

```
clEnqueueReadBuffer(command_queue, device_write_ptr, CL_FALSE, 0,
    sizeof(int) * 1,
    &kernel_write_size, 0, nullptr, &size_read_event);
clEnqueueReadBuffer(command_queue, device_write_ptr, CL_FALSE,
    DATA_READ_OFFSET,
    kernel_write_size, host_ptr, 1, &size_read_event,
    &data_read_event);
```

`clEnqueueMigrateMemObject` (`clEnqueueReadBuffer` または `clEnqueueWriteBuffer` よりも推奨) を使用すると、サブバッファを使用して同様の方法を使用できます。次にこのサンプル コードを示します (これは API 引数の一部です)。

```
// Create a small subbuffer
cl_buffer_region buffer_info_1 = {0, 1 * sizeof(int)};
cl_mem size_info = clCreateSubBuffer (device_write_ptr, CL_MEM_WRITE_ONLY,
    CL_BUFFER_CREATE_TYPE_REGION,
    &buffer_info_1, &err);
// Map the subbuffer into the host space
auto size_info_host_ptr = clEnqueueMapBuffer(queue, size_info, ..., );

// Read only the subbuffer portion
clEnqueueMigrateMemObjects(queue, 1,
    &size_info, CL_MIGRATE_MEM_OBJECT_HOST, ...);

// Retrieve size information from the already mapped size_info_host_ptr
kernel_write_size = .....

// Create sub-buffer again for required amount
cl_buffer_region buffer_info_2 = {DATA_READ_OFFSET, kernel_write_size};
cl_mem buffer_seg = clCreateSubBuffer (device_write_ptr,
    CL_MEM_WRITE_ONLY,
    CL_BUFFER_CREATE_TYPE_REGION, &buffer_info_2, &err);

// Map the subbuffer into the host space
auto read_mem_host_ptr = clEnqueueMapBuffer(queue, buffer_seg, ...);

// Migrate the subbuffer
clEnqueueMigrateMemObjects(queue, 1,
    &buffer_seg, CL_MIGRATE_MEM_OBJECT_HOST, ...);

// Now use the read data from already mapped read_mem_host_ptr
```

複数のメモリ ポートまたは複数のカーネルで共有されるデバイス バッファ

カーネルのメモリ ポートが少量のデータしか必要としないことがあります。この場合、複数の小さなサイズのバッファを管理および送信すると、パフォーマンスで問題となることがあります。この代わりに、ホストが大きなサイズのバッファを作成して、それを小さなサブバッファに分割するという方法があります。サブバッファはそれぞれメモリ ポートごとにカーネル引数を割り当て、少量のデータを必要とします。これにより、ザイリンクス ランタイムが複数の小型バッファではなく大型バッファを処理するようになるので、パフォーマンスが改善することがあります。

サブバッファが作成されたら、標準的なバッファと同様、ホスト コードで使用できるようになります。

カーネル実行

カーネルが FPGA の 1 つのハードウェア インスタンス (または CU) にコンパイルされる場合、次のように `clEnqueueTask` を使用するのがカーネルを実行する最も簡単な方法です。

```
err = clEnqueueTask(commands, kernel, 0, NULL, NULL);
```

ザイリンクス ランタイムがワークロード (カーネル引数を使用し、OpenCL バッファを介して渡されるデータ) を、カーネルが FPGA で計算負荷の高いタスクをスケジュールします。



重要: `clEnqueueNDRangeKernel` の使用はサポートされますが (OpenCL カーネルのみ)、ザイリンクスでは `clEnqueueTask` を使用することをお勧めしています。

カーネル、複数のカーネル、または FPGA の同じカーネルの複数インスタンスは、さまざまな方法で実行できます。これについては、次のセクションで説明します。

1 回のカーネル実行でタスク全体とデータ並列処理を実行

計算負荷の高いタスク全体を 1 つのカーネル内で定義して、カーネルを一度実行するだけでデータ範囲全体が処理されるようにすることがよくあります。複数のカーネル実行をするとオーバーヘッドができてしまうので、この方法の方が効率的なことが多くあります。この場合、カーネルを 1 回実行するだけでデータの範囲全体が処理されますが、並列処理 (したがってアクセラレーション) はカーネル ハードウェア内の FPGA で達成されます。ほとんどの場合、カーネルは命令レベルの並列処理 (ループ パイプライン) および関数レベルの並列処理 (データフロー) などのさまざまな方法を使用して並列処理を達成します。さまざまなコーディング方法については、[第3章: C/C++ カーネルのプログラミング](#) を参照してください。

ただし、前述の 1 つの `clEnqueueTask` はさまざまな現実的な理由から、常に実現可能なわけではありません。たとえば、カーネル コードですべての計算負荷の高いタスクを 1 つの実行で処理されるようにすると、大きく複雑になりすぎて、最適化ができないことがあります。また、ホストがデータを受信するのに時間がかかったり、すべてのデータを同時に受信しないこともあります。このため、次のセクションに示すように、状況およびアプリケーションによって、`clEnqueueTask` を使用してデータおよびタスクを複数の `clEnqueueTask` コマンドに分割します。

複数の異なるカーネルを使用したタスクの並列処理

FPGA で異なるタスクが並列で実行されるように複数のカーネルを設計できることがあります。複数の `clEnqueueTask` コマンドを順不同のコマンド キューを介して使用すると、異なるタスクを実行する複数のカーネルを FPGA で並列に実行されるようにできます。これにより、FPGA でタスクの並列処理ができるようになります。

空間的なデータ並列処理: 計算ユニット数を増加

1 つのカーネルが複数のハードウェア インスタンス (または CU) にコンパイルされた場合、`clEnqueueTask` を複数回呼び出して、データ並列処理をイネーブルにします。`clEnqueueTask` を呼び出すたびに、異なるデータ セットを並列で処理する別の CU でワークロードがスケジュールされます。

時間的なデータ並列処理: ホストからカーネルへのデータフロー

たとえば、FPGA 上の 1 つの CU しかカーネルに含まれない場合に、ホストが異なるデータ セットで CU を何度も使用する必要があるとします。[clEnqueueMigrateMemObjects の使用](#) に示すように、`clEnqueueMigrateMemObjects` を使用すると、データを次のカーネル実行のために送信する時間よりも前にデバイスのグローバル メモリに送信できるので、カーネル実行でデータ転送レイテンシを隠して、「ソフトウェア パイプライン」をイネーブルにできます。

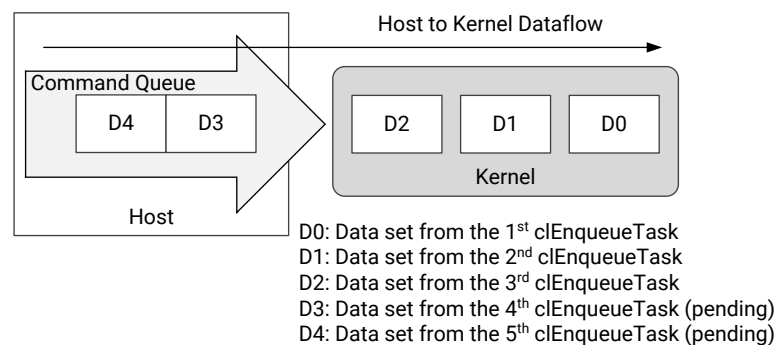
ただし、デフォルトでは、カーネルは現在のデータ セットの処理を終了してからのみ、次のデータ セットを処理し始めることができるようになっています。`clEnqueueMigrateMemObject` を使用すると、データ転送実行時間を隠すことはできますが、カーネル実行はシーケンシャルのままです。

ホストからカーネルのデータフローをイネーブルにすると、カーネルがまだ前のデータ セットを処理している間にカーネルを再開して、パフォーマンスをさらに改善することすら可能です。次のカーネル実行用に前のデータを処理している最中でも新しいデータを受信できるようにカーネルを最適化する場合 (これには [ホストからカーネルのデータフローのイネーブル](#) に示すようにカーネルを特定の 방법으로コンパイルする必要あり)、XRT はカーネルをできるだけ早期に再開して、複数のカーネル実行をオーバーラップさせます。

これにより、ホストとカーネル間の「時間的な並列処理」ができるようになり、カーネル ハードウェアの各セクションが、別の `clEnqueueTask` コマンドからの特定のデータ セットをパイプライン処理できるようになります。ただし、カーネルが次のデータ セットを受信できるようになったら即座に再開できるように、ホストはまだその時間 (ソフトウェア パイプライン) よりも前にコマンド キューを埋める必要があります。

次の図は、このホストからカーネルまでのデータフローの概念を示しています。

図 6: ホストからカーネルのデータフロー



X22774-042519

高度なデザインの場合は、空間的な並列処理 (より多くのハードウェアまたは CU を使用) とソフトウェア パイプライン (`clEnqueueMigrateMemObjects`) を時間的な並列処理と効率的に組み合わせて使用します。必要であれば、すべての方法を組み合わせることができます。

対称および非対称の計算ユニット

カーネル リンク プロセス中、カーネルには FPGA に複数の CU を含めることができます。

対称計算ユニット

計算ユニット (CU) は同じようにグローバル メモリに接続されていない場合 (同じ `--sp` オプションを使用していない場合)、非対称 CU として認識されます。非対称 CU として認識された場合は、ザイリンクス ランタイムでそれらを交互に使用できます。対称 CU のグループのいずれか 1 つのインスタンスを開始するには、`clEnqueueTask` を呼び出します。

非対称計算ユニット

計算ユニット (CU) は同じようにグローバル メモリに接続されていない場合 (同じ `--sp` オプションを使用していない場合)、非対称 CU として認識されます。同じ入力 (および出力) バッファ設定を使用した場合は、これらの CU の両方を交互に使用できませんので、ザイリンクス ランタイムで実行に依存することになります。

カーネル ハンドルおよび計算ユニット

`clSetKernelArg` が指定したカーネル オブジェクトに対して最初に呼び出されると、ザイリンクス ランタイムはこのカーネルの後に続く実行に対称 CU のグループを選択します。`clEnqueueTask` が呼び出されると、そのグループの対称 CU をどれでも使用できるようになります。

指定したカーネルの CU すべてが対称の場合は、1 つのカーネル オブジェクトだけでこれらのどの CU にでもアクセスできますが、非対称 CU がある場合は、アプリケーションがその非対称 CU グループ分のカーネル オブジェクトを作成して、すべてが使用できるようにする必要があります。

特定の計算ユニット用のカーネル オブジェクトの作成

2019.1 リリースから、ザイリンクス ランタイムに特殊な CU または CU のグループ用にカーネル ハンドルを取得する機能が追加されました。次は、その構文です。

```
// Create kernel object only for a specific compute unit
kernel1 = clCreateKernel(program, "<kernel_name_1>:{comp_unit_name_1}",
&err);

// Create kernel object for two specific compute units
kernel1 = clCreateKernel(program, "<kernel_name_1>:
{comp_unit_name_1,comp_unit_name_2}", &err);
```

これにより、アプリケーション内でどの特定の CU インスタンスを使用するか制御できます。これは、非対称 CU がある場合や、CU の明示的なロードおよび優先度管理を実行する場合に役立つことがあります。

計算ユニット名を使用して非対称計算ユニットすべてのハンドルを取得

カーネルに対称ではない CU も含まれる場合、CU 名の付いた `clCreateKernel` を使用できます。この場合、ホストで `cl_kernel` ハンドルを使用して各対称 CU グループが個別に管理されるようにする必要があります。次は、その例です。

mykernel カーネルに 5 つの CU (K1、K2、K3、K4、K5) が含まれるとします。K1、K2、および K3 という CU はデバイス上に対称接続があるので、1 つの対称 CU のグループで、K4 と K5 という CU も別の対称 CU のグループです。次のコード例は、2 つの `cl_kernel` ハンドルを使用して、この 2 つの対称 CU グループを管理するところを示しています。

```
// Kernel handle for Symmetrical compute unit group 1: K1,K2,K3

    cl_kernel kernel_handle1 = clCreateKernel(program,"mykernel:
{K1,K2,K3}",&err);

    for(i=0; i<3; i++) {
        // Creating buffers for the kernel_handle1
        .....
        // Setting kernel arguments for kernel_handle1
        .....
        // Enqueue buffers for the kernel_handle1
        .....
        // Possible candidates of the executions K1,K2 or K3
        clEnqueueTask(commands, kernel_handle1, 0, NULL, NULL);

        //
    }

// Kernel handle for Symmetrical compute unit group 1: K4, K5

    cl_kernel kernel_handle2 = clCreateKernel(program,"mykernel:
{K4,K5}",&err);

    for(int i=0; i<2; i++) {
        // Creating buffers for the kernel_handle2
        .....
        // Setting kernel arguments for kernel_handle2
        .....
        // Enqueue buffers for the kernel_handle2
        .....
        // Possible candidates of the executions K4 or K5
        clEnqueueTask(commands, kernel_handle2, 0, NULL, NULL);
    }
```

イベントの同期化

OpenCL `clEnqueueXXX` API 呼び出しはすべて非同期です。これらのコマンドはコマンドキューに追加されるとすぐに返されます。コマンド間のデータの依存性を解決するため、`clWaitForEvents` または `clFinish` などの API 呼び出しを使用してホストプログラムの実行を停止またはブロックできます。

次に、`clWaitForEvents` および `clFinish` コマンドの使用例を示します。

```
err = clEnqueueTask(command_queue, kernel, 0, NULL, NULL);

// Execution will wait here until all commands in the command queue are
// finished
clFinish(command_queue);

// Read back the results from the device to verify the output
cl_event readevent;
int host_mem_output_ptr[MAX_LENGTH]; // host memory for output vector

clEnqueueReadBuffer(command_queue, dev_mem_ptr, CL_TRUE, 0, sizeof(int) *
number_of_words,
```

```
host_mem_output_ptr, 0, NULL, &readevent );

clWaitForEvents(1, &readevent); // Wait for clEnqueueReadBuffer event to
finish

// Check Results
// Compare Golden values with host_mem_output_ptr
```

上記の例に同期化 API がどのように追加されているかに注目してください。

1. `clFinish` API は、カーネルの実行が終了するまでホストが実行されないようにするために使用されています。このようにしないと、ホストが FPGA バッファからデータを読み出すのが早すぎ、無効なデータが読み出される可能性があります。
2. FPGA メモリからローカル ホスト マシンへのデータ転送は、`clEnqueueReadBuffer` を使用して実行されます。`clEnqueueReadBuffer` の最後の引数は、この特定の読み出しコマンドを識別して、イベントをクエリするのに使用可能な (またはこの特定コマンドが終了するまで待機する) イベント オブジェクトを返します。`clWaitForEvents` はその 1 つのイベントを指定し、データ転送が終了してからホスト側のメモリからのデータがチェックされるように待機します。

後処理および FPGA のクリーンアップ

ホスト コードの最後には、適切な解放関数を使用して、割り当てられたリソースをすべて解放する必要があります。リソースが適切に解放されないと、SDAccel 環境で正しいパフォーマンス関連のプロファイルおよび解析レポートを生成できないことがあります。

```
clReleaseCommandQueue(Command_Queue);
clReleaseContext(Context);
clReleaseDevice(Target_Device_ID);
clReleaseKernel(Kernel);
clReleaseProgram(Program);
free(Platform_IDs);
free(Device_IDs);
```

まとめ

前のトピックで説明したように、SDAccel 環境で推奨されるホスト アプリケーションのコーディング スタイルは次のとおりです。

1. 必要に応じて、デバッグ用に各 OpenCL API 呼び出しの後にエラー チェックを追加します。
2. SDAccel 環境では、1 つまたは複数のカーネルは個別に `.xclbin` ファイルにあらかじめコンパイルされます。カーネル バイナリからプログラムをビルドするには、`clCreateProgramWithBinary` API を使用します。
3. バッファのエンキュー操作前にカーネル引数 (`clSetKernelArg`) を設定するバッファを使用します。
4. ホスト コードと FPGA の間のデータ転送には `clEnqueueMigrateMemObjects` を使用します。
5. `posix_memalign` を使用してホスト メモリ ポインターを 4K 境界に合わせます。
6. FPGA で同時実行コマンドを実行する順不同コマンド キューを使用することをお勧めします。
7. 同期化コマンドを使用して、非同期 OpenCL API 呼び出しの依存性を解決します。

C/C++ カーネルのプログラミング

SDAccel™ 環境では、カーネル コードはアルゴリズムの計算負荷の高い部分であり、FPGA でアクセラレーションすることが意図されています。SDAccel 環境では、C、OpenCL™、および RTL で記述されたカーネル コードがサポートされます。このガイドでは、主に C カーネルのコーディング スタイルを説明します。

ランタイム中は、C/C++ カーネル実行ファイルがホスト コード実行ファイルから呼び出されます。ホスト コードとカーネル コードを別々に開発およびコンパイルすると、片方が C で、もう片方が C++ で記述された場合に、命名方法に違いがでる可能性があります。この問題を回避するには、カーネル関数宣言をヘッダー ファイル内に記述して、`extern "C"` リンケージで囲むことをお勧めします。

```
extern "C" {  
    void kernel_function(int *in, int *out, int size);  
}
```

データ型

`int`、`float`、`double` などのネイティブでサポートされている C データ型を使用した方がコードを短時間で記述および検証できるので、最初にコードを記述する際はこれらのデータ型を使用するのが一般的です。ただし、コードはハードウェアにインプリメントされるので、ハードウェアで使用されるすべての演算子のサイズはアクセラレータ コードで使用するデータ型によります。そのため、デフォルトのネイティブ C/C++ データ型を使用すると、ハードウェア リソースが大きく低速なものとなり、カーネルのパフォーマンスが制限される可能性があります。その代わりにビット精度のデータ型を使用して、コードがハードウェアのインプリメンテーション用に最適化されるようにします。ビット精度または任意精度のデータ型を使用すると、小型で高速なハードウェア演算子が得られます。これにより、より多くのロジックをプログラマブル ロジックに配置できるようになり、また消費電力を抑えつつロジックをより高いクロック周波数で実行できるようになります。

コードでネイティブの C/C++ データ型ではなくビット精度のデータ型を使用することを考慮してください。



推奨: コードでネイティブの C/C++ データ型ではなくビット精度のデータ型を使用することを考慮してください。

次のセクションで、`xocc` コンパイラによりサポートされる最も一般的な任意精度データ型 (任意精度整数型および任意精度固定小数点型) について説明します。これらのデータ型は C/C++ のカーネルにのみ使用し、OpenCL カーネル (またはホスト コード内) には使用しないようにしてください。

任意精度整数型

任意精度の整数データ型は、ヘッダー ファイル `ap_int.h` で符号付きの整数場合は `ap_int`、符号なし整数の場合は `ap_uint` と定義されています。任意精度の整数データ型を使用するには、次の手順に従います。

- ソース コードにヘッダー ファイル `ap_int.h` を追加します。
- ビット型を、`ap_int<N>` または `ap_uint<N>` (`N` はビット サイズを表す 1 ~ 1024 の値) に変更します。

次の例に、ヘッダー ファイルの追加方法と、2 つの変数を 9 ビット整数および 10 ビットの符号なし整数を使用してインプリメントする方法を示します。

```
#include "ap_int.h"
ap_int<9> var1 // 9 bit signed integer
ap_uint<10> var2 // 10 bit unsigned integer
```

任意精度固定小数点データ型

既存アプリケーションの中には、ほかのハードウェア アーキテクチャ用に記述されているために、浮動小数点データ型が使用されているものもあります。ただし、固定小数点型の方が、終了するのに多くのクロック サイクルを必要とする浮動小数点型よりも向いていることもあります。アプリケーションおよびアクセラレータに浮動小数点型を使用するのか、固定小数点型を使用するのか選択する際には、消費電力、コスト、生産性、および精度などのトレードオフに注意してください。

『ザイリンクス デバイスでの INT8 に最適化した深層学習の実装』(WP486) で説明するように、機械学習のようなアプリケーションに浮動小数点ではなく固定小数点演算を使用すると、消費電力効率が上がり、必要な消費電力合計を削減できます。浮動小数点型の範囲がすべて必要な場合を除き、固定小数点型で同じ精度をインプリメントでき、より小型で高速なハードウェアにできることがよくあります。この変換方法の例は、『浮動小数点から固定小数点への変換による消費電力およびコストの削減』(WP491) を参照してください。

固定小数点データ型では、整数ビットおよび少数ビットとしてデータが記述されます。固定小数点データ型には、`ap_fixed` ヘッダーが必要で、符号付きと符号なしの両方がサポートされます。

- ヘッダー ファイル: `ap_fixed.h`
- 符号付き固定小数点: `ap_fixed<W,I,Q,O,N>`
- 符号なし固定小数点: `ap_ufixed<W,I,Q,O,N>`
 - `W` = 合計幅 <1024 ビット
 - `I` = 整数ビット幅。`I` の値は幅 (`W`) と等しいかそれ以下である必要があります。小数部を表すためのビット数は `W` から `I` を差し引いた値です。整数幅を指定するには、定数の整数式のみを使用します。
 - `Q` = 量子化モード。`Q` の指定には、あらかじめ定義されている列挙値のみを使用できます。使用できる値は、次のとおりです。
 - `AP_RND`: 正の無限大への丸め。
 - `AP_RND_ZERO`: 0 への丸め。
 - `AP_RND_MIN_INF`: 負の無限大への丸め。
 - `AP_RND_INF`: 無限大への丸め。
 - `AP_RND_CONV`: 収束丸め。
 - `AP_TRN`: 切り捨て。`Q` を指定しない場合、これがデフォルト値です。
 - `AP_TRN_ZERO`: 0 への切り捨て。
 - `O` = オーバーフロー モード。`Q` の指定には、あらかじめ定義されている列挙値のみを使用できます。使用できる値は、次のとおりです。
 - `AP_SAT`: 飽和。
 - `AP_SAT_ZERO`: 0 への飽和。
 - `AP_SAT_SYM`: 対称飽和。
 - `AP_WRAP`: 折り返し。`O` を指定しない場合、これがデフォルト値です。

- AP_WRAP_SM: 符号絶対値の折り返し。
- 。 N = オーバーフロー WRAP モードでの飽和ビット数。このパラメーター値には定数の整数式のみを使用します。デフォルト値は0です。



ヒント: `ap_fixed` および `ap_ufixed` データ型には省略定義を使用でき、`W` および `I` だけが必須で、その他のパラメーターはデフォルト値に割り当てられます。ただし、`Q` または `N` を定義する場合は、デフォルト値だけを指定した場合でも、これらの前にパラメーターを指定する必要があります。

次のコード例では、`ap_fixed` 型を使用して、符号付き 18 ビット変数 (6 ビットが 2 進小数点より上位の整数部で、12 ビットが 2 進小数点より下位の小数部) を定義しています。量子化モードは正の無限大 (`AP_RND`) へ丸めらるよう設定されています。オーバーフロー モードと飽和ビットが指定されていないので、デフォルトの `AP_WRAP` と 0 が使用されます。

```
#include <ap_fixed.h>
...
    ap_fixed<18,6,AP_RND> my_type;
...
```

変数に異なるビット数 (`W`) や精度 (`I`) が含まれるような計算を実行する場合は、2 進小数点が自動的に揃えられます。固定小数点データ型の詳細は、『Vivado Design Suite ユーザー ガイド: 高位合成』(UG902)の「C++ の任意精度固定小数点型」を参照してください。

インターフェイス

ホスト マシンと FPGA デバイス上のカーネルの間のデータ転送には、グローバル メモリ バンクを介したデータ転送とスカラー データ転送があります。

メモリのデータ入力および出力

カーネルでの計算で処理されるメインのデータは通常大量であり、FPGA ボード上のグローバル メモリ バンクを介して転送する必要があります。ホスト マシンは、データの大きな塊を 1 つまたは複数のグローバル メモリ バンクに転送します。カーネルは、これらのグローバル メモリ バンクからデータに、理想的にはバーストでアクセスします。カーネルが計算を終了すると、結果のデータがグローバル メモリ バンクを介してホスト マシンに戻されます。

カーネル インターフェイスを記述する際、グローバル メモリ バンクに対するインターフェイスはプラグマを使用して記述します。

メモリ データ

```
void cnn( int *pixel, // Input pixel
          int *weights, // Input Weight Matrix
          int *out, // Output pixel
          ... // Other input or Output ports

#pragma HLS INTERFACE m_axi port=pixel offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=weights offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=out offset=slave bundle=gmem
```

上記の例では、3 つの大型データ インターフェイスがあります。`pixel` および `weights` という 2 つの入力と、`out` という 1 つの出力です。グローバル メモリ バンクに接続されたこれらの入力と出力は、`HLS INTERFACE m_axi` プラグマを使用して C コードで指定されています。

bundle キーワードでは、ポート名を指定します。コンパイラは一意のバンドル名ごとにポートを作成します。同じ名前を別のインスタンスに使用した場合は、それらのインターフェイスが同じポートにマップされます。

ポートを共有すると、FPGA リソースは節約できますが、すべてのメモリ転送が1つのポートを通ることになるので、カーネルのパフォーマンスが制限されてしまうことがあります。複数のポートを別のバンドル名を使用して作成すると、カーネルの帯域幅とスループットを増加できます。

```
void cnn( int *pixel, // Input pixel
         int *weights, // Input Weight Matrix
         int *out, // Output pixel
         ... // Other input or Output ports

#pragma HLS INTERFACE m_axi port=pixel offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=weights offset=slave bundle=gmem1
#pragma HLS INTERFACE m_axi port=out offset=slave bundle=gmem
```

前述の例の場合、bundle 属性を使用して gmem と gmem1 の2つのポートが作成されています。カーネルは gmem ポートを介して pixel および out にアクセスしますが、weights には gmem1 ポートを介してアクセスされます。このため、カーネルが pixel と weights に並列にアクセスできるようになるので、カーネルのスループットが改善される可能性があります。

この可能性に留意して、これらの異なるポートを異なるグローバル メモリ バンクに接続する必要もあります。これは、--sp オプションを使用して xocc リンク段階で実行されます。このオプションの詳細は、[システム アーキテクチャの設定](#)を参照してください。

メモリ インターフェイスのデータ幅に関する考慮事項

SDAccel 環境では、グローバル メモリとカーネルの間の最大データ幅は 512 ビットです。データ転送レートを最大にするには、この最大データ幅を使用することをお勧めします。最大ビット幅を利用するには、カーネル コードを変更する必要があります。

上記の例では、ネイティブの整数型が使用されているので、データ転送に最大帯域幅は使用されません。[データ型](#)に説明されているように、任意精度型 ap_int または ap_uint を使用すると、ビット精度の高いデータ幅を達成できます。

```
void cnn( ap_uint<512> *pixel, // Input pixel
         int *weights, // Input Weight Matrix
         ap_uint<512> *out, // Output pixel
         ... // Other input or output ports

#pragma HLS INTERFACE m_axi port=pixel offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=weights offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=out offset=slave bundle=gmem
```

上記の例では、ap_uint データ型を使用した出力(out) インターフェイスにより、512 ビットの最大転送幅が使用されるようになっています。

メモリ インターフェイスのデータ幅は2のべき乗で指定する必要があります。データ幅が32未満の場合は、ネイティブ C++ データ型を使用します。32を超え、2のべき乗で増加していく場合は、ap_int/ap_uint を使用します。

バースト読み出しおよび書き込み

カーネルからグローバル メモリ バンク インターフェイスへのアクセスには、長いレイテンシがあります。そのため、グローバル メモリ データ転送はバーストで実行する必要があります。バーストを推論するには、次に示すように、パイプライン ループ コーディング スタイルをお勧めします。

```
hls::stream<datatype_t> str;

INPUT_READ: for(int i=0; i<INPUT_SIZE; i++) {
    #pragma HLS PIPELINE
    str.write(inp[i]); // Reading from Input interface
}
```

上記のコード例では、パイプライン処理された for ループが使用され、入力メモリ インターフェイスからデータを読み出し、内部 `hls::stream` 変数に書き込んでいます。上記のコーディング スタイルにより、データはグローバル メモリ バンクからバーストで読み出されます。

上記の for ループを異なる関数内に含め、最上位で dataflow 最適化を適用するコーディング スタイルをお勧めします。

```
top_function(datatype_t * m_in, // Memory data Input
             datatype_t * m_out, // Memory data Output
             int inp1,          // Other Input
             int inp2) {        // Other Input
    #pragma HLS DATAFLOW

    hls::stream<datatype_t> in_var1; // Internal stream to transfer
    hls::stream<datatype_t> out_var1; // data through the dataflow region

    read_function(m_in, inp1); // Read function contains pipelined for loop
                                // to infer burst

    execute_function(in_var1, out_var1, inp1, inp2); // Core compute function

    write_function(out_var1, m_out); // Write function contains pipelined for
    loop
                                // to infer burst
}
```



ヒント: データフロー最適化については、後で説明します。

スカラー データ入力

スカラー入力は通常、ホスト マシンから直接読み込まれる制御変数です。これらは、メイン カーネルの計算が実行されるプログラム データまたはパラメーターと考えることができます。これらのカーネル入力は、ホスト側からの書き込みのみです。これらのインターフェイスは、カーネル コードで次のように指定します。

```
void process_image(int *input, int *output, int width, int height) {
    #pragma HLS INTERFACE s_axilite port=width bundle=control
    #pragma HLS INTERFACE s_axilite port=height bundle=control
}
```

上記の例では、width および height を指定する 2 つのスカラー入力があります。これらの入力は、`#pragma HLS INTERFACE s_axilite` を使用して指定します。これらのデータ入力は、グローバル メモリ バンクを使用せずに、ホスト マシンから直接カーネルに送信されます。



重要: 現在のところ、SDAccel 環境では各カーネルに 1 つの制御インターフェイス バンドルのみがサポートされています。そのため、すべてのスカラー データ入力の `bundle` 名は同じにする必要があります。前の例では、同じ `bundle` 名の `control` がすべての制御入力に対して使用されています。

ホストからカーネルのデータフローのイネーブル

カーネル実行 で説明したように、カーネルが前のトランザクションからのデータを処理中にさらにデータを受信できる場合、SDAccel ランタイムは次のデータ バッチを送信できます。これで、カーネルがハードウェアの別の部分で同時に複数のデータ セットを処理するようになるので、レイテンシおよびパフォーマンスが改善しますが、次の `ap_ctrl_chain` プラグマを使用してカーネルをコンパイルする必要があります。

```
void kernel_name( int *inputs,
    ... )// Other input or Output ports
{
#pragma HLS INTERFACE ..... // Other interface pragmas
#pragma HLS INTERFACE ap_ctrl_chain port=return bundle=control
```



重要: ホストからカーネルのパイプラインの利点を生かすには、カーネルをループ レベルおよびタスク レベルでパイプライン処理する必要があります (ループ パイプラインおよびデータフローなどのカーネル パイプライン方法に関する情報は、この章の後半で説明します)。

ループ

ループは、高パフォーマンスのアクセラレータには重要です。ループは通常、高度に分散された並列 FPGA アーキテクチャを利用するためパイプライン処理されるか展開され、CPU で実行するよりもパフォーマンスが上がります。

デフォルトでは、ループはパイプライン処理も展開もされません。ハードウェアでは、ループの各反復を実行するのに少なくとも 1 クロック サイクルかかります。ハードウェアの面から考えると、ループの本体ではクロックまで待機することが暗示されます。ループの次の反復は、前の反復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

デフォルトでは、ループの反復は前の反復が終了してから開始されます。たとえば次に示すループの例では、ループの 1 回の反復で 2 つの変数が加算され、結果が 3 つ目の変数に格納されます。ハードウェアでこのループの 1 回の反復を終了するのに 3 サイクルかかるとします。また、ループの変数 `len` は 20 だとします (カーネルで `vadd` ループが 20 回反復実行される)。このループのすべての演算を終了するには、合計 60 クロックサイクル (20 反復 * 3 サイクル) かかります。

```
vadd: for(int i = 0; i < len; i++) {
    c[i] = a[i] + b[i];
}
```



ヒント: ループには、上記のコード例のように常にラベルを付けることをお勧めします (`vadd:...`)。このようにしておくと、SDAccel 環境でデバッグしやすくなります。ラベルが原因でコンパイル中に警告メッセージが表示されますが、無視しても問題ありません。

ループをパイプライン処理すると、後続の反復はパイプラインで実行されます。つまり、ループの後続の反復は重複させて、ループ本体の別のセクションで同時に実行されます。ループをパイプライン処理するには、HLS PIPELINE プラグマをイネーブルにします。プラグマはループの本体内に記述します。

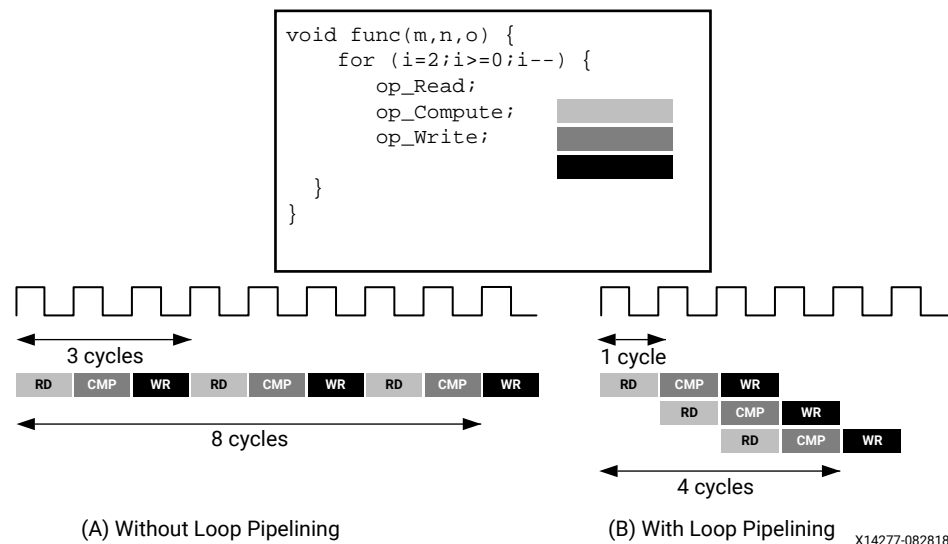
```
vadd: for(int i = 0; i < len; i++) {
    #pragma HLS PIPELINE
    c[i] = a[i] + b[i];
}
```

上記の例では、ループの各反復 (読み込み、追加、書き出し) に 3 サイクルかかっていました。パイプライン処理なしの場合、ループの後続の反復は 3 サイクルごとに開始します。パイプライン処理すると、ループの後続の反復が 2 サイクルごと、または各サイクルで開始するようにできます。

ループの次の反復を開始するまでにかかるサイクル数は、パイプライン ループの開始間隔 (II) と呼ばれます。II=2 はループの連続する反復が 2 サイクルごとに開始され、II=1 (理想的) はループの各反復が各サイクルで開始されることを意味します。pragma HLS PIPELINE を使用すると、コンパイラは常に II=1 のパフォーマンスを達成しようとします。

次の図は、パイプライン処理ありとなしのループの違いを示しています。この図の (A) はデフォルトの順次演算を示しています。各入力は 3 クロック サイクルごとに処理され (II=3)、最後の出力が書き出されるまでに 8 クロック サイクルかかっています。

図 7: ループのパイプライン処理



(B) に示すパイプライン処理されたループでは、入力サンプルが各クロック サイクルで読み出され、最終的な出力は 4 クロック サイクル後に書き込まれるようになります。同じハードウェア リソースを使用して、開始間隔 (II) とレイテンシの両方を向上できます。



重要: ループをパイプライン処理すると、そのループ内の入れ子のループがすべて 展開されます。

ループ内にデータ依存性がある場合、II=1 を達成できず、開始間隔が 1 より大きな値になることがあります。ループのデータ依存性については、[ループ依存性](#)を参照してください。

ループ展開

コンパイラでは、ループを部分的または完全に展開し、複数のループ反復を並列で実行するようにもできます。これには、`HLS_UNROLL` プラグマを使用します。ループを展開すると、並列処理が多くなるので、デザインをかなり高速にできますが、ループ反復の演算すべてが並列で実行されるので、ハードウェアをインプリメントするのに多くのプログラマブル ロジック リソースが必要になります。この結果、リソース量が多くなって容量の問題が発生し、カーネル コンパイル処理時間が遅くなってしまうことがあります。そのため、ループ本体が小さいループ、または反復回数の少ないループを展開することをお勧めします。

```
vadd: for(int i = 0; i < 20; i++) {
    #pragma HLS UNROLL
    c[i] = a[i] + b[i];
}
```

上記の例では、ループ本体に `pragma HLS UNROLL` が追加されており、コンパイラにループを完全に展開するよう指示しています。データの依存性で許容されれば、ループの 20 回の反復すべてが並列実行されます。

ループを完全に展開すると多量のデバイス リソースが使用されますが、部分的に展開するだけの場合、ハードウェア リソースに大きな影響を与えずにパフォーマンスを向上させることができます。

部分展開されたループ

ループを完全に展開するには、ループの範囲が定数である必要があります (上記の例では 20)。ただし、範囲が可変のループでも、部分展開は可能です。ループを部分展開するということは、特定数の反復のみが並列実行されるということです。

次のコード例は、部分展開を示しています。

```
array_sum:for(int i=0;i<4;i++){
    #pragma HLS UNROLL factor=2
    sum += arr[i];
}
```

上記の例では、`UNROLL` プラグマに係数 2 が指定されています。これは、ループ本体を手動で複製し、その 2 つのループを半分の反復回数分だけ同時に実行するのと同じことです。次のコードは、その例です。この変換により、上記のループの 2 回の反復が並列に実行されます。

```
array_sum_unrolled:for(int i=0;i<2;i+=2){
    // Manual unroll by a factor 2
    sum += arr[i];
    sum += arr[i+1];
}
```

ループ内のデータの依存性がパイプライン処理されたループの開始間隔 (II) に影響するのと同様に、展開されたループでも、データの依存性により可能な場合にのみの演算が並列実行されます。ループの 1 回の反復での演算に前の反復からの結果が必要な場合は並列実行できませんが、1 つの反復からのデータが使用可能になるとすぐに次の反復が実行されます。



推奨: まず、`PIPELINE` でループをパイプライン処理をしてから、小型のループ本体を `UNROLL` で限られた反復回数分展開して、パフォーマンスを改善していくことをお勧めします。

ループ依存性

ループ内のデータ依存性は、ループのパイプライン処理またはループの展開結果に影響することがあります。これらのループ依存性は、ループの1回の反復内またはループ内の異なる反復間で発生します。ループ依存性を理解するには、極端な例を見てみるのとわかりやすいです。次のコード例では、ループの結果がそのループの継続/終了条件として使用されています。次のループを開始するには、前のループの各反復が終了する必要があります。

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

このループはパイプライン処理できません。ループの前の反復が終了するまで次の反復を開始できないからです。

xocc を使用してさまざまなタイプの依存性を処理するには、コンパイラの結果を使用する高位合成について詳細に理解しておく必要があります。Vivado HLS での依存性の詳細は、『Vivado Design Suite ユーザー ガイド: 高位合成』(UG902) を参照してください。

入れ子のループ

入れ子のループはコード記述で一般的に使用されます。入れ子のループ構造内のループがどのようにパイプライン処理されるか理解することが、必要なパフォーマンスの達成につながります。

HLS PIPELINE プラグマが別のループ内で入れ子になったループに適用されると、xocc コンパイラはループを平坦化して1つのループを作成し、PIPELINE プラグマをその作成されたループに適用します。ループを平坦化すると、カーネル パフォーマンスを改善しやすくなります。

コンパイラでは、次のタイプの入れ子のループを平坦化できます。

1. 完全入れ子ループ:
 - 内側のループのみにループ本体が含まれます。
 - ループ宣言間に指定されるロジックまたは演算はありません。
 - すべてのループ範囲は定数です。
2. 半完全入れ子ループ:
 - 内側のループのみにループ本体が含まれます。
 - ループ宣言間に指定されるロジックまたは演算はありません。
 - 内側のループ範囲は定数にする必要がありますが、外側のループ範囲は変数にできます。

次のコード例は、完全入れ子ループの構造を示しています。

```
ROW_LOOP: for(int i=0; i< MAX_HEIGHT; i++) {
    COL_LOOP: For(int j=0; j< MAX_WIDTH; j++) {
        #pragma HLS PIPELINE
        // Main computation per pixel
    }
}
```

上記の例は、入力ピクセルデータに対して計算を実行する2つのループを含む入れ子のループ構造を示しています。ほとんどの場合、サイクルごとに1ピクセル処理するのが望ましいので、PIPELINEは入れ子のループの本体構造に適用されます。この例の場合は完全入れ子ループなので、コンパイラで入れ子ループ構造を平坦化できます。

前の例の入れ子ループには、2つのループ宣言間にロジックが含まれていません。ROW_LOOPとCOL_LOOPの間にロジックはないので、すべての処理ロジックがCOL_LOOPに含まれます。また、両方のループの反復数は固定されています。これら2つの条件を満たすことが、xoccコンパイラでループを平坦化し、PIPELINE制約を適用するのに役立ちます。



推奨: 外側のループの境界が変数の場合でも、コンパイラでループを平坦化することは可能です。内側のループの範囲は、定数にするようにしてください。

シーケンシャル ループ

デザインに複数のループがある場合、デフォルトではこれらがオーバーラップせずに順に実行されます。このセクションでは、シーケンシャルループのデータフロー最適化の概念について説明します。次のようなコードがあるとします。

```
void adder(unsigned int *in, unsigned int *out, int inc, int size) {

    unsigned int in_internal[MAX_SIZE];
    unsigned int out_internal[MAX_SIZE];
    mem_rd: for (int i = 0 ; i < size ; i++){
        #pragma HLS PIPELINE
        // Reading from the input vector "in" and saving to internal variable
        in_internal[i] = in[i];
    }
    compute: for (int i=0; i<size; i++) {
        #pragma HLS PIPELINE
        out_internal[i] = in_internal[i] + inc;
    }

    mem_wr: for(int i=0; i<size; i++) {
        #pragma HLS PIPELINE
        out[i] = out_internal[i];
    }
}
```

上記の例は、mem_rd、compute、およびmem_wrの3つのシーケンシャルループを示しています。

- mem_rd ループはメモリ インターフェイスから入力ベクター データを読み出し、内部ストレージに格納します。
- compute ループは、内部ストレージからデータを読み出してインクリメント演算を実行し、結果を別の内部ストレージに保存します。
- mem_wr ループは、内部ストレージからデータを読み出してメモリに書き込みます。

メモリのデータ入力および出力に示すように、このコード例ではメモリの入力/出力インターフェイスに対する読み出しと書き込みに2つの個別のループを使用してバースト読み出し/書き込みを推論しています。

デフォルトでは、これらのループはオーバーラップせずに順に実行されます。mem_rd ループが入力データをすべて読み出すまで、compute ループの処理は開始しません。同様に、compute ループがデータを処理し終わってからmem_wr ループがデータの書き込みを開始します。ただし、これらのループの処理をオーバーラップさせて、mem_rd (または compute) ループがデータを処理し終えるまで待たずに、compute (または mem_wr) ループが処理を実行するのに十分なデータが使用可能になったらすぐに開始されるようにできます。

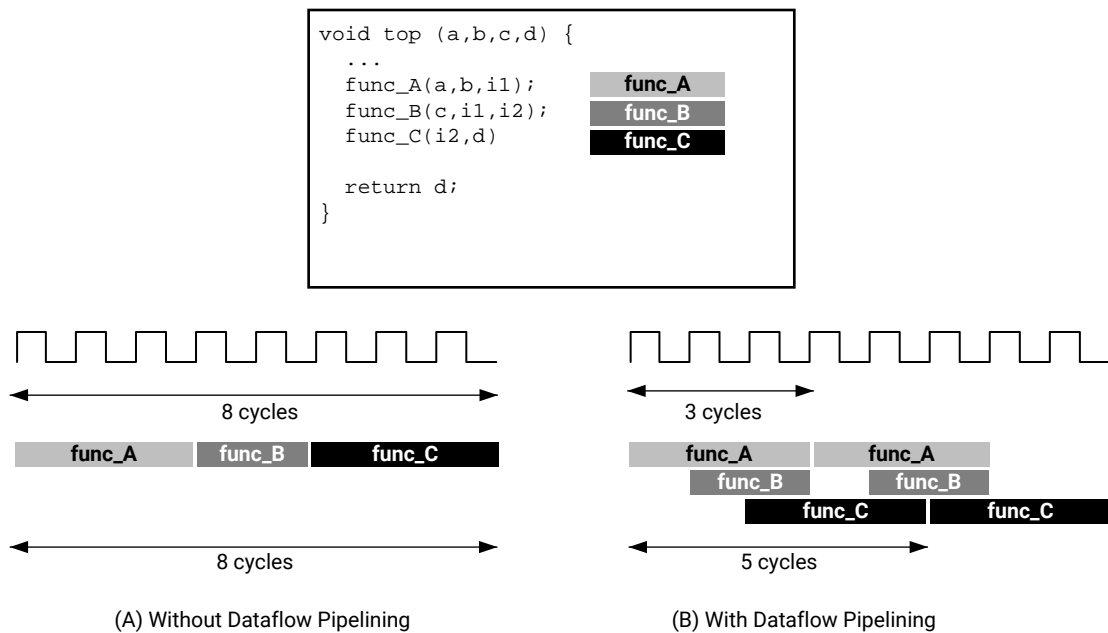
ループの実行は、**データフロー最適化**に示すように、データフロー最適化を使用してオーバーラップさせることができます。

データフロー最適化

データフロー最適化は、カーネル ハードウェア関数タスク レベルのパイプラインおよび並列処理をイネーブルにすることで、カーネルのパフォーマンスを改善する優れた手法で、スループットを高めてレイテンシを下げるために、xocc コンパイラで複数のカーネル関数が同時実行されるようスケジュールできるようになります。これは、タスクレベルの並列処理とも呼ばれます。

次の図に、データフロー パイプライン処理の概念を示します。デフォルトでは、func_A、func_B、func_C の順に実行されて終了しますが、HLS DATAFLOW プラグマをイネーブルにすると、データが使用可能になった直後に各関数が実行されるようスケジュールできます。この例の場合、元の top 関数のレイテンシと間隔は 8 クロック サイクルです。DATAFLOW 最適化を使用すると、間隔は 3 クロック サイクルに削減されます。

図 8: データフロー最適化



X14266-083118

データフローのコード例

データフローのコード例では、次に注意してください。

- コンパイラでデータフロー最適化をイネーブルにするため、HLS DATAFLOW プラグマが適用されています。これは、PS と PL 間のインターフェイスであるデータ ムーバーではなく、アクセラレータを介したデータのフローを示します。
- データフロー領域内の各関数間のデータ転送チャネルとして stream クラスが使用されています。



ヒント: `stream` クラスは、プログラマブルの FIFO メモリ回路を推論します。このメモリ回路はソフトウェアプログラミングのキューとして動作し、関数間をデータレベルで同期化してパフォーマンスを向上します。
`hls::stream` クラスの詳細は、『Vivado Design Suite ユーザーガイド: 高位合成』(UG902) を参照してください。

```
void compute_kernel(ap_int<256> *inx, ap_int<256> *outx, DTYPE alpha) {
    hls::stream<unsigned int>inFifo;
    #pragma HLS STREAM variable=inFifo depth=32
    hls::stream<unsigned int>outFifo;
    #pragma HLS STREAM variable=outFifo depth=32

    #pragma HLS DATAFLOW
    read_data(inx, inFifo);
    // Do computation with the acquired data
    compute(inFifo, outFifo, alpha);
    write_data(outx, outFifo);
    return;
}
```

データフロー最適化の正規形式

ザイリンクスでは、正規形式を使用してデータフロー領域内でコードを記述することをお勧めしています。関数およびループのデータフロー最適化には、正規形式があります。

- 関数: 関数内のデータフローの正規形式のコーディングガイドラインは、次のとおりです。
 1. データフロー領域内では、次のタイプの変数のみを使用します。
 - a. ローカルの非スタティック スカラー/配列/ポインター変数。
 - b. ローカルのスタティック `hls::stream` 変数。
 2. 関数呼び出しは、データを前方向にのみ送信します。
 3. 配列または `hls::stream` には、プロデューサー関数 1 つと コンシューマー関数 1 つのみが含まれます。
 4. 関数引数 (データフロー領域外から入ってくる変数) は読み出されるか書き込まれますが、両方が実行されることはありません。読み出しと書き込みを同じ関数引数で実行する場合は、読み出しが書き込みよりも前に発生する必要があります。
 5. ローカル変数 (前方向へのデータ転送する変数) は読み出しよりも前に書き込まれる必要があります。

次のコード例は、関数内のデータフローの正規形式を示しています。上記のコードでは、最初の関数 (`func1`) で入力を読み出され、最後の関数 (`func3`) で出力が書き込まれます。各関数で作成された出力値は、次の関数に入力パラメーターとして渡されます。

```
void dataflow(Input0, Input1, Output0, Output1) {
    UserDataTpe C0, C1, C2;
    #pragma HLS DATAFLOW
    func1(read Input0, read Input1, write C0, write C1);
    func2(read C0, read C1, write C2);
    func3(read C2, write Output0, write Output1);
}
```

- ループ: ループ本体内のデータフローの正規形式のコーディングガイドラインには、上記で定義される関数のコーディングガイドラインが含まれるほか、次も指定されます。
 1. 初期値は 0。
 2. ループ条件は、ループ変数と定数、またはループ本体内で変化しない変数との比較結果で決まります。
 3. 1 ずつインクリメント。

次のコード例は、ループ内のデータフローの正規形式を示しています。

```
void dataflow(Input0, Input1, Output0, Output1) {
    UserDataTpe C0, C1, C2;
    for (int i = 0; i < N; ++i) {
        #pragma HLS DATAFLOW
        func1(read Input0, read Input1, write C0, write C1);
        func2(read C0, read C0, read C1, write C2);
        func3(read C2, write Output0, write Output1);
    }
}
```

データフローのトラブルシューティング

次のような場合、xocc コンパイラで DATAFLOW 最適化が実行されないことがあります。

1. シングル プロデューサー コンシューマー違反。
2. タスクのバイパス。
3. タスク間のフィードバック。
4. タスクの条件付き実行。
5. 複数の exit 条件またはループ内で定義された条件を持つループ。

これらのいずれかがデータフロー領域内で発生する場合は、データフロー最適化が問題なく実行されるようにするためにコードを記述し直す必要があることもあります。

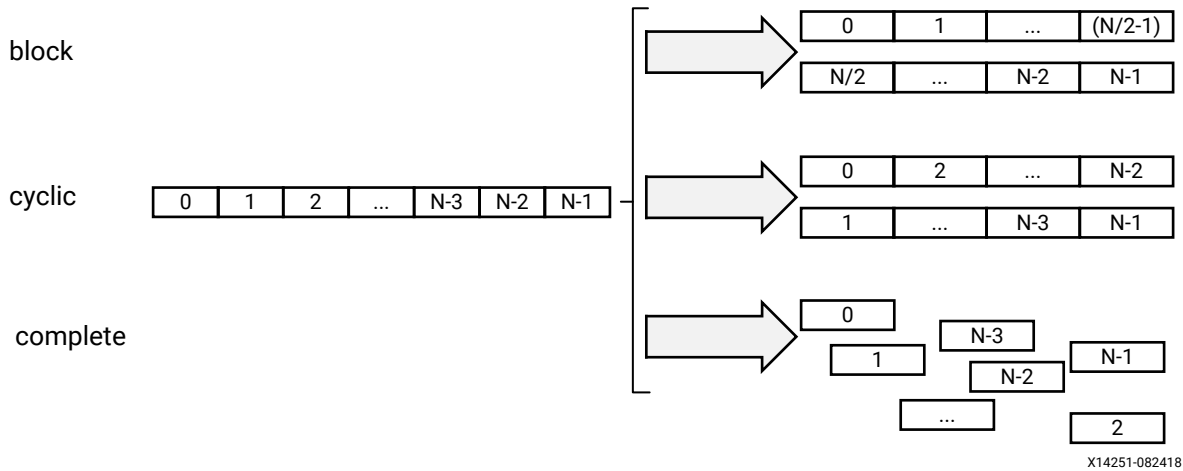
配列コンフィギュレーション

SDAccel コンパイラでは、大型の配列は PL 領域にあるブロック RAM (BRAM) にマップされます。これらの BRAM のアクセス ポイント (またはポート) は最大 2 つなので、ハードウェアにインプリメントされたときに配列のすべての要素に並列にアクセスできず、アプリケーションのパフォーマンスが制限されることがあります。

パフォーマンス要件によっては、配列の一部またはすべての要素に同じクロック サイクルでアクセスすることが必要な場合があります。これには、`#pragma HLS ARRAY_PARTITION` を使用して、コンパイラで配列の要素が分割され、より小さな配列や個別レジスタにマップされるようにします。コンパイラには、次の図に示すように、3 つの配列分割方法があります。3 つの分割方法は、次のとおりです。

- `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`: 配列が個別要素に分割されます。これは、メモリの個別レジスタへの分解に相当します。これは `ARRAY_PARTITION` プラグマの場合デフォルトです。

図9: 配列の分割

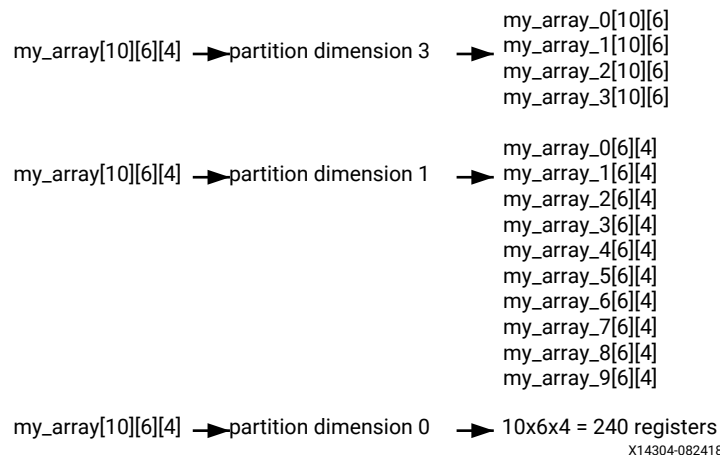


block および cyclic 分割では、`factor` オプションを使用して作成する配列の数を指定できます。前の図では、`factor` オプション 2 が使用され、配列が 2 つの小さな配列に分割されています。配列の要素数がこの係数の整数倍ではない場合、後の配列に含まれる要素数は少なくなります。

多次元配列を分割する際は、`dimension` オプションを使用して、分割する次元を指定できます。次の図に、次のコード例を `dimension` オプションを使用して 3 つの方法で分割した場合を示します。

```
void foo (...) {
    // my_array[dim=1][dim=2][dim=3]
    // The following three pragma results are shown in the figure below
    // #pragma HLS ARRAY_PARTITION variable=my_array dim=3 <block|cyclic>
    factor=2
    // #pragma HLS ARRAY_PARTITION variable=my_array dim=1 <block|cyclic>
    factor=2
    // #pragma HLS ARRAY_PARTITION variable=my_array dim=0 complete
    int my_array[10][6][4];
    ...
}
```

図10: 配列の次元の分割



dimension を 3 に設定すると 4 つの配列に、1 に設定すると 10 個の配列に分割されます。dimension を 0 に設定すると、すべての次元が分割されます。

分割の重要性

配列の complete 分割では、すべての配列エレメントが個別レジスタにマップされます。これにより、すべてのレジスタに同じサイクルで同時にアクセスできるので、カーネル パフォーマンスが改善します。



注意: 大型の配列の complete 分割すると、PL 領域が大量に使用されるので、注意が必要です。場合によっては、コンパイル プロセスの速度が遅くなり、容量が足りなくなることがあります。このため、配列は必要な場合にのみ分割する必要があります。特定の次元を分割するか、block または cycle 分割を実行することを考慮してみてください。

分割する次元の選択

A と B が 2 つの行列を示す 2 次元配列で、次のような行列乗算アルゴリズムがあるとします。

```
int A[64][64];
int B[64][64];

ROW_WISE: for (int i = 0; i < 64; i++) {
    COL_WISE : for (int j = 0; j < 64; j++) {
        #pragma HLS PIPELINE
        int result = 0;
        COMPUTE_LOOP: for (int k = 0; k < 64; k++) {
            result += A[i][k] * B[k][j];
        }
        C[i][j] = result;
    }
}
```

PIPELINE プラグマにより、ROW_WISE および COL_WISE ループが一緒に平坦化され、COMPUTE_LOOP が完全に展開されます。COMPUTE_LOOP の各反復 (k) を同時に実行するには、行列 A の各列と行列 B の各行に並列でアクセスできるようにする必要があります。このため、行列 A は 2 つ目の次元で分割し、行列 B は最初の次元で分割する必要があります。

```
#pragma HLS ARRAY_PARTITION variable=A dim=2 complete
#pragma HLS ARRAY_PARTITION variable=B dim=1 complete
```

cyclic および block 分割の選択

ここでは、同じ行列乗算アルゴリズムを使用して、cyclic および block 分割を選択し、下位のアルゴリズムの配列アクセス パターンを理解することで、適切な係数を指定します。

```
int A[64 * 64];
int B[64 * 64];
#pragma HLS ARRAY_PARTITION variable=A dim=1 cyclic factor=64
#pragma HLS ARRAY_PARTITION variable=B dim=1 block factor=64

ROW_WISE: for (int i = 0; i < 64; i++) {
    COL_WISE : for (int j = 0; j < 64; j++) {
        #pragma HLS PIPELINE
        int result = 0;
        COMPUTE_LOOP: for (int k = 0; k < 64; k++) {
```

```

        result += A[i * 64 + k] * B[k * 64 + j];
    }
    C[i* 64 + j] = result;
}
}

```

ここではコード A と B は 1 次元配列であるとします。行列 A の各列と行列 B の各行に並列でアクセスするには、`cyclic` と `block` 分割を上記の例のように使用します。行列 A の各列に並列でアクセスするため、`cyclic` 分割が行サイズとして指定した `factor` (この場合 64) で適用されています。同様に、行列 B の各行に並列でアクセスするため、`block` 分割が行サイズとして指定した `factor` (この場合 64) で適用されています。

キャッシュを使用した配列アクセスの削減

配列はアクセスポート数が制限された BRAM にマップされるので、配列の繰り返しにより、アクセラレータのパフォーマンスが制限されることがあります。アルゴリズムの配列アクセスパターンを理解し、データをローカルでキャッシュして配列アクセスを制限して、カーネルのパフォーマンスを改善するようにしてください。

次のコード例では、配列へのアクセスにより最終インプリメンテーションのパフォーマンスが制限されます。この例では、`mem[N]` 配列が 3 回アクセスされ、それらを合計して結果が作成されています。

```

#include "array_mem_bottleneck.h"
dout_t array_mem_bottleneck(din_t mem[N]) {
    dout_t sum=0;
    int i;
    SUM_LOOP:for(i=2;i<N;++i)
        sum += mem[i] + mem[i-1] + mem[i-2];
    return sum;
}

```

上記のコード例を次のように変更すると、`II=1` でパイプライン処理できるようになります。先行読み出しを実行し、データアクセスを手動でパイプライン処理することで、ループの各反復で指定される配列読み出しを 1 回だけにしていきます。これにより、パフォーマンスを達成するためには、シングルポート BRAM だけが必要となります。

```

#include "array_mem_perform.h"
dout_t array_mem_perform(din_t mem[N]) {
    din_t tmp0, tmp1, tmp2;
    dout_t sum=0;
    int i;
    tmp0 = mem[0];
    tmp1 = mem[1];
    SUM_LOOP:for (i = 2; i < N; i++) {
        tmp2 = mem[i];
        sum += tmp2 + tmp1 + tmp0;
        tmp0 = tmp1;
        tmp1 = tmp2;
    }
    return sum;
}

```



推奨: アルゴリズムによって、パイプラインパフォーマンスを改善するため、ローカルレジスタにキャッシュして配列アクセスを最小限に抑えることを考慮してください。

配列のコンフィギュレーションに関する詳細は、『Vivado Design Suite ユーザーガイド: 高位合成』(UG902: [英語版](#)、[日本語版](#)) の「配列」セクションを参照してください。

関数のインライン展開

C コードは通常、複数の関数で構成されます。デフォルトでは、各関数が `xocc` コンパイラにより個別にコンパイルおよび最適化されます。関数本体に対して固有のハードウェア モジュールが生成され、必要に応じて再利用されます。

パフォーマンスの面から、一般的には関数をインライン展開するか、関数の階層を解除する方が良い結果が得られます。このようにすると、`xocc` コンパイラで関数の境界を越えてグローバルに最適化を実行できるようになります。たとえば、パイプライン処理されたループ内で関数を呼び出す場合、関数をインライン展開するとコンパイラにより積極的に最適化を実行できるようになり、ループのパイプライン パフォーマンスが向上します (開始間隔 (II) を削減)。

次の `INLINE` プラグマを関数本体に配置すると、関数がインライン展開されます。

```
foo_sub (p, q) {  
    #pragma HLS INLINE  
    ....  
    ...  
}
```

ただし、関数本体が非常に大きく、メインのカーネル関数で複数回呼び出される場合は、関数をインライン展開すると、使用されるリソース量が多くなりすぎて容量の問題が発生することがあります。このような場合は関数をインライン展開せずに、`xocc` コンパイラで関数とそのローカル コンテキスト内で個別に最適化されるようにします。

まとめ

前のトピックで説明したように、C/C/C++ を使用した FPGA アクセラレーションのカーネルをコード記述する際には、重要な点がいくつかあります。

1. 任意精度データ型 `ap_int` および `ap_fixed` を使用することを考慮します。
2. カーネル インターフェイスを理解して、スカラー インターフェイスを使用するかメモリ インターフェイスを使用するかを決定します。リンク段階で別の DDR メモリ バンクを指定する場合は、`bundle` キーワードを使用して異なる名前を指定します。
3. メモリ インターフェイスに対する読み出しおよび書き込みにはバースト コーディング スタイルを使用します。
4. メモリ データのデータ入力および出力入力および出力の幅を選択する際は、データ転送に DDR バンクの全幅を使用することを考慮します。
5. パイプライン処理およびデータフローを使用して最大限のパフォーマンスを得られるようにします。
6. `xocc` コンパイラで平坦化を実行してパイプラインを効果的に適用できるように、完全または半完全な入れ子のループを記述します。
7. 反復回数が少なく、ループ本体内の演算数が少ないループを展開します。
8. 配列のアクセス パターンを理解し、配列全体に `complete` 分割を適用するのではなく、特定の次元に `complete` 分割を適用するか、`block` または `cyclic` 分割を適用します。
9. カーネルのパフォーマンスを向上するため、ローカル キャッシュを使用して配列へのアクセスを最小限に抑えます。
10. 関数 (特にパイプライン処理された領域内) をインライン展開することを考慮します。データフロー領域内の関数はインライン展開しないでください。

システム アーキテクチャの設定

第 1 章: SDAccel のコンパイル フローおよび実行モデル に示すように、SDAccel™ 環境のカーネルのビルド プロセスには次の 2 つの段階があります。

1. コンパイル段階: コンパイル プロセスは `xocc -c` オプションで制御します。コンパイル段階の最後に、各カーネル関数が個別の `.xo` ファイルにコンパイルされます。この時点では、`xocc` コンパイラにより、C/C++ コードおよびプラグマからハードウェアの意図する機能が抽出されます。`xocc` コンパイラの詳細は、『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)を参照してください。
2. リンク段階: リンク段階は `xocc -l` オプションで制御します。リンク プロセスでは、すべての `.xo` ファイルが FPGA ハードウェアに統合されます。

カーネルのリンク プロセスは、SDAccel 環境ランタイム パフォーマンスを向上するため必要に応じてカスタマイズできます。この章では、その手法をいくつか示します。

カーネルの複数のインスタンス

デフォルトでは、1 つのカーネルから 1 つのハードウェア インスタンスがインプリメントされます。ホストで同じカーネルが複数回実行される場合、複数のカーネル実行は同じハードウェア インスタンスで順次実行されます。カーネルのコンパイル(リンク段階)をカスタマイズすると、1 つのカーネルに対して複数のハードウェア インスタンスを作成できます。これにより複数のカーネル呼び出しを別のハードウェア インスタンスでオーバーラップさせて同時実行できるので、パフォーマンスが向上します。

カーネルの複数のインスタンスを作成するには、リンク中に `xocc --nk` オプションを使用します。

たとえば、カーネル `foo` に 2 つのハードウェア インスタンスを作成するには、次のコードを使用します。

```
# xocc --nk <kernel name>:<number of instance>
xocc --nk foo:2
```

デフォルトでは、インスタンス名は `<kernel_name>_1` および `<kernel_name>_2` のようになります。ただし、次のコードを使用すると、このデフォルトのインスタンス名を変更できます。

```
# xocc --nk <kernel name>:<no of instance>:<name 1>.<name 2>...<name N>
xocc --nk foo:3:fooA.fooB.fooC
```

この例では、FPGA プログラマブル ロジックに `fooA`、`fooB`、`fooC` という 3 つのまったく同じコピーまたはカーネル `foo` のハードウェア インスタンスがインプリメントされています。

カーネル ポートとグローバル メモリの接続

デフォルトでは、すべてのカーネル メモリ インターフェイスが同じグローバル メモリ バンクに接続されます。そのため、メモリ バンクとデータ間の転送ができるのは一度に 1 つのカーネル ポートのみなので、アプリケーションのパフォーマンスが制限されます。

ただし、既成の SDAccel プラットフォームには複数のグローバル メモリ バンクが含まれています。リンク段階では、どのカーネル ポート (またはインターフェイス) をグローバル メモリ バンクに接続するか指定できます。

帯域幅を最大にして、データ転送を最適化して、全体的なパフォーマンスを改善するには、カーネルとメモリの接続を正しく設定しておくことが重要です。

次のような例があるとします。

```
void cnn( int *image, // Read-Only Image
          int *weights, // Read-Only Weight Matrix
          int *out, // Output Filters/Images
          ... // Other input or Output ports

#pragma HLS INTERFACE m_axi port=image offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=weights offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=out offset=slave bundle=gmem
```

この例には、`image` および `weights` というカーネルの 2 つのメモリ インターフェイス入力が見えています。両方を同じメモリ バンクに接続した場合、これら両方の入力のカーネルへの同時転送は不可能です。

`image` および `weights` 入力を別のメモリ バンク接続にするには、次の手順に従います。

1. これらの入力に別のバンドル名を指定します。これについては、[メモリのデータ入力および出力](#)で説明されています。ここにもコード例を示します。

```
void cnn( int *image, // Read-Only Image
          int *weights, // Read-Only Weight Matrix
          int *out, // Output Filters/Images
          ... // Other input or Output ports

#pragma HLS INTERFACE m_axi port=image offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=weights offset=slave bundle=gmem1
#pragma HLS INTERFACE m_axi port=out offset=slave bundle=gmem
```



重要: `bundle=` 名を指定する場合は、すべて小文字を使用し、`--sp` オプションを使用して特定のメモリ バンクに割り当てる必要があります。

上記の例では、メモリ インターフェイス入力 `image` と `weights` に異なるバンドル名が割り当てられています。

2. `--sp` オプションを使用してリンク段階でカーネル ポートとグローバル メモリの接続を指定します。

```
--sp <kernel_instance_name>.<interface_name>:<bank name>
```

説明:

- [カーネルの複数のインスタンス](#) に示すように、`<kernel_instance_name>` は `--nk` オプションで指定したカーネルのインスタンス名です。
- `<interface_name>` は HLS INTERFACE プラグマで定義される `m_axi_` で始まるインターフェイス バンドルの名前、指定した場合は `bundle=` 名になります。

注記: `<interface_name>` (`image`, `weights`) を使用することもできます。



ヒント: ポートがバンドルの一部として指定されていない場合は、`<interface_name>` を `m_axi_` なしで `port=` を使用して指定します。

- `<bank_name>`: `DDR[0]`、`DDR[1]` などのバンク名を指定します。4 つの DDR を含むプラットフォームでは、バンク名は `DDR[0]`、`DDR[1]`、`DDR[2]`、および `DDR[3]` となります。プラットフォームの中には、PLRAM および HBM メモリ バンクをサポートするものもあります。

上記の例で `cnn` カーネルの 1 つのインスタンスを考慮すると、`--sp` オプションは次のように指定できます。

```
--sp cnn_1.m_axi_gmem:DDR[0] \
--sp cnn_1.m_axi_gmem1:DDR[1]
```

注記: 最高 15 のカーネル ポートを指定したメモリ バンクに接続できます。このため、デザイン全体のポートが 15 個を超える場合は、デフォルトのマッピングを使用できないので、`--sp` オプションを使用して接続を別のバンクに分散する必要があります。

まとめ

このセクションでは、カーネルのコンパイルをカスタマイズして、実行中システム パフォーマンスを向上する 2 つの方法を説明しました。

1. カーネルがホスト コードから複数呼び出される場合は、`xocc --nk` オプションを使用して FPGA ファブリックに複数のカーネル インスタンスを作成することを考慮してください。
2. 同時アクセスを可能にするため、`xocc --sp` を使用してグローバル メモリからカーネル メモリ インターフェイスへの接続をカスタマイズすることを考慮してください。

ホストおよびカーネルのデザインによって、これらのオプションを使用してザイリンクス FPGA のカーネルのアクセラレーションを向上できます。

SDAccel ストリーミング プラットフォーム

ホストとカーネル間のストリーミング データ転送

SDAccel™ 2019.1 リリースからは、グローバル メモリを介さず、ホストからカーネルおよびカーネルからホストへのダイレクト ストリーミングをサポートする新しいプログラミング モデルが SDAccel に含まれるようになりました。この機能は、従来のグローバル メモリを使用した既存のホストからカーネル、カーネルからホストへのデータ転送に追加されたものです。ストリーミングを使用すると、次のような利点があります。

- ホスト アプリケーションは、カーネルからのデータ サイズを必ずしも知る必要がありません。
- ホスト メモリのデータは、必要になると即座にカーネルに転送できます。同様に、処理されたデータを必要な場合に転送し戻すこともできます。

このプログラミング モデルは、より大きくて遅いグローバル メモリ バンクと比較すると最小限のストレージしか使用しないので、パフォーマンスおよび消費電力を改善できることがあります。

ホスト コーディング ガイドライン

ザイリンクスでは、拡張 API としてストリーミング操作用に新しい OpenCL™ API を提供しています。

- `clCreateStream()`: 読み出しまたは書き込みストリームを作成します。
- `clReleaseStream()`: 作成したストリームと関連するメモリを空けます。
- `clWriteStream()`: データをストリームに書き込みます。
- `clReadStream()`: ストリームからデータを取得します。
- `clPollStreams()`: デバイスのストリームをポーリングして終了します。ノンブロッキング ストリーム操作にのみ必要です。

典型的な API フローは次のようになります。

- `clCreateStream` で必要な数の読み出し/書き込みストリームを作成します。
 - ストリームはコマンド キューを使用しないので、直接 OpenCL デバイス オブジェクトに接続する必要があります。ストリーム自体はデータを特定の方向 (ホストからカーネルまたはカーネルからホスト) に渡すだけのコマンド キューです。
 - ストリーム書き出し/読み込み操作 (ホスト側からの視点) を示すには、正しいフラグを使用する必要があります。
 - ストリームのデバイスへの接続方法を指定するには、定義済みの拡張ポインター (`cl_mem_ext_ptr_t`) を使用して、ストリームが関連付けられたカーネルとその引数を示します。

次のコードでは、読み出しストリーム (`read_stream`) および書き込みストリーム (`write_stream`) が作成されます。

```
#include <CL/cl_ext_xilinx.h> // Required for Xilinx Extension

// Device connection specification of the stream through extension
// pointer
cl_mem_ext_ptr_t ext; // Extension pointer
ext.param = kernel; // The .param should be set to kernel
// (cl_kernel type)
ext.obj = nullptr;

// The .flag should be used to denote the kernel argument
// Create write stream for argument 3 of kernel
ext.flags = 3;
cl_stream write_stream = clCreateStream(device_id,
CL_STREAM_WRITE_ONLY, CL_STREAM, &ext, &ret);

// Create read stream for argument 4 of kernel
ext.flags = 4;
cl_stream read_stream = clCreateStream(device_id, CL_STREAM_READ_ONLY,
CL_STREAM, &ext, &ret);
```

- 残りのストリーム以外のカーネル引数を設定し、カーネルをエンキューします。次のコードは、典型的なカーネル引数 (バッファやスカラーなどのストリーム以外の引数) の設定とカーネル エンキューを示しています。

```
// Set kernel non-stream argument (if any)
clSetKernelArg(kernel, 0, ..., ...);
clSetKernelArg(kernel, 1, ..., ...);
clSetKernelArg(kernel, 2, ..., ...);
// Argument 3 and 4 are not set as those are already specified during
// the clCreateStream through extension pointer

// Schedule kernel enqueue
clEnqueueTask(commands, kernel, . . . );
```

- `clReadStream` および `clWriteStream` で読み出しおよび書き込み転送を開始します。
 - 読み出しおよび書き込みリクエストに関連する `cl_stream_xfer_req` 属性が使用されるところに注意してください。
 - 転送メカニズムを示すには、`.flag` を使用します。
 - `CL_STREAM_EOT`: 現在のところ、EOT (End of Transfer) 信号で転送の終わりを示すことで、問題のないストリーム転送メカニズムになるようになっています。このフラグは、現在のリリースでは必須です。
 - `CL_STREAM_NONBLOCKING`: デフォルトでは、読み出し転送および書き込み転送はブロッキングです。ノンブロッキング転送の場合は、`CL_STREAM_NONBLOCKING` を設定する必要があります。
 - 転送に関連するストリングを指定するには、`.priv_data` を使用します。これにより、ストリーミングの終了をポーリングする際に特定の転送が終了したことを示すことができます。ノンブロッキング バージョンの API を使用する場合にのみ必要です。

次のコード例の場合、ストリーム読み出しおよび書き込み転送がノンブロッキング方式を使用して実行されます。

```
// Initiate the READ transfer
cl_stream_xfer_req rd_req {0};

rd_req.flags = CL_STREAM_EOT | CL_STREAM_NONBLOCKING;
rd_req.priv_data = (void*)"read"; // You can think this as tagging the
// transfer with a name
```

```
clReadStream(read_stream, host_read_ptr, max_read_size, &rd_req, &ret);

// Initiating the WRITE transfer
cl_stream_xfer_req wr_req {0};

wr_req.flags = CL_STREAM_EOT | CL_STREAM_NONBLOCKING;
wr_req.priv_data = (void*)"write";

clWriteStream(write_stream, host_write_ptr, write_size, &wr_req, &ret);
```

- すべてのストリームをポーリングして終了します。ノンブロッキング転送の場合、読み出し/書き込み転送を確実に終了するためのポーリング API が提供されています。ブロッキング バージョンの API の場合、ポーリングは必要ありません。
 - ポーリング リクエストには、`cl_streams_poll_req_completions` を使用する必要があります。
 - `clPollStreams` はブロッキング API です。すべてのストリーム リクエストが終了したことを示す通知を受け取るか、指定したタイムアウトになったら、ホスト コードに実行を戻します。

```
// Checking the request completion
cl_streams_poll_req_completions poll_req[2] {0, 0}; // 2 Requests

auto num_compl = 2;
clPollStreams(device_id, poll_req, 2, 2, &num_compl, 5000, &ret);
// Blocking API, waits for 2 poll request completion or 5000ms,
// whichever occurs first
```

- ホストでストリーム データを読み出して使用します。
 - ポーリング リクエストが問題なく終了したら、ホストがホスト ポインターからデータを読み出すことができます。
 - ホストはホストに転送されたデータのサイズをチェックすることもできます。この目的の場合、ホストは `priv_data` をマッチングさせて、`cl_streams_poll_req_completions` structure からの `nbytes` (転送されるバイト数) をフェッチして、正しいポーリング リクエストを見つける必要があります。

```
for (auto i=0; i<2; ++i) {
    if(rd_req.priv_data == poll_req[i].priv_data) { // Identifying the
                                                    // read transfer
        // Getting read size, data size from kernel is unknown
        ssize_t result_size=poll_req[i].nbytes;
    }
}
```

関数プロトタイプおよび引数の説明を含むヘッダー ファイルは、[サイリンクス ランタイム GitHub リポジトリ](#)から入手できます。



重要: ストリーミング カーネルに複数の CU が含まれる場合、ホスト コードは各 CU に一意の `cl_kernel` オブジェクトを使用する必要があります。ホスト コードには `clCreateKernel` に `<kernel_name>`: `{compute_unit_name}` を付けて、各 CU を取得して、それらのストリームを作成して、個別にエンキューする必要があります。

カーネル コーディング ガイドライン

システム ベースの C カーネルを開発する基本的なガイドラインは、次のとおりです。

- `hls::stream` を `qdma_axis<D,0,0,0>` データ型で使します。 `qdma_axis` データ型には `ap_axi_sdata.h` ヘッダー ファイルが必要です。

- `qdma_axis<D,0,0,0>` は、ストリーミング プラットフォームを使用する際、ホストとカーネル間のデータ転送に使用される特殊なクラスで、ホストと関連するストリーミング カーネル インターフェイスでのみ使用され、ほかのカーネルと関連するものには使用されません。テンプレートのパラメーター `<D>` はデータ幅を示します。残りの 3 つのパラメーターは現在のリリースでは使用しないように、0 に設定する必要があります。
- 次のコードは、入力ストリーム 1 つと出力ストリーム 1 つを含む単純なカーネル インターフェイスを示しています。

```
#include "ap_axi_sdata.h"
#include "hls_stream.h"

//qdma_axis is the HLS class for stream data transfer between host and
kernel for streaming platform
//It contains "data" and two sideband signals (last and keep) exposed to
the user via class member function.
typedef qdma_axis<64,0,0,0> datap;

void kernel_top (
    hls::stream<datap> &input,
    hls::stream<datap> &output,
    ..... , // Other Inputs/Outputs if any
)
{
    #pragma HLS INTERFACE axis port=input
    #pragma HLS INTERFACE axis port=output
}
```

- `qdma_axis` データ型には、カーネル コード内で使用する必要のある 3 つの変数が含まれます。
 - `data`: `qdma_axis` には `ap_uint<D>` が含まれ、`.get_data()` および `.set_data()` メソッドでアクセスされます。
 - `D` は 8、16、32、64、128、256、または 512 ビット幅である必要があります。
 - `last`: `last` 変数は入力ストリームおよび出力ストリームの最後の値を示します。入力ストリームからの読み込みがあると、`last` でストリームの終わりが検出されます。同様に、カーネルが出力ストリームに書き込んでホストに転送される際にも、`last` を設定してストリームの終わりを示す必要があります。
 - `get_last/set_last`: ストリームの最後のデータを示すのに使用される `last` 変数を取得/設定します。
 - `keep`: `keep` は、特殊な状況で、`last` データをより少ないバイト数に切り捨てるために使用されます。ただし、`keep` はストリームの `last` データ以外のデータには使用できません。このため、ほとんどの場合、`keep` を 1 に設定してカーネルからすべてのデータを出力させるようにする必要があります。
 - `get_keep/set_keep`: `keep` 変数を取得/設定します。
 - `last` データの前のデータすべてに対して `keep` を 1 に設定し、データのすべてのバイトが有効であることを示す必要があります。
 - `last` データに対しては、カーネルにより少ないバイトを送信する柔軟性があります。たとえば、4 バイトを転送する際、カーネルは次のように `set_keep()` 関数を使用して、1 バイト、2 バイト、または 3 バイトを送信して、`last` データを切り捨てることができます。
 - `last` データが 1 バイト => `.set_keep(1)`
 - `last` データが 2 バイト => `.set_keep(3)`
 - `last` データが 3 バイト => `.set_keep(7)`
 - `last` データが 4 バイト (`last` 以外のデータすべてと同様) => `.set_keep(-1)`

- 次のコードは、input ストリームがどのように読み込まれるかを示しています。.last により、最後のデータが決定されています。

```
// Stream Read
// Using "last" flag to determine the end of input-stream
// when kernel does not know the length of the input data
hls::stream<ap_uint<64> > internal_stream;
while(true) {
    datap temp = input.read(); // "input" -> Input stream
    internal_stream << temp.get_data(); // Getting data from the
    stream
    if(temp.get_last()) // Getting last signal to determine the
    EOT (end of transfer).
        break;
}
```

- 次のコードは、output ストリームがどのように書き出されるかを示しています.set_keep はすべてのデータに対して -1 に設定されています。また、カーネルは set_last() を使用して、ストリームの last データを指定しています。



重要: ホストおよびカーネル システムが正しく機能するようにするには、last ビットを設定することが大変重要です。

```
// Stream Write
for(int j = 0; j <....; j++) {
    datap t;
    t.set_data(...);
    t.set_keep(-1); // keep flag -1 , all bytes are valid
    if(... ) // check if this is last data to be write
        t.set_last(1); // Setting last data of the stream
    else
        t.set_last(0);
    output.write(t); // output stream from the kernel
}
```

カーネル間のストリーミング データ転送

SDAccel 環境では、2 つのカーネル間のストリーミング データ転送もサポートされます。1 つ目のカーネルが計算の一部を実行し、2 つ目のカーネルは 1 つ目のカーネルからの出力データを受信してから、残りを処理します。SDx™ 2019.1 バージョンより前のバージョンでは、1 つのカーネルから別のカーネルにデータを転送するには、グローバルメモリを介する方法しかありませんでした。2019.1 バージョンからはカーネル間のストリーミングがサポートされるようになったので、グローバルメモリを介して転送しなくてもデータがカーネル間を転送できるようになり、パフォーマンスが改善するようになりました。

ホスト コーディング ガイドライン

カーネル間のストリーミング データ転送では、カーネル間のデータ転送に使用されるカーネル ポートにホストコードからの clSetKernelArg が必要ない点にのみ注意してください。ホストコードで clSetKernelArg コマンドを使用して、ホストに直接接続されるほかのカーネル ポート引数を設定する必要があります。

カーネル コーディング ガイドライン

データを直接別のカーネル ストリーミング インターフェイスに直接送信または受信するカーネル ストリーミング インターフェイスは `hls::stream` で、`ap_axiu<D,0,0,0>` データ型を使用して定義する必要があります。
`ap_axiu<D,0,0,0>` データ型には、ヘッダー ファイル `ap_axi_sdata.h` が必要です。



重要: ザイリンクスでは、前のセクションで説明したように、ホストからカーネル、カーネルからホストへの転送に `qdma_axis` データ型を使用することを必須条件にしていますが、カーネル間ストリーミング データ転送には `ap_axiu` データ型を使用する必要があります。これらのデータ型はどちらも SDAccel リリースに含まれる `ap_axi_sdata.h` ファイル内で定義されます。

次の例は、プロデューサーおよびコンシューマー カーネルのストリーミング インターフェイスを示しています。

```
// Producer kernel
// Producing stream output to another kernel on the FPGA
// The below code segment ignores all other inputs and outputs, if any

void kernel1 (.... , hls::stream<ap_axiu<32, 0, 0, 0> >& stream_out) {
#pragma HLS interface axis port=stream_out

    for(int i = 0; i < ....; i++) {
        int a = ..... ; // Internally generated data
        ap_axiu<32, 0, 0, 0> v; // temporary storage for ap_axiu
        v.data = a; // Writing the data
        stream_out.write(v); // Writing to the output stream.
    }
}

// Consumer kernel
// Consuming stream input from another kernel on the FPGA
// The below code segment ignores all other inputs and outputs, if any
void kernel2 (hls::stream<ap_axiu<32, 0, 0, 0> >& stream_in, .... ) {
#pragma HLS interface axis port=stream_in

    for(int i = 0; i < ....; i++) {
        ap_axiu<32, 0, 0, 0> v = stream_in.read(); // Reading from the
        Input stream
        int a = v.data; // Extract the data

        // Do further processing
    }
}
```

カーネルのリンク

また、`xocc` リンク (-l) 段階で `--sc` オプションを使用して、プロデューサー カーネルのストリーミング出力ポートをコンシューマー カーネルのストリーミング入力ポートに接続してください。

```
#Syntax:: xocc -l --sc <Source streaming port>:<Destination streaming port>
xocc -l --sc <kernel1 instance name>.stream_in:<kernel2 instance
name>.stream_out
```

OpenCL インストーラブル クライアント ドライバ ロード

システムには、それぞれ独自のドライバと OpenCL バージョンを持つ複数の OpenCL™ を含めることができます。SDAccel™ 環境では、OpenCL インストーラブル クライアント ドライバ (ICD) 拡張 (cl_khr_icd) がサポートされます。これにより、OpenCL を複数インプリメンテーションして、同じシステム内に共存させることができます。ICD ロードは、インストールしたすべてのプラットフォームを監視して、API 呼び出しの標準ハンドラを提供します。

アプリケーションはインストールされたプラットフォームのリストから OpenCL プラットフォームを選択できます。ICD はアプリケーションで指定したプラットフォーム ID に基づいて、OpenCL ホスト呼び出しを正しいランタイムに送信します。

ザイリンクスでは OpenCL ICD ライブラリを提供していないので、次のライブラリをシステムに合わせてインストールする必要があります。

Ubuntu

Ubuntu の場合、ICD ライブラリがディストリビューションにパッケージされます。次のパッケージをインストールします。

- ocl-icd-libopencl1
- opencl-headers
- ocl-icd-opencl-dev

Linux

RHEL/CentOS 7.X の場合、EPEL 7 を使用し、次のパッケージをインストールします。

- ocl-icd
- ocl-icd-devel
- opencl-headers

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 は、SDSoC™ および SDAccel™ 開発環境と共にインストールされます。DocNav を開くには、次のいずれかを実行します。

- 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

1. 『SDAccel 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』 ([UG1238](#))
2. 『SDAccel 環境プロファイリングおよび最適化ガイド』 ([UG1207](#))
3. 『SDAccel 環境チュートリアル: 入門』 ([UG1021](#))
4. [SDAccel™ 開発環境ウェブ ページ](#)

5. [Vivado® Design Suite 資料](#)
6. 『Vivado Design Suite ユーザー ガイド: IP インテグレーターを使用した IP サブシステムの設計』 (UG994)
7. 『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)
8. 『Vivado Design Suite ユーザー ガイド: パーシャル リコンフィギュレーション』 (UG909)
9. 『Vivado Design Suite ユーザー ガイド: 高位合成』 (UG902)
10. 『UltraFast 設計手法ガイド (Vivado Design Suite 用)』 (UG949)
11. 『Vivado Design Suite プロパティ リファレンス ガイド』 (UG912)
12. [Khronos Group ウェブ ページ](#): OpenCL 規格の資料
13. [ザイリンクス Virtex® UltraScale+™ FPGA VCU1525 アクセラレーション開発キット](#)
14. [ザイリンクス Kintex® UltraScale™ FPGA KCU1500 アクセラレーション開発キット](#)
15. [ザイリンクス Alveo™ ウェブ ページ](#)

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、ザ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、ザイリンクスは、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、また、(2) ザイリンクスは、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、ザ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。ザイリンクスは、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、ザ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> で見られるザイリンクスの販売条件を参照してください。IP コアは、ザ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。ザ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ザ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> で見られるザ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

オートモーティブ製品 (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、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、また、

商標

© Copyright 2018-2019 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ザ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。