

SDAccel 環境ユーザー ガイド

UG1023 (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 実行モデル	セクションをアップデート。
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 設計手法	ソフトウェア中心の説明をアップデート。
第 2 章: はじめに	UG1352 へのリンクと情報を追加。
アプリケーション プロジェクトの作成	jSch を追加し、説明をアップデート。
第 4 章: SDAccel のプログラム	セクションをアップデート。
ランタイムの設定	4 のコードの説明をアップデート。
カーネル言語サポート	セクションをアップデート。
C/C++ カーネルの記述	セクションをアップデート。
スカラー	セクションをアップデート。
ストリーミング	セクションを追加。
ホスト アプリケーションのビルド	セクションをアップデート。
ホスト アプリケーションのリンク	注記を追加。
ハードウェアのビルド	リンクを削除してヘッダーをアップデート。
複数のカーネル インスタンスの作成	clCreateSubDevices の説明を追加。
メモリ リソースへのカーネル インターフェイスのマッピング	セクションをアップデート。
カーネル間ストリーミング接続	セクションを追加。
計算ユニットの SLR への割り当て	カーネル SLR および DDR のリンクを追加。
システム見積もりレポート	report_level コードをアップデート。
コンパイル	コードに \ を追加。
リンク	コードに \ を追加、注記を追加。
コンパイル	説明をアップデート。
リンク	説明をアップデート。
sdaccel.ini ファイルの使用	sdaccel.ini の段落をアップデートし、ランタイム グループに cpu_affinity エントリを追加。
カーネルのソフトウェア要件	制御および割り込みの表をアップデート。
RTL Kernel ウィザードの [General Settings] ページ	図をアップデート。
[Kernel options]	カーネル制御インターフェイスを追加。
RTL カーネルをザイリンクス オブジェクト ファイルにパッケージ	セクションの最後にリンクを追加。
Vivado HLS を使用した SDAccel カーネルの作成	SDAccel ボトム アップの説明とデバイスの選択 GUI をアップデート。
SDAccel への Vivado HLS カーネル プロジェクトの取り込み	最初の段落をアップデート。
付録 A: サンプル デザインの概要	GitHub へのリンクと説明を追加。
付録 C: 便利なコマンド ライン ユーティリティ	アップデートして UG1279 に追加。

セクション	改訂内容
2019 年 1 月 24 日 バージョン 2018.3	
SDAccel を使用したアクセラレーションのベスト プラクティス	説明をアップデート。
XOCC リンクおよびコンパイル オプション	SLR リンクを追加。
カーネル言語サポート	SystemVerilog を追加。
ランタイムの設定	説明の 5 番をアップデート。
インプリメンテーション結果の制御	説明をアップデート。
レポート生成の制御	説明をアップデート。
デバッグの機能および手法	表に注記を追加。
第 9 章: RTL カーネル	説明をアップデート。
RTL カーネルとして RTL デザインを使用するための要件	説明をアップデート。
カーネルのインターフェイス要件	説明をアップデート。
カーネルのソフトウェア要件	説明をアップデート。
割り込み	説明をアップデート。
[Scalars] ページ	引数タイプの説明をアップデート。
RTL カーネル タイプのプロジェクト フロー	図のサイズを変更。
RTL カーネルでのクロックの管理	セクションを追加。
Vivado HLS を使用した SDAccel カーネルの作成	図と説明をアップデート。
xclbinutil ユーティリティ	説明をアップデート。
クロック	リンクを追加してコードをアップデート。
2018 年 12 月 5 日 バージョン 2018.3	
SDAccel 実行モデル	図をアップデート。
SDAccel 設計手法	表を順序付きリストに移動して、説明をアップデート。
SDAccel を使用したアクセラレーションのベスト プラクティス	説明をアップデート。
第 3 章: SDAccel プロジェクトの作成	説明および図をアップデート。
アプリケーション プロジェクトの作成	セクションをアップデート。
SDx GUI の理解	図をアップデート。
SDx の [Assistant] ビュー	図をアップデート。
XOCC リンクおよびコンパイル オプション	セクションを追加。
SDx プロジェクトのエクスポート	[Export SDx Projects] ダイアログ ボックスの図をアップデート。
SDx プロジェクトのインポート	[Import Projects] ダイアログ ボックスの図をアップデート。
ホスト アプリケーションのコード記述	説明をアップデート。
C/C++ カーネルの記述	説明をアップデートし、ポインター引数およびスカラーをセクションに分割。
第 5 章: システムのビルド	プロジェクト エディター ビューの図とセクション ヘディングをアップデート。
ハードウェアのビルド	サンプル コードを \$ でクリーンアップし、SDAccel リンキング、複数計算ユニットのインスタンスシート、および XOCC コンパイル設定の図をアップデート。[Assistant] ビューの最上位カーネルの図を削除。
メモリ リソースへのカーネル インターフェイスのマップ	説明をアップデート。
計算ユニットの SLR への割り当て	説明をアップデート。
インプリメンテーション結果の制御	説明をアップデート。

セクション	改訂内容
レポート生成の制御	説明をアップデート。
システム見積もりレポート	タイトルおよび最初の段落をアップデート。
プロファイル サマリ レポート	説明をアップデート。
アプリケーション タイムライン	説明、ヒント、およびリンクをアップデート。
波形ビューおよびライブ波形ビューアー	説明をアップデート。
カーネル SLR および DDR メモリの割り当て	セクションを追加。
デバッグの機能および手法	表をアップデート。
システム	説明をアップデート。
第 8 章: コマンド ラインでのアプリケーションのビルド	説明をアップデート。
コンパイル	説明をアップデート。
ハードウェアのビルド	セクション内の説明をアップデート。
sdaccel.ini ファイルの使用	3 つの表に分割。
カーネルのソフトウェア要件	セクションをアップデート。
カーネルのインターフェイス要件	説明および S_AXI_CONTROL エントリをアップデートし、レジスタを追加。
割り込み	リンクを追加。
RTL Kernel ウィザード	説明をアップデート。
[Kernel options]	説明を追加。
[Streaming Interfaces] ページ	セクションを追加。
RTL カーネルをサイリンクス オブジェクト ファイルにパッケージ	説明をアップデート。
第 10 章: SDAccel への HLS カーネル デザインの統合	章を追加。
便利なコマンド ライン ユーティリティ	付録を追加。
カーネルの SLR 割り当て	付録を削除。
2018 年 10 月 2 日 バージョン 2018.2.xdf	
第 1 章: SDAccel の概要	サポートされるプラットフォームを Alveo™ U200 および U250 データセンター アクセラレータ カードを含めるようにアップデート。
資料全体	xbsak および xbinst コマンドを xbutil コマンドに置換。
アプリケーション プロジェクトの作成	必要なプラットフォームを別々にインストールする必要があることを示す注記を追加。
付録 E: 新規ターゲット プラットフォームへの移行	付録を追加。
メモリ リソースへのカーネル インターフェイスのマッピング	--sp オプションを使用してカーネルのインターフェイス/ポートすべてを割り当てる必要があることを示す注記を追加。
2018 年 8 月 13 日 バージョン 2018.2	
付録 F: ボードのインストールとデバッグ	削除。 ボードのインストール手順は、該当するボードのユーザー ガイドを参照してください。 詳細は、『KCU1500 ボード ユーザー ガイド』 (UG1260) および『VCU1525 リコンフィギュラブル アクセラレーション プラットフォーム ユーザー ガイド』 (UG1268) を参照してください。
2018 年 7 月 2 日 バージョン 2018.2	
資料全体	マイナーな編集。
ボードのインストールおよびデバッグ手順	内容をコマンド ライン ベースのインストール フローに変更。

セクション	改訂内容
第 9 章: RTL カーネル	重複する内容を削除。
第 7 章: 第 7 章: アプリケーションおよびカーネルのデバッグ	MicroBlaze デバッグ関する記述を削除。
2018 年 6 月 20 日 バージョン 2018.2	
ボードのインストール、プログラム、およびデバッグ	付録を追加。
2018 年 6 月 6 日 バージョン 2018.2	
大部分の内容を大幅にアップデート。	
第 3 章: SDAccel プロジェクトの作成	[Assistant] ビューの説明を追加。
第 4 章: SDAccel のプログラム	ホスト アプリケーションのコード記述に内容を追加。
第 5 章: システムのビルド	ホスト アプリケーションのビルドおよびハードウェアのビルドプロセスの詳細を追加。
第 7 章: アプリケーションおよびカーネルのデバッグ	デバッグ プロセスの概要を示し、詳細は別のユーザー ガイドに移動。
第 8 章: コマンド ラインでのアプリケーションのビルド	ホスト コードおよびカーネル コードのコンパイルおよびリンクに使用するコマンドの説明を追加。
第 9 章: RTL カーネル	内容を大幅にアップデート。
付録 B: ディレクトリ構造	アプリケーション プロジェクトのディレクトリ構造を説明。
カーネルの SLR 割り当て	--xp オプションを使用したカーネル配置の制御について説明。
付録 F: プライベート デバッグ ネットワーク用の JTAG フォールバック	RTL カーネルのリモート デバッグの問題を説明。
2018 年 4 月 4 日 バージョン 2018.1	
xbsak コマンドおよびオプション	--apm コマンド オプションを --spm に変更。
ザイリンクス OpenCL コンパイラを使用した OpenCL カーネルのコンパイル	--pk コマンド オプションを --profile_kernel に変更。

目次

改訂履歴.....	2
第 1 章: SDAccel の概要.....	8
SDAccel を使用したソフトウェア アクセラレーション.....	8
SDAccel 実行モデル.....	9
SDAccel のビルド プロセス.....	11
SDAccel 設計手法.....	13
SDAccel を使用したアクセラレーションのベスト プラクティス.....	15
第 2 章: はじめに.....	16
第 3 章: SDAccel プロジェクトの作成.....	17
SDx ワークスペースの使用.....	17
アプリケーション プロジェクトの作成.....	18
SDx GUI の理解.....	21
SDx の [Assistant] ビュー.....	22
SDx プロジェクトのエクスポートとインポート.....	27
ソースの追加.....	32
第 4 章: SDAccel のプログラム.....	36
ホスト アプリケーションのコード記述.....	36
カーネル言語サポート.....	39
第 5 章: システムのビルド.....	43
ホスト アプリケーションのビルド.....	44
ハードウェアのビルド.....	45
ビルド ターゲット.....	54
第 6 章: プロファイリングおよび最適化.....	57
設計ガイダンス.....	58
システム見積もりレポート.....	59
HLS レポート.....	60
プロファイル サマリ レポート.....	61
アプリケーション タイムライン.....	63
波形ビューおよびライブ波形ビューアー.....	64
カーネル SLR および DDR メモリの割り当て.....	66
第 7 章: アプリケーションおよびカーネルのデバッグ.....	70

デバッグの機能および手法.....	70
第 8 章: コマンド ラインでのアプリケーションのビルド.....	73
ホストのビルド.....	73
ハードウェアのビルド.....	75
sdaccel.ini ファイルの使用.....	77
[emconfigutil] 設定.....	80
第 9 章: RTL カーネル.....	81
RTL カーネルとして RTL デザインを使用するための要件.....	81
RTL Kernel ウィザード.....	86
RTL カーネルの手動開発フロー.....	101
RTL 設計の推奨事項.....	105
第 10 章: SDAccel への HLS カーネル デザインの統合.....	109
Vivado HLS を使用した SDAccel カーネルの作成.....	110
SDAccel への Vivado HLS カーネル プロジェクトの取り込み.....	114
既知の制限.....	114
付録 A: サンプル デザインの概要.....	115
サンプル デザインのインストール.....	115
ローカル コピーの使用.....	117
付録 B: ディレクトリ構造.....	119
コマンド ライン.....	119
GUI.....	120
付録 C: 便利なコマンド ライン ユーティリティ.....	122
付録 D: プラットフォームおよびリポジトリの管理.....	123
付録 E: 新規ターゲット プラットフォームへの移行.....	125
デザインの移行.....	125
リリースの移行.....	130
カーネル配置の変更.....	131
タイミングの解決.....	137
付録 F: プライベート デバッグ ネットワーク用の JTAG フォールバック.....	139
JTAG フォールバック手順.....	139
付録 G: その他のリソースおよび法的通知.....	140
ザイリンクス リソース.....	140
Documentation Navigator およびデザイン ハブ.....	140
参考資料.....	140
お読みください: 重要な法的通知.....	141

SDAccel の概要

SDAccel™ 環境は、標準プログラミング言語を使用して、FPGA でアクセラレーションされたデータセンター アプリケーションを開発して配布するためのフレームワークを提供します。SDAccel 環境には、Eclipse ベースの統合設計環境 (IDE) と、FPGA リソースを効率的に活用するよう最適化されたコンパイラを使用した、ソフトウェア開発フローが含まれます。アクセラレーション アプリケーションの開発者は、使い慣れたソフトウェア プログラミング ワークフローを使用して、FPGA またはハードウェア デザインの経験がなくても、FPGA アクセラレーションを利用できます。アクセラレーション カーネルの開発者は、ハードウェア中心設計手法を使用して、標準プログラミング言語を HLS コンパイラでコンパイルして、ソフトウェア コンポーネントおよびハードウェア コンポーネントの両方を含むヘテロジニアス アプリケーションを生成できます。ソフトウェア コンポーネント (アプリケーション) は C/C++ と OpenCL™ API 呼び出しを使用して開発し、ハードウェア コンポーネント (カーネル) は C/C++、OpenCL、または RTL を使用して開発します。SDAccel 環境ではさまざまな設計手法を使用でき、ソフトウェア コンポーネントからでもハードウェア コンポーネントからでも開発を開始できます。

ザイリンクス FPGA には、プロセッサで実行可能などの関数でもインプリメントできるカスタム アーキテクチャなど、従来の CPU/GPU アクセラレーションと比べて多くの利点があり、消費電力を低く抑えながら高パフォーマンスを達成できます。ザイリンクス デバイスでソフトウェアをアクセラレーションできる利点を活用するには、アプリケーションで計算負荷の高い部分をアクセラレーションします。これらの関数をカスタム ハードウェアにインプリメントすると、パフォーマンスと消費電力の理想的なバランスを達成できます。SDAccel 環境には、アプリケーションのパフォーマンスをプロファイリングし、アクセラレーションできる部分を特定するためのツールとレポートが含まれています。また、キャッシュ、メモリ、バス使用量の自動ランタイム インストールメンテナーも提供され、ハードウェア上でのリアルタイム パフォーマンスを確認できます。

SDAccel 環境は、ザイリンクス Alveo™ データセンター カードなどのアクセラレーション ハードウェア プラットフォームをターゲットにしています。このアプリケーション プラットフォームは、計算負荷の高いアプリケーション、特にライブ ビデオ トランスコーディング、データ解析、機械学習を使用する人工知能 (AI) アプリケーション用に開発されています。SDAccel 環境と互換性を持つサードパーティ アクセラレーション プラットフォームも多数あります。

アクセラレーション ML 推論アプリケーションを最適化および運用するザイリンクス機械学習 (ML) スイートなどの多数の FPGA アクセラレーション ライブラリは、SDAccel 環境から使用できます。定義済みのアクセラレータ関数には、人口知能 (Caffe、MxNet、TensorFlow などの一般的な機械学習フレームワークを多数含む)、ビデオ処理、暗号化、ビッグ データ解析などのターゲット アプリケーションがあります。これらのザイリンクスおよびサードパーティ開発者から提供されている定義済みアクセラレータ ライブラリを、アクセラレーション アプリケーション プロジェクトに統合すると、開発期間を短縮できます。

SDAccel 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 FPGA 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。カーネル用に SDAccel 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り、パフォーマンスを大幅に向上できる可能性があります。既存のアプリケーションを FPGA でアクセラレーションするようリターゲットすることもできますが、FPGA アーキテクチャを理解し、それに基づいてホストおよびカーネル コードを変更することにより、パフォーマンスを大幅に向上できます。ホストおよびカーネル コードの記述およびそれらの間のデータ転送に関する詳細は、『SDAccel 環境 プログラマ ガイド』([UG1277](#)) を参照してください。

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つの値の加算のような単純な演算でも、複数クロックサイクルで実行される複数のアセンブリ命令になります。これが、キャッシュヒット率を上げて命令ごとのプロセッササイクル数を削減するためにアルゴリズムを再構築するのにソフトウェアエンジニアが時間を費やす理由です。

一方、FPGA は本質的に並列処理デバイスであり、プロセッサ上で実行可能などんな演算関数でもインプリメントできます。ザイリンクス FPGA 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。すべての計算で同じ ALU が共有されるプロセッサとは異なり、FPGA では演算がプロセッシングリソースの設定可能な配列に分配されて実行されます。FPGA コンパイラにより、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固有の回路が作成されます。FPGA プログラミングファブリックは、アクセラレーション関数を定義してインプリメントする空のキャンバスとして機能します。

SDAccel コンパイラは、スケジューリング、パイプライン処理、およびデータフローの処理により FPGA ファブリックの機能を活用します。

- **スケジューリング:** 異なる演算間のデータおよび制御の依存性を特定し、各演算をいつ実行するかを決定するプロセスです。コンパイラにより隣接する演算間および時間を越えた依存性が解析され、演算を同じクロックサイクルで実行したりデータフロー依存性で許容される場合に関数呼び出しをオーバーラップさせるため、演算がグループ化されます。
- **パイプライン処理:** 演算または関数の独立した段階をオーバーラップさせることにより、アルゴリズムのハードウェアインプリメンテーションの命令レベルの並列処理を増加する手法です。機能が等価になるよう元のソフトウェアインプリメンテーションのデータ依存性は保持されますが、必要な回路は独立した段のチェーンに分割されます。チェーンのすべての段は、同じクロックサイクルで並列実行されます。パイプライン処理は詳細な最適化で、次の関数呼び出しまたは演算を開始する前に現在の関数呼び出しまたは演算を完全に終わらせる必要があるという CPU の制限を取り除きます。
- **データフロー:** タスクレベルの並列処理をインプリメントし、FPGA にインプリメントした複数の関数を並列にパイプライン方式で実行できるようにします。コンパイラにより、プログラムの異なる関数間の相互関係が入力と出力に基づいて評価され、このレベルの並列処理が抽出されます。ソフトウェア実行では、この変換は 1つのカーネル内の関数の並列実行に適用されます。

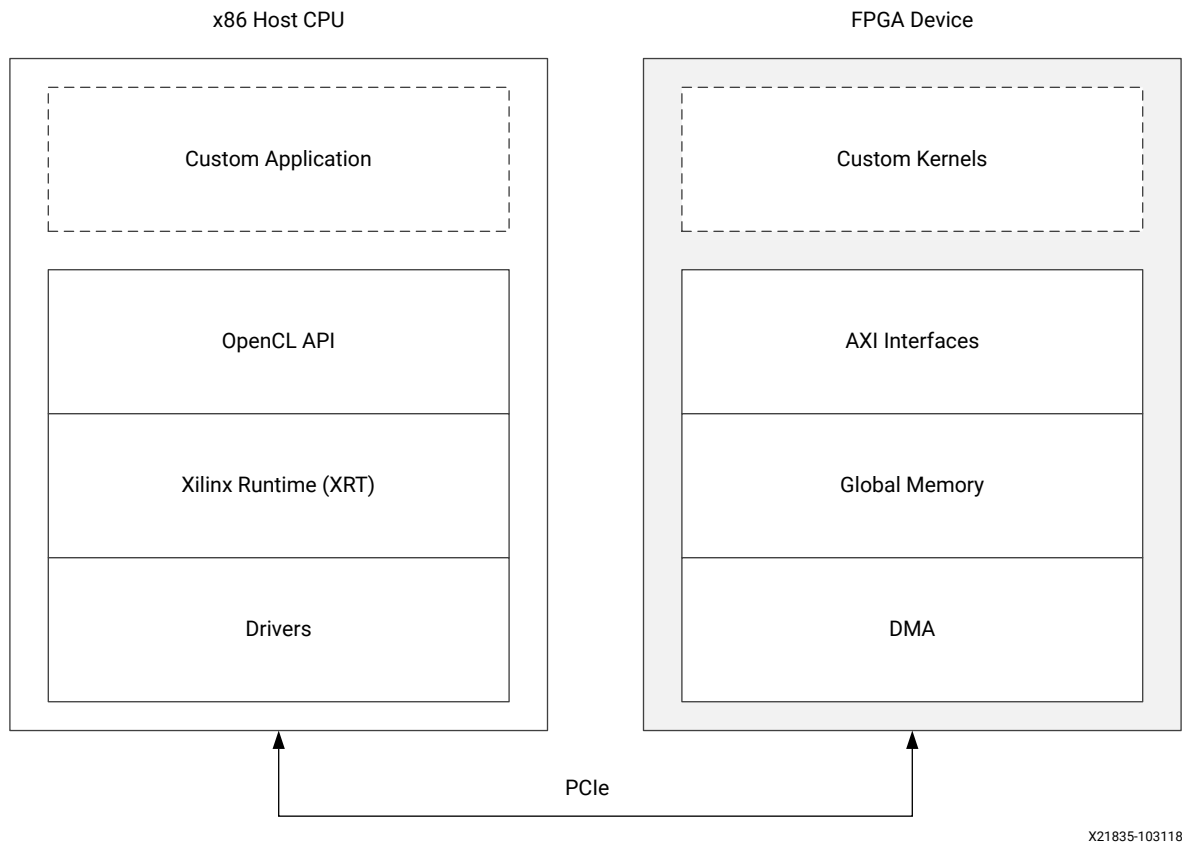
ザイリンクス FPGA のもう 1つの利点は、動的にリコンフィギュレーション可能であるということです。コンパイル済みのプログラムをプロセッサに読み込むのと同様に、ランタイムで FPGA をリコンフィギュレーションすると、FPGA のリソースを別の目的に使用して、アクセラレーションアプリケーション実行としてカーネルを追加します。これにより、1つの SDAccel アクセラレータボードで 1つのアプリケーション内の複数の関数を順次または同時にアクセラレーションできます。

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ン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 1: SDAccel アプリケーションのアーキテクチャ



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 PCIe 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。

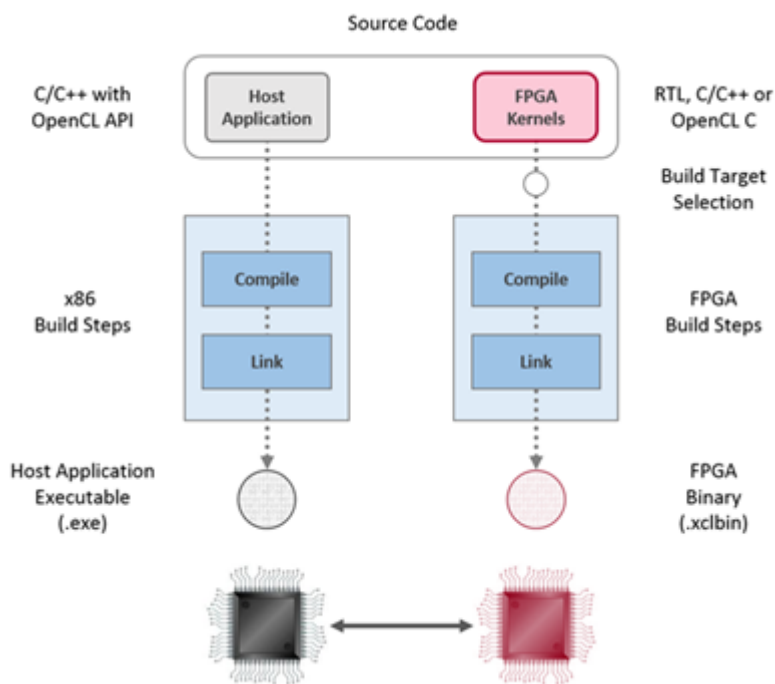
SDAccel のビルド プロセス

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 バイナリはサイリンクス `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 2: ソフトウェア/ハードウェアのビルド プロセス



X22015-112618

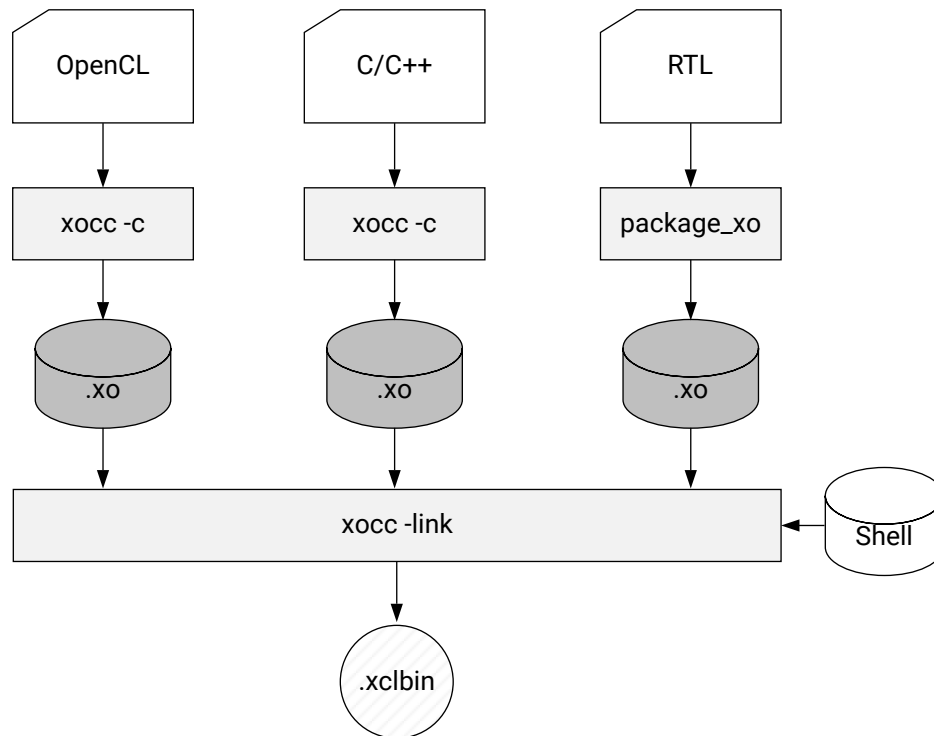
1. GCC を使用したホスト アプリケーションのビルド プロセス:

- 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) をサイリンクス 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 3: 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 設計手法

SDAccel 環境は、主に次の 2つのユース ケースをサポートします。

- ソフトウェア中心設計: ソフトウェア プログラマが記述したアプリケーションをプロファイリングして計算負荷の高い関数またはボトルネックを特定し、それをアクセラレーションすることによりアプリケーションのパフォーマンスを向上します。
- ハードウェア中心設計: アクセラレーション カーネル開発者が最適化されたカーネルを作成し、それをアプリケーション開発者がライブラリ エlementとして呼び出します。カーネル言語は、この設計手法特定のものではありません。ソフトウェア中心フローでもカーネルに C/C++、OpenCL または RTL のいずれかを使用できます。この 2つの設計手法の大きな違いは、ソフトウェア アプリケーションで開始するかカーネルで開始するかです。

これらの 2つのユース ケースを組み合わせ、ソフトウェアおよびハードウェアの開発者がアクセラレータ カーネルを定義し、それらを使用するアプリケーションを開発できます。これらを組み合わせる手法には、異なる開発者または異なる企業が開発したさまざまなアプリケーション コンポーネントが使用されます。アクセラレーションするアプリケーションに定義済みのカーネル ライブラリを利用するか、すべてのアクセラレーション関数をチームで開発できます。

ソフトウェア中心設計

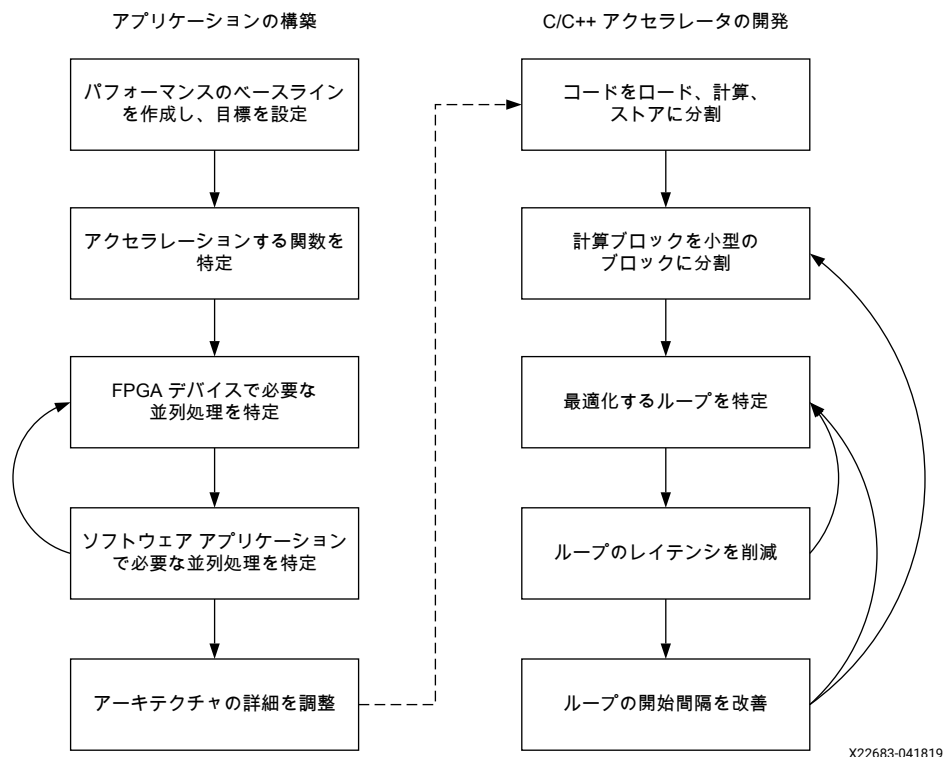
この設計手法には、主に 2つの段階があります。

1. アプリケーションのアーキテクチャ
2. C/C++ カーネルの開発

最初の段階では、どのソフトウェア ファンクションを FPGA カーネルにマップするか、どれくらい並列処理が必要か、どのように送信するかなど、アプリケーション アーキテクチャに関する重要事項を決定します。

次の段階では、カーネルをインプリメントします。この段階では主に、ソース コードを構築して、必要なコンパイラ プラグマを指定し、必要なカーネル アーキテクチャを作成してパフォーマンス目標を達成できるようにします。

図 4: 設計手法の概要



X22683-041819

SDAccel ツールでの設計手法の詳細は、『SDAccel 設計手法ガイド』 ([UG1346](#)) を参照してください。

ハードウェア中心設計

ハードウェア中心設計手法フローでは、カーネルの開発および最適化に焦点を置き、高度な FPGA 設計手法を活用します。詳細は、『SDAccel 環境プロファイリングおよび最適化ガイド』 ([UG1207](#)) を参照してください。ハードウェア中心設計手法フローでは、通常次の手順を使用します。

1. アプリケーションの機能とパフォーマンスのベースラインを作成し、ハードウェアでアクセラレーションする関数を特定します。
2. サイクルバジェットとパフォーマンス要件を見積もり、アクセラレータのアーキテクチャとインターフェイスを定義します。
3. アクセラレータを開発します。
4. 機能とパフォーマンスを検証します。必要に応じて繰り返します。
5. タイミングおよびリソース使用率を最適化します。必要に応じて繰り返します。
6. カーネルを SDAccel にインポートします。
7. サンプル ホスト コードを開発し、実際のカーネルと同じインターフェイスを含むダミー カーネルを使用してテストします。
8. ハードウェアエミュレーションを使用するか実際のハードウェア上で実行して、カーネルがホスト コードで正しく機能するかを検証します。必要に応じて繰り返します。
9. アクティビティ タイムライン、プロファイル サマリ、およびソース コードのタイマーを使用してパフォーマンスを計測し、ホスト コードのパフォーマンスを最適化します。必要に応じて繰り返します。

SDAccel を使用したアクセラレーションのベスト プラクティス

SDAccel 環境でアプリケーション コードおよびハードウェア関数を開発する際は、次の事項を考慮してください。その他の情報は、『SDAccel 環境プロファイリングおよび最適化ガイド』(UG1207) を参照してください。

- 入力および出力のデータ量に対して計算時間の比率が高い関数をアクセラレーションします。FPGA カーネルを使用すると計算時間は大幅に短縮されますが、データ量により転送レイテンシが追加されます。
- 自己完結型の制御構造を持ち、ホストとの定期的な同期を必要としない関数をアクセラレーションします。
- ホストからグローバル デバイス メモリに大型のデータ ブロックを転送します。小型の転送を複数実行するよりも、1 つの大型転送を実行する方が効率的です。帯域幅テストを実行して最適な転送サイズを検出します。
- ホストにデータをコピーするのは、必要なときのみに行います。カーネルによりグローバル メモリに書き込まれたデータは、別のカーネルで直接読み出すことができます。メモリ リソースには、PLRAM (サイズは小さいが最短レイテンシで高速アクセスが可能)、HBM (中程度のサイズで多少のレイテンシあり)、DDR (サイズは大きいがレイテンシは最長になるので低速アクセス) などが含まれます。
- 複数のグローバル メモリ リソースを活用して、帯域幅を複数のカーネルに均等に分配します。
- 512 ビット幅のバーストを実行して、カーネルとグローバル メモリ間の帯域幅を最大限にします。
- カーネル内のローカル メモリにデータをキャッシュします。ローカル メモリにアクセスする方が、グローバル メモリにアクセスするよりもかなり高速です。
- ホスト アプリケーションで、イベントおよびノンブロッキング トランザクションを使用して、複数の要求を並列にオーバーラップさせて実行します。
- FPGA では、タスク レベルの並列処理を活用できるよう異なるカーネルを使用し、データ レベルの並列処理を活用できるよう複数の CU を使用して、複数のタスクを並列実行することによりパフォーマンスをさらに向上します。
- カーネル内でデータフローを使用したタスク レベルと、ループ展開とループのパイプラインを使用した命令レベルの並列処理を活用して、スループットを最大にします。
- 一部のザイリンクス FPGA には、複数のパーティション (SLR (Super Logic Region) と呼ばれる) が含まれます。カーネルをカーネルがアクセスするグローバル メモリ バンクと同じ SLR に配置します。
- ソフトウェアおよびハードウェア エミュレーションを使用してコードの周波数を検証し、正しく機能することを確認します。
- SDAccel ガイダンス レポートを頻繁に参照します。このレポートには、プロジェクトについて明確で実用的なアドバイスが示されます。

はじめに

『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](#)) の手順に従って SDAccel™ ツール スイートをダウンロードおよびインストールします。

『Get Moving with Alveo』 ([UG1352](#)) を参照することもお勧めします。この資料には、FPGA アクセラレーション アプリケーション用のソフトウェア コード開発の基礎がわかりやすく説明されています。

ザイリンクス [GitHub](#) からのサンプル デザインのダウンロードおよび使用法は、[付録 A: サンプル デザインの概要](#) を参照してください。

注記: SDAccel ツール スイートには、ビットストリーム、オブジェクト コード、実行ファイルを生成するためのツールすべてが含まれます。ザイリンクス Vivado[®] Design Suite およびソフトウェア開発キット (SDK) ツールを個別にインストールした場合、これらのインストールを SDAccel 環境とまとめないようにしてください。ツールが SDAccel インストール (Vivado Design Suite および SDAccel ツールを含む) から実行されていることを確認してください。



重要: SDAccel アプリケーションは、Linux OS でしか実行できません。SDAccel 環境のソフトウェア要件は、『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](#)) を参照してください。

SDAccel プロジェクトの作成

SDx™ ツール内では、IDE GUI を使用して SDAccel™ プロジェクトを作成できます。このセクションでは、SDx ワークスペースをセットアップし、SDAccel プロジェクトを作成して、IDE の主要な機能を使用する方法を説明します。

SDAccel 環境には、SDx IDE に加えて、[第 8 章: コマンドラインでのアプリケーションのビルド](#)に説明されているようにスクリプトフローをサポートするコマンドラインインターフェイスが含まれます。すべてのコマンドのリストは、『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。

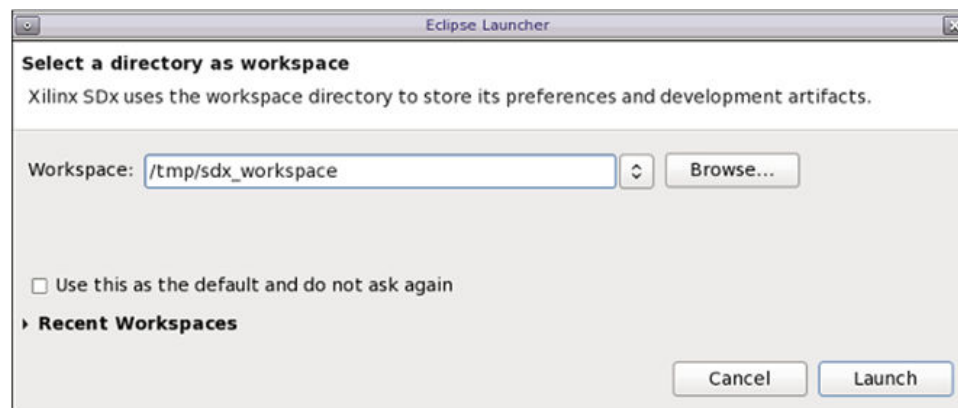
SDx ワークスペースの使用

1. コマンドラインから次のコマンドを使用して SDx IDE を直接起動します。

```
$ sdax
```

2. SDx IDE が開き、次の図に示すワークスペースを選択するダイアログ ボックスが表示されます。

図 5: SDx ワークスペースの指定



重要: SDx コマンドを入力するため新しいシェルを開いた場合は、まず `settings64` および `setup` スクリプトをツール環境に読み込みます。詳細は、『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](#)) を参照してください。



重要: 開発と運用の両方に 1 つのコンピューターを使用する場合は、SDx ツールと `xbutil` ボード インストール ユーティリティを実行する際に別のターミナルを開いてください。同じターミナルから両方のツールを実行すると、環境変数に悪影響を及ぼし、ツールで問題が発生する可能性があります。

SDx ワークスペースは、プロジェクト、ソース ファイル、ツールでの処理結果を含むフォルダーです。プロジェクトごとに個別のワークスペースを定義したり、プロジェクトのタイプ別にワークスペースを定義したりできます。SDAccel プロジェクトのワークスペースを定義するには、次の手順に従います。

1. [Browse] ボタンをクリックしてワークスペースを指定するか、[Workspace] フィールドに適切なパスを入力します。
2. [Use this as the default and do not ask again] をオンにすると、指定したワークスペースがデフォルトになり、今後 SDx を起動したときにこのダイアログボックスは表示されなくなります。
3. [Launch] をクリックします。



ヒント: 現在のワークスペースを変更するには、SDx IDE で [File] → [Switch Workspace] クリックします。

これで SDx ワークスペースが作成されたので、プロジェクトを含めます。プラットフォームおよびアプリケーションプロジェクトが作成され、SDAccel プラットフォームを作成する SDx ツールフローが記述されます。

アプリケーション プロジェクトの作成



ヒント: サンプル デザインは、SDAccel ツールのインストールに含まれますが、ザイリンクス [GitHub](#) リポジトリからもダウンロードできます。詳細は、[付録 A: サンプル デザインの概要](#) を参照してください。

1. SDx IDE を起動したら、新規プロジェクトを作成できます。[File] → [New] → [SDx Project] をクリックするか、これが最初に SDx IDE を起動した場合であれば、ウェルカム画面で [Create Application Project] をクリックします。
2. New SDx Project ウィザードが開きます。
3. [Create a New SDx Application Project] ページでプロジェクトを名前を指定します。[Project name] フィールドにプロジェクト名を指定します。

図 6: [Create a New SDx Application Project] ページ

4. [Use default location] がデフォルトでオンになっており、プロジェクトは SDx ワークスペースのフォルダーに保存されます。このオプションをオフにすると、[Location] でプロジェクトを保存するディレクトリを選択できます。
5. ディレクトリを指定すると、[Choose file system] でデフォルト ファイル システム JSch ([default]) または Eclipse Remote File System Explorer ([RSE]) を選択できます。



重要: プロジェクトのディレクトリとして SDx ワークスペースの親フォルダーを指定することはできません。

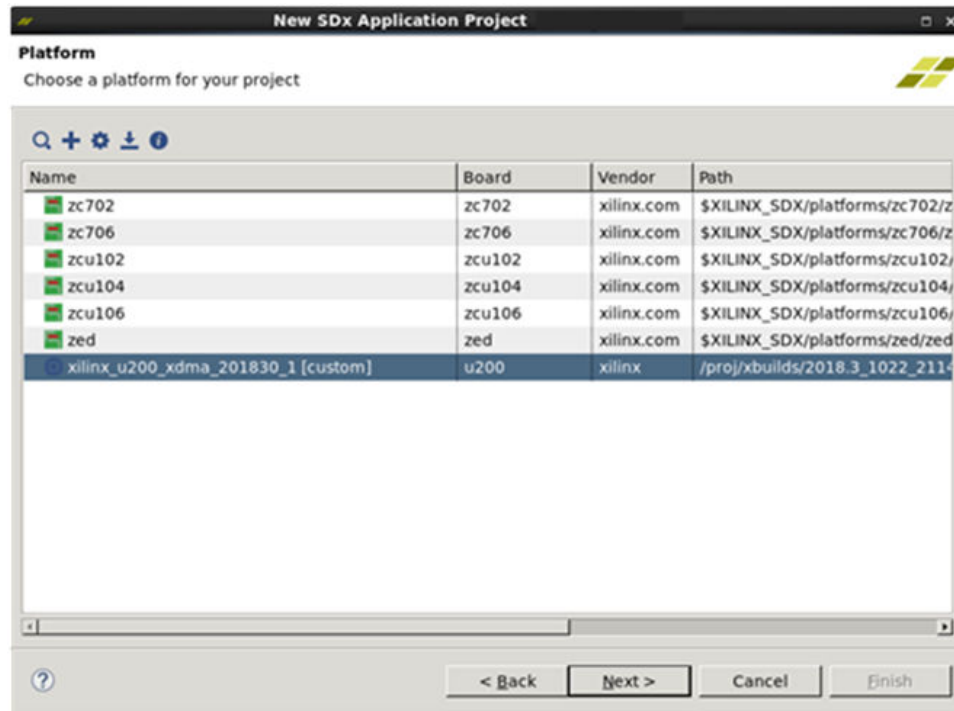
6. [Next] をクリックします。

[Platform] ページには、使用可能なインストール済みプラットフォームが表示されます。追加プラットフォームのインストールの詳細は、『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](#)) の「プラットフォーム別パッケージのインストール」を参照してください。



重要: プラットフォームの選択によりこの後のプロセスが決定されるので、プロジェクトには正しいプラットフォームを選択してください。

図 7: SDAccel プラットフォームの指定

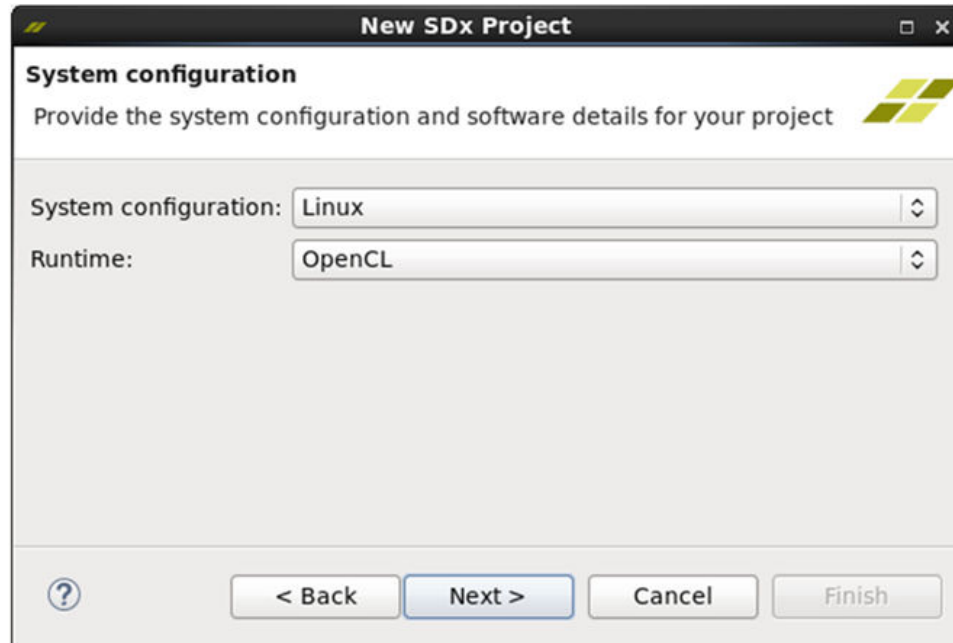


プラットフォームには、ベースハードウェアデザインを記述するシェルと、接続されたアクセラレータで使用されるインターフェイスを記述するメタデータが含まれます。SDAccel では、さまざまなボード用のプラットフォームが提供されています。

リポジトリにはカスタム定義またはサードパーティのプラットフォームを追加できます。詳細は、[付録 D: プラットフォームおよびリポジトリの管理](#)を参照してください。

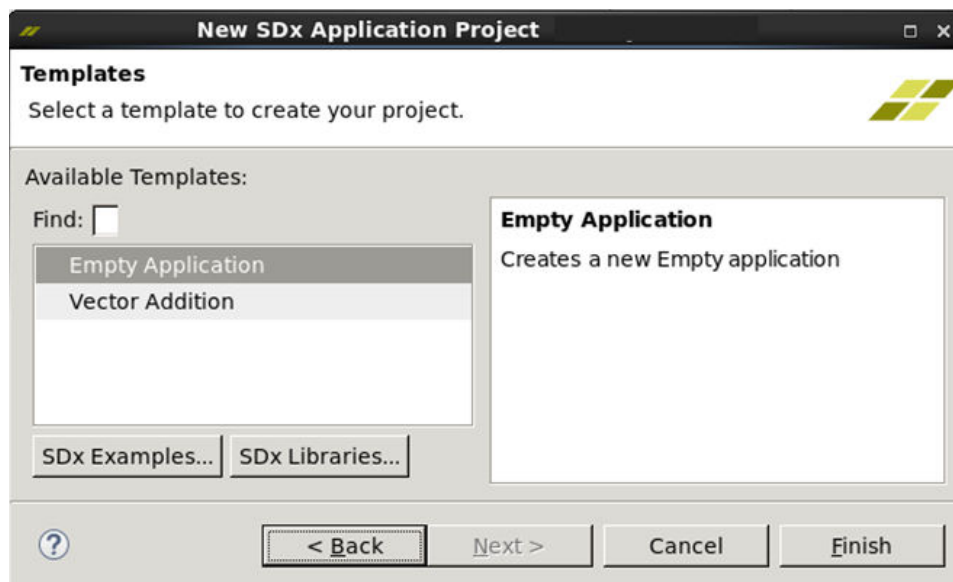
1. プロジェクトのターゲットプラットフォームを選択するには、そのプラットフォームを選択して、[Next] をクリックします。
2. 次の図に示す [System Configuration] ページが開きます。このページでは、システムコンフィギュレーション ([System Configuration]) とランタイム ([Runtime]) をドロップダウンリストから選択します。[System Configuration] ページでは、ハードウェアプラットフォームで実行されるソフトウェア環境を定義します。ハードウェアプラットフォームのプロセッサのオペレーティングシステムと使用可能なランタイム設定を指定します。

図 8: システム設定の指定



3. システム コンフィギュレーション ([System configuration]) を選択して [Next] をクリックすると、次の図に示す [Templates] ページが表示されます。新規プロジェクトのアプリケーション テンプレートを指定します。SDx ツールのインストールエリアの `samples` ディレクトリに、ソース コードの複数のサンプル テンプレートが含まれます。
4. 最初は、[Template] ページには [Empty Application] と [Vector Addition] のみがリストされます。追加の SDAccel サンプルにアクセスするには、[SDx Examples] ボタンをクリックします。

図 9: アプリケーションテンプレート



5. [SDx Examples] ダイアログ ボックスが開いたら、[SDAccel Examples] の [Download] をクリックします。[OK] をクリックします。ダウンロードされたサンプルが [Templates] ページにリストされます。

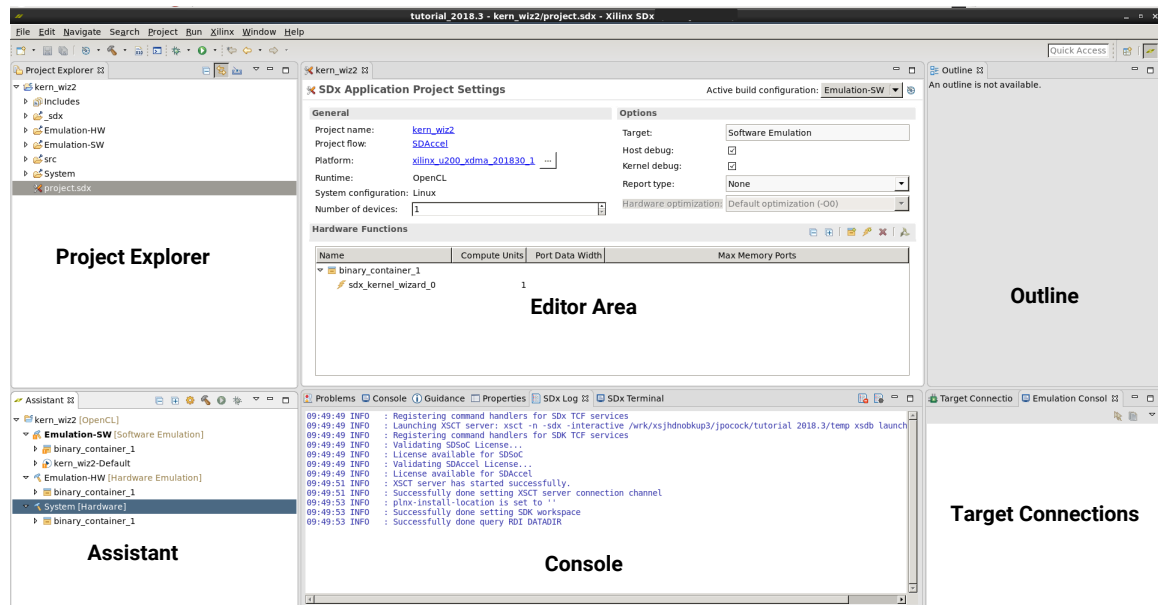
注記:SDxは、ネットワーク設定によって、GitHub リポジトリからサンプルをダウンロードすることもあるので、特定のプロキシ設定が必要なこともあります。

6. テンプレート プロジェクトは、SDx ツールとアクセラレーション カーネルについて学ぶため、または新規プロジェクトの基礎として使用できます。テンプレートの選択は必須です。[Empty Application] を選択すると、空のプロジェクトが作成され、ファイルをインポートして、最初からプロジェクトを作成できます。
7. [Finish] をクリックし、New SDx Project ウィザードを閉じてプロジェクトを開きます。

SDx GUI の理解

SDx IDE でプロジェクトを開くと、ワークスペースにビューおよびエディターが特定の配置 (IDE ではパースペクティブと呼ばれる) で表示されます。次の図に示す SDx パースペクティブ (デフォルト) が開きます。

図 10: SDx - デフォルト パースペクティブ



デフォルト パースペクティブには、次のビューおよびエディターが含まれます。


- [Project Explorer] ビュー: プロジェクト フォルダーおよびソース ファイル、ビルド ファイル、ツールで生成されるレポートをツリー ビューで表示します。
- [Assistant] ビュー: SDAccel アプリケーションの設定の表示および編集、アプリケーションのビルドおよび実行、プロファイリングおよびデバッグセッションの実行、レポートを開くなどの作業を実行できます。
- エディター エリア: プロジェクト設定、ビルド コンフィギュレーションを表示し、プロジェクトを操作するための多くのコマンドにアクセスできます。
- コンソール エリア: コマンド コンソール、デザインのガイダンス、プロジェクト プロパティ、ログ、ターミナルビューなどの複数のビューを表示します。
- [Outline] ビュー: エディター エリアで開いている現在のソース ファイルのアウトラインを表示します。

- [Target Connections] ビュー: Vivado ハードウェア サーバー、TCF (Target Communication Framework)、およびクイックエミュレーター (QEMU) ネットワーキングなどの SDx ツールに接続されるさまざまなターゲットのステータスを表示します。

ビューを閉じるには、ビューのタブ上の [Close] ボタンをクリックします。ビューを開くには、[Window] → [Show View] をクリックし、該当するビューをクリックします。ビューは IDE 内でドラッグアンドドロップして並べ替えることができます。

[Window] → [Perspective] → [Save Perspective As] をクリックすると、ビューの配置をパースペクティブとして保存できます。これにより、最初のプロジェクト編集、レポート解析、デバッグなどのに合わせてパースペクティブを定義できます。パースペクティブとして保存しない変更は、ワークスペースに保存されます。ビューのデフォルトの配置を復元するには、[Window] → [Perspective] → [Reset Perspective] をクリックします。

別のパースペクティブを開くには、[Window] → [Perspective] → [Open Perspective] をクリックします。

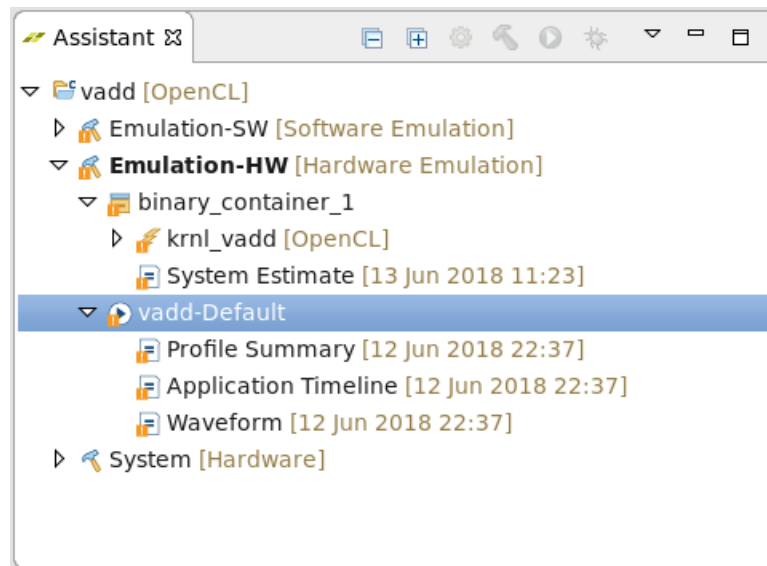
SDx (デフォルト) パースペクティブを復元するには、メインツールバーの右側にある [SDx] ボタン  をクリックします。

SDx の [Assistant] ビュー

[Assistant] ビューは、設定、ビルド、ランタイム、プロファイル、デバッグ、およびレポートを管理する SDx 中心のプロジェクトツリーを示します。[Project Explorer] ビューを補足するビューで、デフォルトで [Project Explorer] ビューの下に表示されます。

次に、[Assistant] ビューとそのツリー構造の例を示します。拡張ハードウェアエミュレーションフローでは、バイナリコンテナの内容と、[Profile Summary]、[Application Timeline]、および [Waveform] のデバッグレポートが示されます。

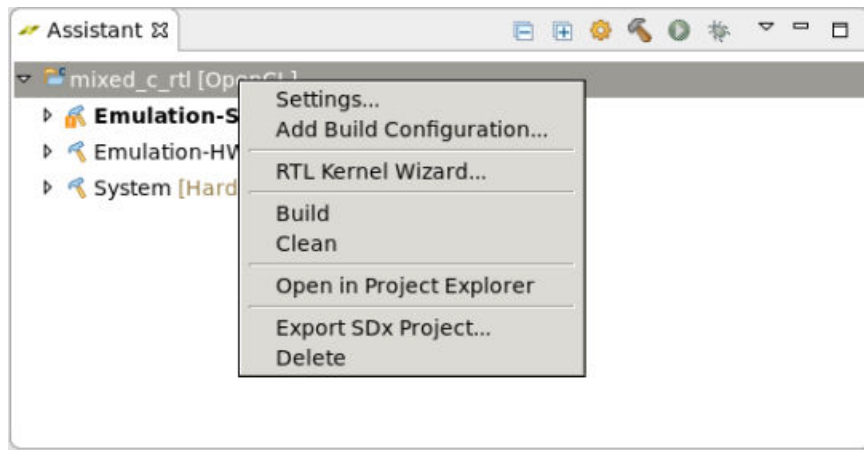
図 11: [Assistant] ビューとそのツリー構造の例



ツリーの各アイテムには、そのアイテム特定の操作を含む右クリックメニューがあります。これらの操作には、ダイアログボックス、レポート、またはビューを開く、プロセスの開始、外部タスクの実行などの操作があります。

たとえば、プロジェクト (次の例の場合 mixed_c_rtl) を右クリックすると、次のメニューが表示されます。

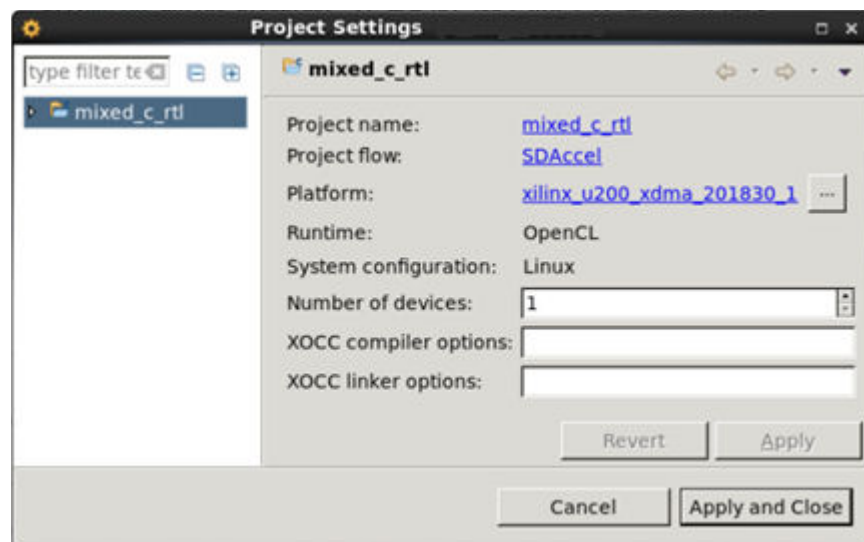
図 12: [Assistant] ビューでの右クリック メニュー



ヒント: アクションが無効になっている、もしくは淡色表示になっている場合は、プロジェクトには該当する情報が無いことを意味します。

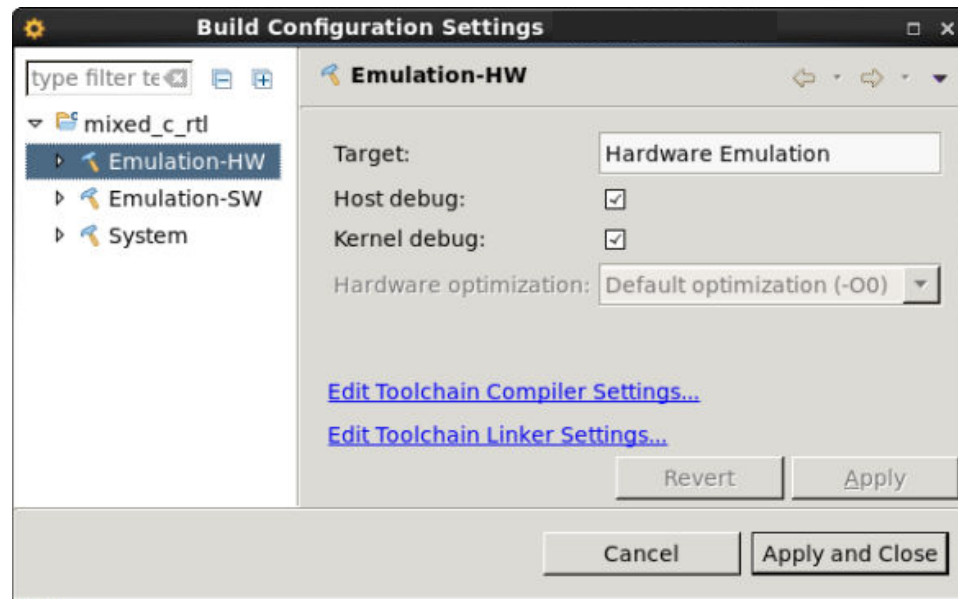
[Settings] をクリックして [Project Settings] ダイアログ ボックスを開きます。

図 13: プロジェクト設定



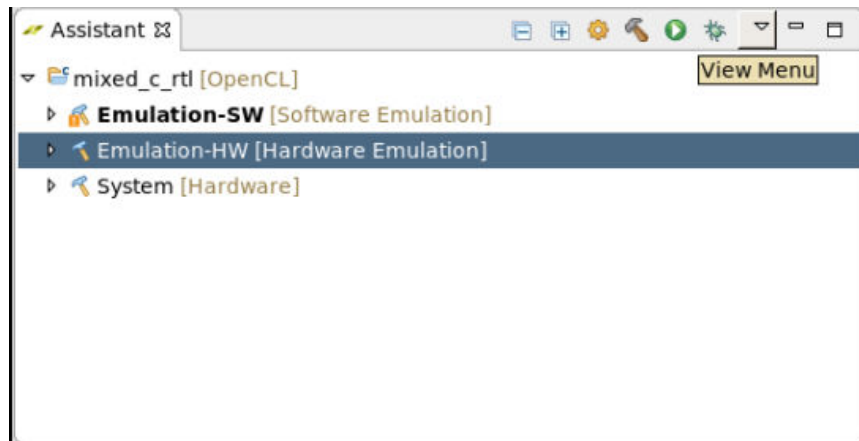
ツリーにリストされているさまざまなアイテムの設定を選択できます。たとえば、[Emulation-HW] ビルド コンフィギュレーションの設定を選択すると、次が表示されます。[Assistant] ビューでは、デザイン オブジェクトを探して、その設定を簡単に表示およびアップデートできます。

図 14: [Emulation-HW]



[View] メニューには [Assistant] ビューの動作に関するオプションが含まれますが、これらのオプションはプロジェクトデータには影響しません。次の図に示す下向き矢印のアイコンをクリックして、オプションを表示します。

図 15: [Assistant] ビューの [View] メニュー



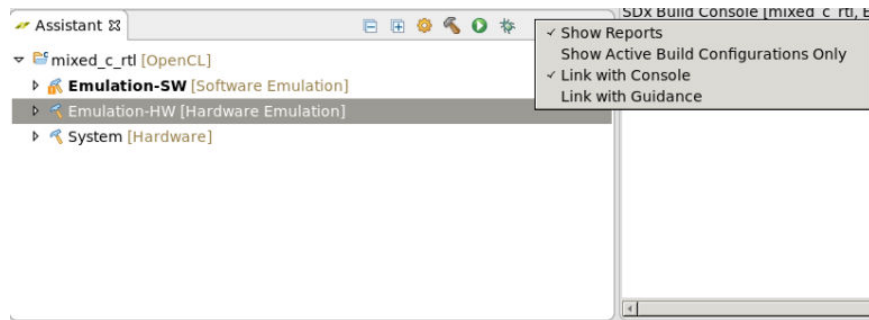
次のオプションが表示されます。

- [Show Reports]: ツリーにレポートを表示します。オフにすると、ツリーにレポートは表示されなくなります。レポートは、通常はプロジェクトを特定の設定でビルドまたは実行した後、プロジェクトにそのレポートが存在する場合にのみ表示されます。
- [Show Active Build Configurations Only]: 各プロジェクトのアクティブなビルド コンフィギュレーションのみを表示します。アクティブビルド コンフィギュレーションは、最後にビルドされたコンフィギュレーションです。[Project] → [Build Configurations] → [Set Active] または [Project] → [Build Configurations] → [Manage] をクリックすると、アクティブビルド コンフィギュレーションを標準 CDT メソッドを表示するよう変更できます。

[Assistant] ビューを使用して特定のビルドを繰り返す場合、現在のビルド コンフィギュレーションのみを表示すると便利です。

- [Link with Console]: [Assistant] ビューのツリーでビルド コンフィギュレーションまたはビルド コンフィギュレーションの下位のアイテムを選択している場合に、[Console] ビューのビルド コンソールを現在選択されているものに自動的に切り替えます。オフにすると、[Assistant] ビューでの選択を変更してもコンソールは自動的に切り替えられません。
- [Link with Guidance]: コンソール エリアの [Guidance] ビューを [Assistant] ビューのツリーで現在選択されているものに自動的に切り替えます。

図 16: [Assistant] ビューの [View] メニューのオプション

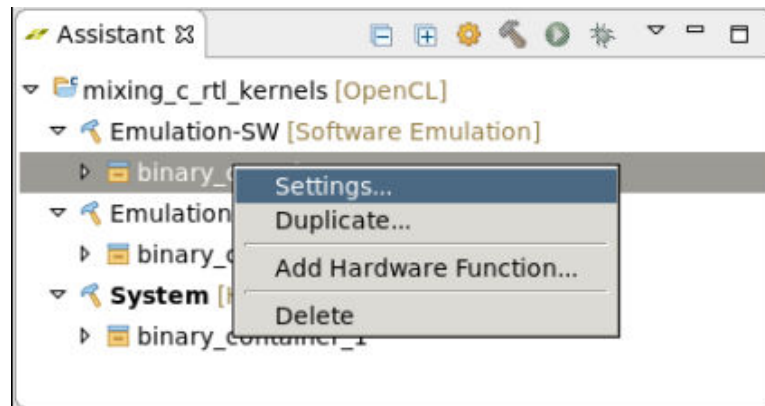


このように、[Assistant] ビューから数回クリックするだけで、ツールのさまざまな機能にアクセスできます。

XOCC リンクおよびコンパイル オプション

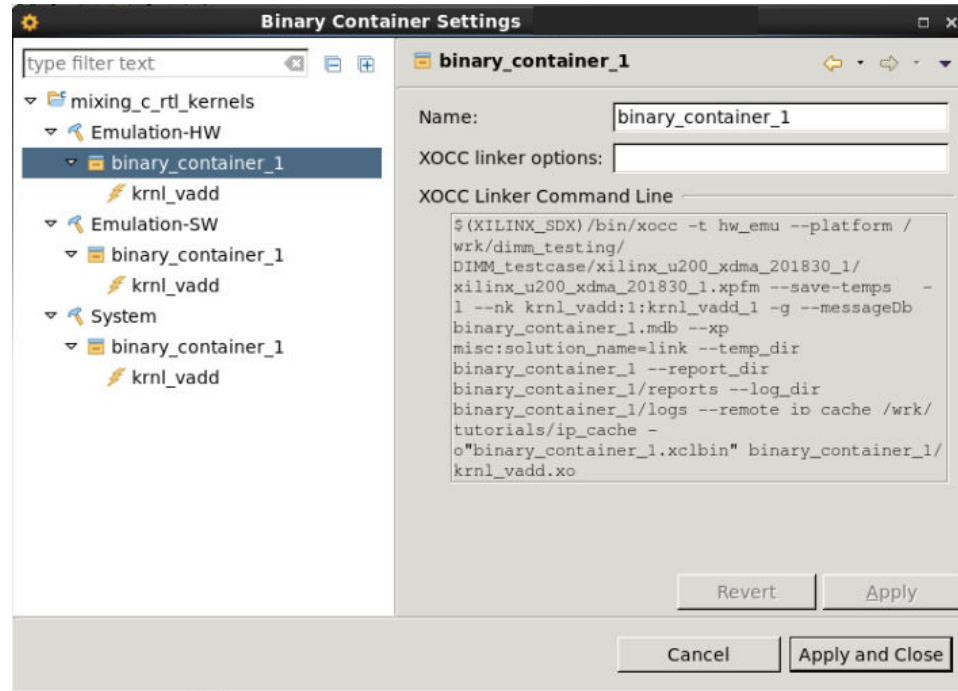
[Assistant] ビューを使用すると、xocc コンパイル オプションとリンク オプションの両方をアップデートできます。まず、[Assistant] ビューで `binary_container` フォルダを右クリックし、[Settings] をクリックします。

図 17: [Assistant] ビューからの XOCC 設定



次のような [Binary Container Settings] ダイアログ ボックスが表示されます。xocc リンカー オプションを追加するには、[XOCC linker options] フィールドにオプションを直接入力します。追加したオプションが `makefile` ([Project Explorer] の下にあり) でアップデートされ、ビルド中に適用されます。アップデートされたオプションは [XOCC Linker Command Line] ボックスにも表示されます。

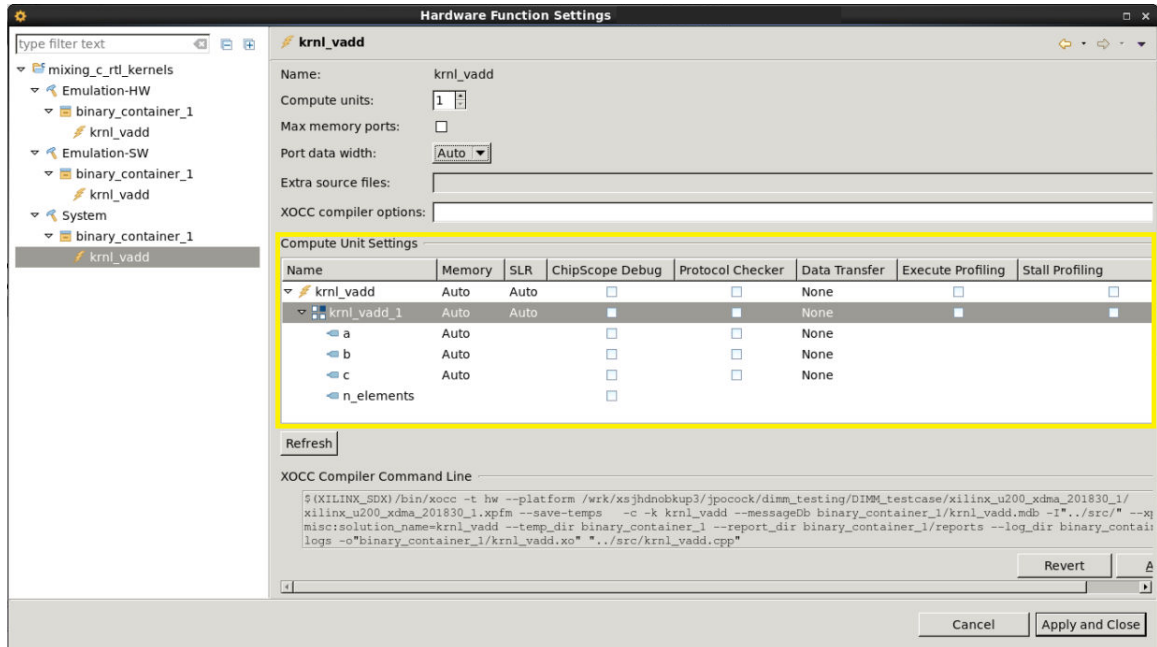
図 18: XOCC リンカー オプション



xocc リンカー オプションは、[Emulation-HW]、[Emulation-SW]、[System] コンフィギュレーション間で少し異なります。ターゲット (-t オプション) は該当するコンフィギュレーションと一致します。また、[System] コンフィギュレーションではデバッグはイネーブルになりません。

xocc コンパイル オプションを特定のカーネルに追加するには、binary_container フォルダでそのカーネルをクリックします。たとえば、次の図の場合、[System] コンフィギュレーションで krnl_vadd カーネルを選択しています。コンパイル オプションは、[XOCC compile options] フィールドに直接入力できます。追加したオプションはそのカーネルにだけ適用され、ほかのカーネルでは共有されません。[XOCC Compiler Command Line] にアップデートしたオプションが表示されます。

図 19: XOCC コンパイラ オプション



カーネルのインスタンス (計算ユニット) の数を設定したり、ポートのデータ幅を設定するオプションがあります。これらのオプションを変更すると、それに関連する `xocc` コンパイル オプションが自動的に生成されて `xocc` コンパイラ コマンドラインに追加されます。

`xocc` リンカー オプションと同様、コンパイラ オプションも [Emulation-HW]、[Emulation-SW]、[System] コンフィギュレーション間で少し異なります。ターゲット (`-t` オプション) は該当するコンフィギュレーションと一致します。また、[System] コンフィギュレーションではデバッグはイネーブルになりません。

[Emulation-HW] および [System] コンフィギュレーションの場合、[Compute Unit Settings] エリア (上記の図の黄色部分) も表示され、さらにリンカー オプションおよびコンパイル オプションを設定できるようになっています。これらの追加オプションは [Emulation-SW] コンフィギュレーションには適用されないため、[Compute Unit Settings] エリアは表示されません。

[Emulation-HW] コンフィギュレーションの場合は、メモリおよび SLR (SLR の詳細は [Super Logic Region](#) を参照) 割り当てのオプションが表示されます。[System] コンフィギュレーションの場合は、追加でプロトコルチェッカーおよびプロファイリングロジックオプションが表示されます。選択したオプションによって、`xocc` リンカー オプションとコンパイル オプションの両方がアップデートされます。

SDx プロジェクトのエクスポートとインポート

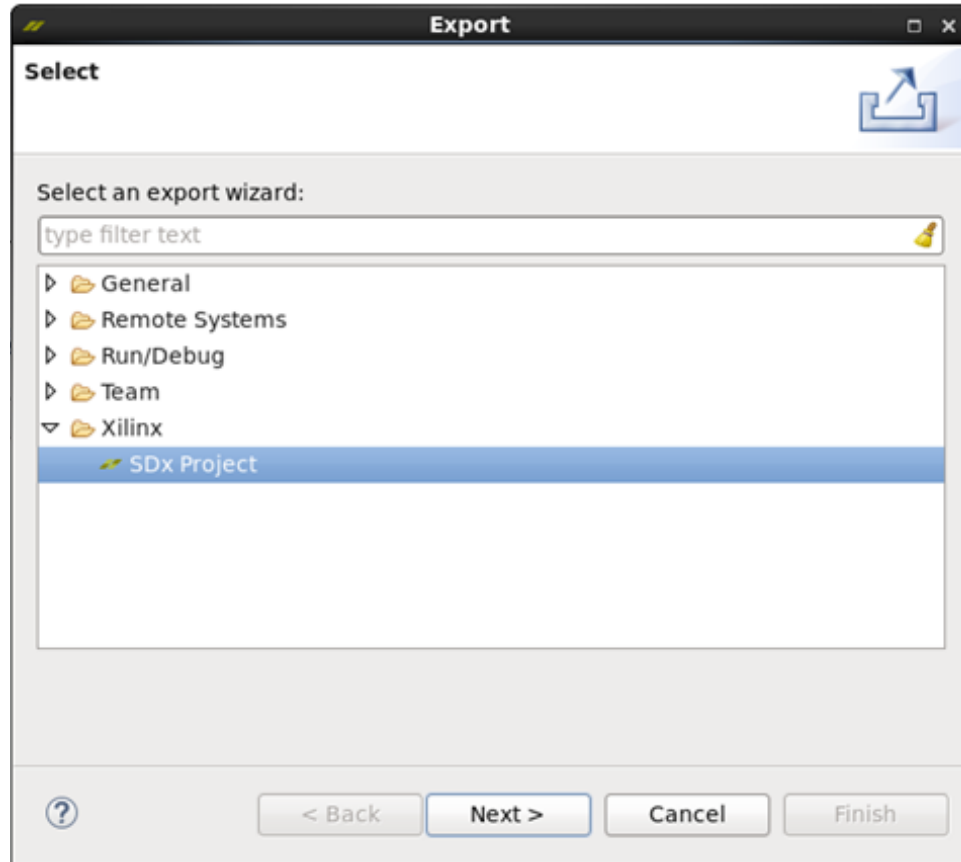
SDx では、ワークスペース内の 1 つまたは複数の SDx プロジェクトを簡単にエクスポート/インポートできます。関連するプロジェクトビルドフォルダーを含めることもできます。

SDx プロジェクトのエクスポート

プロジェクトをエクスポートすると、プロジェクトに関連するすべてのファイルを含めて ZIP ファイルにアーカイブし、別のワークスペースにインポートできます。

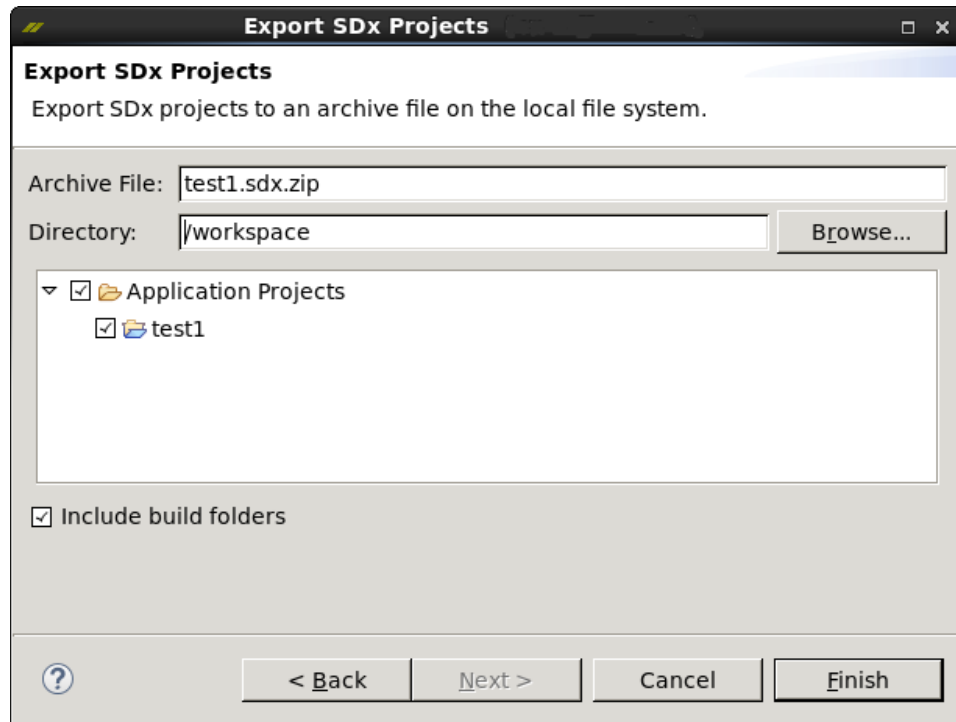
1. プロジェクトをエクスポートするには、[File] → [Export] をクリックします。
2. Export ウィザードで、次の図に示すようにインポートするプロジェクトを選択し、[Next] をクリックします。

図 20: Export ウィザードの選択



3. [Export SDx Projects] ダイアログ ボックスが開き、次の図に示すようにワークスペースのプロジェクトが表示されます。アーカイブに含めるプロジェクトのチェック ボックスをオンにします。アーカイブ ファイルの名前とファイルを保存するディレクトリを指定します。オプションで関連のプロジェクトビルド フォルダーのチェック ボックスをオンにすると、これらのビルド フォルダーも含めることができます。ビルド フォルダーには、レポートおよび BIT ファイルなど、ビルド関連のファイルがすべて含まれます。
4. [Finish] をクリックしてアーカイブ ZIP ファイルを保存します。

図 21: エクスポート ファイル名とビルド フォルダの選択

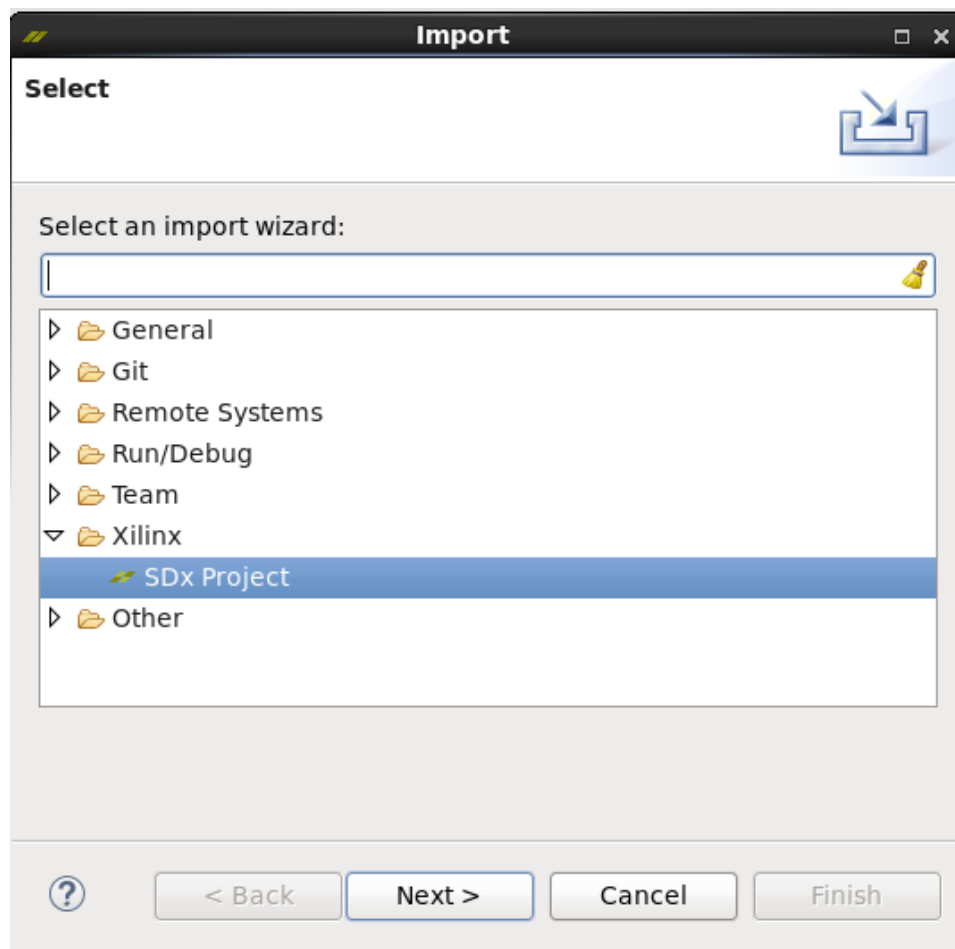


SDx プロジェクトがアーカイブされ、異なるワークスペースにインポートできるようになります。

SDx プロジェクトのインポート

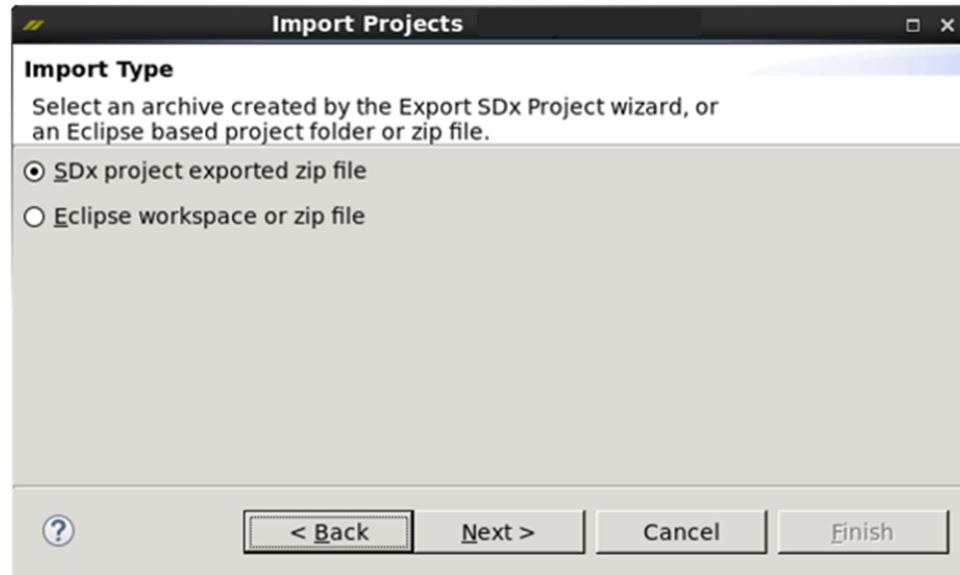
1. SDx プロジェクトをインポートするには、[File] → [Import] をクリックします。
2. Import ウィザードで、次の図に示すように [Xilinx] フォルダの下に [SDx Project] インポート ウィザードを選択し、[Next] をクリックします。

図 22: インポート ウィザードの選択



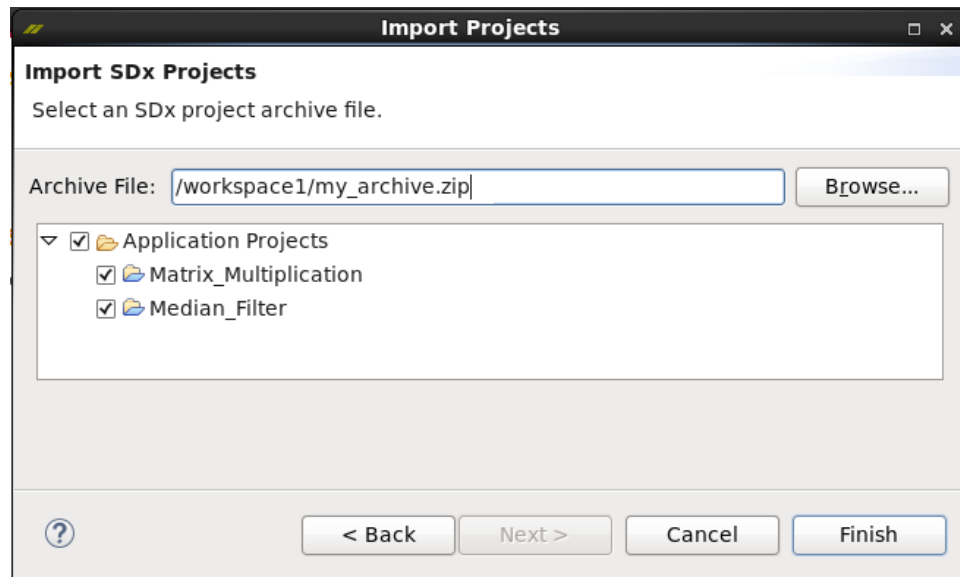
3. インポート ファイル タイプを選択する [Import Project] ダイアログ ボックスが開きます。次の図に示す SDx プロジェクトのエクスポートされた ZIP ファイルを選択して、[Next] をクリックします。

図 23: [Import Projects] ダイアログ ボックスの [Import Type] ページ



4. [Import SDx Projects] ページが開きます。
5. [Browse] をクリックしてアーカイブ ファイルを選択します。アーカイブ プロジェクトが表示されます。
6. インポートするプロジェクトのチェック ボックスをオンにし、[Next] をクリックします。次の図では、両方のプロジェクトをインポートするよう選択しています。
7. [Finish] をクリックしてプロジェクトをワークスペースにインポートします。

図 24: インポートするアーカイブ プロジェクトの選択



ソースの追加

プロジェクトには、ホストアプリケーションコード、カーネル関数、コンパイル済み .xo ファイルなど、さまざまなソース ファイルが含まれています。プロジェクトにソース ファイルを追加するには、SDx IDE でプロジェクトを開


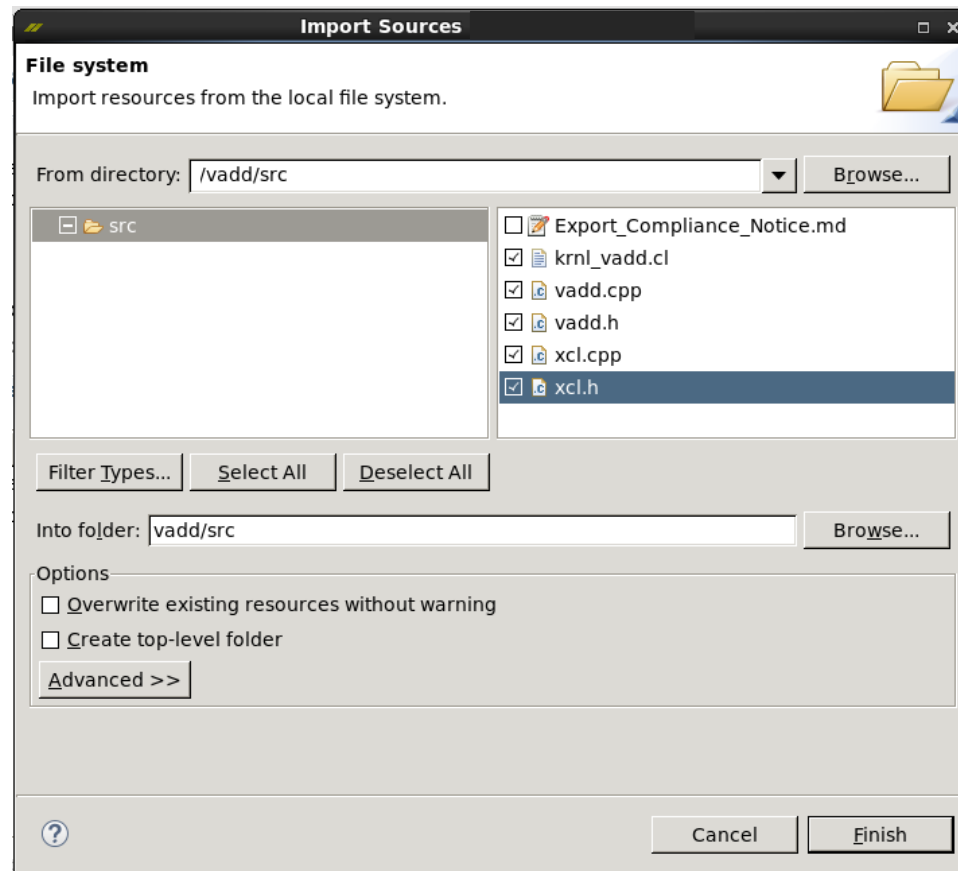
き、[Project Explorer] ビューで `import sources icon` アイコン  をクリックします。次の図に示す [Import Sources] ダイアログ ボックスが表示されます。

図 25: [Import Source] ダイアログ ボックス



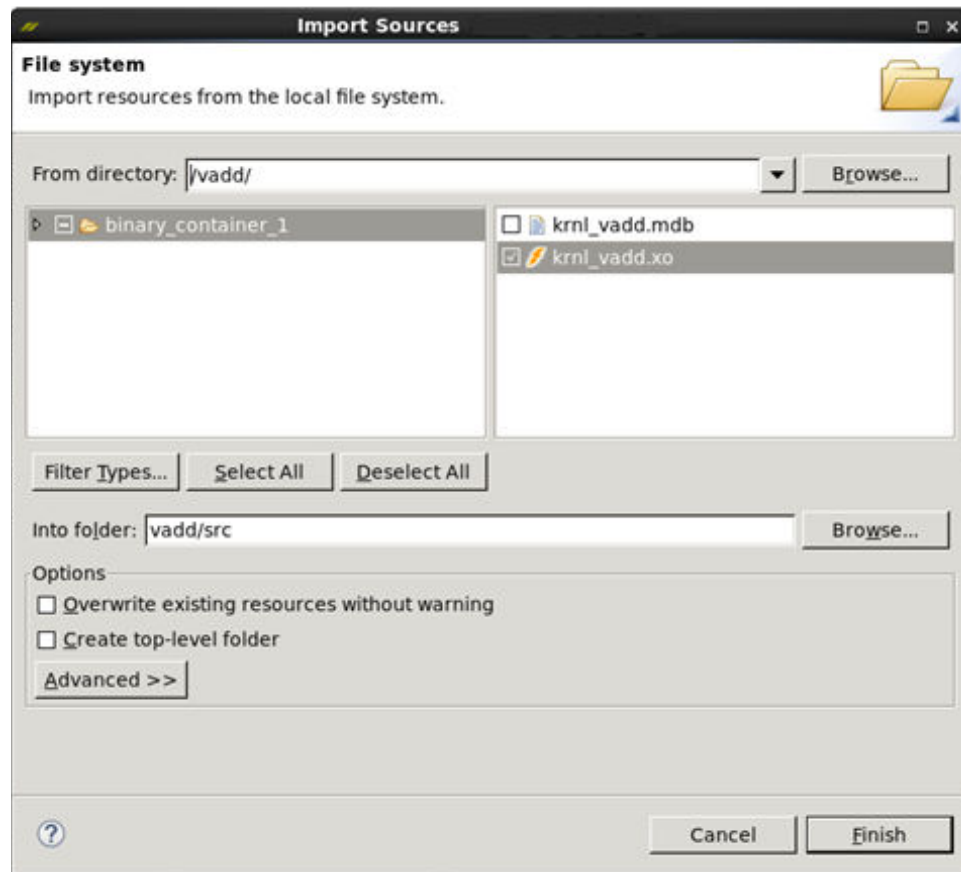
1. このダイアログ ボックスで [Browse] をクリックし、ソースをインポートするディレクトリを選択します。追加するソースを選択し、[Finish] をクリックします。
2. 選択したディレクトリのソース ファイルが表示されます。インポートするソースのチェック ボックスをオンにし、[Finish] をクリックします。次の図では、C/C++、OpenCL™、およびヘッダー ファイルがプロジェクトにインポートされます。



重要: ソース ファイルをワークスペースにインポートすると、ファイルがワークスペースにコピーされます。ワークスペースを削除すると、ファイルへの変更はすべて失われます。

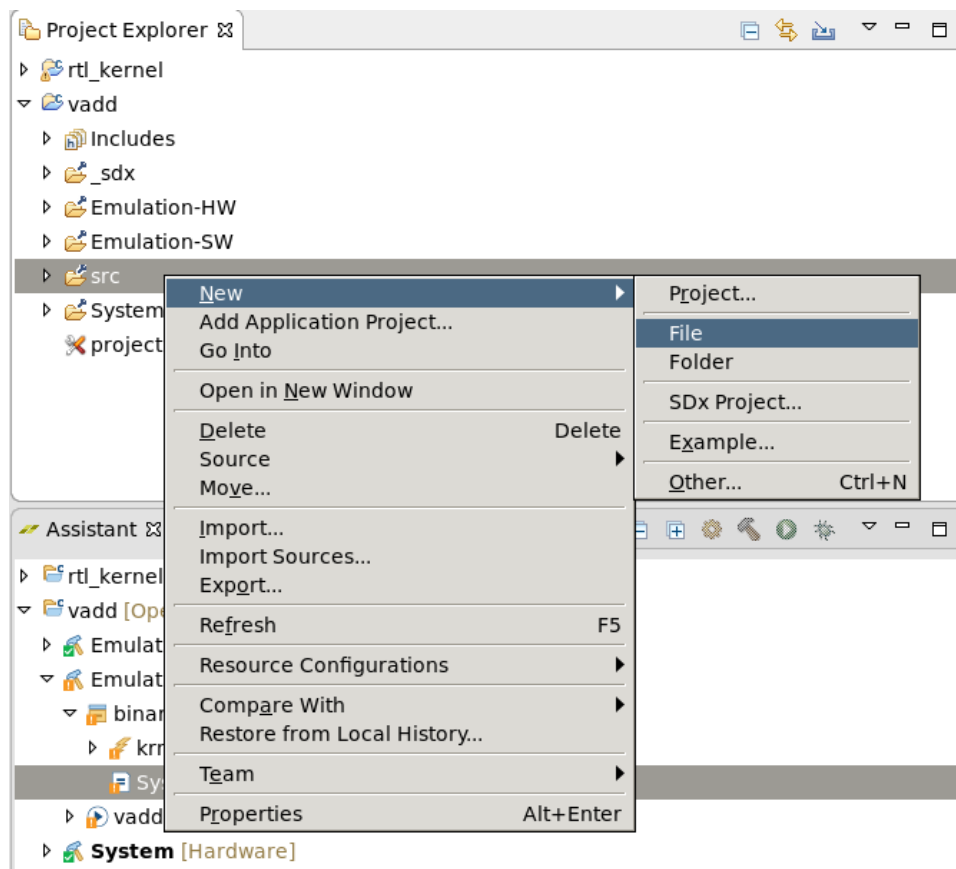
3. 同様に、コンパイル済みカーネル (.xo ファイル) を [Import Sources] をクリックしてプロジェクトにインポートできます。次の図では、`krnl_vadd.xo` ファイルがプロジェクトにインポートされます。

図 26: .xo ソースのインポート



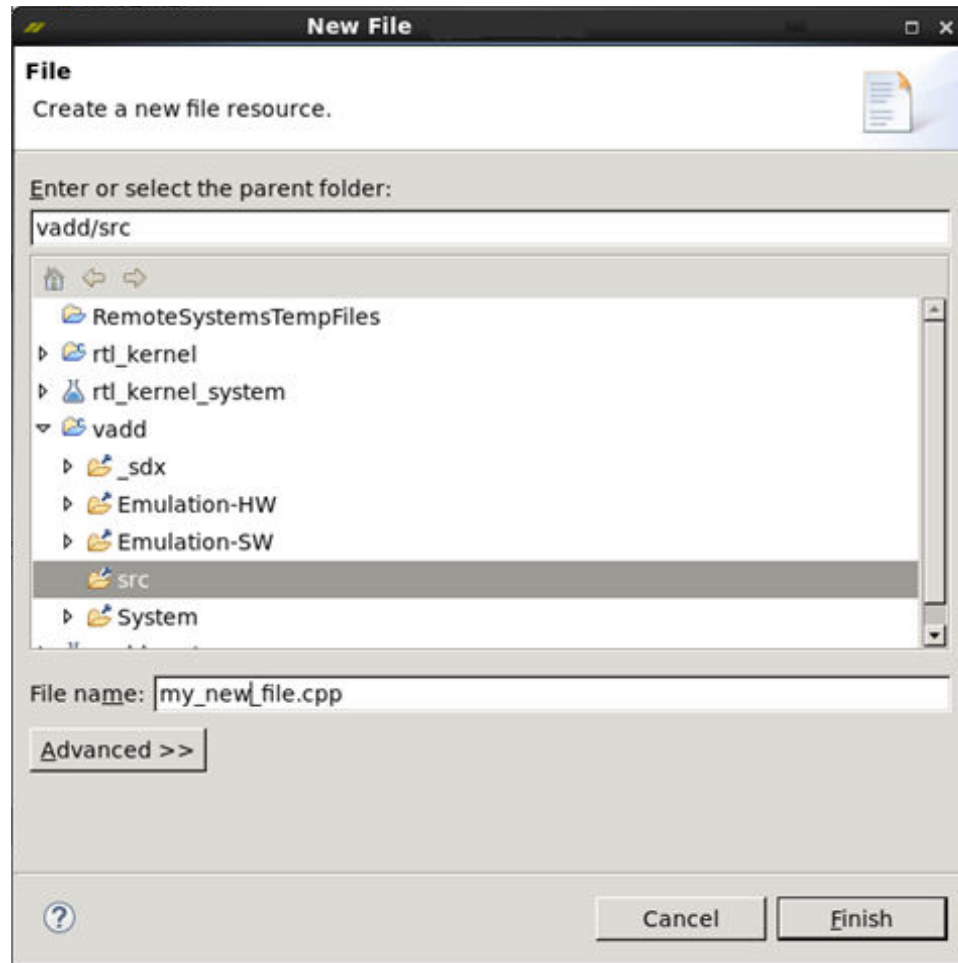
4. ソースのインポートに加え、新しいソース ファイルを GUI で作成および編集できます。SDx IDE でプロジェクトを開き、src フォルダを右クリックして [New] → [File] をクリックします。

図 27: 新規ファイルの作成



5. 次の図に示すように、ファイルを作成するフォルダーを選択してファイル名を入力します。[Finish] をクリックすると、ファイルがプロジェクトに追加されます。

図 28: 新規ファイル名の指定



6. ソースファイルをプロジェクトに追加したら、コンフィギュレーション、コンパイル、アプリケーションを実行できます。ソースファイルを開くには、[Project Explorer] ビューで `src` フォルダを展開し、ファイルをダブルクリックします。

SDAccel のプログラム

ザイリンクス FPGA で実行するカーネル用に SDAccel™ 環境で生成されるカスタム プロセッシング アーキテクチャでは、パフォーマンスを大幅に向上できる可能性があります。この可能性を最大限に活用するため、ホストおよびカーネルコードを FPGA でアクセラレーションするのに適したものにする必要があります。

ホスト アプリケーションはホスト x86 サーバーで実行され、SDAccel ランタイムを使用して FPGA カーネルとの通信を制御します。ホスト アプリケーションは、OpenCL™ API を使用して C/C++ で記述されます。カスタム カーネルは、SDAccel プラットフォーム上のザイリンクス FPGA 内で実行されます。

SDAccel ハードウェア プラットフォームには、ホストとカーネルの間でデータを転送するのに使用するグローバル メモリ バンクが含まれます。また、サポートされるプラットフォームでは、ホストとカーネルの間の転送に直接ストリーミングを使用できます。ホスト x86 マシンと SDAccel アクセラレータ間の通信には PCIe[®] バスが使用されます。

このセクションでは、ザイリンクス ランタイム (XRT) をセットアップするためのホスト アプリケーション コードの記述方法、カーネル バイナリを SDAccel プラットフォームに読み込む方法、ホスト アプリケーションとカーネルの間でのデータの転送方法、FPGA 上のカーネルをホスト アプリケーションで適切なときにトリガーする方法を説明します。

FPGA ファブリックでは、複数のカーネルを同時に実行することも可能です。そのため、1 つのカーネルの複数のインスタンスを作成したり、1 つのデバイス上に複数のカーネルをコンフィギュレーションしたりして、ホスト アプリケーションのパフォーマンスを向上できます。FPGA 上で実行されるカーネルには、プラットフォームまたはほかのカーネルに接続するため、1 つまたは複数のインターフェイスを含めることができます。FPGA で実行するカーネルの数、カーネルによりアクセスされるメモリ バンクの接続、およびホストとカーネル間のストリーミング接続は、ビルド プロセス中に `xocc` リンク オプションで指定されます。

詳細は[ハードウェアのビルド](#)、またはより詳しい説明は『SDAccel 環境プログラマ ガイド』([UG1277](#))を参照してください。ホスト アプリケーション、カーネル コード、およびそれらの間のデータ転送に関する詳細は、このガイドを参照してください。

ホスト アプリケーションのコード記述

ホスト アプリケーションを作成する際は、SDAccel ランタイムの設定、カーネルのプログラムおよび起動、ホスト アプリケーションとカーネル間でのデータ転送、およびアプリケーションの主な関数のアドレス指定に必要なオーバーヘッドを管理する必要があります。

ランタイムの設定

各ホスト アプリケーション内で、OpenCL プラットフォームおよびデバイス ID を特定する環境を設定し、コンテキストを指定し、コマンドをキューに入れ、プログラムをビルドし、1 つまたは複数のカーネルを生成する必要があります。プログラムによりカーネルを特定して設定し、ホスト コードとカーネルの間でデータを転送します。ホスト コードでは、このプロセスに次の手順を使用できます。



ヒント: 次のコード例は、[IDCT サンプル デザイン](#)からのものです。

1. `clGetPlatformIDs` および `clGetPlatformInfo` コマンドを使用してザイリンクス プラットフォームを特定し、OpenCL ランタイム環境を設定します。次に例を示します。

```
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];
    }
}
```

2. `clGetDeviceIDs` コマンドを使用して、キューにあるカーネルで使用可能なプラットフォーム上のザイリンクス デバイスを特定します。デバイス ID を取得するには、前の手順で得られたプラットフォーム ID が必要です。次に例を示します。

```
clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_ACCELERATOR, 1, &device_id,
    NULL);
```

3. `clCreateContext` を使用してコンテキストを設定します。コンテキストはワーク アイテムを実行する環境で、コマンド キューからのトランザクションを割り当てるデバイスを特定します。次に、コンテキストを作成する例を示します。

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

4. `clCreateCommandQueue` を使用してコマンド キューを定義します。コマンド キューは、実行を待機するコマンドのリストです。コマンド キューは、コマンドを発行された順に処理するか、コマンドを実行する準備ができたらすぐに実行できるようにアウト オブ オーダーで処理するかを設定できます。FPGA 上でのカーネルの同時実行には、1 つの順不同コマンド キューまたは複数の順次コマンド キューを使用します。次に例を示します。

```
// Create out-of-order Command Queue
cl_command_queue commands = clCreateCommandQueue(context, device_id,
    CL_QUEUE_PROFILING_ENABLE | CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE
    , &err);
```

5. ホスト アプリケーションによりコマンド キューに渡されるカーネルを含み、コンフィギュレーションするプログラムをホスト コードで設定します。`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。`clCreateProgramWithBinary` コマンドを使用すると、FPGA バイナリ (.xclbin) がデバイスにダウンロードされ、`cl_program` ハンドルが戻されます。次に、これらの API 呼び出しを使用して、プログラムを作成する例を示します。

```
char *fpga_bin;
size_t fpga_bin_size;
fpga_bin_size = load_file_to_memory(binaryName, &fpga_bin);

cl_program program = clCreateProgramWithBinary(context, 1,
    (const cl_device_id*) &device_id, &fpga_bin_size,
    (const unsigned char**) &fpga_bin, NULL, &err);
```

FPGA デバイスとのデータ転送

プログラムが確立されたら、カーネルをトリガーする前に、カーネルに必要なデータを SDAccel プラットフォームに転送できます。カーネルとデータを送受信するには、[clCreateBuffer](#)、[clEnqueueReadBuffer](#)、および [clEnqueueWriteBuffer](#) コマンドを使用するのが最も簡単ですが、トランザクションの前に必要なデータを転送するには、[clEnqueueMigrateMemObjects](#) コマンドを使用します。このコマンドを使用すると、アプリケーションでのレイテンシが削減されます。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 | 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);
```



ヒント: デフォルトでは、すべてのカーネルからのすべてのメモリインターフェイスが1つのグローバルメモリバンクに接続されます。デフォルトの接続を変更すると、メモリバンクの接続をカスタマイズできます。これにより、複数のカーネルで異なるグローバルメモリバンクからの読み出しおよび書き込みデータを同時に処理できるので、カーネルのパフォーマンスが向上します。詳細は、[メモリリソースへのカーネルインターフェイスのマップ](#)を参照してください。

カーネルの設定

プログラムが確立されたら、カーネルを設定し、カーネルを実行して、ホストアプリケーションとカーネルの間でのイベント同期を管理できます。

1. [clCreateKernel](#) コマンドを使用して、プログラムおよび読み込まれた FPGA バイナリからカーネルを作成します。

```
// Create Kernel
cl_kernel krnl = clCreateKernel(program, "krnl_idct", &err);
```

2. [clSetKernelArg](#) を使用してカーネル引数を設定します。このコマンドを使用すると、カーネルの引数を設定できます。

```
// Set the kernel arguments
clSetKernelArg(mKernel, 0, sizeof(cl_mem), &mInBuffer[0]);
clSetKernelArg(mKernel, 1, sizeof(cl_mem), &mInBuffer[1]);
clSetKernelArg(mKernel, 2, sizeof(cl_mem), &mOutBuffer[0]);
clSetKernelArg(mKernel, 3, sizeof(int), &m_dev_ignore_dc);
clSetKernelArg(mKernel, 4, sizeof(unsigned int), &mNumBlocks64);
```

3. カーネルを FPGA で実行するようスケジュールするには、[clEnqueueTask](#) を使用します。カーネルを実行するリクエストはコマンドキューに配置され、キューの設定によって、順番を待つか、準備ができたら実行されます。

```
clEnqueueTask(mQ, mKernel, 1, &inEvVec[mCount], &runEvVec[mCount]);
```

4. `clEnqueueTask` (および `clEnqueueMigrateMemObjects`) コマンドは非同期で、コマンドがコマンド キューに追加されるとすぐに返るので、ホスト アプリケーション内のイベントのスケジューリングを管理する必要があります。場合があります。ホスト アプリケーションのコマンド間のデータ依存性を解決するため、`clWaitForEvents` または `clFinish` コマンドを使用して、ホスト プログラムの実行を一時停止または阻止できます。次に例を示します。

```
// Execution waits until all commands in the command queue are finished
clFinish(command_queue);

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

カーネルの設定の詳細は、『SDAccel 環境プログラマ ガイド』 ([UG1277](#)) を参照してください。

カーネル言語サポート

SDAccel 環境では、OpenCL C、C/C++、および RTL (SystemVerilog、Verilog、または VHDL) で記述されたカーネルがサポートされます。同じアプリケーションで異なるカーネル タイプを使用できます。ただし、各カーネル特定の要件およびコーディング スタイルに従う必要があります。

OpenCL C および C/C++ から作成されたカーネルは、ソフトウェアおよびアルゴリズムの開発に適しています。既存の C/C++ アプリケーションから開始して、その一部を簡単にアクセラレーションできます。

すべてのカーネルには、次が必要です。

- スカラー引数を渡し、カーネルを開始/停止するため、制御レジスタにアクセスするのに使用する 1 つのスレーブ AXI4-Lite インターフェイス。
- 次のインターフェイスの少なくとも 1 つ (両方のインターフェイスを使用可能):
 - グローバル メモリと通信するための AXI4 マスター インターフェイス。
 - カーネル間で、またはホストと直接データを転送するための AXI4-Stream インターフェイス。

OpenCL C カーネルの記述

SDAccel 環境では、OpenCL C 言語コンストラクトと OpenCL 1.0 エンベデッド プロファイルのビルトイン関数がサポートされます。次に、SDAccel 環境でコンパイル可能な行列乗算用の OpenCL C カーネルの例を示します。

```
__kernel __attribute__ ((reqd_work_group_size(16,16,1)))
void mult(__global int* a, __global int* b, __global int* output)
{
    int r = get_local_id(0);
    int c = get_local_id(1);
    int rank = get_local_size(0);
    int running = 0;
    for(int index = 0; index < 16; index++){
        int aIndex = r*rank + index;
        int bIndex = index*rank + c;
        running += a[aIndex] * b[bIndex];
    }
    output[r*rank + c] = running;
    return;
}
```



重要: `math.h` などの標準 C ライブラリは、OpenCL C カーネルでは使用できません。OpenCL ビルトイン C 関数を使用してください。

OpenCL C カーネルの場合、AXI4-Lite インターフェイスが自動的に生成され、AXI4-Lite メモリ マップ インターフェイスが関数定義の `__global` 指示子に基づいて生成されます。

C/C++ カーネルの記述

SDAccel 環境では、C/C++ で記述したカーネルがサポートされます。上記の行列乗算カーネルは、C/C++ コードで次のように記述できます。このように記述されたカーネルであれば、SDAccel 環境で Vivado® HLS で利用可能な最適化手法がすべてサポートされます。唯一注意が必要な点は、カーネルの記述が特定の関数シグネチャに準拠している必要があることです。

デフォルトでは、HLS 用に C/C++ で記述されたカーネルには、関数パラメーター データの転送に使用される物理インターフェイスに関する想定は何もないということに注意してください。HLS では、コードに組み込まれたプラグマを使用して、関数ポート用に生成するのに使用する物理インターフェイスをコンパイラに指示します。関数が有効な HLS C/C++ カーネルとして処理されるようにするには、各関数引数に Vivado HLS インターフェイス プラグマが必要です。

```
void mmult(int *a, int *b, int *output)
{
#pragma HLS INTERFACE m_axi port=a offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=b offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=output offset=slave bundle=gmem
#pragma HLS INTERFACE s_axilite port=a bundle=control
#pragma HLS INTERFACE s_axilite port=b bundle=control
#pragma HLS INTERFACE s_axilite port=output bundle=control
#pragma HLS INTERFACE s_axilite port=return bundle=control

    const int rank = 16;
    int running = 0;
    int bufa[256];
    int bufb[256];
    int bufc[256];
    memcpy(bufa, (int *) a, 256*4);
    memcpy(bufb, (int *) b, 256*4);

    for (unsigned int c=0;c<rank;c++){
        for (unsigned int r=0;r<rank;r++){
            running=0;
            for (int index=0; index<rank; index++) {
#pragma HLS pipeline
                int aIndex = r*rank + index;
                int bIndex = index*rank + c;
                running += bufa[aIndex] * bufb[bIndex];
            }
            bufc[r*rank + c] = running;
        }
    }

    memcpy((int *) output, bufc, 256*4);
    return;
}void mmult(int *a, int *b, int *output)
```

カーネルを C++ で定義する際は、カーネルをターゲットとする関数を `extern "C" {...}` で囲みます。extern "C" を使用すると、コンパイラ/リンカーで C の命名規則および呼び出し規則が使用されます。

C/C++ カーネルの場合、AXI4-Lite および AXI4 メモリ マップ インターフェイスにマップするのにインターフェイス プラグマを使用します。RTL カーネルの場合、これらのインターフェイスの追加はユーザーの責任となります。

ポインター引数

すべてのポインターはグローバル メモリにマップされます。データは、異なるバンクにマップ可能な AXI インターフェイスを介してアクセスされます。メモリ インターフェイス仕様には、次の2つのプラグマが必要です。

1. 1つ目のプラグマでは、AXI メモリ マップ インターフェイスがどの引数にアクセスするかを定義します。オフセットは常に必要です。offset=slave は、<variable_name> 配列のオフセットがカーネルの AXI スレーブ インターフェイスを介して使用できるようになることを意味します。

```
#pragma HLS INTERFACE m_axi port=<variable name> offset=slave
bundle=<AXI_MM_name>
```

2. 2つ目のプラグマは、AXI スレーブ インターフェイス用です。スカラー (およびポインター オフセット) が1つの AXI スレーブ制御インターフェイス (control という名前にする必要あり) にマップされます。

```
#pragma HLS INTERFACE s_axilite port=<variable name> bundle=control
```

注記: 4.x 以前のバージョンのプラットフォームでは、元の名前で大文字/小文字がどのように使用されていたかにかかわらず、arg_name を大文字にして接頭辞 M_AXI_ を付けたインターフェイス名 M_AXI_ARG_NAME が使用されていました。

現在のプラットフォーム (v5 以降) では、小文字のインターフェイス名 m_axi_arg_name が使用されます。arg_name は小文字にする必要があり、接頭辞 m_axi_ が付きます。

スカラー

スカラーは定数入力と考慮され、s_axilite にマップする必要があります。制御インターフェイスは次のコマンドで指定します。

```
#pragma HLS INTERFACE s_axilite port=<variable name> bundle=control
```

これらのプラグマの使用方法については、『SDx プラグマ リファレンス ガイド』 ([UG1253](#)) を参照してください。

カーネルのグローバル メモリ ポインターには、C++ 任意精度データ型を使用できます。これらは、値渡しされるスカラー カーネル入力にはサポートされません。

ストリーミング

ストリーミングでは、グローバル メモリを使用せずに、データをカーネルに直接ストリーミングできます。グローバル メモリは使用されないため、パフォーマンスを向上して消費電力を削減できますが、追加の FPGA メモリ (ブロック RAM) が必要です。

ストリーミングは、次の2つのタイプに分けられます。

1. ホストからカード (H2C) とカードからホスト (C2H) ストリーミング
2. カーネル間 (K2K) ストリーミング

H2C および C2H では、カード上のホストとカーネルの間でデータがストリーミングされます。H2C および C2H ストリーミングは、xilinx_u200_qdma_201910_1 などの一部の QDMA でのみ使用可能です。H2C および C2H ストリーミングでは、ユーザーが明示的に接続を作成する必要はなく、システム リンカーにより自動的に接続されます。

ホストとカード間でデータをストリーミングするのに加え、カード間 (K2K) ストリーミングもサポートされます。カーネル間の直接ストリーミングが可能です。H2C および C2H ストリーミングとは異なり、K2K ストリーミングはすべてのプラットフォームでサポートされます。ただし K2K ストリーミングでは、ソースおよびデスティネーションカーネルストリーミングインターフェイス間の接続を指定する必要があります。これは、xocc リンク中に実行されます。

各ストリーミングインターフェイスに対して、次のプラグマを指定する必要があります。

```
#pragma HLS interface axis port=<port_name>
```

『SDAccel 環境プログラマ ガイド』 (UG1277) の「SDAccel ストリーミング プラットフォーム」付録に、ホスト/カーネルおよびカーネル/カーネル間のデータ転送におけるホストとカーネルのコーディング ガイドラインが記載されています。また、C/C++ カーネルの詳細は、『SDAccel 環境プログラマ ガイド』 (UG1277) の「C/C++ カーネルのプログラミング」の章を参照してください。

RTL カーネルの記述

RTL カーネルを SDAccel 環境フレームワークで使えるようにするには、ソフトウェアとハードウェア両方の要件に従う必要があります。ソフトウェアの側面からは、RTL カーネルは [カーネルのソフトウェア要件](#) で説明するレジスタ定義に従っている必要があります。

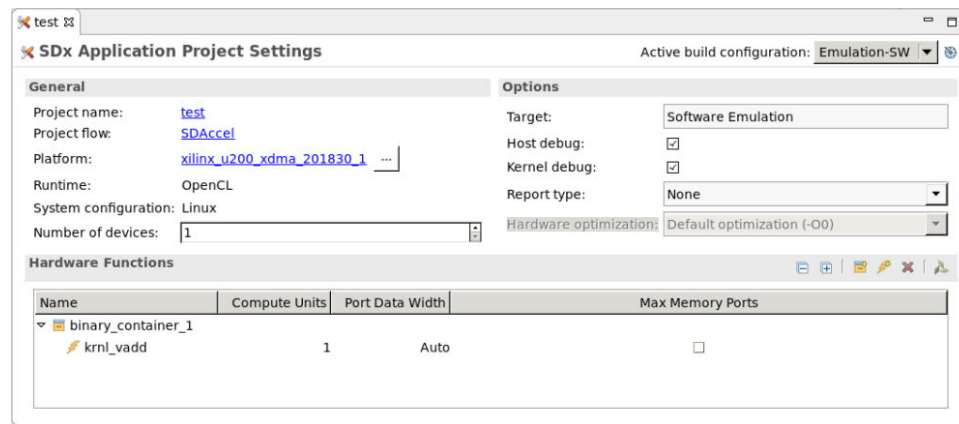
ハードウェアの側面からは、[カーネルのインターフェイス要件](#) で説明するように、インターフェイスが必要です。


RTL カーネルの作成および使用の詳細は、[第9章: RTL カーネル](#)を参照してください。

システムのビルド

システムのビルドには、システムのハードウェア (カーネル) およびソフトウェア (ホスト コード) の両方をビルドする必要があります。次の図に示すプロジェクトエディター ビューには、ビルド コンフィギュレーションの最上位ビューが表示されます。アクティブビルド コンフィギュレーションについて、プロジェクト名、現在のプラットフォーム、選択されているシステム コンフィギュレーション (OS およびランタイム) などの一般情報が示されます。選択されているビルド ターゲット、ホストおよびカーネルのデバッグをイネーブルにするオプションなど、複数のビルド オプションも含まれています。ビルド ターゲットの詳細は [ビルド ターゲット](#)、デバッグ オプションの詳細は [第 7 章: アプリケーションおよびカーネルのデバッグ](#) を参照してください。

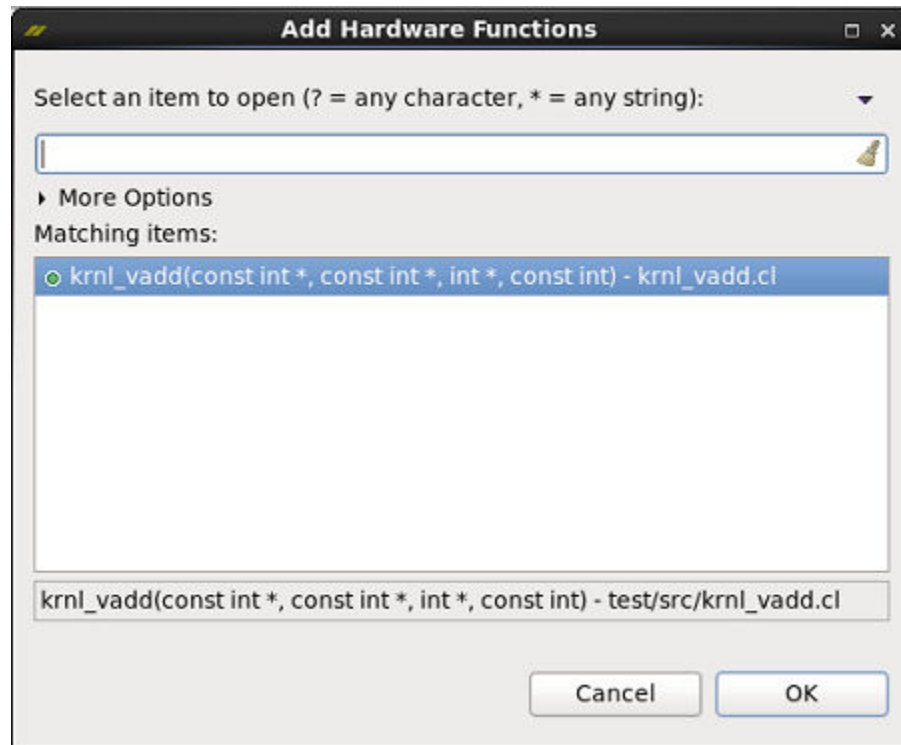
図 29: プロジェクトのエディター ビュー



エディター ビューの下部には、プロジェクトで使用されている現在のカーネルがリストされます。カーネルはバイナリ コンテナの下にリストされます。上記の例では、カーネル `krnl_vadd` は `binary_container_1` に追加されています。バイナリ コンテナを追加するには、 をクリックします。バイナリ コンテナの名前を変更するには、デフォルト名をクリックして新しい名前を入力します。

カーネルをバイナリ コンテナを追加するには、 をクリックします。プロジェクトで定義されているカーネルのリストが表示されます。次の図に示すように、[Add Hardware Functions] ダイアログ ボックスでカーネルを選択します。

図 30: バイナリ コンテナへのハードウェア関数の追加



複数のカーネル インスタンスの作成に説明されているように、カーネルの [Compute Units] 列にカーネルにインスタンス化して使用するインスタンスの数を入力します。

アクティブビルド コンフィギュレーションのさまざまなオプションを指定したら、[Build] コマンド (🔧) をクリックしてビルド プロセスを開始できます。

SDAccel™ 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) が生成されます。SDAccel 環境では、2つの個別のビルド フローが管理されます。

- ホストコード (ソフトウェア) ビルド
- カーネルコード (ハードウェア) ビルド

SDAccel では、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。次のセクションで、ホストコードおよびカーネルコードをビルドして選択したビルドターゲットを生成する手順を説明します。

ホスト アプリケーションのビルド

OpenCL™ API 呼び出しを使用して C/C++ で記述されたホスト アプリケーションは、GNU コンパイラ コレクション (GCC) に基づくザイリンクス C++ コンパイラ (xcpp) を使用してビルドします。各ソース ファイルはオブジェクト ファイル (.o) にコンパイルされ、ザイリンクス SDAccel ランタイム共有ライブラリとリンクされて、実行ファイル (.exe) が作成されます。



ヒント: `xcpp` は GCC に基づいており、ここには記載されていませんが、多数の標準 GCC オプションをサポートします。詳細は、[GCC オプションインデックス](#)を参照してください。

ホスト アプリケーションのコンパイル

ホスト アプリケーションのソース ファイルは、`-c` オプションを使用してコンパイルし、オブジェクト ファイル(`.o`) を生成します。

```
xcpp ... -c <file_name1> ... <file_nameN>
```

`-o` オプションを使用して、出力オブジェクト ファイルの名前を指定することもできます。

```
xcpp ... -o <outut_file_name>
```

`-g` オプションを使用すると、デバッグ情報を生成できます。

```
xcpp ... -g
```

ホスト アプリケーションのリンク

生成されたオブジェクト ファイル(`.o`) がザイリンクス SDAccel ランタイム共有ライブラリとリンクされ、実行ファイル(`.exe`) が作成されます。リンクを実行するには、`-l` オプションを使用します。

```
xcpp ... -l <object_file1.o> ... <object_fileN.o>
```

注記: ホストのコンパイルとリンクは、1つの手順に統合できます。`-c` および `-l` オプションは必須ではなく、ソース入力ファイルのみが必要です。

GUI フローでは、ビルド (🔧) コマンドをクリックすると、ホスト コードとカーネル コードがコンパイルおよびリンクされます。

ハードウェアのビルド

カーネル コードは C、C++、OpenCL C、RTL のいずれかで記述され、GCC を基にしたコマンドラインユーティリティである `xocc` コンパイラによりビルドされます。`xocc` の最終出力は、カーネル `.xo` ファイルとハードウェアプラットフォーム(`.dsa`)をリンクする FPGA バイナリ(`.xclbin`)です。`.xclbin` は、カーネルのコンパイルとリンクを必要とする2段階のビルドプロセスで生成されます。

`xocc` はスタンドアロン (理想的にはスクリプトまたは `make` のようなビルドシステム) で使用可能であり、SDx™ IDE で完全にサポートされています。

ターゲットのビルド

コンパイルは選択したビルド ターゲットによって異なります。詳細は、[ビルド ターゲット](#)を参照してください。ビルド ターゲットを指定するには、次に示すように `xocc -target` オプションを使用します。

```
xocc --target sw_emul|hw_emul|hw ...
```

- ソフトウェアエミュレーション (sw_emu) では、カーネルソースコードがエミュレーション中に使用されます。
- ハードウェアエミュレーション (hw_emu) では、合成済み RTL コードがハードウェアエミュレーションフローに使用されます。
- システムビルド (hw) の場合、xocc により FPGA バイナリが生成され、システムをハードウェア上で実行できます。

カーネルのコンパイル

コンパイルでは、xocc によりカーネル アクセラレータ関数 (C/C++ または OpenCL 言語で記述) がライブラリ オブジェクト (.xo) ファイルにコンパイルされます。各カーネルが個別の .xo ファイルにコンパイルされます。これは xocc の -c/--compile モードです。

RTL で記述されたカーネルは、package_xo コマンド ライン ユーティリティを使用してコンパイルされます。このユーティリティは、xocc -c と同様、この後のリンク段階で使用される .xo ファイルを生成します。詳細は、[第9章: RTL カーネル](#)を参照してください。

カーネルのリンク

前述のように、カーネルのコンパイル プロセスでは、カーネルが OpenCL、C、C++、または RTL のいずれで記述されていても、ライブラリ オブジェクト ファイル (.xo) が生成されます。リンク段階では、異なるカーネルからの .xo ファイルがシェ尔にリンクされ、ホストコードに必要な FPGA バイナリ コンテナ ファイル (.xclbin) が作成されます。

ファイルをリンクする xocc コマンドは、次のとおりです。

```
$ xocc -l <kernel_object_file>.xo -o <binary_platform_file>.xclbin
```

入力ファイル <kernel_object_file> は複数指定でき、<binary_platform_file> は xclbin 出力ファイルの名前です。

複数のカーネル インスタンスの作成

リンク段階では、--nk xocc オプションを使用してカーネルのインスタンス数 (計算ユニットと呼ぶ) を指定できます。これにより、FPGA の異なるデバイス リソースを使用してアプリケーションのランタイムで同じカーネル関数を並列実行し、ホストアプリケーションのパフォーマンスを向上できます。

注記: --nk オプションの詳細は、『SDAccel 環境プログラマ ガイド』 ([UG1277](#)) および 『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。

コマンドラインフローでは、xocc --nk オプションで .xclbin ファイルにインスタンス化してカーネルのインスタンス数を指定します。コマンドの構文は次のとおりです。

```
$ xocc -nk <kernel name>:<no of instances>:<name1>.<name2>...<nameN>
```

たとえば、foo カーネルは計算ユニット fooA、fooB、fooC を使用して3回インスタンス化されています。

```
$ xocc --nk foo:3:fooA.fooB.fooC
```

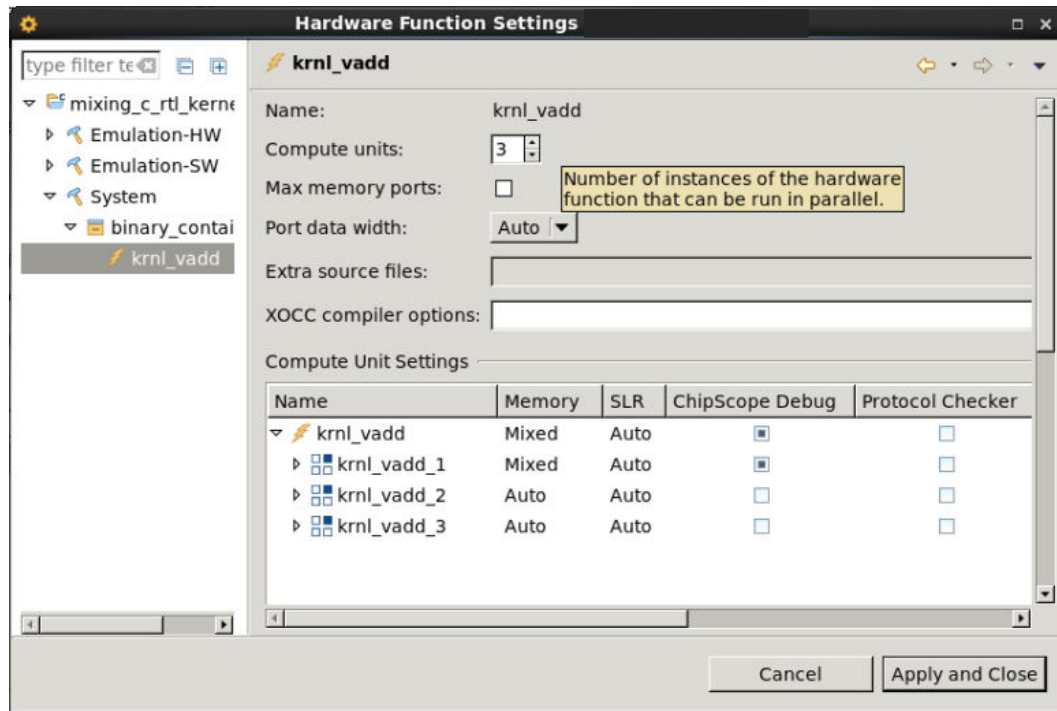


ヒント: カーネル インスタンス名はオプションですが、--sp などのオプションには必要なので、指定するようにしてください。

GUI フローで計算ユニットの数を指定するには、[Assistant] ビューで最上位カーネルを右クリックし、[Settings] をクリックします。

[Project Settings] ダイアログ ボックスで、インスタンス化するカーネルを選択して、計算ユニットの値をアップデートします。次の図では、カーネル `krnl_vadd` が3回 (CU 3つ) インスタンス化されます。

図 31: 複数の計算ユニットのインスタンス化



上の図では、`krnl_vadd` カーネルの3つの計算ユニット (`krnl_vadd_1`、`krnl_vadd_2` および `krnl_vadd_3`) が FPGA バイナリ (`.xclbin`) にリンクされます。

カーネルのさまざまなインスタンスにアクセスするには、ホスト コードの OpenCL API `clCreateSubDevices` を使用して、デバイスをカーネルインスタンスを1つずつ含む複数のサブデバイスに分割できます。詳細は、『SDAccel 環境プログラマ ガイド』 (UG1277) の「サブデバイス」セクションを参照してください。

メモリ リソースへのカーネル インターフェイスのマップ

リンク段階では、カーネルのメモリ ポートが PLRAM および DDR を含むメモリ リソースに接続されます。指定しない場合、これらのリソースへの接続は `xocc` リンク時に自動的に実行されますが、ザイリンクスではパフォーマンスを最適なものにするためこれらの接続を指定することをお勧めします。詳細は、『SDAccel 環境プログラマ ガイド』 (UG1277) および『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。

SDAccel プラットフォームは、さまざまなメモリ リソースにアクセスできます。計算ユニットからインスタンスの異なるメモリ リソースに入力ポートと出力ポートをマップすることにより、入力データおよび出力データに同時にアクセスできるようになるので、全体的なパフォーマンスが向上します。

計算ユニットからメモリ リソースにインターフェイスをマップするには、リンクで `xocc --sp` オプションを使用します。

ホストアプリケーションのコード記述方法の詳細は、『SDAccel 環境プログラマ ガイド』(UG1277)の「FPGA デバイスのバッファ転送」を参照してください。

計算ユニットのメモリ インターフェイスをメモリ リソースに割り当てる指示子は、次のとおりです。

```
--sp <compute_unit>.<mem_interface>:<memory>
```

説明:

- `compute_unit`: 計算ユニット (CU) の名前。
- `mem_interface`: 計算ユニットのメモリ インターフェイスまたは関数引数の名前。
- `memory`: メモリ リソース。

各メモリ インターフェイス接続に、個別の指示子が必要です。



ヒント: カーネル、ポート、および引数名などのカーネル情報を取得するには、`.xo` ファイルがある場合は `kernelinfo` コマンドラインツールを、`.xclbin` ファイルがある場合は `platforminfo` を使用します。ツールの詳細は、『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。

次の例では、`vadd_1` という名前の CU から DDR[3] メモリに `m_axi_gmem` というメモリ インターフェイスを割り当てています。

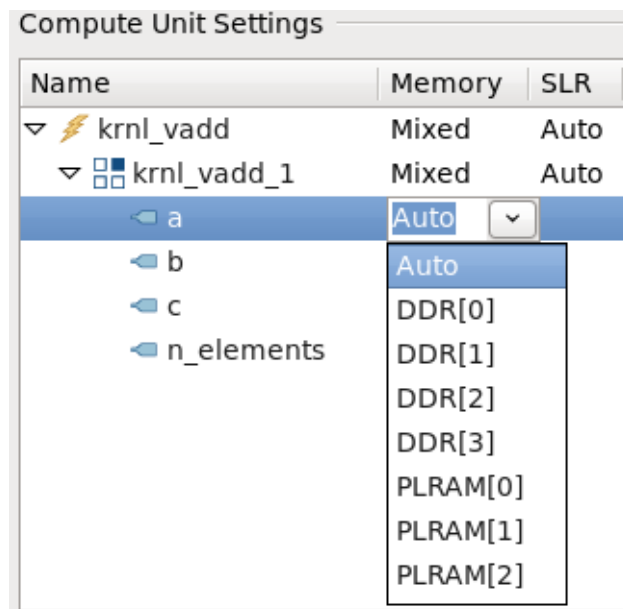
```
xocc ... --sp vadd_1.m_axi_gmem:DDR[3]
```

SDx GUI で、[複数のカーネル インスタンスの作成](#) に説明される SDx GUI を使用した方法を使用して、`--sp` オプションを追加できます。[Assistant] ビューで最上位カーネルを右クリックし、[Settings] をクリックします。[Project Settings] ダイアログ ボックスで、[XOCC Linker Options] フィールドに `--sp` オプションを入力します。

GUI を使用して `xocc` コンパイルに指示子を追加するには、[Assistant] ビューの [System] の下でカーネルを右クリックし、[Settings] をクリックします。

ハードウェア関数設定のダイアログ ボックスが表示され、[Compute Unit Settings] フィールドでメモリ インターフェイスのマップを変更できます。特定の引数の CU のメモリ リソース マッピングを変更するには、その引数の [Memory] 設定をクリックして必要なメモリ リソースに変更します。次の図は、引数 `a` を選択したところを示しています。

図 32: 計算ユニットのメモリ設定



すべてのCU引数に対して同じメモリリソースを選択するには、そのCU(上記の例の場合は `krnl_vadd_1`) をクリックして必要なメモリリソースに変更します。



重要: `--sp` オプションを使用してカーネルインターフェイスをメモリバンクに割り当てる際は、カーネルのすべてのインターフェイスに対して `--sp` オプションを指定する必要があります。詳細は、『SDAccel 環境プログラマガイド』(UG1277) の「DDR バンクとカーネルの接続のカスタマイズ」を参照してください。

カーネル間ストリーミング接続

カーネル間 (K2K) ストリーミングは、カーネル間の直接ストリーミングです。ソースおよびデスティネーションカーネルストリーミングインターフェイス間の接続を指定する必要があります。これは、次に示すように、`xocc` リンクで `-sc` オプションを使用して指定します。

```
xocc -l --sc <kernel_instance_name>.<source streaming port>:<kernel_instance_name>.<destination streaming port>
```

たとえば、次の2つのカーネルのストリーミングポートを接続するとします。

1. `data_out` という出力ストリーミングポートを持つインスタンス `CU_A`。
2. `data_in` という入力ストリーミングポートを持つインスタンス `CU_B`。

次のコマンドを使用します。

```
xocc -l --sc CU_A.data_out:CU_B.data_in
```

計算ユニットの SLR への割り当て

計算ユニット (CU) は `--slr` を使用した `xocc` リンク中に SLR (Super Logic Region) に割り当てられます。コマンドライン指示子の構文は次のとおりです。

```
--slr <compute_unit>:<SLR_NUM>
```

`compute_unit` は CU の名前、`SLR_NUM` は CU が割り当てられる SLR の数です。

たとえば、`xocc ... --slr vadd_1:SLR2` では `vadd_1` という名前の CU が SLR2 に割り当てられます。

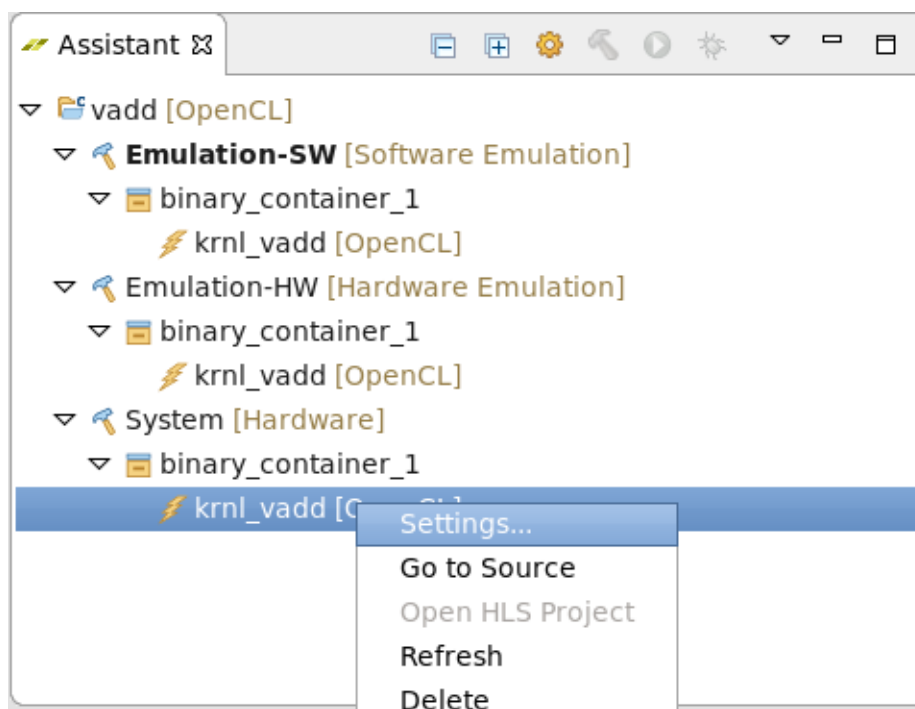
`--slr` 指示子は、CU ごとに別々に適用する必要があります。たとえば、次の例の場合、`--slr` が 3 回使用されて、3 つすべての CU が SLR に割り当てられます。`krnl_vadd_1` および `krnl_vadd_2` は SLR1 に、`krnl_vadd_3` は SLR2 に割り当てられます。

```
--slr krnl_vadd_1:SLR1 --slr krnl_vadd_2:SLR1 --slr krnl_vadd_3:SLR2
```

`--slr` 指示子がない場合、CU はどの SLR にでも配置されます。CU SLR マップの推奨事項は、[カーネル SLR および DDR メモリの割り当て](#) を参照してください。

SDx GUI フローで CU を SLR に割り当てるには、次の図に示すように [System] または [Emulation-HW] の下のカーネルを右クリックし、[Settings] をクリックします。

図 33: `xocc` リンク設定



これでハードウェア関数を設定するダイアログボックスが開きます。[Compute Unit Settings] フィールドで該当する CU の [SLR] 設定をクリックして、メニューから SLR を選択します。[Auto] を選択した場合、どの SLR にでも CU が配置されるようになります。

図 34: 計算ユニットの SLR 設定

Compute Unit Settings			
Name	Memory	SLR	Chips
▼ ⚡ krnl_vadd	Mixed	Auto	
▼ 🧩 krnl_vadd_1	Mixed	SLR0	▼
🔌 a	Auto	Auto	
🔌 b	Auto	SLR0	
🔌 c	Auto	SLR1	
🔌 n_elements	Auto	SLR2	

インプリメンテーション結果の制御

コンパイルまたはリンクの際、`--xp` オプションを使用して SDAccel ハードウェア エミュレーションおよびシステムビルドで生成されるハードウェアを詳細に制御できます。

`--xp` オプションは、Vivado® Design Suite を設定するパラメーターに対応しています。たとえば、`--xp` オプションでハードウェアインプリメンテーションの最適化、配置、およびタイミング結果を設定できます。

また、`--xp` オプションはエミュレーションおよびコンパイル オプションを設定するためにも使用できます。これらのパラメーターの例には、クロック マージンの設定、カーネルのデータフロー領域で使用される FIFO の深さの指定、カーネル AXI インターフェイスにバッファリングする未処理の書き込みおよび読み出しの数の指定が含まれます。すべてのパラメーターおよび有効な値のリストは、『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。



ヒント: これらのパラメーターを最大限に活用するには、『Vivado Design Suite ユーザー ガイド: 高位合成』(UG902) およびツールに精通していることが必要です。詳細は、『Vivado Design Suite ユーザー ガイド: インプリメンテーション』(UG904) を参照してください。

コマンドライン フローでは、パラメーターが `param:<param_name>=<value>` と指定されます。

- `param`: 必須のキーワード。
- `param_name`: 適用するパラメーターの名前。
- `value`: パラメーターに適した値。



重要: `xocc` リンカーでは、パラメーターまたは値が有効であるかは確認されません。注意して有効な値を指定してください。そうしないと、ダウンストリーム ツールが正しく機能しない可能性があります。

次に例を示します。

```
$ xocc --xp param:compiler.enableDSASIntegrityCheck=true
--xp param:prop:kernel.foo.kernel_flags="-std=c++0x"
```

次のように、xocc コマンドで param を使用するたびに --xp オプションを繰り返す必要があります。

```
$ xocc --xp param:compiler.enableDSAIntegrityCheck=true
--xp param:prop:kernel.foo.kernel_flags="-std=c++0x"
```

param 値は xocc.ini ファイル内に指定して、各オプションをそれぞれ別の行 (--xp オプションなし) で指定することもできます。

xocc.ini は、--xp 設定を含む初期化ファイルです。ビルド コンフィギュレーションと同じディレクトリにあるファイルを使用します。

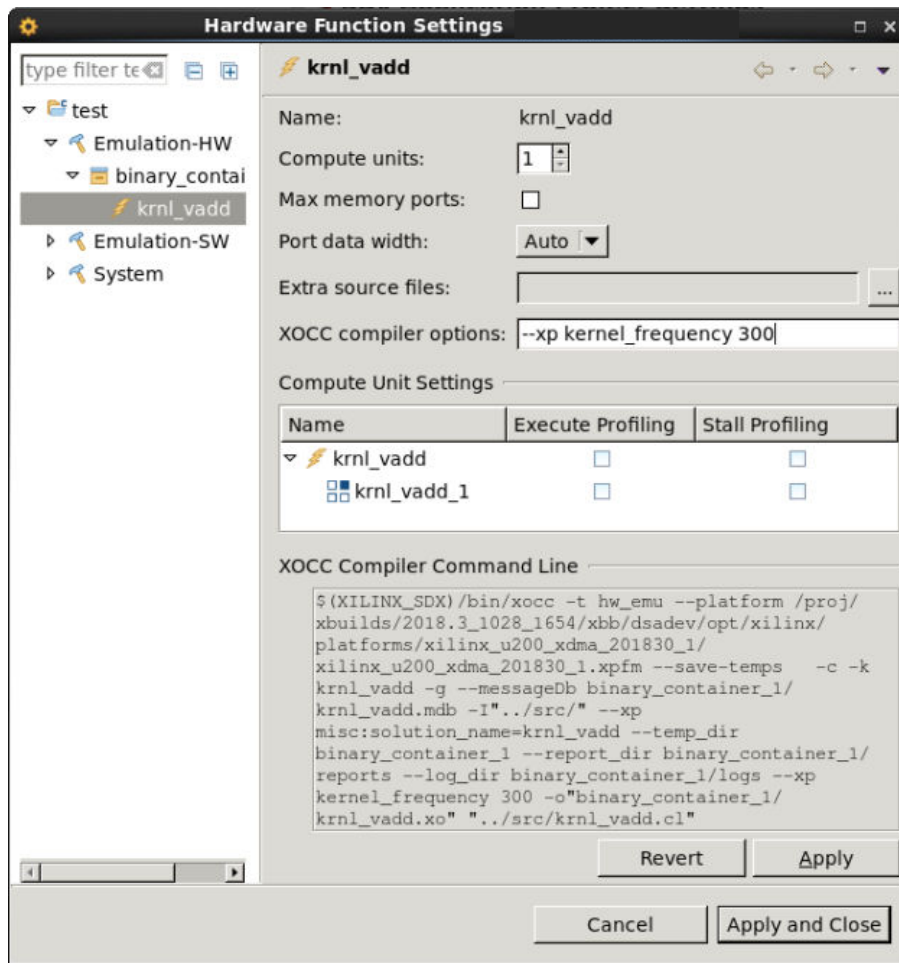
```
param:compiler.enableDSAIntegrityCheck=true
param:prop:kernel.foo.kernel_flags="-std=c++0x"
```

GUI フローでは、xocc.ini がいない場合はアプリケーションで GUI ビルド設定が使用されます。Makefile フローでは、xocc.ini がいない場合は makefile 内の設定が使用されます。

SDx GUI で、[複数のカーネルインスタンスの作成](#) に説明されているのと同僚の方法を使用して、--xp オプションを追加できます。[Assistant] ビューで最上位カーネルを右クリックし、[Settings] をクリックします。[Project Settings] ダイアログボックスで、[XOCC Linker Options] フィールドに --xp オプションを入力します。

[Assistant] ビューでカーネルを右クリックして、xocc コンパイラ オプションおよび --xp パラメーターをカーネルに追加することもできます。次の図は、krnl_vadd カーネルの --xp 設定を示しています。

図 35: XOCC コンパイル設定



レポート生成の制御

xocc-R オプションは、ハードウェア エミュレーションおよびシステム ターゲットのリンク段階で生成されるレポートのレベルを制御します。生成するレポートの数が少ないほうが実行は高速です。

コマンドライン オプションは次のとおりです。

```
$ xocc -R <report_level>
```

<report_level> は次の report_level オプションのいずれかにします。

- -R0: 最小限のレポートを生成し、中間デザインチェックポイント (DCP) は生成しません。
- -R1: R0 のレポートと次のレポートを生成します。
 - 各カーネル用にレビューするデザイン特性を特定します (report_failfast)。
 - 最適化後のフル デザイン用にレビューするデザイン特性を特定します (report_failfast)。
 - 最適化後の DCP を保存します。

- -R2 :R1 のレポートと次のレポートを生成します。
 - インプリメンテーションの各段階の後に DCP を含む Vivado デフォルト レポートを生成します。
 - 配置後に各 SLR 用にレビューするデザイン特性をします (report_failfast)。



ヒント: report_failfast は、デバイス使用率に関して発生する可能性のある問題、クロック制約の問題、および達成不可能なターゲット周波数 (MHz) を検出するユーティリティです。

-R オプションは、[複数のカーネルインスタンスの作成](#)に説明されているように、SDx GUI でも追加できます。

- [Assistant] ビューで最上位カーネルを右クリックし、[Settings] をクリックします。
- [Project Settings] ダイアログ ボックスで、[XOCC Linker Options] フィールドに -R オプションを入力します。

ビルド ターゲット

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。デバッグおよび検証に使用されるソフトウェアおよびハードウェア エミュレーション用の 2 つのエミュレーション ターゲットと、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 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

ソフトウェア エミュレーションの主な目的は、論理的に正しいことを確認し、アプリケーションをカーネルに分割することです。ソフトウェア エミュレーションでは、ホスト 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にコンパイルされます。高速コンパイル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のプログラマ モデルが保持されます。ソフトウェア エミュレーションでは、コンパイルおよび実行時間は CPU と同じです。ソフトウェア エミュレーションの実行に関する詳細は、『SDAccel 環境デバッグ ガイド』([UG1281](#)) を参照してください。

SDAccel 開発環境では、CPU でのソフトウェア エミュレーションは、CPU/GPU プログラミングで典型的な反復開発プロセスと同じです。この開発スタイルでは、プログラマは開発しながら繰り返しアプリケーションをコンパイルして実行します。

RTL カーネルでは、C モデルが関連付けられている場合にソフトウェア エミュレーションがサポートされます。ソフトウェア エミュレーションフローをサポートするため、RTL カーネル ウィザードのパッケージ段階に RTL カーネルに C モデル ファイルを関連付けるオプションがあります。

ハードウェア エミュレーション

ソフトウェア エミュレーションフローは機能の正しさを確認するには良い方法ですが、FPGA 実行ターゲットでの正確性は保証されません。ハードウェア エミュレーションフローでは、カーネルのインスタンス化であるカスタム計算ユニットをハードウェア上で運用する前に、カスタム計算ユニット用に生成されたロジックが正しいかを確認できます。

SDAccel 環境では、アプリケーションの各カーネルに対して 1 つ以上のカスタム計算ユニットが生成されます。各カーネルはハードウェア モデル (RTL) にコンパイルされます。エミュレーション中はカーネルがハードウェア シミュレーションで実行されますが、システムの残りの部分には C シミュレータが使用されます。これにより、SDAccel 環境で FPGA 計算ファブリックで実行されるロジックの機能をテストできます。

また、ハードウェアエミュレーションではパフォーマンスおよびリソースの見積もりが示されるので、デザインに関する有益な情報を得ることができます。

ハードウェアエミュレーションはソフトウェアエミュレーションよりも時間がかかるので、ザイリンクスではデバッグおよび検証用に小型のデータセットを使用することをお勧めしています。



重要: ハードウェアエミュレーションで使用される DDR メモリ モデルおよびメモリ インターフェイス ジェネレーター (MIG) モデルは、高位シミュレーション モデルです。これらのモデルはシミュレーションのパフォーマンスには良いですが、レイテンシには近似値が使用されるので、カーネルのようにサイクル精度ではありません。そのため、プロファイル サマリ レポートに示されるパフォーマンス値は概算値でしかないので、異なるカーネル インプリメンテーション間のパフォーマンスを比較するための一般ガイダンスとしてのみ使用してください。

システム

ビルド ターゲットがシステムの場合、xocc によりデザインに対して合成およびインプリメンテーションが実行され、デバイス用の FPGA バイナリが生成されます。バイナリには、バイナリ コンテナの各計算ユニットのカスタム ロジックが含まれます。そのため、このビルド段階は、SDAccel ビルド フローのほかの段階よりも時間がかかるのが一般的です。ただし、カーネルは実際のハードウェアで実行されるので、実行は非常に高速です。

カスタム計算ユニットの生成には、コンパイル フローの計算ユニット ジェネレーターである Vivado 高位合成 (HLS) ツールが使用されます。すべてのコーディング スタイルで最大パフォーマンスを達成するために計算ユニットを自動的に最適化するのは不可能なので、コンパイラに追加のユーザー入力が必要になります。カスタム計算ユニットへのカーネル演算のインプリメンテーションを最適化するために SDAccel 環境で追加可能なユーザー入力については、『SDAccel 環境 プロファイリングおよび最適化ガイド』(UG1207) を参照してください。

計算ユニットがすべて生成されると、これらのユニットは、ソリューションのターゲット デバイスで提供されるインフラストラクチャ エlement に接続されます。デバイスのインフラストラクチャ Element は、OpenCL アプリケーションをサポートするために定義されているメモリ、制御、I/O のデータ プレーンのすべてです。SDAccel 環境によりカスタム計算ユニットとベース デバイス インフラストラクチャがまとめられ、アプリケーション実行中にザイリンクス デバイスをプログラムするために使用する FPGA バイナリが生成されます。



重要: SDAccel 環境では常に有効な FPGA ハードウェア デザインが生成され、カーネルからグローバル メモリへのデフォルト接続が実行されます。ザイリンクスでは、最適な接続を明示的に定義することをお勧めします。詳細は、[カーネル SLR および DDR メモリの割り当て](#) を参照してください。

ターゲットの指定

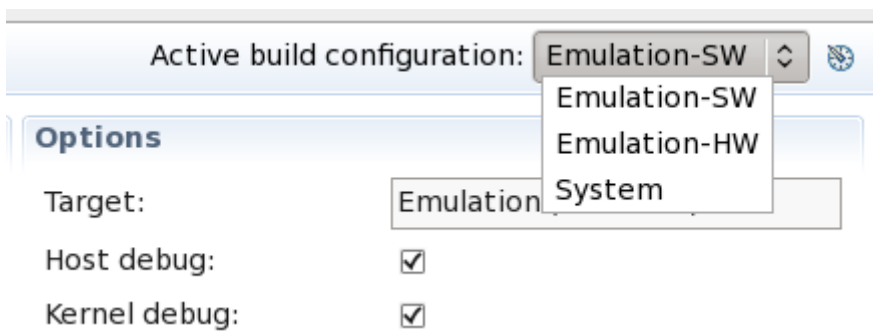
コマンドラインからターゲット ビルドを指定するには、次のコマンドを使用します。

```
xocc --target sw_emu|hw_emu|hw ...
```

GUI では、[Project Editor] ウィンドウの [Active build configuration] ドロップダウン リストからターゲット ビルドを選択できます。この場合、次の図に示すように、3 つの選択肢があります。

- [Emulation-SW]
- [Emulation-HW]
- [System]

図 36: アクティブビルドコンフィギュレーション



ヒント: コンパイルターゲットは、[Build] (🔧) コマンドから、または [Project] → [Build Configurations] → [Set Active] をクリックして割り当てることもできます。

アクティブビルドコンフィギュレーションを設定したら、[Project] → [Build Project] をクリックしてシステムをビルドします。

推奨されるビルドフローの詳細は、[デバッグフロー](#)を参照してください。

プロファイリングおよび最適化

SDAccel™ 環境では、コンパイル中にシステムおよびカーネルのリソースとパフォーマンスに関するさまざまなレポートが生成されます。エミュレーションおよびシステム モードのコンフィギュレーションの両方で、アプリケーションの実行中にはプロファイリング データも収集されます。次のようなレポートが生成されます。

- ホストおよびデバイスのタイムライン イベント
- OpenCL™ API の呼び出しシーケンス
- カーネルの実行シーケンス
- AXI トランザクションを含む FPGA トレース データ
- カーネルの開始および停止信号

これらのレポートとプロファイリング データは、アプリケーションのパフォーマンスのボトルネックを特定し、デザインを最適化してパフォーマンスを向上するために使用できます。

アプリケーションの最適化には、アプリケーション ホスト コードとハードウェアでアクセラレーションされるカーネルの両方の最適化が必要です。ホスト コードはデータ転送とカーネル実行がスムーズに実行されるように最適化する必要があり、カーネルではパフォーマンスとリソース使用量が適切なものになるよう最適化する必要があります。

SDAccel でアルゴリズムを最適化する際には、システムのリソース使用量とパフォーマンス、カーネル最適化、ホスト最適化、および PCIe® 帯域幅最適化の 4 つのエリアがあります。次の SDAccel レポートおよびグラフィカルツールは、これらのエリアをプロファイリングおよび最適化するのに役立ちます。

- システム見積もり
- 設計ガイダンス
- HLS レポート
- プロファイル サマリ
- アプリケーション タイムライン
- 波形ビューおよびライブ波形ビューアー

SDAccel GUI または xocc/makefile フローでアクティブ ビルドを実行すると、レポートが自動的に生成されます。

3 つのビルド コンフィギュレーションの対して個別のレポート セットが生成され、該当するレポート ディレクトリに保存されます。



重要: 高位合成 (HLS) レポートおよび HLS ガイダンスは、C および OpenCL カーネルのハードウェア エミュレーションおよびシステム ビルド コンフィギュレーションに対してのみ生成され、RTL カーネルには生成されません。

プロファイル サマリおよびアプリケーション タイムライン レポートは、3 つのビルド コンフィギュレーションすべてに対して生成され、デフォルトのアプリケーション サブディレクトリに保存されます。

レポートはウェブ ブラウザーまたは SDAccel GUI のスプレッドシート ビューアーで表示できます。SDx™ 統合設計環境からこれらのレポートを開くには、[Assistant] ビューでレポートをダブルクリックします。

この後のセクションで、さまざまなレポートおよびグラフィカル表示ツールについて説明し、デザインのプロファイリングおよび最適化に使用する方法を示します。各レポートの詳細、最適化手順、およびコーディングガイドラインは、『SDAccel 環境プロファイリングおよび最適化ガイド』(UG1207)を参照してください。

設計ガイダンス

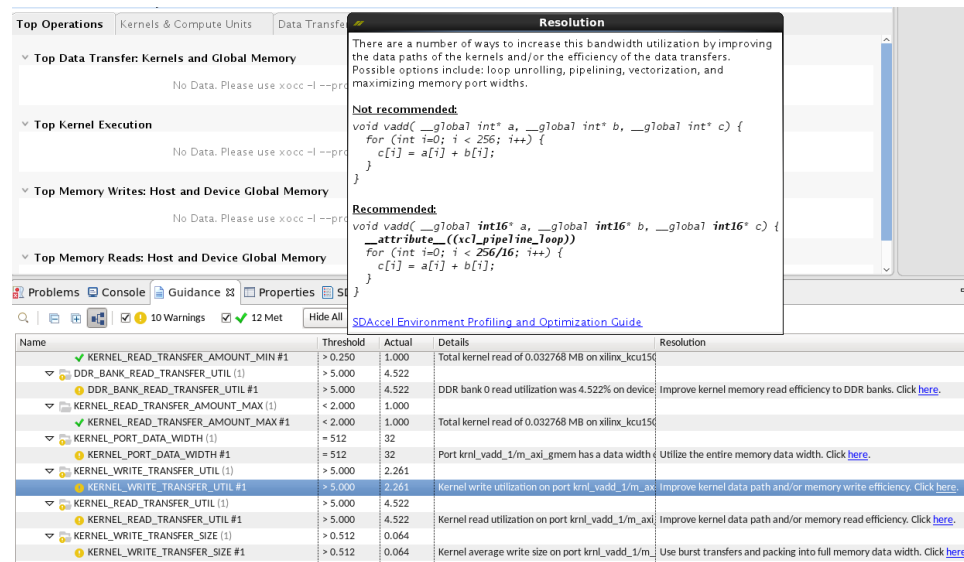
SDAccel 環境には、デザインで検出された問題に対して、ソフトウェアアプリケーション開発者に実行可能なガイダンスをその場で提供する包括的な設計ガイダンス ツールがあります。ガイダンスは、HLS から、および xocc から起動した場合は SDx プロファイラーおよび Vivado® Design Suite から生成されます。生成された設計ガイダンスには、エラー、アドバイザー、警告、およびクリティカル警告の重要度があり、ソフトウェアエミュレーション、ハードウェアエミュレーション、システムビルド中に表示されます。

ガイダンスには、ハイパーリンク、例、および資料へのリンクが含まれています。これにより、問題をすばやく特定できるので生産性が向上し、SDAccel ツールの使用方法を短時間で習得できます。

設計ガイダンスは、SDx GUI でデザインをビルドまたは実行すると自動的に生成され、SDx GUI のコンソールエリアの [Guidance] ビューに表示されます。ガイダンスの上にカーソルを置くと、ソリューションおよび推奨事項がハイライトされます。

次の図に、SDx GUI で表示されるガイダンスの例を示します。カーネルの帯域幅の使用を増加する方法が説明されています。リンクをクリックすると、拡張ビューに実行可能なガイダンスが表示されます。この例では、グローバルメモリ帯域幅の使用を最大限にするためのガイダンスが表示されています。

図 37: 設計ガイダンスの例



Name	Threshold	Actual	Details	Resolution
✓ KERNEL_READ_TRANSFER_AMOUNT_MIN #1	> 0.250	1.000	Total kernel read of 0.032768 MB on xilinx_kcu150	
▼ DDR_BANK_READ_TRANSFER_UTIL (#1)	> 5.000	4.522	DDR bank 0 read utilization was 4.522% on device	Improve kernel memory read efficiency to DDR banks. Click here .
⚠ DDR_BANK_READ_TRANSFER_UTIL (#1)	> 5.000	4.522	DDR bank 0 read utilization was 4.522% on device	
▼ KERNEL_READ_TRANSFER_AMOUNT_MAX (#1)	< 2.000	1.000	Total kernel read of 0.032768 MB on xilinx_kcu150	
✓ KERNEL_READ_TRANSFER_AMOUNT_MAX #1	< 2.000	1.000	Total kernel read of 0.032768 MB on xilinx_kcu150	
▼ KERNEL_PORT_DATA_WIDTH (#1)	= 512	32	Port krnl_vadd_1/m_axi_gmem has a data width of 32	Utilize the entire memory data width. Click here .
⚠ KERNEL_PORT_DATA_WIDTH #1	= 512	32	Port krnl_vadd_1/m_axi_gmem has a data width of 32	
▼ KERNEL_WRITE_TRANSFER_UTIL (#1)	> 5.000	2.261	Kernel write utilization on port krnl_vadd_1/m_axi	Improve kernel data path and/or memory write efficiency. Click here .
⚠ KERNEL_WRITE_TRANSFER_UTIL #1	> 5.000	2.261	Kernel write utilization on port krnl_vadd_1/m_axi	
▼ KERNEL_READ_TRANSFER_UTIL (#1)	> 5.000	4.522	Kernel read utilization on port krnl_vadd_1/m_axi	Improve kernel data path and/or memory read efficiency. Click here .
⚠ KERNEL_READ_TRANSFER_UTIL #1	> 5.000	4.522	Kernel read utilization on port krnl_vadd_1/m_axi	
▼ KERNEL_WRITE_TRANSFER_SIZE (#1)	> 0.512	0.064	Kernel average write size on port krnl_vadd_1/m_axi	Use burst transfers and packing into full memory data width. Click here .
⚠ KERNEL_WRITE_TRANSFER_SIZE #1	> 0.512	0.064	Kernel average write size on port krnl_vadd_1/m_axi	



ヒント: [Assistant] ビューでビルド コンフィギュレーションを右クリックし、[Show Guidance] をクリックします。

xocc の各コマンドライン実行(コンパイルおよびリンクを含む)に対して HTML ガイダンス レポートが生成されます。レポートファイルは、--report_dir ディレクトリの特定の .xo 名の下に生成されます。

レポート ファイルの名前は次のようになります。<output> は次の .xo 名になります。

- xocc コンパイルの場合は `xocc_compile_<output>_guidance.html`
- xocc リンクに対しては `xocc_link_t_guidance.html`

プロファイルの設計ガイダンスは、プロファイリング結果を解釈し、パフォーマンスを向上するために焦点を置く部分を判断するのに有益です。レポートの特定の詳細および設計ガイダンスの追加情報は、『SDAccel 環境プロファイリングおよび最適化ガイド』(UG1207)を参照してください。

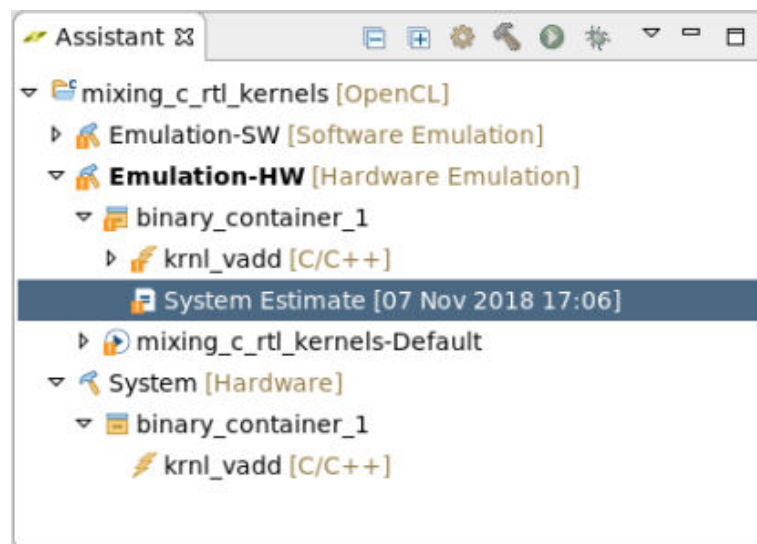
システム見積もりレポート

SDAccel HLS で生成されるシステム見積もりレポートには、FPGA リソース使用量およびハードウェアでアクセラレーションされたカーネルが動作可能な周波数が示されます。このレポートはハードウェアエミュレーションおよびシステムビルドに対して自動的に生成され、次のように [Assistant] ビューの該当するディレクトリの下に含まれます。



ヒント: システムビルドに対するシステム見積もりレポートには見積もりではなく実際のリソース使用量が示されるので、ハードウェアエミュレーションビルドに対するシステム見積もりレポートよりも生成に時間がかかります。ザイリンクスでは、システムビルドを実行する前にハードウェアエミュレーションで試行を繰り返して最適化することをお勧めしています。

図 38: システム見積もりの [Assistant] ビュー



レポートには、リソース使用量および見積もり周波数を含むユーザーカーネルの全体的な情報が含まれます。この結果をデザイン最適化のガイドとして使用できます。たとえば、ターゲット周波数が満たされていない場合、ソースコードを見直す必要があるかもしれません。

次の図に、サンプルレポートを示します。krnl_vadd カーネルの情報がレポートされています。

- 見積もり動作周波数は 411 MHz ですが、これはターゲット周波数 300 MHz を超えています。
- ベストケースのレイテンシは 1 クロック サイクルです。
- FPGA リソース使用量の見積もりは、フリップフロップ 2353 個、LUT 3948 個、ブロック RAM 3 個で、DSP は使用されません。

図 39: システム見積もり

Report name: system_estimate_krnل_vaddBuild configuration: Unknown

Project name: vadd

Created: 13 Apr 2018 21:25

1=====

2Version: xocc v2018.2.0 (64-bit)

3Build: SW Build 2196058 on Thu Apr 12 13:21:30 MDT 2018

4Copyright: Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.

5Created: Fri Apr 13 21:25:48 2018

6=====

7

8-----

9Design Name: krnl_vadd

10Target Device: xilinx:kcu1500:dynamic:5.0

11Target Clock: |

12Total number of kernels: 1

13-----

14

15Kernel Summary

16Kernel Name Type Target OpenCL Library Compute Units

17-----

18krnl_vadd clc fpga0:OCL_REGION_0 krnl_vadd 1

19

20

21-----

22OpenCL Binary: krnl_vadd

23Kernels mapped to: clc_region

24

25Timing Information (MHz)

26Compute Unit Kernel Name Module Name Target Frequency Estimated Frequency

27-----

28krnl_vadd_1 krnl_vadd krnl_vadd 300.300293 411.522614

29

30Latency Information (clock cycles)

31Compute Unit Kernel Name Module Name Start Interval Best Case Avg Case Worst Case

32-----

33krnl_vadd_1 krnl_vadd krnl_vadd 2 ~ 0 1 undef undef

34

35Area Information

36Compute Unit Kernel Name Module Name FF LUT DSP BRAM

37-----

38krnl_vadd_1 krnl_vadd krnl_vadd 2353 3948 0 3

39-----

40

コマンドラインフローを使用する場合、システム見積もりレポートなどのレポートを生成するには、次のオプションを使用します。

```
xocc .. --report_level <arg>
```

arg は生成するレポートのレベルを指定します。

システム見積もりレポートの詳細は、『SDAccel 環境プロファイリングおよび最適化ガイド』(UG1207)を参照してください。--report_level xocc オプションの詳細は、『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。

HLS レポート

HLS レポートは、ハードウェアエミュレーションおよびシステムビルドで生成され、ユーザーカーネルの高位合成(HLS)プロセスに関する詳細を提供します。このプロセスでは、FPGAに機能をインプリメントするため、C/C++およびOpenCLカーネルをハードウェア記述言語に変換します。カスタムハードウェアロジックのFPGAリソース使用量の見積もり、動作周波数、レイテンシ、およびインターフェイス信号が示されます。これらの詳細は、カーネルの最適化に役立つ多数の情報を提供します。

HLS レポートを開くには、[Assistant] ビューでレポートをダブルクリックします。次に、HLS レポートの例を示します。

図 40: HLS レポート

Report name: krnl_vadd.design

Project name: test_prj

Created: 01 May 2018 16:24

Module

krnl_vadd

Current Module : krnl_vadd

Synthesis Report for 'krnl_vadd'

General Information

Date: Tue May 1 16:24:05 2018

Version: 2018.2 (Build 2215266 on Mon Apr 30 21:53:11 MDT 2018)

Project: krnl_vadd

Solution: solution

Product family: virtexuplus

Target device: xcvu9p-fsgd2104-2L-e

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	3.33	2.433	0.90

Latency (clock cycles)

Summary

Latency		Interval		Type
min	max	min	max	
1	54043198934220800	1	54043198934220800	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	1248	-
FIFO	-	-	-	-	-
Instance	2	-	796	1068	-
Memorv	1	-	0	0	-

コマンドラインから実行すると、このレポートは次のディレクトリに保存されます。

`_x/<kernel_name>.<target>.<platform>/<kernel_name>/<kernel_name>/solution/syn/report`

システム見積もりレポートの詳細は、『SDAccel 環境プロファイリングおよび最適化ガイド』(UG1207)を参照してください。

プロファイル サマリ レポート

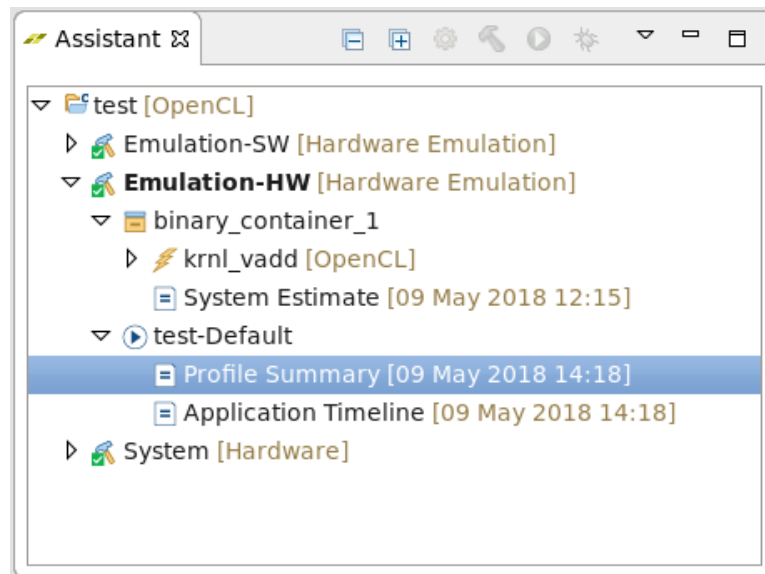
プロファイル サマリには、全体的なアプリケーションパフォーマンスに関する注釈付きの詳細が表示されます。プログラムの実行中に生成されたすべてのデータがSDAccelで収集され、カテゴリ別に表示されます。プロファイル サマリでは、実際のデータ転送とカーネル実行の数値と統計をより詳細に掘り下げることができます。



ヒント: プロファイル サマリ レポートは、すべてのビルド コンフィギュレーションで自動的に生成されます。ただし、ソフトウェア エミュレーションビルドでは、レポートにカーネル実行効率およびデータ転送効率の下にデータ転送の詳細は含まれません。この情報は、ハードウェア エミュレーションおよびシステムビルド コンフィギュレーションでのみ生成されます。

SDx IDE でプロファイル サマリ レポートを開くには、次の図に示すように、[Assistant] ビューで [Profile Summary] をダブルクリックします。

図 41: プロファイル サマリ レポートを開く



次に、プロファイル サマリ レポートの例を示します。

図 42: プロファイル サマリ

Report name: Profile Summary (sdaccel_profile_summary)

Project name: vadd

Created: 13 Apr 2018 21:34

Build configuration: Unknown

Top Operations

Kernels & Compute Units

Data Transfers

OpenCL APIs

▼ Top Data Transfer: Kernels and Global Memory

Device	Compute Unit	Number Of Transfers	Average Bytes per Transfer	Transfer Efficiency (%)	Total Data Transfer (MB)	Total Write (MB)	Total Read (MB)	Transfer Rate (MB/s)	Average Bandwidth Utilization (%)
xilinx_kcu1500_dynamic_5_0-0	All	768	64.000	1.563	0.049	0.016	0.033	792.877	6.883

▼ Top Kernel Execution

Kernel Instance Address	Kernel	Context ID	Command Queue ID	Device	Start Time (ms)	Duration (ms)	Global Work Size	Local Work Size
0x1e21a60	krnl_vadd	0	0	xilinx_kcu1500_dynamic_5_0-0	0.014	0.062	1:1:1	1:1:1

▼ Top Memory Writes: Host and Device Global Memory

Buffer Address	Context ID	Command Queue ID	Start Time (ms)	Duration (ms)	Buffer Size (KB)	Writing Rate (MB/s)
0x1000	0	0	0.0	N/A	32.768	N/A
0x1000	0	0	10194.700	N/A	32.768	N/A

▼ Top Memory Reads: Host and Device Global Memory

Buffer Address	Context ID	Command Queue ID	Start Time (ms)	Duration (ms)	Buffer Size (KB)	Reading Rate (MB/s)
0x9000	0	0	40296.800	N/A	16.384	N/A

レポートには複数のタブがあります。次の表に、各タブの説明を示します。

表 1: プロファイル サマリ

タブ	説明
[Top Operations]	カーネルおよびグローバル メモリ。最上位演算のサマリを示します。FPGA とデバイス メモリ間の最上位データ転送のプロファイル データを表示します。
[Kernels & Compute Units]	すべてのカーネルおよび計算ユニットのプロファイル データを表示します。
[Data Transfers]	ホストおよびグローバル メモリ。ホストとデバイス メモリ間の PCIe リンクを介したすべての読み出しおよび書き込み転送のプロファイル データを表示します。カーネルとグローバル メモリ間のデータ転送がイネーブルの場合、その情報も表示します。
[OpenCL APIs]	ホスト アプリケーションで実行されるすべての OpenCL C ホスト API 関数のプロファイル データを表示します。

コマンドラインを使用している場合は、プロファイル サマリ データはリンク段階で `--profile_kernel` オプションを使用して生成します。 `--profile_kernel` 構文は次のとおりです。

```
--profile_kernel <[data]:<[kernel_name|all]:[compute_unit_name|all]:  
[interface_name|all]:[counters|all]>
```

詳細は、『SDAccel 環境 プロファイリングおよび最適化ガイド』 ([UG1207](#)) を参照してください。

アプリケーション タイムライン

アプリケーション タイムラインはホストとデバイスのイベント情報を収集し、共通のタイムラインに表示します。これは、システムの全体的な状態とパフォーマンスを視覚的に表示して理解するのに役立ちます。これらのイベントには、次のものがあります。

- ・ ホスト コードからの OpenCL API 呼び出し。
- ・ 計算ユニット、AXI トランザクションの開始/停止を含むデバイス トレース データ。
- ・ ホスト イベントおよびカーネルの開始/停止。

このグラフィカル表示により、カーネル同期および効率的な並列実行に関する問題を特定できます。

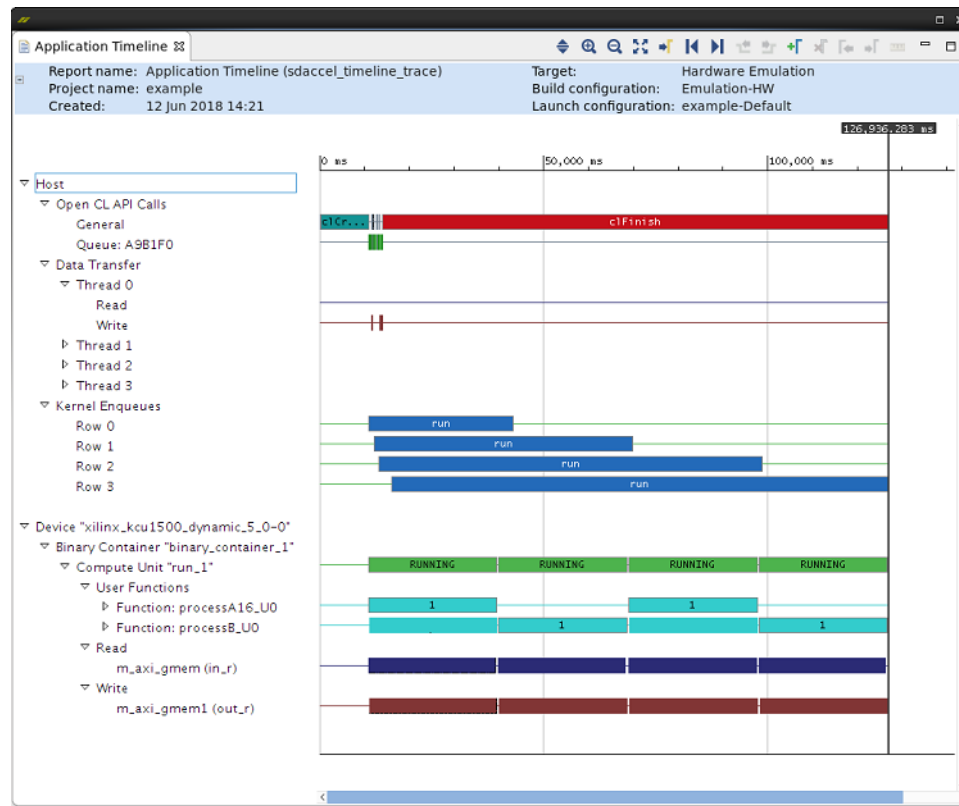


ヒント: デフォルトでは、タイムラインおよびデバイス トレース データは、ハードウェア エミュレーション中のみ収集され、システム ビルドでは収集されません。システム ビルドのデバイス プロファイリングは介入的であり、全体的なパフォーマンスに悪影響を与える可能性があります。この機能は、システム パフォーマンスのデバッグにのみ使用してください。システム テスト中にデータを収集するには、実行コンフィギュレーション設定をアップデートします。詳細は、『SDAccel 環境 プロファイリングおよび最適化ガイド』 ([UG1207](#)) を参照してください。

[Reports] ウィンドウで [Application Timeline] をダブルクリックし、[Application Timeline] ウィンドウを開きます。

次の図に、ホストおよびデバイスのイベントを共通のタイムラインに表示する [Application Timeline] ウィンドウの例を示します。ホスト アクティビティが画像の上部に、カーネル アクティビティが画像の下部に表示されます。ホスト アクティビティには、プログラムの作成、カーネルの実行、およびグローバル メモリとホスト間のデータ転送が含まれます。カーネル アクティビティには、読み出し/書き込みアクセス、およびグローバル メモリとカーネル間の転送が含まれます。この情報は、アプリケーション実行の詳細を理解し、パフォーマンスを向上できる部分を特定するのに有益です。

図 43: アプリケーション タイムライン



コマンドライン フローでもタイムラインデータの収集をイネーブルにできますが、表示には GUI を使用する必要があります。コマンド フローおよび GUI フローを使用してタイムラインデータ コレクションをイネーブルにして表示する方法は、『SDAccel 環境プロファイリングおよび最適化ガイド』(UG1207)を参照してください。

波形ビューおよびライブ波形ビューアー

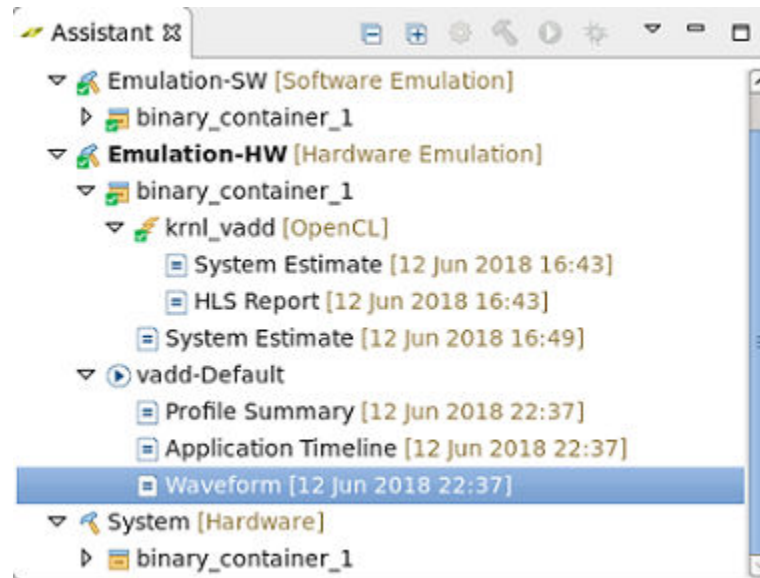
SDx 開発環境では、ハードウェア エミュレーションを実行したときに波形ビューが生成されます。エミュレーションの結果がシステム レベル、計算ユニット (CU) レベル、および関数レベルで詳細に表示されます。詳細には、カーネルとグローバル メモリ間のデータ転送、カーネル パイプ間のデータフローが含まれます。これらの詳細を利用すると、システム レベルから個別の関数呼び出しまでのパフォーマンスのボトルネックが理解でき、アプリケーションが最適化しやすくなります。

ライブ波形ビューアーは、波形ビューアーに似ていますが、さらに下位レベルの詳細を示します。ハードウェア開発で使用されるサイリンクス ツールである xsim を使用して開くこともできます。

波形ビューおよびライブ波形ビューアーのデータを収集するには、ランタイムでハードウェア エミュレーション中にシミュレーション波形を生成する必要があり、時間もディスク容量も消費するので、デフォルトでは実行されません。GUI およびコマンド ラインの両方で波形ビューおよびライブ波形ビューアーのデータ収集をイネーブルにする手順は、『SDAccel 環境プロファイリングおよび最適化ガイド』(UG1207)を参照してください。

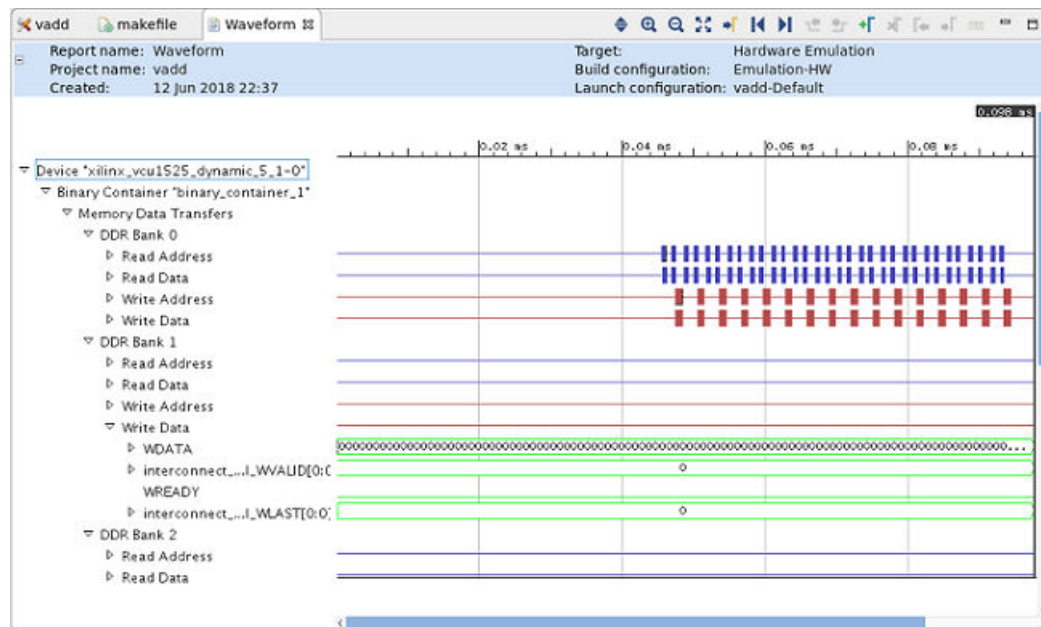
[Assistant] ビューで [Waveform] をダブルクリックすると、波形ビューが開きます。

図 44: 波形ビューを開く



波形ビューが次の例のように表示されます。

図 45: 波形ビューの例

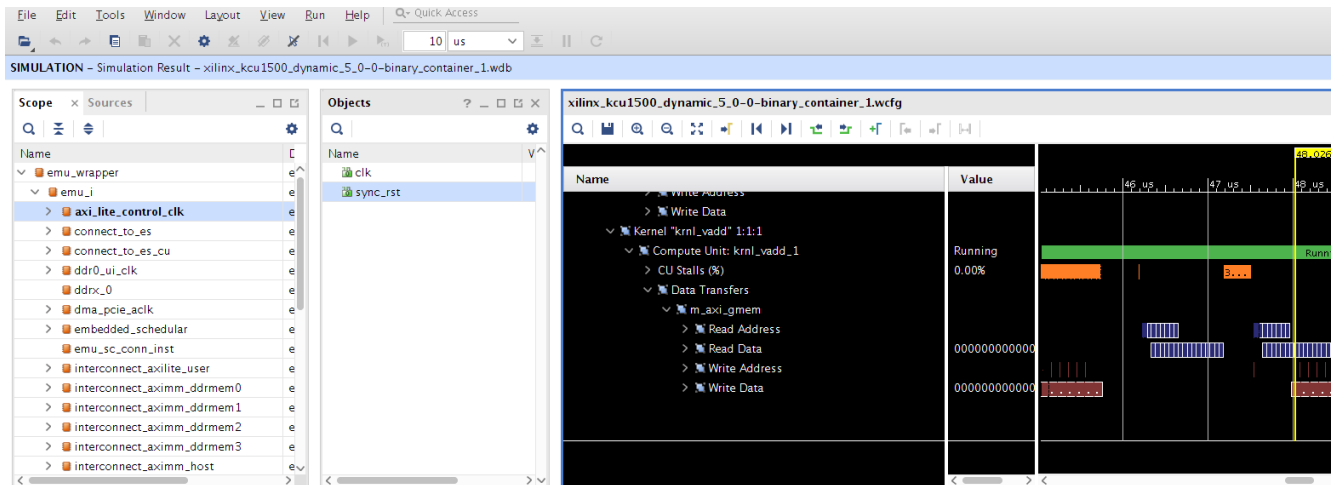


ライブ波形ビューアーは、[Run Configuration Main] タブで [Launch Live Waveform] をクリックすると開くことができます。[Launch Live Waveform] を選択しない場合は、Linux コマンド ラインで `xsim` を使用して波形 (.wdb) を開くことができます。 .wdb ファイルは、プロジェクトディレクトリの Emulation-HW/<kernel_name>-Default サブディレクトリにあります。Linux ラインコマンドで次のコマンドを実行して `xsim` を開きます。

```
xsim -gui <filename.wdb> &
```

次の図に、`xsim` ライブ波形ビューアーの例を示します。

図 46: ライブ波形ビューアーの例



カーネル SLR および DDR メモリの割り当て

周波数およびリソースの点でデザインの結果の品質を満たすためには、カーネル計算ユニット (CU) インスタンスおよび DDR メモリ リソースのフロアプランが重要になります。フロアプランには、CU (カーネル インスタンス) を明示的に SLR に割り当てたり、CU を DDR メモリ リソースにマップしたりする必要があります。フロアプランする際には、CU のリソース使用率と DDR メモリの帯域幅の要件を必ず考慮してください。

最大のザイリンクス FPGA は複数のスタックド シリコン ダイで作成されており、各スタックは SLR (Super Logic Region) と呼ばれ、DDR インスタンスを含め、決まった量のリソースおよびメモリを含んでいます。使用可能なデバイス SLR リソース (カスタム ロジックに使用可能) については、『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) を参照してください。または、『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) で説明する platforminfo ユーティリティを使用して表示することもできます。

実際のカーネル リソース使用率の値を使用することで、CU は SLR をまたいで分散しやすくなるので、どれか 1 つの SLR 内で密集が発生する可能性を削減できます。システム見積もりレポートには、デザインサイクルの早期にカーネルで使用されるリソース (LUT、フリップフロップ、ブロック RAM など) の数がリストされます。レポートは、コマンドラインまたは GUI を使用して、ハードウェアエミュレーションおよびシステム コンパイル中に生成できます。詳細は、[システム見積もりレポート](#) を参照してください。

この情報と使用可能な SLR リソースを使用すると、CU を SLR に割り当てやすくなり、1 つの SLR が使用過多にならないようにできます。SLR の密集が少ないほど、ツールでデザインを FPGA リソースにマップして、パフォーマンスターゲットを満たしやすくなります。メモリ リソースおよび CU のマップについては、それぞれ [メモリ リソースへのカーネルインターフェイスのマップ](#) および [計算ユニットの SLR への割り当て](#) を参照してください。

注記: 計算ユニットは使用可能な DDR メモリ リソースのいずれにでも接続できますが、SLR に割り当てる際は、カーネルの帯域幅要件を考慮する必要もあります。DDR 帯域幅の割り当てと最適化については、『SDAccel 環境プロファイリングおよび最適化ガイド』 (UG1207) を参照してください。

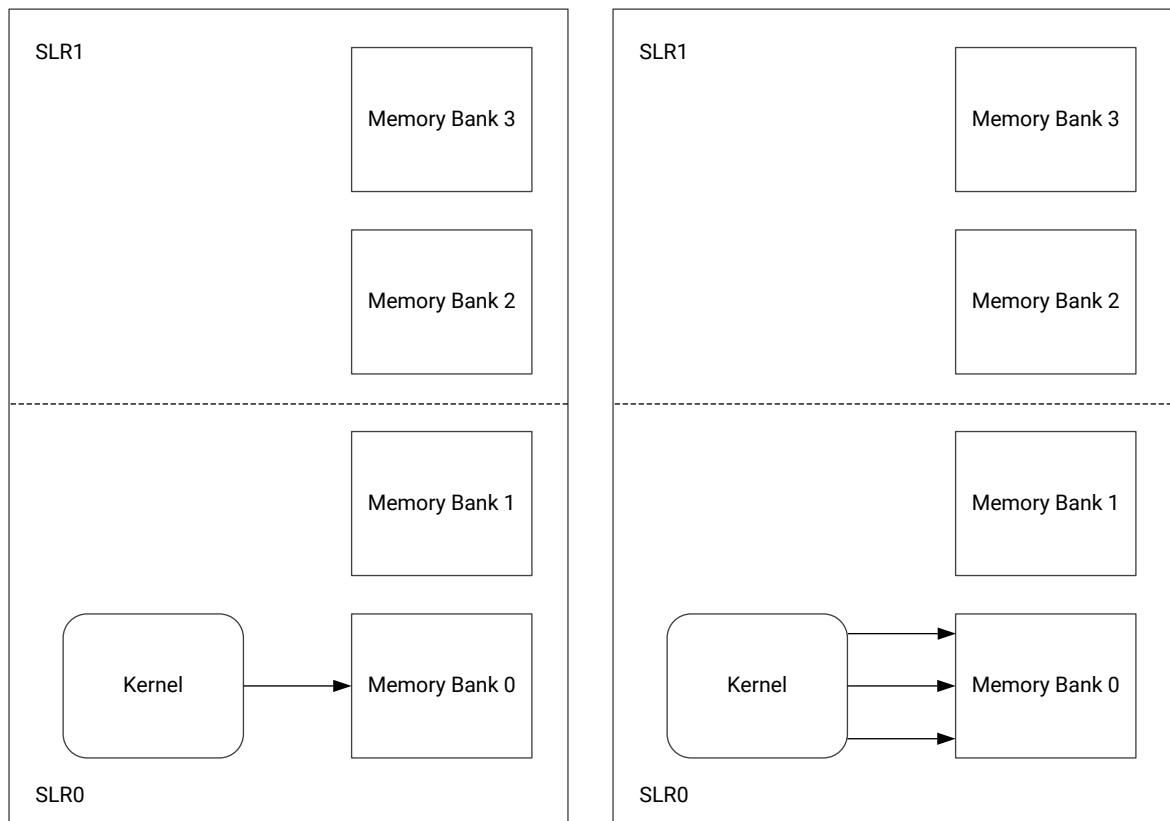
CU を SLR に割り当てたら、CU マスター AXI ポートのいずれかを DDR メモリ リソースにマップします。ザイリンクスでは CU と同じ SLR にある DDR メモリ リソースへ接続することをお勧めしています。この方が、制限のある SLR をまたぐ接続リソースの競合を削減できるからです。また、SLR 間の接続で SLL (Super Long Line) 配線リソースが使用されると、標準の SLR 間配線よりも遅延が大きくなります。

SLR 領域をまたいで別の SLR にある DDR リソースに接続することが必要なことはありますが、`--sp` および `--slr` 指示子の両方が明示的に指定される場合は、ツールで自動的にクロッシング ロジックが追加され、SLL 遅延の影響を最小限に抑えて、タイミング クロージャが改善されるようになっています。

複数のメモリ バンクにアクセスするカーネルのガイドライン

DDR メモリ リソースは、プラットフォームの SLR (Super Logic Region) をまたいで分散されます。SLR 間をまたぐ接続の数は制限されるので、カーネルを DDR メモリ リソースとの接続数が最も多い SLR に配置するのが一般的です。これにより、SLR をまたぐ接続の競合が削減し、SLR をまたぐためにロジック リソースが消費されるのを回避できます。

図 47: 同じ SLR 内のカーネルおよびメモリ



X22194-010919

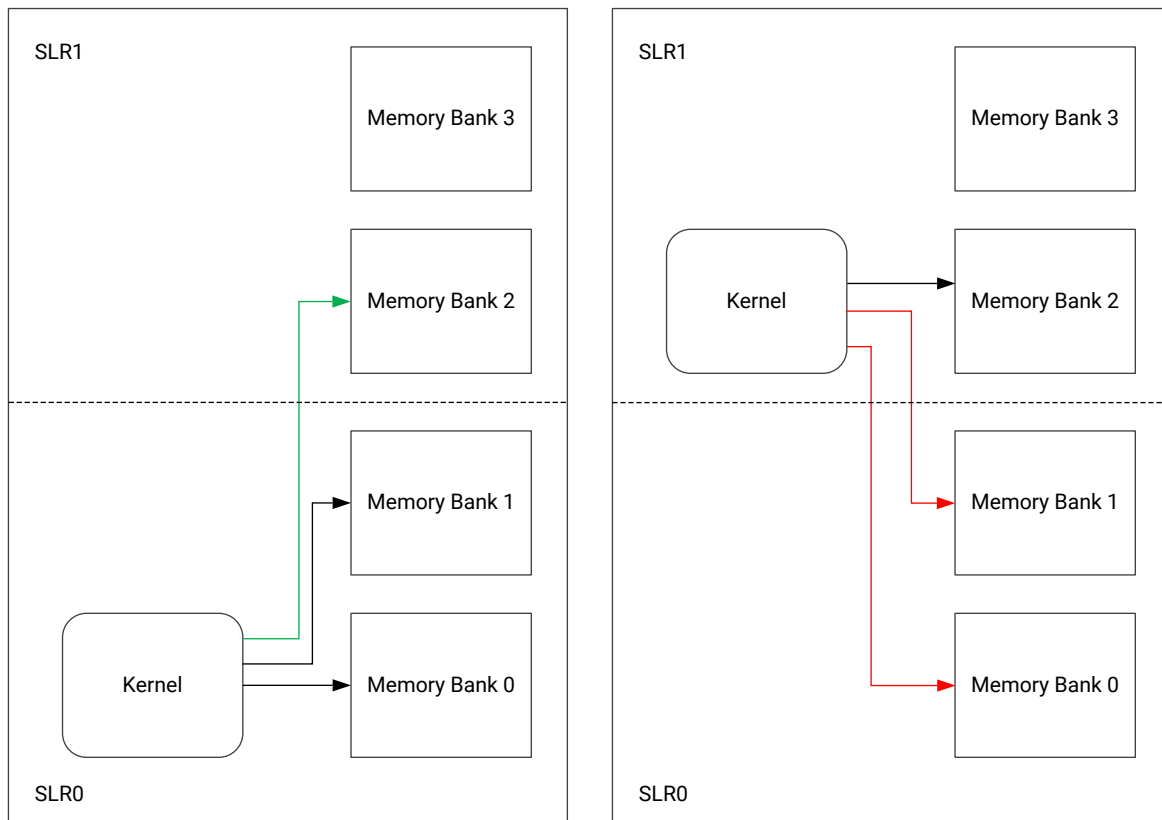
注記: 左の画像では、1つの AXI インターフェイスが1つのメモリ バンクに接続されています。右の画像では、複数の AXI インターフェイスが同じメモリ バンクに接続されています。

上の図に示すように、カーネルに1つの AXI インターフェイスがあり、それが1つのメモリ バンクにのみマップされている場合、『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) で説明されている `platforminfo` ユーティリティにより、そのカーネルのメモリ バンクに接続された SLR がリストされます。この場合、カーネルと同じ SLR 内に含めるのが最適です。この例では、デザイン ツールでカーネルが自動的に追加の入力なしでその SLR に配置される可能性があります、次の場合は明示的に SLR を割り当てる必要があります。

- デザインに同じメモリ バンクにアクセスするカーネルが多数含まれる場合。
- カーネルにメモリ バンクの SLR に含まれない特殊なロジック リソースが必要な場合。

カーネルに複数の AXI インターフェイスがあり、すべてのインターフェイスが同じメモリバンクにアクセスする場合、1つの AXI インターフェイスを含むカーネルと同様に処理でき、カーネルを AXI インターフェイスがマップされているメモリバンクと同じ SLR に配置する必要があります。

図 48: 隣接 SLR のメモリバンク



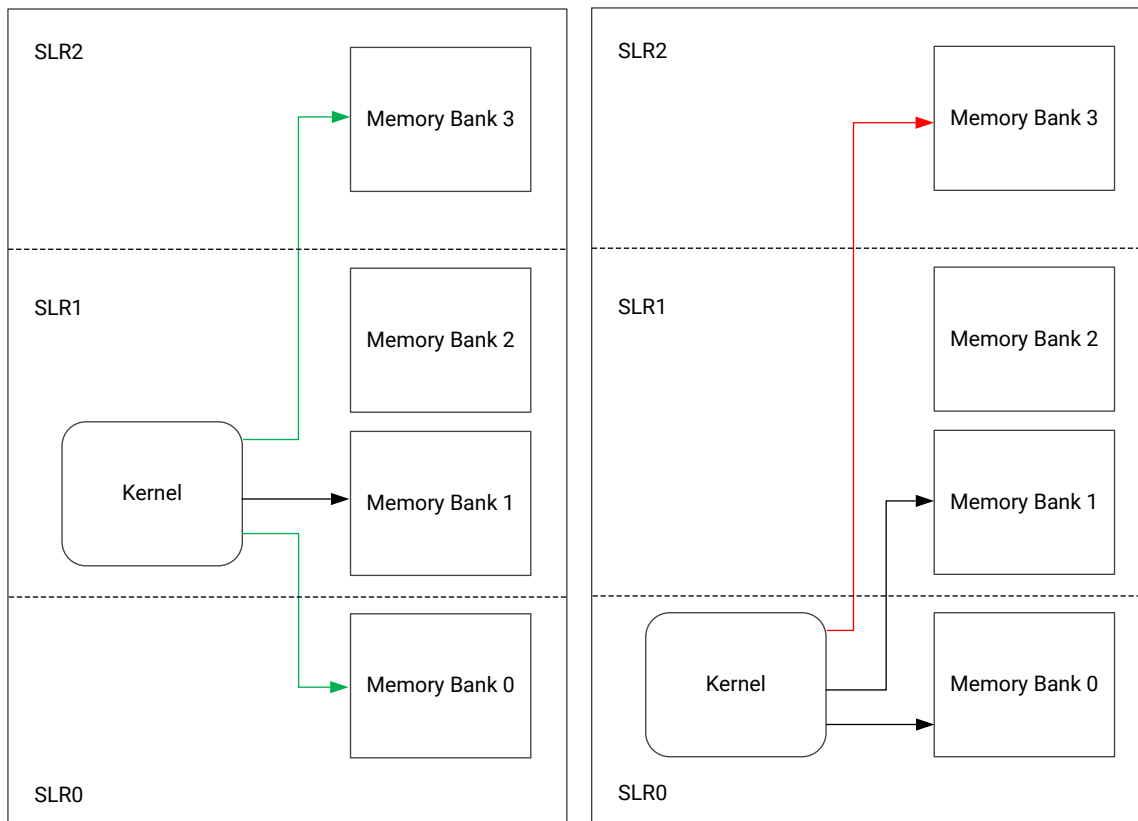
X22195-010919

注記: 左の画像は、カーネルが SLR0 にある場合に、1つの SLR クロッシングが必要なところを示しています。右の画像は、カーネルがメモリバンクにアクセスするために2つの SLR クロッシングが必要なところを示しています。

カーネルに複数の AXI インターフェイスがあり、異なる SLR にある複数のメモリバンクに接続されている場合は、カーネルがアクセスするメモリバンクの大部分が含まれる SLR にカーネルを配置することをお勧めします。これにより、このカーネルに必要な SLR をまたぐ接続数が最小限に抑えられ、ユーザー デザイン内のほかのカーネルでメモリバンクに接続するために使用可能な SLR をまたぐリソースが増えます。

カーネルが別の SLR にあるメモリバンクに接続される場合は、[カーネル SLR および DDR メモリの割り当て](#)のように SLR の割り当てを明示的に指定します。

図 49: 2つ離れたSLRのメモリバンク



X22196-010919

注記:左の画像は、すべてのメモリ マップド バンクにアクセスするのに2つのSLRクロッシングが必要なところを示しています。右の画像は、すべてのメモリ マップド バンクにアクセスするのに3つのSLRクロッシングが必要なところを示しています。

プラットフォームに含まれるSLRが3つ以上になると、上の図に示すように、最も多くマップされるメモリバンクのすぐ隣ではないSLRのメモリバンクにカーネルがマップされることもあります。このような場合、離れたメモリバンクにアクセスするために複数のSLR境界をまたぐ必要があるため、SLRをまたぐリソースの使用量が増加します。このようなリソース使用量の増加を避けるには、カーネルを中間のSLRに配置して、隣接するSLRにまたぐリソースのみが使用されるようにします。

アプリケーションおよびカーネルのデバッグ

SDAccel™ 環境は、ホスト コードとカーネル コード、およびそれらの通信をデバッグするための、アプリケーションレベルのデバッグ機能を提供します。これらの機能および手法は、ソフトウェア中心フローと、より詳細で下位のハードウェア中心フローに分類できます。

また、ハードウェア中心のデバッグでは、ハードウェア上で実行されているデザインを PCIe® (ザイリンクス仮想ケーブル XVC を使用) および JTAG (USB-JTAG ケーブルを使用) の両方を使用してデバッグできます。

デバッグの機能および手法

デザインのデバッグには、いくつかの機能および手法を使用できます。次の表に、3つのビルド コンフィギュレーションでのデバッグに使用可能な機能と手法を示します。各機能および手法については、『SDAccel 環境デバッグ ガイド』 (UG1281) を参照してください。

表 2: 異なるビルド コンフィギュレーションでのデバッグに使用可能な機能および手法

機能/手法	OS	ホスト	カーネル	FPGA (プラットフォーム)
ソフトウェア エミュレーション	dmesg	GDB	GDB	xbutil
ハードウェア エミュレーション	dmesg	GDB	GDB カーネル波形ビューアー	xbutil
system	dmesg	GDB	カーネル波形ビューアー ILA	xbutil

注記:

1. dmesg は Linux コマンドです。
2. GDB は GNU デバッガーです。
3. xbutil はザイリンクスの提供するユーティリティです。

これらの機能および手法は、次の表に示すように、ソフトウェア中心およびハードウェア中心のデバッグ機能に分類できます。

表 3: ソフトウェア中心およびハードウェア中心のデバッグ機能および手法

ソフトウェア中心	ハードウェア中心
GNU デバッガー (GDB)	カーネル波形ビューアー
ザイリンクス ユーティリティ xbutil	ILA (Integrated Logic Analyzer)
Linux dmesg	なし

ソフトウェア中心およびハードウェア中心のデバッグ機能を使用することにより、機能の問題からプロトコルの問題やボードのハングアップまでのさまざまな問題を特定してデバッグできます。

デバッグ フロー

推奨されるアプリケーションレベルのデバッグ フローには、次の3つのレベルのデバッグがあります。

- ソフトウェア エミュレーション (sw_emu) を実行し、アルゴリズムの機能を確認します。
- ハードウェア エミュレーション (hw_emu) を実行し、カスタム ハードウェアを作成して生成されたロジックが正しいかどうかと、FPGA のパフォーマンスを確認します。
- システムビルド (ハードウェア) hw をビルドし、カスタム ハードウェアをインプリメントします。

それぞれデザインに関する特定の情報を提供するので、デバッグしやすくなります。すべてのフローは、統合された GUI フローおよびパッチ フローでサポートされ、基本的なコンパイル時間およびランタイム設定オプションを使用します。次に、各フローを簡単に説明します。

ソフトウェア エミュレーション

ソフトウェア エミュレーションは、C/C++ または OpenCL™ で記述されたホストおよびカーネルが機能的に正しいかどうかを検証するのに使用できます。ホストおよびカーネル コードのデバッグには、GDB を使用できます。ソフトウェア エミュレーションはコンパイル時間が短く、すばやく実行できるので、すべての操作モードでアプリケーションが正しく機能するまでソフトウェア エミュレーションを繰り返すことをお勧めします。

ハードウェア エミュレーション

ハードウェア エミュレーションでは、ホスト コードの検証、ホストおよびカーネルのパフォーマンスのプロファイリング、FPGA リソース使用量の見積もり、ハードウェアの正確なモデル (RTL) を使用したカーネルの検証を実行できます。ハードウェア エミュレーションはソフトウェア エミュレーションよりも時間がかかります。そのため、ザイリンクスではデバッグおよび検証用に小型のデータセットを使用することをお勧めしています。ホストおよびカーネルのデバッグには、GDB を使用できます。見積もられたカーネル パフォーマンスが十分になるまで、ハードウェア エミュレーションを繰り返し実行します。最適化の詳細は、『SDAccel 環境プロファイリングおよび最適化ガイド』(UG1207) を参照してください。

システム

最後にハードウェア実行(システム)で、実際のハードウェア上で完成したシステムを検証し、カーネルが正しく実行され、システム パフォーマンスが満たされることを確認します。SDAccel にはハードウェア デバッグ機能が含まれており、波形解析、カーネル アクティビティ レポート、およびメモリ アクセス解析を実行してハードウェアのクリティカルな問題を特定できます。ハードウェア デバッグではハードウェア モデル全体に追加のロジックを組み込む必要があり、FPGA リソースおよびパフォーマンスに影響します。この追加ロジックは、最終的なコンパイルから削除できます。

GNU デバッグ

GNU デバッグ (GDB) では、ブレークポイントの追加、変数の確認、カーネルまたはホスト コードのデバッグを実行できます。このソフトウェア デバッグ フローを使用すると、デバッグを実行して機能をすばやく検証できます。SDAccel には、アプリケーションから OpenCL ランタイム環境の内容を確認するための特別な GDB 拡張機能が含まれます。これらを使用して、ホストとカーネルの間のプロトコル同期化問題をデバッグできます。

SDAccel 環境では、すべてのフローで GDB ホスト プログラムのデバッグがサポートされますが、カーネルのデバッグはソフトウェアおよびハードウェア エミュレーション フローでのみサポートされます。まず、`xocc` コマンドラインに `-g` オプションを渡すか、GUI オプションで適切なボックスをオンにして、バイナリ コンテナでデバッグ情報を生成する必要があります。

ソフトウェアおよびハードウェア エミュレーション フローでは、アクセラレーションされたカーネル コードのデバッグの操作には制限があります。このコードはソフトウェア エミュレーション フローでは前処理され、ハードウェア エミュレーション フローではハードウェア 記述言語 (HDL) に変換されるので、特にハードウェア エミュレーションではすべての位置にブレークポイントを設定できるわけではありません。

詳細は、『SDAccel 環境デバッグ ガイド』 ([UG1281](#)) を参照してください。

Linux の dmesg

システムのハングアップをデバッグするのは困難ですが、SDAccel では `dmesg` Linux コマンドを使用して、Linux およびハードウェア プラットフォームとの通信をデバッグできます。

ソフトウェアまたはハードウェアがロックアップしている場合、`dmesg` コマンドを使用して、トランザクションの記録およびカーネル情報メッセージを表示できます。詳細なレポートは、問題を特定して解決するのに役立ちます。

カーネル波形ビューアー

SDAccel では、ハードウェア エミュレーション モードの GUI フローで波形ベースの HDL デバッグを実行できます。波形は、Vivado ユーザーが使い慣れた Vivado® 波形ビューアーに開きます。カーネル インターフェイスおよび内部信号を表示でき、再開、HDL ブレークポイントなどのデバッグ制御や、HDL コード ルックアップ、波形マーカーもサポートされます。また、最上位 DDR データ転送 (バンクごと) と、計算ユニットの停止、ループパイプラインのアクティビティ、カーネル特定の詳細も示されます。

詳細は、『SDAccel 環境プロファイリングおよび最適化ガイド』 ([UG1207](#)) を参照してください。

ILA

SDAccel では、System ILA (Integrated Logic Analyzers) をデザインに挿入して、カーネルとグローバル メモリ間のインターフェイス信号を使用して AXI トランザクション レベルのアクティビティを取り込んで表示できます。ILA では、たとえば 1 つまたは複数の信号をトリガーするカスタム イベントを供給し、システム速度で波形をキャプチャできるようにすることが可能です。波形は、ビューアーで表示して解析することによりプロトコル違反やパフォーマンスの問題をデバッグするために使用でき、アプリケーションのハングアップなどの困難な状況をデバッグする際には不可欠です。この下位のハードウェア中心のデバッグ手法は、Vivado ユーザーには使い慣れたものであるはずですが。詳細は、『Vivado Design Suite ユーザー ガイド: プログラムおよびデバッグ』 ([UG908](#)) を参照してください。

注記: ILA コアには、信号データをキャプチャして格納するためのロジックおよびローカル メモリなど、システム リソースが必要です。

デザインに System ILA を挿入するには、`xocc` コマンドで `--dk` オプションを指定します。

次に例を示します。

```
$ xocc --dk chipscope:<compute_unit_name>:<interface_name>
```

キャプチャされたデータは、ザイリンクス 仮想ケーブル (XVC) を介して Vivado ツールからアクセスできます。

コマンド ラインでのアプリケーションのビルド

SDAccel™ GUI を使用してプロジェクトを作成し、ハードウェアでアクセラレーションするアプリケーションをビルドするに加え、ツールへのパスが設定されたコマンドまたはシェル ウィンドウのコマンド ラインインターフェイスを使用して SDx™ システム ツールを起動できます。



ヒント: コマンド シェルを設定するには、`<install_dir>/SDx/<version>` ディレクトリ (`<install_dir>` は SDx ソフトウェアのインストール フォルダー、`<version>` はソフトウェア リリース) から、Linux の場合は `settings64.sh` または `settings64.csh` ファイルを `source` コマンドで読み込みます。

第 5 章: システムのビルド に説明されているように、SDAccel アプリケーション プロジェクトは、ホスト アプリケーションのソフトウェア ビルド プロセスとアクセラレーション カーネルのハードウェア ビルド プロセスの 2 つのプロセスでコンパイルされます。ホスト アプリケーションは、GCC 互換コンパイラである `xcpp` を使用してコンパイルされます。

SDAccel のザイリンクス オープン コード コンパイラ (`xocc`) はコマンド ライン コンパイラで、ユーザーのソース コードを入力として取り込み、Vivado® インプリメンテーション ツールを実行して、FPGA ベースのアクセラレータ カーネルのプログラムに必要なビットストリームなどのファイルを生成します。OpenCL™ C、C/C++、および RTL (SystemVerilog、Verilog、または VHDL) で記述されているカーネルがサポートされます。

この章では、コマンド ライン フローを使用して、コマンド ラインまたはスクリプトからソフトウェアおよびハードウェア コンポーネントをビルドする方法を説明します。`xocc` ツールおよび関連オプションの詳細は、『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。

ホストのビルド

コンパイル

ホスト コード (C/C++ で OpenCL API を使用して記述) をザイリンクス C++ (`xcpp`) コンパイラでコンパイルし、ホスト CPU で実行するホスト実行ファイル (`.exe` ファイル) を生成します。`xcpp` は、標準 `gcc` コンパイラと標準 `gcc` オプションおよびコマンド ライン引数を使用するラッパーです。標準 `gcc` オプションおよびコマンド ライン引数については、ここでは詳細には説明しません。



ヒント: `xcpp` は GCC に基づいており、ここには記載されていませんが、多数の標準 GCC オプションをサポートします。詳細は、[GCC オプション インデックス](#) を参照してください。

次に、デザインをコンパイルする `xcpp` コマンドの例を示します。

```
xcpp -DSDX_PLATFORM=xilinx_vcu1525_dynamic_5_1 \
-I/${XILINX_XRT}/include/ \
-I/${XILINX_VIVADO}/include/ -g -Wall -c -o vadd.o vadd.cpp \
```

使用されているオプションの説明は、次のとおりです。

表 4: ホスト コードのコンパイル

コード	説明
<code>xcpp</code>	
<code>-DSDX_PLATFORM=xilinx_vcu1525_dynamic_5_1</code>	マクロを定義
<code>-I/\${XILINX_XRT}/include/</code>	ヘッダー ファイルのディレクトリを含有
<code>-I/\${XILINX_VIVADO}/include/</code>	ヘッダー ファイルのディレクトリを含有
<code>-I<user include directory></code>	ユーザーのインクルード ファイル ディレクトリ
<code>-g</code>	デバッグ情報を生成
<code>-Wall</code>	警告すべて
<code>-c</code>	コンパイル
<code>-o vadd.o</code>	出力ファイル <code>vadd.o</code> を指定
<code>vadd.cpp</code>	ソース ファイル

リンク

生成されたオブジェクト ファイル (`.o`) がザイリンクス SDAccel ランタイム共有ライブラリとリンクされ、実行ファイル (`.exe`) が作成されます。

次に、デザインをリンクする `xcpp` コマンドの例を示します。

```
xcpp -o vadd.exe -Wl -lxilinuxopencl -lpthread -lrt -lstdc++ \
-L/${XILINX_XRT}/lib/ \
-rpath,/lnx64/lib/csim vadd.o \
```

注記: `-lxilinuxopencl` に対してホスト アプリケーションをリンクする際は、`-rpath-link` が必要です。 `-lOpenCL` とリンクする際は、`-rpath-link` は必要ありません。

使用されているオプションの説明は、次のとおりです。

表 5: ホスト コードのリンク

コード	説明
<code>xcpp</code>	
<code>-o vadd.exe</code>	出力ファイル <code>vadd.exe</code> を作成
<code>-Wl</code>	警告レベルを指定
<code>-lxilinuxopencl -lpthread -lrt -lstdc++</code>	特定ライブラリ ファイルとリンク
<code>-L/\${XILINX_XRT}/lib/</code>	ライブラリ ファイルのディレクトリを検索
<code>-rpath,/lnx64/lib/csim</code>	ランタイム検索パスを指定
<code>vadd.o</code>	オブジェクト コード ファイル

ハードウェアのビルド

コンパイル

システムをビルドする最初の段階は、カーネル アクセラレータ関数のコンパイルです。コンパイルには、`xocc` コンパイラを使用します。カーネルを正しくコンパイルするには、複数の `xocc` オプションを使用する必要があります。次に、これらのオプションについて説明します。

- カーネル ソース ファイルは、`xocc` コマンドに直接ソース ファイルをリストすることにより指定します。複数のソース ファイルを指定できます。
- `-k/--kernel` オプションは、ソース ファイルに関連付けるカーネル名を指定します。

```
xocc ... -k <kernel_name> <kernel_source_file> ... <kernel_source_file>
```

- カーネルのターゲットとなるプラットフォームを指定する必要があります。プラットフォームを指定するには、`--platform xocc` オプションを使用します。

```
xocc ... --platform <platform_name>
```

- ビルド ターゲットを指定する必要があります。`-t / --target xocc` オプションで指定します。

```
xocc ... -t <build_target>
```

デフォルトでは、`build_target` は `hw` に設定されます。[ビルド ターゲット](#)で説明するように、`build_target` は次のいずれかに設定できます。

- `sw_emu`: ソフトウェア エミュレーション
- `hw_emu`: ハードウェア エミュレーション
- `hw`: ターゲット ボード上でのビルド

- `-o` オプションを使用すると、オプションで生成される出力ファイルの名前を指定できます。デフォルトの出力ファイル名は `<kernel>.xo` です。

```
xocc .. -o <xo_kernel_name> .xo
```

- `-R/--report_level` オプションを使用すると、オプションでシステム レポートおよび見積もりレポートを生成できます。また、`-report_dir`、`--log_dir`、および `-temp_dir` オプションを使用すると、`report`、`log`、および `temp` ディレクトリを指定できます。これらのオプションは、生成されたファイルを整理するのに便利です。

```
xocc ... --report_level 2 --report_dir <report_dir_name>
```

- `xocc -c/--compile` オプションを使用してカーネルをコンパイルします。これにより、この後のリンク段階で使用する `.xo` ファイルが生成されます。

```
xocc ... -c
```

次に、コマンド例を示します。

```
xocc -c -k krnl_vadd --platform xilinx_u200 -t sw_emu vadd.cl vadd.h \
-o krnl_vadd.xo --report_level 2 --report_dir reports
```

これにより、次が実行されます。

- カーネルをコンパイル: `-c`
- カーネル名を指定: `-k krnl_vadd`
- プラットフォームを指定: `--platform xilinx_u200`
- ターゲットソフトウェアをエミュレーション: `-t sw_emu`
- ソース入力ファイル: `vadd.cl` および `vadd.h`
- 出力ファイル名を指定: `-o krnl_vadd.xo`
- レベル 2 レポートを生成して `reports` ディレクトリに保存: `--report_level 2 --report_dir reports`

リンク

第 5 章: システムのビルドで説明されているように、ビルド プロセスの 2 番目の段階では、1 つまたは複数のカーネルをプラットフォームにリンクして、バイナリ コンテナ `xclbin` ファイルを作成します。コンパイルと同様、リンクにも複数のオプションが必要です。

- `.xo` ソース ファイルは、`xocc` コマンドに直接ソース ファイルをリストすることにより指定します。複数のソース ファイルを指定できます。

```
xocc ... <kernel_xo_file.xo> ... <kernel_xo_file.xo>
```

- プラットフォームを指定するには、`--platform` オプションを使用します。コンパイル時に指定したプラットフォームを指定する必要があります。

```
xocc ... --platform <platform_name>
```

- ビルド ターゲットを指定するには、`-t` オプションを使用します。コンパイル時に指定したのと同じターゲットを指定する必要があります。

```
xocc ... -t <build_target>
```

- コンパイル段階と同様、`-o` オプションを使用して生成される出力ファイルの名前を指定できます。リンク段階の出力ファイルは、`.xclbin` ファイルです。デフォルトの出力ファイル名は `a.xclbin` です。

```
xocc .. -o <xclbin_name>.xclbin
```

- [複数のカーネル インスタンスの作成](#) に示すように、`--nk` オプションを使用して `.xclbin` ファイルのカーネルに指定した数の計算ユニットをインスタンス化します。計算ユニットのインスタンスの名前を指定するかどうかはオプションですが、ザイリンクスでは指定することをお勧めしています。

```
xocc ... --nk <kernel_name>: <compute_units>:<kernel_name1>:
...:<kernel_nameN>
```

- [メモリ リソースへのカーネル インターフェイスのマップ](#) で説明されているように、`--sp` オプションを使用すると、オプションでカーネル インターフェイスとターゲット DDR バンクの接続を指定できます。各インターフェイスを特定のバンクにマップするには、複数の `--sp` オプションを指定します。

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

- リンクは、`-l/--link` オプションを使用しても実行できます。

```
xocc ... -l
```

次に、コマンド例を示します。

```
xocc -l --platform xilinx_u200 -t sw_emu --nk krnl_vadd:1:krnl_vadd1 \
--sp krnl_vadd1.m_axi_gmem:DDR[3] -o vadd.xclbin krnl_vadd.xo
```

これにより、次が実行されます。

- カーネルをリンク: `-l`
- プラットフォームを指定: `--platform xilinx_u200`
- ターゲットソフトウェアをエミュレーション: `-t sw_emu`
- `krnl_vadd1` という計算ユニットを作成:
 - `--nk krnl_vadd:1:krnl_vadd1`
- `krnl_vadd1`、ポート `m_axi_gmem` を DDR bank3 にマップ:
 - `--sp krnl_vadd1.m_axi_gmem:DDR[3]`
- 出力ファイル名を指定: `-o vadd.xclbin`
- ソース入力ファイル: `krnl_vadd.xo`

sdaccel.ini ファイルの使用

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、およびアクセラレーションボードシステム実行において、ホストアプリケーションおよびカーネルの実行中にデバッグ、プロファイリング、メッセージの記録を制御するため、SDAccel ランタイムライブラリにはさまざまなパラメーターがあります。これらの制御パラメーターはランタイム初期化ファイルで指定されます。

コマンドラインを使用する場合は、`sdaccel.ini` という名前のランタイム初期化ファイルを手動で作成する必要があります。ファイルの場所を定義するには、`SDACCEL_INI_PATH` 環境変数でファイルを含むディレクトリへのパスを指定します。デフォルトでは、`SDACCEL_INI_PATH`、`.exe` パス、`.ini` ファイルの現在のディレクトリの順でファイルが検索されます。ファイルがない場合は、空の文字列が返されます。

SDx GUI を使用している場合は、ユーザーの実行コンフィギュレーションに基づいて自動的に `sdaccel.ini` が作成され、ホスト実行ファイルと同じディレクトリに保存されます。

ランタイムライブラリは、`sdaccel.ini` がホスト実行ファイルと同じディレクトリにあるかどうかを確認し、スタートアップ中にこのファイルからパラメーターを自動的に読み出します。

ランタイム初期化ファイルのフォーマット

ランタイム初期化ファイルはテキスト形式のファイルで、キーのグループおよびその値が記述されています。セミコロン (;) またはシャープ記号 (#) で始まる行はコメント行です。グループ名、キー、キーの値では、大文字/小文字が区別されます。

次の例は、プロファイル タイム ラインでトレースできるようにし、ランタイム時に生成されるメッセージをコンソールに表示させます。

```
#Start of Debug group
[Debug]
timeline_trace = true

#Start of Runtime group
[Runtime]
runtime_log = console
```

次の表に、サポートされるグループ、キー、有効なキー値、およびキーの機能の簡単な説明を示します。

表 6: デバッググループ

キー	有効な値	説明
debug	[true false]	カーネル デバッグをイネーブルまたはディスエーブルにします。 <ul style="list-style-type: none"> • true: イネーブル • false: ディスエーブル • デフォルト: false
profile	[true false]	OpenCL コードのプロファイリングをイネーブルまたはディスエーブルにします。 <ul style="list-style-type: none"> • true: イネーブル • false: ディスエーブル • デフォルト: false
timeline_trace	[true false]	プロファイル タイムライントレースをイネーブルまたはディスエーブルにします。 <ul style="list-style-type: none"> • true: イネーブル • false: ディスエーブル • デフォルト: false
device_profile	[true false]	デバイス プロファイリングをイネーブルまたはディスエーブルにします。 <ul style="list-style-type: none"> • true: イネーブル • false: ディスエーブル • デフォルト: false

表 7: ランタイムグループ

キー	有効な値	説明
api_checks	[true false]	OpenCL API チェックをイネーブルまたはディスエーブルにします。 <ul style="list-style-type: none"> • true: イネーブル • false: ディスエーブル • デフォルト: true

表 7: ランタイム グループ (続き)

キー	有効な値	説明
runtime_log	null console syslog filename	ランタイムのログを出力する場所を指定する。 <ul style="list-style-type: none"> • null: ログを出力しない。 • console: ログを stdout に出力。 • syslog: ログを Linux システム ログに出力。 • filename: ログを指定ファイルに出力。 例: runtime_log=my_run.log • デフォルト: null
cpu_affinity	1 つまたは複数の整数	すべてのランタイム スレッドを指定の CPU に固定します。例: cpu_affinity={4,5,6}
polling_throttle	整数	ランタイム ライブラリがデバイス ステータスを取得する時間の間隔をマイクロ秒で指定する。デフォルト: 0

表 8: エミュレーショングループ

キー	有効な値	説明
aliveness_message_interval	整数	メッセージを出力する時間間隔を秒で指定します。 デフォルト: 300
print_infos_in_console	[true false]	ユーザー コンソールへのエミュレーション情報メッセージの表示を制御します。エミュレーション情報メッセージは、emulation_debug.log というファイルに保存されます。 <ul style="list-style-type: none"> • true: ユーザー コンソールに出力 • false: ユーザー コンソールに出力しない • デフォルト: true
print_warnings_in_console	[true false]	ユーザー コンソールへのエミュレーション警告メッセージの表示を制御します。エミュレーション警告メッセージは、emulation_debug.log というファイルに保存されます。 <ul style="list-style-type: none"> • true: ユーザー コンソールに出力 • false: ユーザー コンソールに出力しない • デフォルト: true
print_errors_in_console	[true false]	ユーザー コンソールへのエミュレーションエラー メッセージの表示を制御します。エミュレーションエラー メッセージは、emulation_debug.log というファイルに保存されます。 <ul style="list-style-type: none"> • true: ユーザー コンソールに出力 • false: ユーザー コンソールに出力しない • デフォルト: true
enable_oob	[true false]	エミュレーション中の範囲外アクセスの診断をイネーブルまたはディスエーブルにします。範囲外アクセスがあった場合は警告が出力されます。 <ul style="list-style-type: none"> • true: イネーブル • false: ディスエーブル • デフォルト: false

表 8: エミュレーショングループ (続き)

キー	有効な値	説明
launch_waveform	[off batch gui]	<p>エミュレーション中の波形の保存および表示方法を指定します。</p> <ul style="list-style-type: none"> off: シミュレータの波形 GUI を起動し、wdb ファイルを保存しません。 batch: シミュレータの波形 GUI を起動せず、wdb ファイルを保存します。 gui: シミュレータの波形 GUI を起動し、wdb ファイルを保存します。 デフォルト: off <p>注記: 波形を保存してシミュレータの GUI に表示するには、カーネルをデバッグをイネーブル (xocc -g) にしてコンパイルする必要があります。</p>

[emconfigutil] 設定

エミュレーション コンフィギュレーション ファイルを作成する際に使用する emconfigutil コマンドとオプションは、[emconfigutil] セクションの [Command] フィールドで指定できます。

emconfigutil およびそのオプションに関する詳細は、『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。

図 50: [emconfigutil] 設定



RTL カーネル

ハードウェア エンジニアの多くは既存の RTL IP (Vivado® IP インテグレーター ベース デザインを含む) を持っていたり、問題なく RTL にカーネルをインプリメントして、Vivado を使用して開発したりできます。SDAccel™ では、RTL デザインを使用できるようになっていますが、ツールフロー内およびランタイム ライブラリ内で使用するには、ソフトウェアおよびハードウェア要件に従う必要があります。



ヒント: RTL カーネルは、『UltraFast 設計手法ガイド (Vivado Design Suite 用)』 (UG949) の推奨事項に従って記述、設計、テストする必要があります。

RTL カーネルとして RTL デザインを使用するための要件

RTL デザインは、インターフェイスとソフトウェア要件の両方を満たしていないと、SDAccel フレームワーク内で RTL カーネルとして使用できません。

これらの要件を満たすため、元の RTL デザインを追加または変更する必要があることもあります。次のセクションで、これについて説明します。

カーネルのインターフェイス要件

SDAccel 実行モデルの要件を満たすには、RTL カーネルが次のインターフェイス要件に従っている必要があります。

- 制御信号にアクセスし、引数を渡すのに使用する AXI4-Lite インターフェイスは 1 つのみ。
- 次のインターフェイスの少なくとも 1 つ (両方を使用可能):
 - メモリと通信するための AXI4 マスター インターフェイス。
 - カーネル間、またはカーネルとホストの間で直接データを転送するための AXI4-Stream インターフェイス。
- 少なくとも 1 つのクロック インターフェイス ポート。

注記: `ap_ctrl_none` カーネル制御インターフェイス オプションを使用する場合は、AXI4 マスター インターフェイスは使用できません。

次の表に、さまざまなインターフェイスの要件を示します。

注記: ポート名を厳密に同じに記述する必要がある場合もあります。

表 9: RTL カーネルおよびポート要件

ポートまたは インターフェイス	説明	コメント
ap_clk	プライマリ クロック入力ポート	<ul style="list-style-type: none"> • 名前を厳密に同じにする必要があります。 • 必須のポート。

表 9: RTL カーネルおよびポート要件 (続き)

ポートまたは インターフェイス	説明	コメント
ap_clk_2	セカンダリ クロック入力ポート	<ul style="list-style-type: none"> 名前を厳密に同じにする必要があります。 オプションのポート。
ap_rst_n	プライマリ アクティブ Low リセット入力ポート	<ul style="list-style-type: none"> 名前を厳密に同じにする必要があります。 オプションのポート。 この信号はタイミングを改善するために内部でパイプライン処理しておく必要があります。 この信号は ap_clk クロック ドメインで同期リセットにより駆動されます。
ap_rst_n_2	オプションのセカンダリ アクティブ Low のリセット入力	<ul style="list-style-type: none"> 名前を厳密に同じにする必要があります。 オプションのポート。 この信号はタイミングを改善するために内部でパイプライン処理しておく必要があります。 この信号は ap_clk_2 クロック ドメインで同期リセットにより駆動されます。
interrupt	アクティブ High の割り込み。	<ul style="list-style-type: none"> 名前を厳密に同じにする必要があります。 オプションのポート。ポートが使用されていない場合は、削除する必要があります。
s_axi_control	唯一の AXI4-Lite スレーブ制御インターフェイス	<ul style="list-style-type: none"> 名前は exact (大文字/小文字の区別あり) にする必要があります。 必須のポート インターフェイス。
AXI4_MASTER	グローバルメモリにアクセスするための 1 つ以上の AXI4 マスター インターフェイス	<ul style="list-style-type: none"> AXI4 マスター インターフェイスにはすべて 64 ビット アドレスが必要です。 グローバル メモリ空間の分割はカーネル開発者が実行します。グローバル メモリの各パーティションがカーネル引数になります。各パーティションのメモリ オフセットは、AXI4-Lite スレーブ インターフェイスを介して制御レジスタ プログラムابلによって設定されます。 AXI4 マスターには WRAP または FIXED タイプのバーストは使用できませんし、サイズが満たないバーストも使用できません。つまり、AxSIZE は AXI データ バスの幅に一致している必要があります。 これらの要件を満たさないユーザー ロジックまたは RTL コードは、ラップしておくかブリッジさせて、これらの要件を満たすようにする必要があります。

カーネルのソフトウェア要件

RTL カーネルには、OpenCL™ および C/C++ カーネルと同じソフトウェアインターフェイス モデルがあります。つまり、RTL カーネルとは、ホスト アプリケーションでは void 戻り値、スカラー引数、およびポインター引数を持つ関数のことです。次はその例です。

```
void mmult(unsigned int length, int *a, int *b, int *output)
```

SDAccel 実行モデルでは、次が検出されます。

- スカラー引数は、AXI4-Lite スレーブ インターフェイスを介してカーネルに直接書き込まれます。
- ポインター引数は、メモリに送信またはメモリから受信されます。

- カーネルは、1つまたは複数の AXI4 メモリ マップ インターフェイスを介してグローバル メモリのデータを読み出し/書き込みするか、ホストとカーネルの間で直接ストリーミングします。
- カーネルは AXI4-Lite スレーブ インターフェイスを使用した制御レジスタ (次を参照) を介してホスト アプリケーションで制御されます。

RTL デザインに別の実行モデルがある場合は、デザインがこの方法で実行できるようにそれを適用する必要があります。

次の表に、SDAccel 環境内でカーネルを使用するために必要なレジスタ マップを示します。制御レジスタはすべてのカーネルに必要ですが、割り込み関連のレジスタは割り込みを含むデザインにのみ必要です。すべてのユーザー定義のレジスタは、0x10 で開始される必要があります。これより下の位置は予約されています。

表 10: アドレス マップ

アドレス	名前	説明
0x0	制御	カーネル ステータスを制御および示します。
0x4	グローバル割り込みイネーブル	ホストへの割り込みをイネーブルにします。
0x8	IP 割り込みイネーブル	割り込みの生成に使用する IP で生成された信号を制御します。
0xC	IP 割り込みステータス	割り込みステータスを示します。
0x10	カーネル引数 (アドレス 0x10 で開始)	スカラーおよびグローバル メモリ引数を含みます。

制御レジスタの定義は、次の 3 つのカーネル操作モードによって異なります。これらのモードの詳細は、『Vivado Design Suite ユーザー ガイド: 高位合成』 (UG902) を参照してください。

- ap_ctrl_none (フリーランニング カーネル)
- ap_ctrl_hs (パイプライン カーネル)
- ap_ctrl_chain (パイプライン カーネル)

選択したカーネルの操作モードに対応した制御レジスタの定義に従う必要があります。

ap_ctrl_none

ap_ctrl_none モードでは、カーネルはリセットから抜けるとすぐに開始し、停止しません。ストリーミング カーネルでのみ使用します ([Streaming Interfaces] ページ を参照)。

表 11: ap_ctrl_none モードでの制御 (0x0)

ビット	Name	説明
31:0	予約済み	予約済み

ap_ctrl_hs

ap_ctrl_hs モードでは、ドライバーが ap_start に 1 を書き込み、ap_start がディassert (入力データが完全に処理されたことを示す) され、ap_done がアassert (出力データが完全に生成されたことを示す) されるのを待ちます。次の表に、このモードでの制御レジスタビットの定義を示します。

ap_ctrl_hs カーネルは、終了後にのみ再度開始できます。実行をパイプライン処理したりオーバーラップしたりすることはできません。

表 12: ap_ctrl_hs モードでの制御 (0x0)

ビット	名前	説明
0	ap_start	カーネルがデータ処理を開始したときにホストによりアサート。ap_ready がアサートされたハンドシェイクでカーネルによりクリア。
1	ap_done	カーネルが出力データの処理を終了したときにカーネルによりアサート。ホストによる読み出しでクリア。
2	ap_idle	カーネルがアイドルになるとカーネルによりアサート (廃止予定)。
3	ap_ready	カーネルが入力データの処理を終了したときにカーネルによりアサート。即時に自己クリア。
6:4	予約済み	予約済み
7	auto_restart ¹	アサートされると、カーネルにより ap_start がアサートに保持されます。ホストによる読み出し/書き込み。
31:8	予約済み	予約済み

注記:

1. auto_restart ビットは、ザイリンクス ランタイムでは使用されません。

ap_ctrl_chain

ap_ctrl_chain モードでは、ドライバーは ap_start をアサートし、次のいずれかが発生するのを待ちます。

- ap_start がディアサートされて入力データが完全に処理されたことが示され、次のバッチを開始できる状態になる。
- ap_done がアサートされて出力データが完全に処理されたことが示され、ap_continue がアサートされてカーネルが操作を継続できる状態になる。

パイプライン実行が望ましい場合は、このモードをお勧めします。次の表に、ap_ctrl_chain モードでの制御レジスタビットの定義を示します。

表 13: ap_ctrl_chain モードでの制御 (0x0)

ビット	名前	説明
0	ap_start	カーネルがデータ処理を開始したときにホストによりアサート。ap_ready がアサートされたハンドシェイクでカーネルによりクリア。
1	ap_done	カーネルが出力データの処理を終了したときにカーネルによりアサート。ap_continue がアサートされたハンドシェイクでカーネルによりクリア。
2	ap_idle	カーネルがアイドルになるとカーネルによりアサート (廃止予定)。
3	ap_ready	カーネルが入力データの処理を終了したときにカーネルによりアサート。即時に自己クリア。
4	ap_continue	カーネルが操作を継続できるようになったときにホストによりアサート。カーネルにより即時クリア。
6:5	予約済み	予約済み
7	auto_restart ¹	アサートされると、ap_start および ap_continue がアサートに保持されます。ホストによる読み出し/書き込み。
31:8	予約済み	予約済み

注記:

1. auto_restart ビットは、ザイリンクス ランタイムでは使用されません。

割り込みレジスタ

次の割り込み関連のレジスタは、カーネルに割り込みがある場合にのみ必要です。

表 14: グローバル割り込みイネーブル (0x4)

ビット	名前	説明
0	グローバル割り込みイネーブル	ホストにより IP 割り込みイネーブル ビットと共にアサートされると、この割り込みがイネーブルになります。ホストによる読み出し/書き込み。
31:1	予約済み	予約済み

表 15: IP 割り込みイネーブル (0x8)

ビット	名前	説明
0	チャンネル 0 (ap_done)	グローバル割り込みイネーブル ビットと共にアサートされると、ap_done がアサートされたときに割り込みがアサートされます。ホストから読み出し/書き込みアクセス、カーネルからは読み出しのみ。
1	チャンネル 1 (ap_ready)	グローバル割り込みイネーブル ビットと共にアサートされると、ap_ready がアサートされたときに割り込みがアサートされます。ホストから読み出し/書き込みアクセス、カーネルからは読み出しのみ。
31:2	予約済み	予約済み

表 16: IP Interrupt Status (0xC)

ビット	名前	説明
0	チャンネル 0 (ap_done)	ap_done により割り込みがアサートされたときに、カーネルによりこの割り込みステータス ビットがアサートされます。ap_done の割り込みをディスエーブルにした場合は、このビットがカーネルによりアサートされることはありません。ホストにより 1 を書き込むことによりこのビットをクリアする必要があります。
1	チャンネル 1 (ap_ready)	ap_ready により割り込みがアサートされたときに、カーネルによりこの割り込みステータス ビットがアサートされます。ap_ready の割り込みをディスエーブルにした場合は、このビットがカーネルによりアサートされることはありません。ホストにより 1 を書き込むことによりこのビットをクリアする必要があります。
31:2	予約済み	予約済み

割り込み

RTL カーネルには、1 つの割り込みがある割り込みポートをオプションで含めることができます。ポート名は `interrupt` にして、アクティブ High にする必要があります。これは、GIE (Global Interrupt Enable) および IER (Interrupt Enable) ビットが両方ともアサートされるとイネーブルになります。さらに、割り込みは IP 割り込みステータス レジスタのアサートされているビットに 1 が書き込まれた場合にのみクリアされます。

`interrupt` ポートをカーネルに追加する場合、この情報を含めて `kernel.xml` ファイルをアップデートする必要があります。`kernel.xml` は、RTL カーネル ウィザードを使用すると自動的に生成されます。ファイルのアップデートの詳細は、[カーネル記述 XML ファイルの作成](#) を参照してください。

RTL Kernel ウィザード

RTL Kernel ウィザードを使用すると、SDAccel のシステムに統合可能なカーネルに RTL IP をパッケージするのに必要な手順の一部を自動化できます。

ウィザードの利点は、次のとおりです。

- SDAccel のシステムに統合可能なカーネルに RTL IP をパッケージするのに必要な手順の一部を自動化します。
- RTL カーネルのソフトウェア機能モデルおよびインターフェイス モデルを順を追って指定できます。
- 入力したインターフェイス情報に基づいて、RTL カーネル インターフェイス要件を満たすカーネルの RTL ラッパーを生成します。
- 制御ロジックおよびレジスタ ファイルを含む AXI4-Lite インターフェイス モジュールを自動的に生成します。AXI4-Lite インターフェイス モジュールは、生成された最上位 RTL カーネル ラッパーに含まれます。
- このラッパーには、サンプルのカーネル IP モジュールが含まれています。これは後でユーザーの RTL IP デザインに置き換える必要があります。ラッパー テンプレートを使用して RTL IP との接続が正しいことを確認する必要があります。
- ウィザードで指定されたソフトウェア機能プロトタイプと動作に一致するように、`kernel.xml` ファイルが生成されます。

RTL カーネル ウィザードは、VADD と呼ばれる単純な加算器 RTL IP を使用するサンプル デザインを含む Vivado プロジェクトを生成します。また、ウィザードでのユーザー入力に基づくインターフェイス、制御ロジック、およびレジスタ マップに一致する関連 RTL ラッパーも生成します。このラッパーを使用すると、SDAccel フレームワークでアクセス可能な RTL カーネルに RTL IP をラップできます。

注記: ウィザードで生成されるコードを使用する必要はありません。上記のソフトウェアおよびインターフェイス要件を満たしていれば、独自の RTL カーネルを生成することもできます。

生成されたラッパーを使用する場合、生成した RTL IP (VADD) をご自身の RTL IP と置き換えて、ラッパーに接続します。

接続には、クロック、リセット、AXI4-Lite インターフェイス、メモリ インターフェイス、およびオプションでストリーミング インターフェイスが含まれます。接続数は、ウィザードで指定されたインターフェイス情報 (2 つの AXI4 メモリ インターフェイスを選択するなど) に基づきます。IP へのこれらの接続を手動で実行し、デザインを検証する必要があります。

RTL Kernel ウィザードにより、最上位 RTL カーネル ラッパーと生成されたファイルの Vivado プロジェクトが生成されます。これにより、RTL カーネルを簡単にアップデートおよび最適化できるようになります。

RTL Kernel ウィザードではさらに、生成された RTL カーネル ラッパーの単純なテストベンチと、サンプル RTL カーネルを実行するためのサンプル ホスト コードも生成されます。このサンプル テストベンチとホスト コードを、ユーザーの RTL IP デザインをテストできるように変更する必要があります。

次のセクションで、RTL Kernel ウィザードの使用方法を説明します。

RTL カーネル ウィザードの起動

RTL カーネル ウィザードは、SDx™ 開発環境または Vivado 統合設計環境 (IDE) から起動できます。SDx 開発環境では、生成されたカーネル/サンプルのホスト コードが SDx プロジェクトに自動的にインポートされるので、よりスムーズに作業できます。

RTL カーネル ウィザードを SDx 開発環境から起動するには、次の手順に従います。

1. SDx 開発環境を起動します。
2. SDx プロジェクト (アプリケーション プロジェクト タイプ) を作成します。
3. [Xilinx] → [RTL Kernel Wizard] をクリックします。

RTL カーネル ウィザードを Vivado から起動するには、次の手順に従います。

1. プラットフォーム上でターゲットにする予定のデバイスを選択して、新しい Vivado プロジェクトを作成します。ターゲット デバイスがわからない場合は、デフォルトのパーツを選択します。
2. [IP Catalog] ボタンをクリックして IP カタログを開きます。
3. IP カタログの検索ボックスに「wizard」と入力します。
4. [SDx Kernel Wizard] をダブルクリックしてウィザードを起動します。

注記: 同じバージョンが使用されるように、SDx インストールから Vivado を使用してください。

RTL Kernel ウィザードの使用

このウィザードでは、カーネル作成のプロセスを複数の手順に分割し、各ページで順にカーネルを設定できるようになっています。ページ間は [Next] および [Back] をクリックして移動します。カーネルを最終確認し、[OK] をクリックしてウィザードの入力に基づいてプロジェクトを作成します。次のセクションで、ウィザードの各ページおよびその設定オプションを説明します。

RTL Kernel ウィザードの [General Settings] ページ

次の図に、RTL Kernel ウィザードの [General Settings] ページを示します。

図 51: RTL Kernel ウィザードの [General Settings] ページ

RTL Kernel Wizard (1.0)

Documentation IP Location Switch to Defaults

General Settings

Kernel identification

Kernel name: sdx_kernel_wizard_0

Kernel vendor: mycompany.com

Kernel library: kernel

Kernel options

Kernel type: RTL

Kernel control interface: ap ctrl hs

Clock and Reset options

Number of clocks: 1

Has reset: 0

< Back Next > Page 2 of 6

OK Cancel

[Kernel Identification]

次は、RTL カーネル ウィザードの [General Settings] ページを示しています。

- [Kernel name]: カーネル名。IP、最上位モジュール、カーネル、C/C++ 論理モデルの名前を指定します。この名前は C および Verilog の命名規則に従う必要があります。Vivado IP インテグレーターの命名規則にも従う必要があるので、アンダースコア () を英数字の間以外には使用できません。
- [Kernel vendor]: ベンダーの名前。『Vivado Design Suite ユーザー ガイド: IP を使用した設計』(UG896) で説明されている VLNV (Vendor/Library/Name/Version) に使用されます。
- [Kernel library]: ライブラリの名前。VLNV で使用されます。同じ識別子の規則に従う必要があります。

[Kernel options]

- [Kernel type]: RTL カーネル タイプは、Verilog 制御レジスタ モジュールを含む Verilog RTL 最上位モジュールと、最上位モジュール内の Verilog カーネル サンプルで構成されます。ブロック デザイン カーネル タイプでも Verilog RTL 最上位モジュールが配布されますが、Verilog RTL 最上位モジュール内に IP インテグレーター ブロック図がインスタンス化されます。ブロック デザインは、制御レジスタをエミュレートするため、ブロック RAM 交換メモリを使用する MicroBlaze™ サブシステムから構成されます。カーネルの制御目的で MicroBlaze の使用方法を示すため、サンプルの MicroBlaze ソフトウェアがプロジェクトと共に配布されています。
- [Kernel control interface]: カーネルの操作モードを選択します。選択可能なモードには、`ap_ctrl_hs`、`ap_ctrl_none`、および `ap_ctrl_chain` があります。詳細は、[カーネルのソフトウェア要件](#) を参照してください。
- [Enable MicroBlaze debug] (一部のコンフィギュレーションのみ): MDM (MicroBlaze Debug Module) をブロック デザイン カーネル タイプの例に追加します。MDM モジュールのバウンダリスキャンインターフェイスはカーネルの最上位に接続されます。デバッグインターフェイスは MicroBlaze インスタンスに接続されます。このオプションは、ザイリンクス仮想ケーブルを介してシステム デバッグをサポートするプラットフォームに対して、[Kernel type] を [Block Design] に選択している場合にのみ設定可能です。

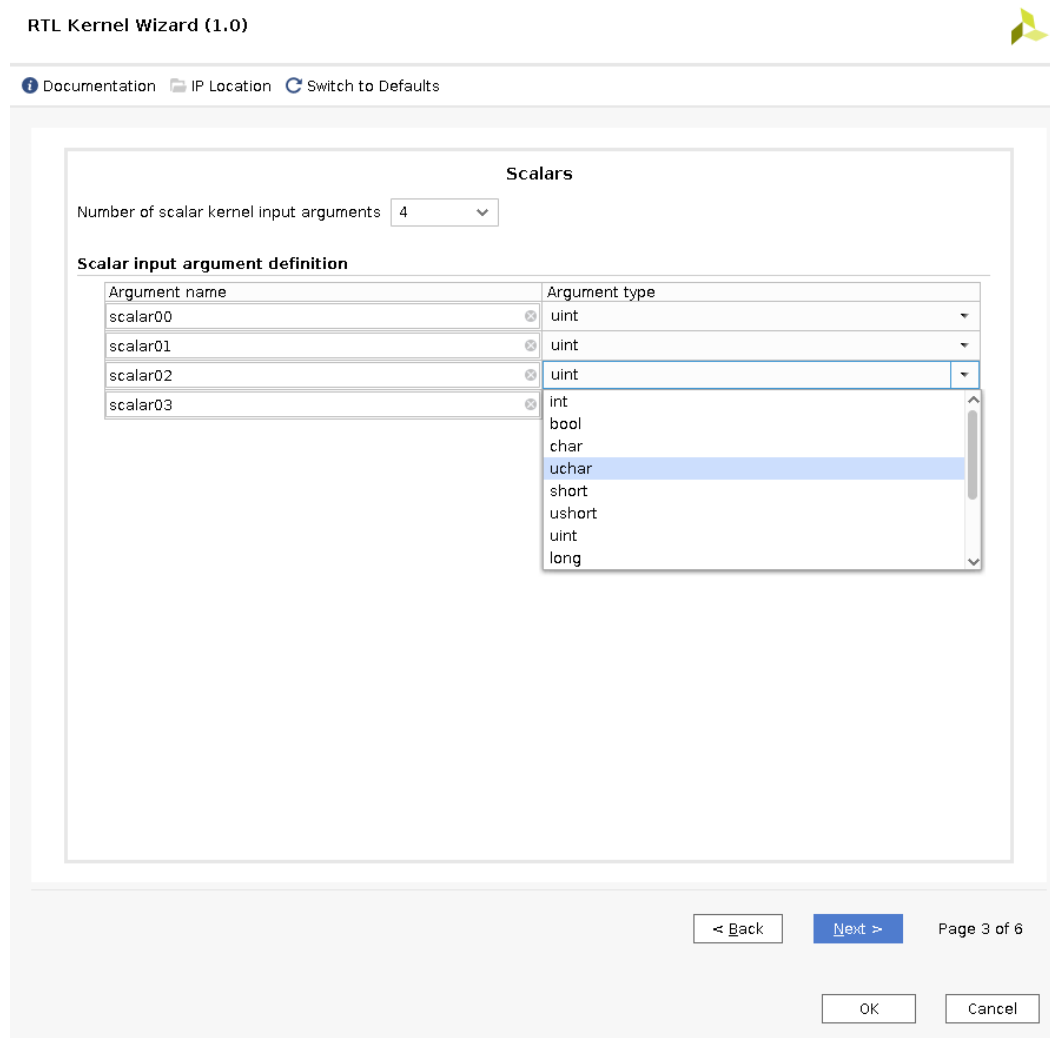
[Clock and Reset options]

- [Number of clocks]: カーネルで使用されるクロックの数を設定します。各カーネルには、`ap_clk` というプライマリクロックと `ap_rst_n` というリセットがあります。カーネルのすべての AXI インターフェイスは、このクロックおよびリセットを使用して駆動されます。[Number of clocks] を 2 に設定すると、カーネル内で使用するためのセカンダリクロックおよび関連リセットが供給されます。セカンダリクロックおよびリセットはそれぞれ `ap_clk_2`、`ap_rst_n_2` という名前です。このセカンダリクロックは、プライマリクロックから独立しており、個別の周波数スケーリングがサポートされます。セカンダリクロックは、カーネルクロックがプライマリクロックが供給される AXI4 インターフェイスよりも高速または低速の場合に便利です。複数のクロックを使用して設計する場合は、どのクロック周波数が使用されている場合でもデータの完全性が確立されているようにするために、正しいクロック乗せ換えテクニックを使用してください。
- [Has reset]: カーネルに最上位リセット入力ポートを含めるかどうかを指定します。リセットを含めないようにすると、大型デザインで配線の密集を緩和できる場合があります。デザインで通常リセットを持つレジスタには、正しくするため適切な初期値を指定する必要があります。このオプションをオンにすると、各クロックにリセットポートが含まれます。ブロック デザイン タイプのカーネルにはリセット入力が必要です。

[Scalars] ページ

スカラー引数は、情報の制御タイプをカーネルに渡すために使用されます。スカラー引数をホストから読み出すことはできません。指定した各引数に対して、ソフトウェアからハードウェアに引数を渡すための制御レジスタが作成されます。次の図を参照してください。

図 52: RTL カーネル ウィザードの [Scalars] ページ



RTL Kernel Wizard (1.0)

Documentation IP Location Switch to Defaults

Scalars

Number of scalar kernel input arguments: 4

Scalar input argument definition

Argument name	Argument type
scalar00	uint
scalar01	uint
scalar02	uint
scalar03	int

int
bool
char
uchar
short
ushort
uint
long

< Back Next > Page 3 of 6

OK Cancel

- [Number of scalar kernel input arguments]: カーネルに渡すスカラー入力引数の数を指定します。指定した各数値に対して、引数名および引数タイプをカスタマイズする行が表示されます。スカラーの最小値要件はありません。ウィザードに使用可能な最大値は 64 です。

[Scalar input argument definition]

スカラー入力引数の定義は次のとおりです。

- [Argument name]: 生成された Verilog 制御レジスタ モジュールで出力信号として使用される引数名です。各引数に ID 値が割り当てられます。この ID 値はホスト ソフトウェアからの引数にアクセスするために使用されます。ID 値引数は、このウィザードのサマリ ページに表示されます。互換性を最大限にするため、引数名はカーネル名と同じ識別規則に従います。

- [Argument type]: 引数のデータ型 (ビット幅) を指定します。これは、生成される Verilog モジュールのレジスタの幅に影響します。使用可能なデータ型は、「6.1.1 Built-in Scalar Data Types」セクションの「[OpenCL C Specification Version 2.0](#)」で指定されているもののみです。この仕様には、データ型ごとに関連するビット幅が示されています。RTL ウィザードでは、引数タイプに関係なく、レジスタ マップのすべてのスカラーに対して 64 ビットを予約します。引数タイプが 32 ビット以下の場合、RTL ウィザードは 64 ビットのうちの上位 32 ビットを予約済みのアドレス位置に設定します。データ型のビット幅が 32 ビットを超える場合は、制御レジスタへの書き込み操作が 2 つ必要になります。

[Global Memory] ページ

グローバル メモリは、カーネルから AXI4 マスター インターフェイスを介してアクセスします (次の図を参照)。

図 53: [Global Memory] ページ

RTL Kernel Wizard (1.0)

Documentation IP Location Switch to Defaults

Global Memory

Number of AXI master interfaces: 2

AXI master definition

Interface name	Width (bytes)	Number of arguments
m00_axi	64	2
m01_axi	64	1

Argument definition

Interface	Argument name
m00_axi	axi00_ptr0
m00_axi	axi00_ptr1
m01_axi	axi01_ptr0

< Back Next > Page 4 of 6

OK Cancel

各 AXI4 インターフェイスの動作は独立しており、各 AXI4 インターフェイスを 1 つまたは複数のメモリ コントローラーに接続して DDR4 などのオフチップメモリに接続できます。グローバルメモリは、主にホストとカーネルの間で大型のデータセットを送受信するため使用され、カーネル間のデータ転送用にも使用できます。最適なパフォーマンスを得るためにこれらのインターフェイスを設計する方法は、[AXI4 インターフェイスのメモリ パフォーマンスの最適化](#)を参照してください。各インターフェイスに対して、開始点として使用可能なサンプル AXI マスター ロジックが RTL カーネルに生成されますが、使用しない場合は破棄できます。

[Global Memory] ページでは、[Number of AXI master interfaces] オプションでカーネル上の AXI マスター インターフェイスの数を指定します。最大は 16 インターフェイスです。各インターフェイスに対し、インターフェイス名、データ幅、関連の引数の数を指定できます。各インターフェイスには、すべての読み出しおよび書き込みチャンネルが含まれます。RTL Kernel ウィザードで付けられるデフォルト名は `m00_axi` および `m01_axi` です。変更しない場合は、これらの名前を `--sp` オプションで DDR バンクを割り当てるときに使用する必要があります。

[AXI master definition]

- [Interface name]: インターフェイスの名前を指定します。互換性を最大限にするため、引数名はカーネル名と同じ識別規則に従います。
- [Width (bytes)]: AXI データチャンネルのデータ幅を指定します。ザイリンクスでは、この幅をメモリ コントローラーの AXI4 スレーブ インターフェイスのネイティブ データ幅に一致させることをお勧めしています。メモリ コントローラーのスレーブ インターフェイスは通常 64 バイト (512 ビット) 幅です。
- [Number of arguments]: このインターフェイスに関連付ける引数の数を指定します。各引数は、カーネルからアクセス可能なグローバルメモリへのデータ ポインターを表します。

[Argument definition]

- [Interface]: 現在の行の対応する列に関連付ける AXI インターフェイスの名前を指定します。前の表で定義したインターフェイス名からコピーされるので、この値は直接変更できません。
- [Argument name]: 関数プロトタイプ シグネチャに表示されるように、ポインター引数の名前を指定します。各引数に ID 値が割り当てられます。この ID 値はホスト ソフトウェアからの引数にアクセスするために使用されます。ID 値引数は、このウィザードのサマリ ページに表示されます。互換性を最大限にするため、引数名はカーネル名と同じ識別規則に従います。生成された Verilog 制御レジスタ モジュールで出力信号として使用される引数名です。

[Streaming Interfaces] ページ

[Streaming Interfaces] ページでは、カーネルの AXI4-Stream インターフェイスを設定できます。ストリーミング インターフェイスは、カーネル間およびカーネルとホスト間 (一部の QDMA ベース プラットフォームのみ) のバス接続に使用できます。カーネル間の通信では、AXI4-Stream 信号セットおよびプロトコルがカーネル間で一致している必要があります。ホストからカーネルおよびカーネルからホストへの通信に使用されるストリーミング インターフェイスは、厳密なプロトコルおよび信号宣言に従う必要があります。QDMA AXI4-Stream プロトコルは、AXI4-Stream プロトコルの `TDATA/TKEEP/TLAST` 信号を使用します。ストリーム トランザクションには、一連の転送が含まれ、最後の転送は `TLAST` 信号がアサートされて終了します。次の図に、設定オプションを示します。ホストとのストリーム転送は、次の要件に従う必要があります。

- AXI4-Stream 転送は `TVALID/TREADY` が両方ともアサートされると発生します。
- `TDATA` 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。
- `TKEEP` (毎バイト) は `TLAST` が 0 の場合はすべて 1 である必要があります。
- `TKEEP` は `TLAST` が 1 の場合に不揃いな尾部の信号を送信するのに使用できます。たとえば、4 バイト インターフェイスの場合、`TKEEP` は `0b0001`、`0b0011`、`0b0111`、`0b1111` のいずれかにしかならず、それぞれ 1 バイト、2 バイト、3 バイト、4 バイトのサイズを指定します。

- TKEEP は(TLAST が1の場合ですら) すべて0にはできません。
- TLAST はパケットの最後にアサートされる必要があります。
- TREADY 入力/TVALID 出力はカーネルが開始しない場合は Low にして、転送を失わないようにする必要があります。


 図 54: [Streaming Interfaces] ページ

RTL Kernel Wizard (1.0)

Documentation IP Location Switch to Defaults

Streaming interfaces

Number of AXI4-Stream interfaces 2

Stream settings

Name	Mode	Width (bytes)
axis00	Master	64
axis01	Slave	64

< Back Next > Page 5 of 6

OK Cancel

- [Number of AXI4-Stream interfaces]: カーネルに存在する AXI4-Stream インターフェイスの数を指定します。最大 32 のインターフェイスをカーネルごとにイネーブルにできます。ザイリンクスでは、インターフェイス数をできるだけ少なくして、使用されるエリア量を削減することをお勧めしています。

[Stream Settings]


- [Name]: インターフェイスの名前を指定します。互換性を最大限にするため、引数名はカーネル名と同じ識別規則に従います。

- [Mode]: インターフェイスの方向を指定します。読み出し専用インターフェイスは AXI4-Stream スレーブ インターフェイスで、データを clWriteStream API を使用して送信できます。書き込み専用インターフェイスは、AXI4-Stream マスターインターフェイスで、ホストは clReadStream API を使用してデータをインターフェイスから受信できます。
- [Width (bytes)]: AXI4-Stream インターフェイスの TDATA 幅 (バイト) を指定します。このインターフェイス幅は、2 のべき乗で 1 ~ 64 バイトに制限されます。

[Summary] ページ

このページには、前のページで選択したオプションから作成される VLNV、ソフトウェア関数プロトタイプ、ハードウェア制御レジスタのサマリが表示されます。関数プロトタイプは、カーネル呼び出しが C 関数であるとしたらどのようなのかを示します。カーネル呼び出し用にカーネル引数を設定する方法の生成されたホスト コード例を参照してください。レジスタ マップは、ホストソフトウェア ID、引数名、ハードウェアレジスタ オフセット、タイプ、および関連インターフェイスの関係を示します。カーネルを生成する前に、このページで設定が正しいことを確認してください。

図 55: RTL Kernel ウィザードの [Summary] ページ

RTL Kernel Wizard (1.0)


Documentation
IP Location
Switch to Defaults

Summary

VLNV: mycompany.com:kernel:sd_x_kernel_wizard_0:1.0

Target platform: Unknown

Function prototype: `void sd_x_kernel_wizard_0(const uint scalar00, const uint scalar01, const uint scalar02, const uint scalar03, global void *axi00_ptr0, global void *axi00_ptr1, global void *axi01_ptr0, hls::stream<qdma_axis<512,0,0,0>> &axis00, hls::stream<qdma_axis<512,0,0,0>> &axis01);`

Register map:

ID	Name	Offset	Type	Interface
N/A	Control	0x000	N/A	S_AXI_CONTROL
0	scalar00	0x010	uint	S_AXI_CONTROL
1	scalar01	0x018	uint	S_AXI_CONTROL
2	scalar02	0x020	uint	S_AXI_CONTROL
3	scalar03	0x028	uint	S_AXI_CONTROL
4	axi00_ptr0	0x030	generic pointer	m00_axi
5	axi00_ptr1	0x038	generic pointer	m00_axi
6	axi01_ptr0	0x040	generic pointer	m01_axi
7	axis00	0x048	host stream (write_only)	axis00
8	axis01	0x050	host stream (read_only)	axis01

Notes: Please see example host code generated in ".sd_x_imports/src/host_example.cpp" for methods used to set the kernel arguments and execute the kernel. The global memory pointers are generic and the kernel should be tuned to handle the data type of the vectors.

< Back
Next >
Page 6 of 6

OK
Cancel

RTL ウィザードで設定したカーネルの最終確認および生成

RTL カーネル ウィザードを SDx から起動した場合は、[OK] をクリックして、サンプル Vivado プロジェクトを開きます。

Vivado から RTL カーネル ウィザードを起動した場合は、[OK] をクリックして次を実行します。

1. [Generate Output Products] ダイアログ ボックスで [Synthesis Option] に [Global] を選択し、[Generate] をクリックしてから、[OK] をクリックします。
2. Vivado の [Sources] ウィンドウで .xci ファイルを右クリックし、[Open IP Example Design] をクリックします。
3. [Open IP Example Design] ダイアログ ボックスで、出力ディレクトリ (またはデフォルトのディレクトリ) を選択し、[OK] をクリックします。サンプル デザインを含む新しい Vivado プロジェクトが開きます。
4. これで、RTL カーネル ウィザードを起動した現在の Vivado プロジェクトを閉じることができます。

割り込み

RTL Kernel ウィザードは、デフォルトで `interrupt` という名前の 1 つの割り込みポートを作成し、制御レジスタ ブロックに割り込みロジックを作成します。これは生成される Verilog コードと関連する `component.xml` および `kernel.xml` ファイルにも記述されます。

割り込みはアクティブ High で、グローバル割り込みイネーブル (GIE) および割り込みイネーブル (IER) レジスタの両方を設定します。デフォルトでは、IER が内部信号 `ap_done` および `ap_ready` を使用して割り込みをトリガーします。

書き込みコマンドにより ISR レジスタの定義されているすべてのビットが 0 になると、割り込みはクリアされます。

RTL Kernel ウィザード Vivado プロジェクト

RTL Kernel ウィザードを使用すると、I/O、制御レジスタ、AXI4 インターフェイスを指定して、RTL カーネルの仕様をカスタマイズできます。その後、カーネルの内容をカスタマイズし、その内容をザイリンクス オブジェクト (xo) ファイルにパッケージします。RTL Kernel ウィザードの設定 GUI を完了すると、Vivado カーネル プロジェクトが生成され、RTL カーネルの作成に必要なファイルが生成されて情報が自動挿入されます。

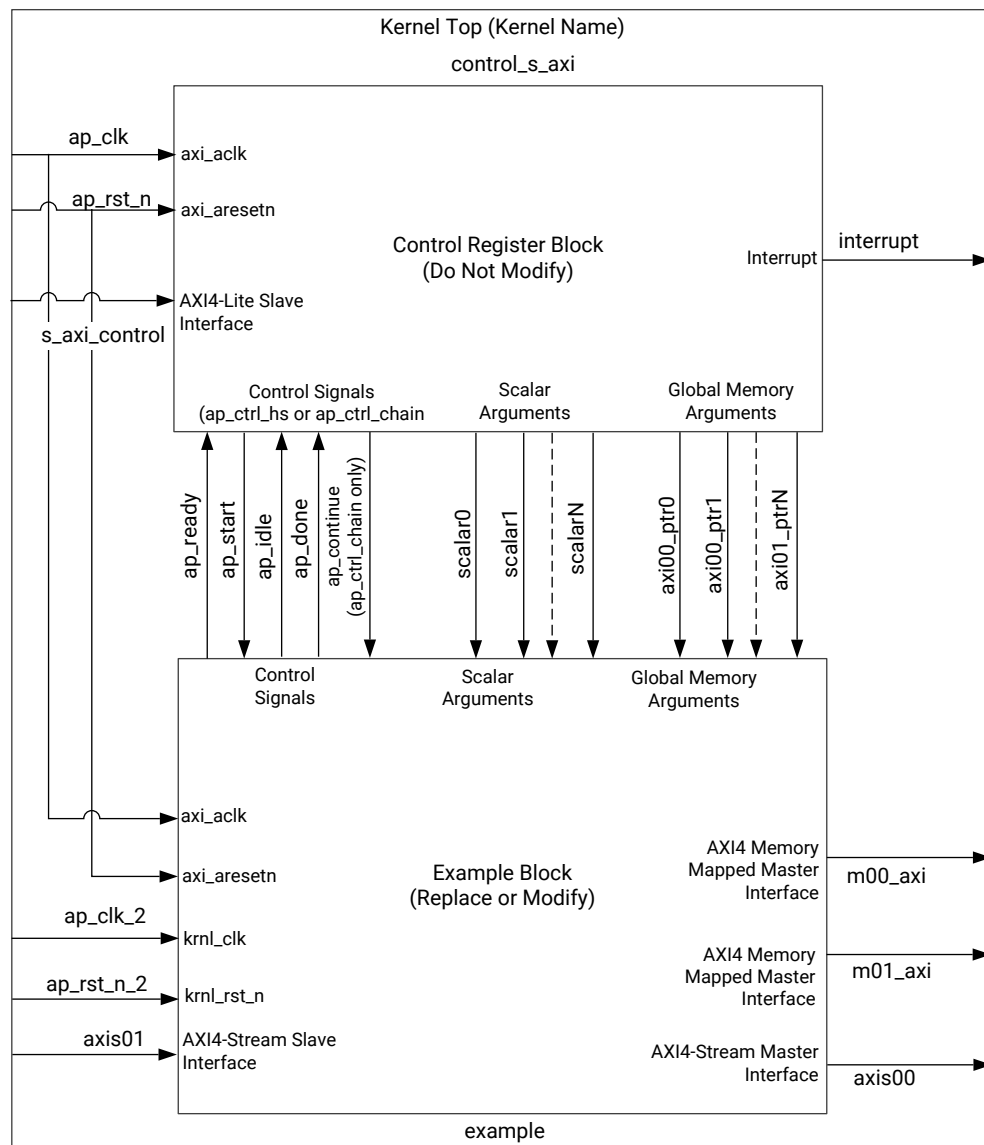
最上位 Verilog ファイルには、必要な入力/出力信号およびパラメーターが含まれています。最上位ポートはカーネル仕様ファイル (`kernel.xml`) と一致しており、RTL/ブロック デザインの残りの部分と統合にすると、アクセラレーション カーネルになります。最上位ファイルで定義されている AXI4 インターフェイスには、効率的でスループットの大きいインターフェイスの生成に必要な AXI4 信号の最低限のセットが含まれています。省略されている信号には、AXI システムのほかの部分に接続したときに、最適化されたデフォルト値が使用されます。これらの最適化されたデフォルト値を使用すると、システムで不要な AXI 機能を省略できるので、エリアが節約され、複雑性も軽減されます。ポート リストにない AXI 信号を含む既存コードで作業を始める場合は、これらの信号を最上位ポートに追加でき、IP パッケージャーでこれらの信号が正しく使用されます。

[Kernel Type] で選択したカーネル タイプによって、最上位ファイルの内容は Verilog サンプルおよび制御レジスタ、またはインスタンス化された IP インテグレーターのブロック デザインが自動挿入されます。

RTL カーネル タイプのプロジェクト フロー

RTL カーネル タイプは、制御レジスタおよび Vadd サブモジュールのサンプル デザインで構成される最上位 Verilog デザインを配布します。次の図に示す Vadd サブモジュールは、単純な加算器関数、AXI4 読み出しマスター、2つの AXI4-Stream インターフェイス、および AXI4 書き込みマスターで構成されています。定義されている AXI4 インターフェイスそれぞれに、個別のサンプル加算器コードがあります。各インターフェイスに最初に関連付けられている引数は、サンプルのデータ ポインターとして使用されます。各サンプルは 16 KB のデータを読み出し、32 ビットの「1を足す」演算を実行し、16 KB のデータを元の位置に書き込みます (読み出しアドレスと書き込みアドレスは同じ)。制御レジスタ モジュールを変更する場合は、このモジュールと Vivado カーネル プロジェクトの imports ディレクトリにある `kernel.xml` ファイルとの一致が保持されるようにしてください。サンプル サブモジュールは、カスタム ロジックに置き換えるか、デザインの開始点として使用できます。

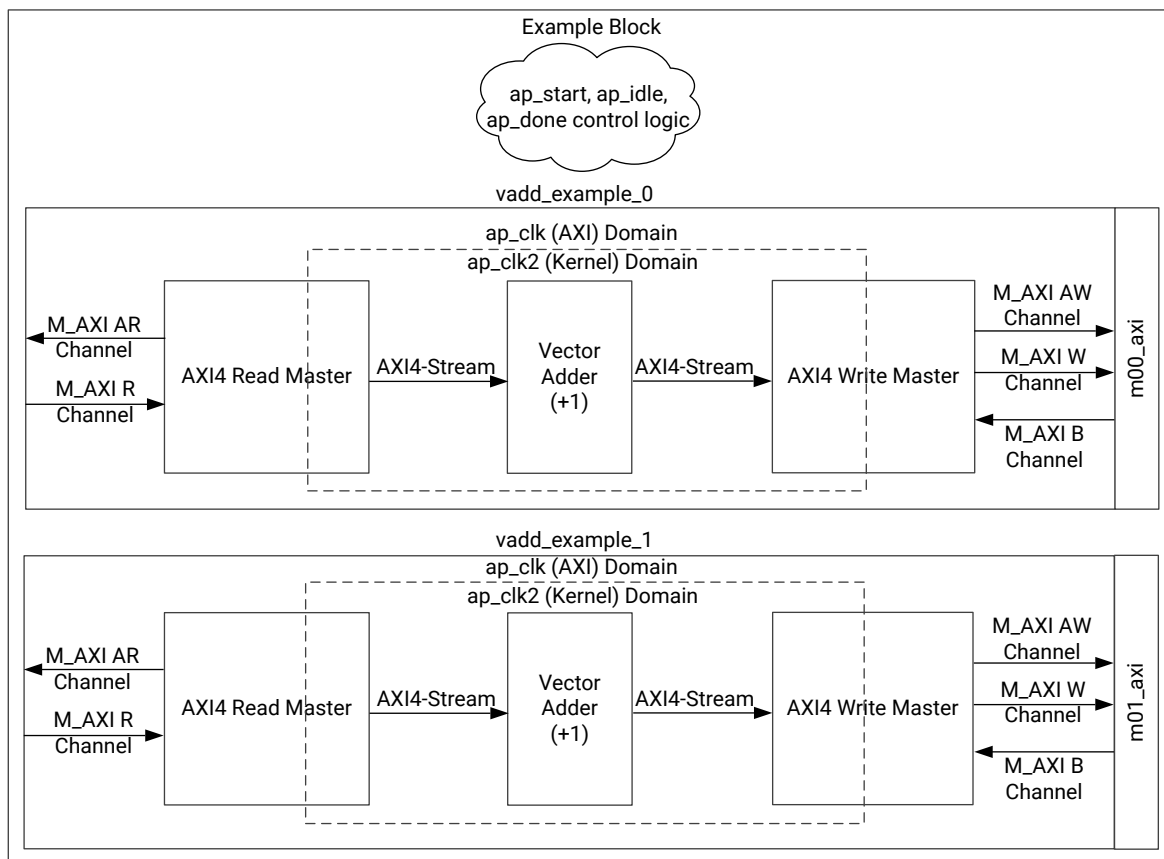
図 56: カーネル タイプ RTL の最上位



X22079-051019

次の図に示す Vadd サブモジュールは、単純な加算器関数、AXI4 読み出しマスター、および AXI4 書き込みマスターで構成されています。定義されている各 AXI4 インターフェイスに、個別のサンプル加算器コードがあります。各インターフェイスに最初に関連付けられている引数は、サンプルのデータポインターとして使用されます。各サンプルは 16 KB のデータを読み出し、32 ビットの「1 を足す」演算を実行し、16 KB のデータを元の位置に書き込みます (読み出しアドレスと書き込みアドレスは同じ)。

図 57: カーネル タイプ RTL の例



X22080-011019

次の表に、カーネルの Vivado プロジェクトのルートにある重要なファイルを説明します。<kernel_name> はウィザードで選択されたカーネルの名前です。

表 17: RTL カーネル ウィザードのソース ファイルとテストベンチ ファイル

ファイル名	説明	配布されるカーネルタイプ
<kernel_name>_ex.xpr	Vivado プロジェクト ファイル	すべて
imports ディレクトリ		
<kernel_name>.v	カーネル最上位モジュール	すべて
<kernel_name>_control_s_axi.v	RTL 制御レジスタ モジュール	RTL
<kernel_name>_example.sv	RTL サンプル ブロック	RTL
<kernel_name>_example_vadd.sv	RTL サンプル AXI4 ベクター加算ブロック	RTL
<kernel_name>_example_axi_read_master.sv	RTL サンプル AXI4 読み出しマスター	RTL

表 17: RTL カーネル ウィザードのソース ファイルとテストベンチ ファイル (続き)

ファイル名	説明	配布されるカーネル タイプ
<kernel_name>_example_axi_write_master.sv	RTL サンプル AXI4 書き込みマスター	RTL
<kernel_name>_example_adder.sv	RTL サンプル AXI4-Stream 加算ブロック	RTL
<kernel_name>_example_counter.sv	RTL サンプル カウンター	RTL
<kernel_name>_exdes_tb_basic.sv	シミュレーション テストベンチ	すべて
<kernel_name>_cmodel.cpp	ソフトウェア エミュレーション用のソフトウ ェア C モデル サンプル。	すべて
<kernel_name>_ooc.xdc	アウト オブ コンテキスト ザイリンクス制約 ファイル	すべて
<kernel_name>_user.xdc	カーネル ユーザー制約用のザイリンクス制約 ファイル。	すべて
kernel.xml	カーネル記述ファイル	すべて
package_kernel.tcl	カーネル パッケージ スクリプトのプロシージ ャ定義	すべて
post_synth_impl.tcl	Tcl インプリメンテーション後ファイル	すべて
sdx_imports ディレクトリ		
src/host_example.cpp	ホスト コード例	すべて
makefile	makefile サンプル	すべて
<kernel_name>_ex.sdk/<kernel_name>_control/src ディレクトリ		
kernel_control.h	MicroBlaze C ヘッダー ファイル	ブロック デザイン
kernel_control.c	MicroBlaze C ファイル	ブロック デザイン
<kernel_name>_ex.sdk/<kernel_name>_control/Debug ディレクトリ		
<kernel_name>_control.elf	MicroBlaze ELF ファイル	ブロック デザイン
<kernel_name>_ex.src/sources_1/<kernel_name>_bd ディレクトリ		
<kernel_name>_bd.bd	Vivado ブロック図ファイル	ブロック デザイン

ブロック デザインのカーネル タイプのプロジェクト フロー

ブロック デザイン カーネル タイプは、カーネルの基本として IP インテグレーター ブロック デザイン (BD) を配布します。制御レジスタのサンプリングおよびカーネルのフローの制御には、MicroBlaze プロセッサ サブシステムが使用されます。MicroBlaze プロセッサ システムでは、レジスタ ファイルの代わりに、ホストとカーネル間の交換メモリにブロック RAM が使用されます。

カーネル実行の制御方法を示すサンプルとして、各 AXI インターフェイスに対して DMA および数学演算のサブブロックが作成されます。このサンプルでは、AXI DataMover IP の制御に MicroBlaze AXI4-Stream インターフェイスが使用され、RTL カーネル タイプのものと同じサンプルが作成されます。また、MicroBlaze コアの ELF ファイルをコンパイルおよびリンクするための SDK プロジェクトも含まれています。この ELF ファイルは Vivado カーネル プロジェクトに読み込まれ、MicroBlaze 命令メモリに直接初期化されます。MicroBlaze プロセッサ プログラムを変更するには、次の手順に従います。

1. デザインがアップデートされている場合は、ハードウェアをエクスポートする必要がある場合があります。このオプションにアクセスするには、[File] → [Export] → [Export Hardware] をクリックします。[Export Hardware] ダイアログ ボックスが表示されたら、[OK] をクリックします。
2. これで、ソフトウェア開発キット (SDK) アプリケーションを実行できるようになります。Vivado メニューから [File] → [Launch] → [SDK] をクリックします。

3. ザイリンクス SDK GUI が表示されたら、[Welcome] タブの右側にある [X] ボタンをクリックしてこのダイアログボックスを閉じます。既に読み込まれている SDK プロジェクトが表示されます。
4. [Project Explorer] ビューの <Kernel Name>_control/src セクションの下にソース ファイルがあります。必要に応じてこれらのファイルを変更します。
5. アップデートが完了したら、[Project] → [Build All] → [Check for errors/warnings and resolve if necessary] をクリックしてソースをコンパイルします。ELF ファイルは GUI で自動的にアップデートされます。
6. 必要であれば、アップデートされたプログラムをテストおよびデバッグするため、シミュレーションを実行します。

シミュレーション テストベンチ

SystemVerilog シミュレーション テストベンチを生成すると、カーネルを実行して動作が正しいことを確認できます。「1 を足す」演算を検証するためのチェッカー機能も自動的に含まれます。この生成されたテストベンチは、カーネル機能を検証する開始点として使用できます。制御レジスタの書き込み/読み出し操作を実行し、カーネルを複数回実行して、簡単なリセットテストも実行します。また、AXI の問題、リセット問題、複数回繰り返したときのバグ、カーネル機能のデバッグにも便利です。ハードウェア エミュレーションと比較すると、ハードウェアのコーナー ケースをより厳しくテストしますが、ホスト コードとカーネルのやり取りはテストしません。

シミュレーションを実行するには Vivado GUI の左側で Flow Navigator → [Run Simulation] をクリックして、[Run Behavioral Simulation] をクリックします。ビヘイビア シミュレーションが予測どおりに機能している場合は、合成後の論理シミュレーションを実行し、合成がビヘイビア モデルと一致していることを確認できます。

アウト オブ コンテキスト合成

Vivado カーネル プロジェクトは、アウト オブ コンテキスト (OOC) モードで合成およびインプリメンテーションを実行するよう設定されています。そのため、デザインのザイリンクス デザイン制約 (XDC) ファイルで合成のデフォルトクロック周波数が指定されています。カーネルがエラーなしに合成されるかどうかを判断するため、合成を実行すると有益です。合成を実行すると、リソース使用率および周波数の見積もりも取得できます。カーネルをパッケージするには、合成までを実行する必要があります。

合成できない場合は、リンク中にエラーが発生し、デバッグが困難になります。合成済み出力は、カーネルを RTL ではなくネットリストとしてパッケージするときに使用できます。カーネル内でブロック デザインが使用されている場合は、カーネルをネットリストとしてパッケージする必要があります。OOC 合成を実行するには、Vivado Flow Navigator → [Synthesis] から [Run Synthesis] をクリックします。

ソフトウェア モデルおよびホスト コード例

「1 を足す」演算の C++ ソフトウェア モデルが imports ディレクトリに含まれています。カーネルと同じ名前で、ファイル拡張子は `cpp` です。このソフトウェア モデルは、カーネルの機能をモデル化するために変更でき、パッケージ段階でカーネルに含めることができます。SDx を使用すると、ソフトウェア エミュレーションをカーネルを使用して実行できます。ハードウェア エミュレーションおよびシステム リンカーでは、常にカーネルのハードウェア記述が使用されます。

`sdx_imports` ディレクトリに、`main.c` という名前のサンプルの C ホスト コードが含まれています。ホスト コードには、プログラムへの引数としてバイナリ コンテナを指定します。ホスト コードを SDx GUI に読み込んだ後に [Run Configuration] → [Arguments] をクリックして [Automatically add binary container(s) to arguments] チェックボックスをオンにすると、この引数が自動的に指定されます。これにより、ホスト コードで `init` 関数の一部としてバイナリが読み込まれます。ホスト コードはカーネルのインスタンス化、バッファの割り当て、カーネルの引数設定、カーネルの実行、「1 を足す」の結果値の収集およびその確認を実行します。

RTL カーネルのパッケージ

Vivado でカーネルを設計しテストしたら、RTL カーネルを生成する最終手順に進みます。この手順では、SDx で使用するため Vivado カーネル プロジェクトをパッケージします。

このプロセスを開始するには、Vivado Flow Navigator → [Project Manager] から [Generate RTL Kernel] をクリックします。3つのパッケージ オプションを含むダイアログ ボックスが表示されます。

- ソースのみのカーネル パッケージを選択すると、RTL デザイン ソースを直接使用してカーネルがパッケージされます。
- 合成済みのカーネル パッケージを選択すると、RTL デザイン ソースと合成済みのキャッシュされた出力を含めてカーネルがパッケージされます。合成済みのキャッシュされた出力を使用すると、フローの後の方で再合成する必要はありません。ターゲット プラットフォームを変更すると、パッケージされたカーネルで合成済みのキャッシュされた出力が使用されなくなり、RTL デザイン ソースだけが使用されるようになります。
- ネットリスト (デザイン チェックポイント (DCP)) ベースのカーネル パッケージを選択すると、カーネルの合成済み出力により生成されたネットリストを使用して、カーネルがブラック ボックスとしてパッケージされます。必要であれば、この出力は暗号化可能です。ターゲットのプラットフォームを変更した場合、カーネルが新しいデバイスにターゲットを変更できなくなる可能性があるため、ソースから生成し直す必要があります。デザインにブロック デザインが含まれている場合は、ネットリスト (DCP) ベースのカーネル パッケージ オプションのみが選択可能です。

また、すべてのカーネル パッケージング タイプで、ソフトウェア エミュレーションで使用可能なソフトウェア モデルを使用してパッケージするオプションもあります。ソフトウェア モデルに複数のファイルが含まれている場合は、ソース ファイル リストのファイル名をスペースで区切るか、GUI を使用して、Ctrl キーを押しながら複数のファイルを選択します。

[OK] をクリックすると、カーネル出力ファイルが生成されます。あらかじめ合成済みのカーネルまたはネットリストのカーネル オプションを選択している場合は、合成を実行できます。合成が前に実行されている場合は、古くてもそのときの出力が使用されます。カーネル ザイリンクス オブジェクト (.xo) ファイルは、Vivado カーネル プロジェクトの `sdx_imports` ディレクトリに生成されます。

この時点で、Vivado カーネル プロジェクトを閉じることができます。Vivado カーネル プロジェクトを SDx GUI から起動した場合、`main.c` という名前のサンプル ホスト コードおよびカーネル ザイリンクス オブジェクト (.xo) ファイルが自動的に SDx ソース フォルダーにインポートされます。

ウィザードで生成した既存 RTL カーネルの変更

SDx GUI で既存の生成済みカーネルを変更することは可能です。カーネルが生成された後、ザイリンクス RTL Kernel ウィザードのメニュー オプションを使用すると、既存カーネルを変更するためのオプションを含むダイアログ ボックスが開きます。[Edit Existing Kernel Contents] を選択すると、Vivado プロジェクトが開き、カーネルを変更して生成し直すことができます。[Re-customize Existing Kernel Interfaces] を選択すると、RTL カーネル ウィザードの設定ダイアログ ボックスが開きます。[Kernel Name] 以外のオプションは変更可能で、前の Vivado プロジェクトが置き換えられます。



重要: アップデートした Vivado カーネル プロジェクトを生成すると、前のプロジェクトのファイルおよび変更はすべて失われます。

RTL カーネルの手動開発フロー

RTL カーネルの作成には RTL Kernel ウィザードの使用を強くお勧めしますが、RTL カーネルはこのウィザードを使用しなくても作成できます。このセクションでは、手動開発フローの手順を詳しく説明します。SDAccel アプリケーション用の RTL カーネルとして RTL デザインをパッケージするには、次の3つの手順があります。

1. RTL ブロックを Vivado IP としてパッケージします。
2. カーネル記述 XML ファイルを作成します。
3. RTL カーネルをザイリンクス オブジェクト (.xo) ファイルにパッケージします。

これらの手順は RTL Kernel ウィザードで自動化されています。完全にパッケージされた RTL カーネルは .xo ファイルとして配布され、.xo という拡張子が付きます。このファイルは、Vivado IP オブジェクト (ソース ファイルを含む) および関連付けられたカーネル XML ファイルを含むコンテナです。.xo ファイルをプラットフォームにコンパイルし、ハードウェアまたはハードウェア エミュレーション フローで実行できます。

Vivado IP として RTL ブロックをパッケージ

RTL カーネルを IP インテグレーターで使用するには、Vivado IP としてパッケージする必要があります。Vivado での IP パッケージの詳細は、『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) を参照してください。

RTL カーネルには、次のインターフェイスのパッケージが必要です。

- AXI4-Lite インターフェイスは、S_AXI_CONTROL という名前でパッケージする必要がありますが、その AXI ポートには異なる名前を指定できます。
- AXI4 インターフェイスは、64 ビット アドレス サポートの AXI4 マスター エンドポイントとしてパッケージする必要があります。



推奨: ザイリンクスでは、AXI4 インターフェイスは AXI メタデータ HAS_BURST=0 および SUPPORTS_NARROW_BURST=0 を含めてパッケージすることをお勧めします。これらのプロパティは、IP レベルの bd.tcl ファイルで設定できます。これは、WRAP および FIXED バースト タイプは使用されず、サイズの満たない (サブサイズ) バーストも使用されないということです。

- ap_clk および ap_clk_2 は、クロック インターフェイスとしてパッケージする必要があります。
- ap_rst_n および ap_rst_n_2 は、アクティブ Low のリセット インターフェイスとしてパッケージする必要があります。
- ap_clk は、AXI4-Lite、AXI4、および AXI4-Stream インターフェイスすべてと関連付けてパッケージする必要があります。

RTL カーネルが IP インテグレーター用に正しくパッケージされていることをテストするには、パッケージ済みのカーネルを IP インテグレーターにインスタンス化してみます。GUI では、クロック、リセット、AXI4-Lite スレーブ、AXI4 マスター、AXI4 スレーブのみのインターフェイスがあるように表示されるはずですが、また、キャンバスにほかのポートは表示されません。AXI インターフェイスのプロパティは、キャンバスでインターフェイスを選択すると表示されます。[Block Interface Properties] ウィンドウで [Properties] タブをクリックし、CONFIG という表エントリを展開表示します。インターフェイスが読み出し専用または書き込み専用の場合は、未使用の AXI チャンネルは削除され、READ_WRITE_MODE が読み出し専用または書き込み専用で設定されます。



重要: RTL カーネルの制約がクロックなどのスタティック エリアにある制約を参照している場合、RTL カーネルの制約 ファイルの処理順序を late に設定して、RTL カーネル制約が正しく適用されるようにする必要があります。

処理順序を late に設定するには、次の2つの方法があります。

1. 制約が .tcl ファイルで指定されている場合は、次のように <: setFileProcessingOrder "late" :> をファイルの .tcl 前文セクションに追加します。

```
<: set ComponentName [getComponentNameString] :>
<: setOutputDirectory "/" :>
<: setFileName $ComponentName :>
<: setFileExtension ".xdc" :>
<: setFileProcessingOrder "late" :>
```

2. 制約が .xdc ファイルで指定されている場合は、component.xml の <spirit:define> の下に次の4行を追加します。component.xml のこの4行は .xdc ファイルの呼び出されるエリアの次に記述する必要があります。次の例の場合、my_ip_constraint.xdc ファイルが呼び出され、その後に定義した late 処理順序が指定されています。

```
<spirit:file>
  <spirit:name>tcl/my_ip_constraint.xdc</spirit:name>
  <spirit:userFileType>tcl</spirit:userFileType>
  <spirit:userFileType>USED_IN_implementation</spirit:userFileType>
  <spirit:userFileType>USED_IN_synthesis</spirit:userFileType>
  <spirit:define>
    <spirit:name>processing_order</spirit:name>
    <spirit:value>late</spirit:value>
  </spirit:define>
</spirit:file>
```

カーネル記述 XML ファイルの作成

カーネル記述 XML ファイルは、SDAccel 環境で使用できるように、RTL カーネルごとに作成する必要があります。ファイル名は kernel.xml にする必要があります。この XML ファイルはレジスタ マップおよびポートのようなカーネル属性を指定します。このファイルは、ランタイムおよび SDAccel フローで必要です。次は kernel.xml ファイルの例です。

```
<?xml version="1.0" encoding="UTF-8"?>
<root versionMajor="1" versionMinor="6">
  <kernel name="sdx_kernel_wizard_0" language="ip_c"
    vlnv="mycompany.com:kernel:sdx_kernel_wizard_0:1.0" attributes=""
    preferredWorkGroupSizeMultiple="0" workGroupSize="1" interrupt="true">
    <ports>
      <port name="s_axi_control" mode="slave" range="0x1000" dataWidth="32"
        portType="addressable" base="0x0"/>
      <port name="m00_axi" mode="master" range="0xFFFFFFFFFFFFFFFF"
        dataWidth="512" portType="addressable" base="0x0"/>
    </ports>
    <args>
      <arg name="axi00_ptr0" addressQualifier="1" id="0" port="m00_axi"
        size="0x8" offset="0x010" type="int*" hostOffset="0x0" hostSize="0x8"/>
    </args>
  </kernel>
</root>
```

次の表で、カーネル XML ファイルを詳しく説明します。

表 18: カーネル XML ファイル

タグ	属性	説明
<root>	versionMajor	現在のリリースの SDAccel では 1 に設定します。
	versionMinor	現在のリリースの SDAccel では 6 に設定します。

表 18: カーネル XML ファイル (続き)

タグ	属性	説明
<kernel>	name	[Kernel name]
	language	RTL カーネルの場合は常に ip_c に設定します。
	vlnv	IP の component.xml にあるベンダー、ライブラリ、名前、バージョン属性と一致している必要があります。たとえば、component.xml には次のタグがあります。 <pre><spirit:vendor>xilinx.com</spirit:vendor> <spirit:library>hls</spirit:library> <spirit:name>test_sincos</spirit:name> <spirit:version>1.0</spirit:version></pre> カーネル XML の vlnv 属性は次のように設定する必要があります。 xilinx.com:hls:test_sincos:1.0
	attributes	予約されています。空の文字列に設定します。
	preferredWorkGroupSizeMultiple	予約されています。0 に設定します。
	workGroupSize	予約されています。1 に設定します。
	interrupt	割り込みがアル場合は true (interrupt="true") に設定し、それ以外の場合は削除します。
<port>	name	ポート名。少なくとも AXI4 マスター ポートが 1 つ、AXI4-Lite スレーブ ポートが 1 つ必要です。カーネル間でデータをストリームする場合は、AXI4-Stream ポートを指定できます。AXI4-Lite インターフェイス名は S_AXI_CONTROL にする必要があります。
	mode	<ul style="list-style-type: none"> AXI4 マスター ポートの場合は master に設定します。 AXI4 スレーブ ポートの場合は slave に設定します。 AXI4-Stream マスター ポートの場合は write_only に設定します。 AXI4-Stream スレーブ ポートの場合は read_only に設定します。
	range	ポートのアドレス空間の範囲です。
	dataWidth	ポートを通過するデータ幅で、デフォルト値は 32 ビットです。
	portType	ポートがアドレス指定可能か、またはストリーミングかを指定します。 <ul style="list-style-type: none"> AXI4 マスターおよびスレーブ ポートの場合は addressable に設定します。 AXI4-Stream ポートの場合は stream に設定します。
	base	AXI4 マスターおよびスレーブ ポートの場合は 0x0 に設定します。このタグは AXI4-Stream ポートには適用されません。

表 18: カーネル XML ファイル (続き)

タグ	属性	説明
<args>	name	カーネルの引数名。
	addressQualifier	有効値: 0: スカラー カーネル入力引数 1: グローバル メモリ 2: ローカル メモリ 3: 定数メモリ 4: パイプ
	id	AXI4 マスターおよびスレーブ ポートのみに適用されます。ID は連続している必要があります。これはカーネル引数の順番を決めるのに使用されます。 AXI4-Stream ポートには適用されません。
	port	arg が接続されているポートを示します。
	size	引数のサイズです。デフォルトは 4 バイトです。
	offset	レジスタ メモリ アドレスを示します。
	type	引数の C データ型です。たとえば、int*, float* のようにします。
	hostOffset	予約されています。0x0 に設定します。
	hostSize	引数のサイズです。デフォルトは 4 バイトです。
	memSize	AXI4 マスターおよびスレーブ ポートには適用されません。 AXI4-Stream ポートの場合、作成される FIFO の深さは memSize で設定されます。
次のタグは、AXI4-Stream ポートの追加情報を指定します。これらは AXI4 マスターまたはスレーブ ポートには適用されません。		
<pipe>	計算ユニットの各パイプに、データバッファリング用の FIFO がコンパイラにより挿入されます。pipe タグは FIFO のコンフィギュレーションを決めます。	
	name	AXI4-Stream ポート用に挿入された FIFO の名前を指定します。同じ計算ユニットで使用されているすべてのパイプの中で、この名前が重複しないように指定する必要があります。
	width	FIFO の幅をバイトで指定します。たとえば、32 ビットの FIFO では 0x4 を指定します。
	depth	FIFO の深さをワード数で指定します。
	linkage	常に internal に設定します。
<connection>	connection タグは、カーネルからパイプ用に挿入された FIFO まで、または FIFO からカーネルまでのハードウェアでの実際の接続を指定します。	
	srcInst	接続のソース インスタンスを指定します。
	srcPort	接続のソース インスタンスのポートを指定します。
	dstInst	接続のデスティネーション インスタンスを指定します。
	dstPort	接続のデスティネーション インスタンスのポートを指定します。

RTL カーネルをザイリンクス オブジェクト ファイルにパッケージ

最終段階では、SDAccel コンパイラでできるように、RTL IP およびその関連カーネルの XML ファイルと一緒にザイリンクス オブジェクト ファイル (.xo) にパッケージします。次のコマンドラインの例では、test_sincos RTL IP および kernel.xml を test.xo というオブジェクト ファイルにパッケージしています。

```
package_xo -xo_path test.xo -kernel_name test_sincos -kernel_xml  
kernel.xml -ip_directory ./ip/
```

package_xo 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) の「package_xo コマンド」を参照してください。package_xo コマンドの使用法の例は、ザイリンクス [GitHub](#) リポジトリを参照してください。

RTL 設計の推奨事項

RTL カーネル ウィザードで SDx フローで使用する RTL デザインのパッケージを実行できますが、RTL カーネルの設計時には『UltraFast 設計手法ガイド (Vivado Design Suite 用)』(UG949) の推奨事項に従う必要があります。

カーネルを設計する際は、インターフェイスおよびパッケージの要件に従うことに加え、パフォーマンス要件も念頭に置いておく必要があります。具体的には、次を考慮します。

- [AXI4 インターフェイスのメモリ パフォーマンスの最適化](#)
- [QoR \(結果の品質\) に関する注意事項](#)
- [デバッグおよび検証での注意事項](#)

これらのトピックについて、次のセクションで説明します。

AXI4 インターフェイスのメモリ パフォーマンスの最適化

AXI4 インターフェイスは通常、プラットフォームの DDR メモリ コントローラーに接続されます。



推奨: 周波数およびリソース使用率を最適にするには、メモリ コントローラーごとに 1 つのインターフェイスを使用することを推奨します。

メモリ コントローラーからベスト パフォーマンスを得るには、次の AXI インターフェイス動作を推奨します。

- ネイティブのメモリ コントローラーの AXI データ幅と一致する AXI データ幅を使用します。通常は 512 ビットです。
- WRAP、FIXED、またはサブサイズのバーストを使用しないでください。
- できる限り大型のバースト転送を使用してください(最高 4 KB までの AXI4 プロトコル)。
- ディアサートされた書き込みストローブは使用しないようにします。ディアサートされたストローブは、読み出し-変更-書き込み操作を実行するため、DDR メモリ コントローラーにエラー訂正コード (ECC) ロジックを作成します。
- パイプライン処理された AXI トランザクションを使用します。
- AXI インターフェイスが 1 つの DDR コントローラーにのみ接続されている場合は、スレッドの使用を避けます。

- カーネルにフルの書き込みトランザクション (ノンブロッキング書き込み要求) を転送する機能がない場合は、書き込みアドレス コマンドの生成を避けます。
- カーネルにバック プレッシュャーのない読み出しデータ (ノンブロッキング読み出し要求) すべてを受信する機能がない場合は、読み出しアドレス コマンドの生成を避けます。
- 読み出し専用または書き込み専用インターフェイスが必要な場合は、プロジェクトをカーネルにパッケージする前に、未使用チャンネルのポートを最上位 RTL ファイルでコメントアウトできます。
- 複数のスレッドを使用すると、カーネルとメモリ コントローラーの間のインフラストラクチャ IP により多くのリソースが必要になります。

RTL カーネルでのクロックの管理

RTL カーネルには、プライマリ クロック (`ap_clk`) とオプションのセカンダリ ブロック (`ap_clk_2`) の2つの外部クロック インターフェイスを含めることができます。どちらのクロックも内部ロジックのクロッキングに使用できますが、すべての外部 RTL カーネル インターフェイスはプライマリ クロックを使用する必要があります。プライマリ クロックとセカンダリ クロックの両方で、独立した自動周波数スケールリングがサポートされます。

RTL カーネル内でさらにクロックが必要な場合は、Clocking Wizard IP または MMCM/PLL プリミティブなどの周波数シンセサイザーを RTL カーネル内にインスタンス化できます。

このため、RTL カーネルはプライマリ クロックだけを使用するか、プライマリ クロックとセカンダリ クロックの両方を使用するか、内部周波数シンセサイザーを使用してプライマリ クロックおよびセカンダリ クロックを使用できます。次は、これら3つの RTL カーネルのクロッキング手法を使用した場合の長所と短所を示しています。

- 1つの入力クロック: `ap_clk`
 - 外部インターフェイスおよび内部カーネル ロジックは同じ周波数で実行されます。
 - クロック乗せ換え (CDC) 問題はありません。
 - `ap_clk` の周波数は、カーネルがタイミングを満たせるように自動的にスケールできます。
- 2つの入力クロック: `ap_clk` および `ap_clk_2`
 - カーネル ロジックはいずれかのクロック周波数で実行できます。
 - 周波数を移動するために適切な CDC 手法が必要です。
 - カーネルがタイミングを満たすため、`ap_clk` および `ap_clk_2` の両方が自動的に周波数を個別にスケールできます。
- カーネル内で周波数シンセサイザーを使用:
 - クロックを生成するためにデバイス リソースが追加が必要です。
 - `ap_clk` および `ap_clk_2` (オプション) インターフェイスが必要です。
 - 生成されたクロックの周波数は、CU ごとに別の周波数にできます。
 - カーネル ロジックは使用可能なクロック周波数のいずれかで実行できます。
 - 周波数を移動するために適切な CDC 手法が必要です。

RTL カーネルで周波数シンセサイザーを使用する場合は、次の制約に注意が必要です。

1. RTL 外部インターフェイスは `ap_clk` でクロック供給されます。
2. 周波数シンセサイザーには、RTL カーネルへの内部クロックとして使用する出力クロックを複数含めることができます。

3. Tcl スクリプトを提供して、Vivado 配置のクロック リソース配置 DRC をダウングレードして、Vivado DRC エラーが起こらないようにする必要があります。Tcl コマンドの例は、次のとおりです。

```
set_property CLOCK_DEDICATED_ROUTE ANY_CMT_COLUMN
[get_nets pfm_top_i/static_region/base_clocking/clkwiz_kernel/inst/
CLK_CORE_DRP_i/clk_inst/clk_out1
```

注記: この制約は、プラットフォームのシェル クロック構造に合わせて編集する必要があります。

4. `xocc --xp` オプションを使用して上記の Tcl スクリプトを指定し、最適化後に Vivado インプリメンテーションで使われるようにします。次に例を示します。

```
--xp vivado_prop:run.impl_1.STEPS.OPT_DESIGN.TCL.POST={<PATH>/<TCL
Script>}
```

5. カーネル (RTL または HLS ベース) で使用可能な 2 つのグローバル クロック入力周波数を指定します。`xocc --kernel_frequency` オプションを使用して、カーネル入力クロック周波数が想定どおりになるようにします。たとえば、1 つのクロックを使用する場合は、次のように指定します。

```
xocc --kernel_frequency 250
```

2 つのクロックの場合、複数の周波数をクロック ID に基づいて指定できます。プライマリ クロックの ID は 0 で、セカンダリ クロックの ID は 1 です。

```
xocc --kernel_frequency 0:250|1:500
```



ヒント: PLL または MMCM 出力クロックが RTL カーネルの演算前にロックされるようにします。RTL カーネルでロックされた信号を使用すると、クロックが正しく動作するようになります。

周波数シンセサイザーを RTL カーネルに追加した場合、生成したクロックが自動的にスケーラブルにはなりません。RTL がタイミング要件を満たしていないと、`xocc` で次のようなエラーが表示されます。

```
ERROR: [VPL-1] design did not meet timing - Design did not meet timing. One
or more unscalable system clocks did not meet their required target
frequency. Please try specifying a clock frequency lower than 300 MHz using
the '--kernel_frequency' switch for the next compilation. For all system
clocks, this design is using 0 nanoseconds as the threshold worst negative
slack (WNS) value. List of system clocks with timing failure.
```

この場合、内部クロック周波数を変更するか、タイミングを満たすようにカーネル ロジックを最適化する必要があります。

QoR (結果の品質) に関する注意事項

タイミングおよびエリアの結果を改善するには次の推奨事項に従ってください。

- すべてのリセット入力をパイプライン処理し、ファンアウトの大きなネットを避けるため、内部でリセットを分配します。
- 必要な制御ロジックのフリップフロップのみをリセットします。
- できる限り、入力および出力信号にレジスタを付けるようにしてください。
- 特に複数のカーネルがインスタンス化される場合は、確実にフィットするように、ターゲットプラットフォームのエリアに比例してカーネルのサイズを検討します。
- スタックドシリコンインターフェイス (SSI) テクノロジーがプラットフォームには使用されていることに注意してください。これらのデバイスには複数のダイがあり、ダイ間をまたぐロジックはフリップフロップからフリップフロップへのタイミングパスである必要があります。

デバッグおよび検証での注意事項

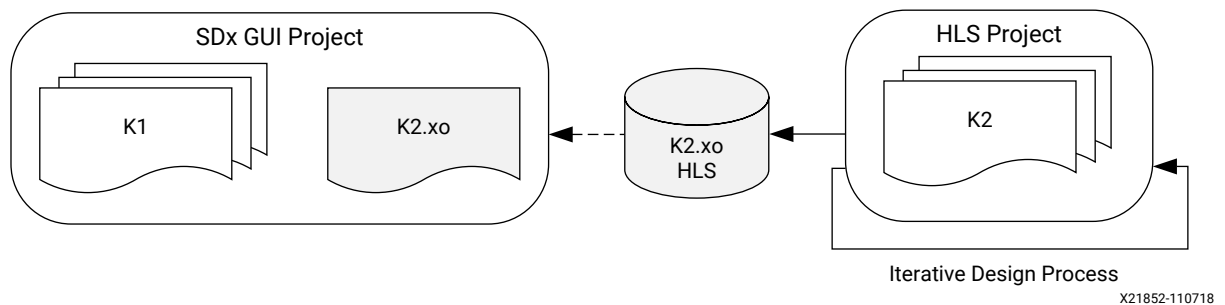
- RTL カーネルは、検証コンポーネント、ランダム化、プロトコルチェッカーなどのアドバンス検証手法を使用して、独自のテストベンチで検証する必要があります。Vivado IP カタログに含まれる AXI Verification IP (VIP) を使用すると、AXI インターフェイスを検証するのに役立ちます。RTL カーネルのサンプル デザインには、サンプル スティミュラス ファイルを含む AXI VIP ベースのテストベンチが含まれています。
- ハードウェア エミュレーション フローでは、ハードウェアで実際の AXI トラフィックで可能なプロトコル信号伝送条件の範囲を正確に表すことができないので、論理検証には使用しないでください。ハードウェア エミュレーションは、ホスト コードのソフトウェア インテグレーションをテストしたり、複数カーネル間の通信を確認したりするために使用します。

SDAccel への HLS カーネル デザインの統合

この文書のアプリケーション中心設計手法で説明する主なフローでは、アクセラレータ カーネルを開発して、トップダウン モデルでプロジェクトのホスト アプリケーションに統合します。つまり、すべてのソース コードが SDAccel™ に提供され、そのセクションがアクセラレータ モジュールに合成されるようになっています。このフローでは Vivado® 高次合成 (HLS) ツールが呼び出され、関数がハードウェアインプリメント可能なアクセラレータ コードに変換されます。

また、SDAccel にはボトムアップ フローも含まれており、Vivado HLS プロジェクトからインポートして HLS ベースのハードウェア カーネルを直接作成することもできます。これにより、最適化を実行して、Vivado HLS プロジェクト内でカーネル パフォーマンスを検証できるようになります。カーネルがパフォーマンスおよびリソース要件を満たすと、出力されたザイリンクス オブジェクト ファイル (.xo) がハンドオフされて、SDx™ プロジェクトに含まれるようになります。ハンドオフ中には、すべてのカーネル Vivado HLS 最適化が維持されます。

図 58: Vivado HLS のデザイン フロー



ボトムアップ フローを使用する利点は、次のとおりです。

- カーネルは完全な SDAccel プロジェクトに統合する前に、設計、検証、最適化できます。
- カーネル別に特定のカーネル最適化が維持されます。
- 独立した Vivado HLS およびプロジェクト ディレクトリを使用すると、アプリケーションとカーネルを分けることができます。
- VHDL プロジェクトは、ライブラリ インスタンス化のような複数の異なるプロジェクトで使用できます。
- チームで協力して生産性を向上できます。

Vivado HLS を使用した SDAccel カーネルの作成

Vivado HLS を実行して SDAccel 用に C/C++ からカーネルを生成する場合、通常の Vivado HLS フローに従って実行されますが、カーネルは SDAccel でアクセラレータとして動作するものなので、SDAccel カーネルの記述ガイドラインに従う必要があります (C/C++ 記述ガイドラインを参照)。何よりもまず、インターフェイスは AXI メモリ インターフェイスとして記述する必要があります。値で呼び出されるスカラー値の場合は例外で、AXI4-Lite インターフェイスにマップされます。これを次の例に示します。

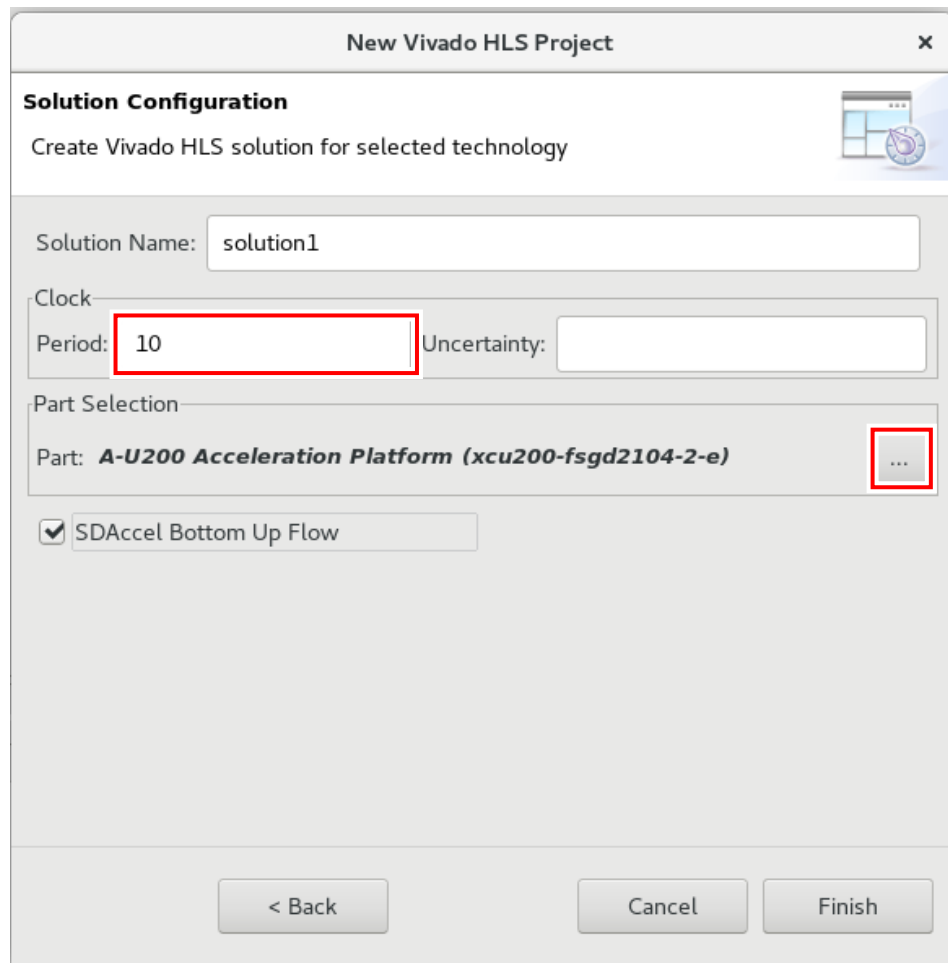
```
void krnl_idct(const ap_int<512> *block,
              const ap_uint<512> *q,
              ap_int<512> *voutp,
              int ignore_dc,
              unsigned int blocks) {
    #pragma HLS INTERFACE m_axi      port=block      offset=slave bundle=p0
    depth=512
    #pragma HLS INTERFACE s_axilite  port=block      bundle=control
    #pragma HLS INTERFACE m_axi      port=q          offset=slave bundle=p1
    depth=2
    #pragma HLS INTERFACE s_axilite  port=q          bundle=control
    #pragma HLS INTERFACE m_axi      port=voutp      offset=slave bundle=p2
    depth=512
    #pragma HLS INTERFACE s_axilite  port=voutp      bundle=control
    #pragma HLS INTERFACE s_axilite  port=ignore_dc  bundle=control
    #pragma HLS INTERFACE s_axilite  port=blocks     bundle=control
    #pragma HLS INTERFACE s_axilite  port=return     bundle=control
}
```



推奨: インターフェイスの ap-datatypes を使用するには、HLS のテストベンチで ap-datatypes を使用する必要があります。これにより、C/C++ シミュレーションの速度が遅くなることがあるので、ネイティブ C/C++ へのマップを考慮する必要があります。ほとんどのホスト コードはネイティブ データ型に基づいているので、それらをカーネル インターフェイスで使用することをお勧めします。

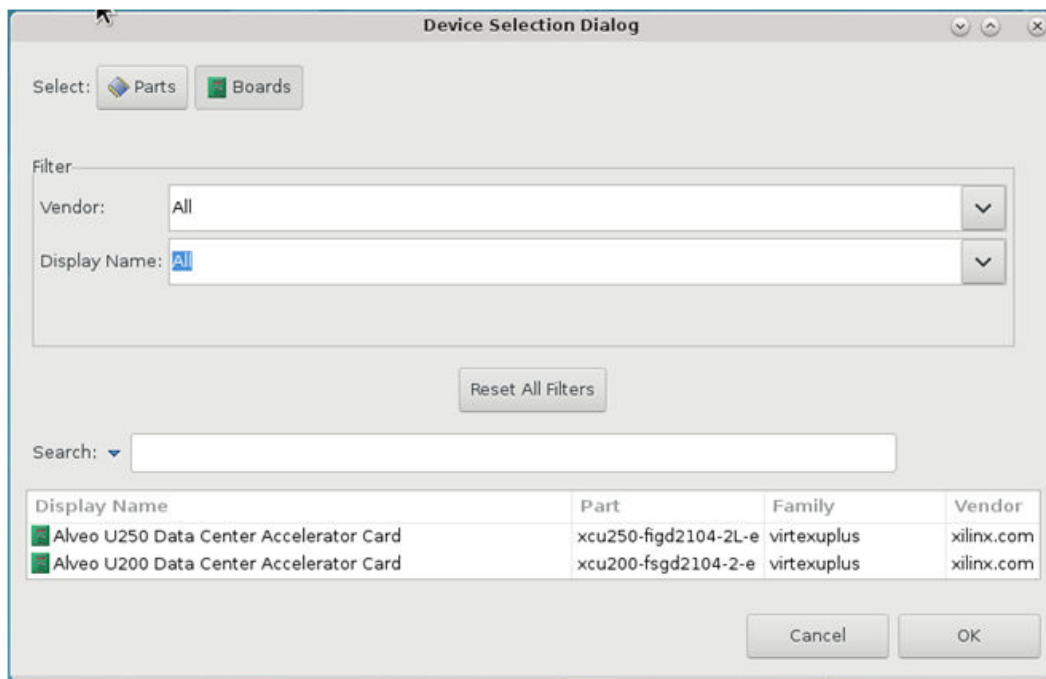
新規プロジェクトの作成については、『Vivado Design Suite ユーザー ガイド: 高位合成』(UG902)を参照してください。SDAccel カーネル プロジェクトでは、次の図に示すように、[SDAccel Bottom Up Flow] チェック ボックスをオンにして、[Clock Period] および [Part Selection] を指定します。

図 59: [New Vivado HLS Project] ダイアログ ボックス



[Browse] ボタンをクリックして [Device Selection Dialog] ダイアログ ボックスを開き、デバイスのリストからアクセラレータ ボードを選択してプラットフォームを選択します。

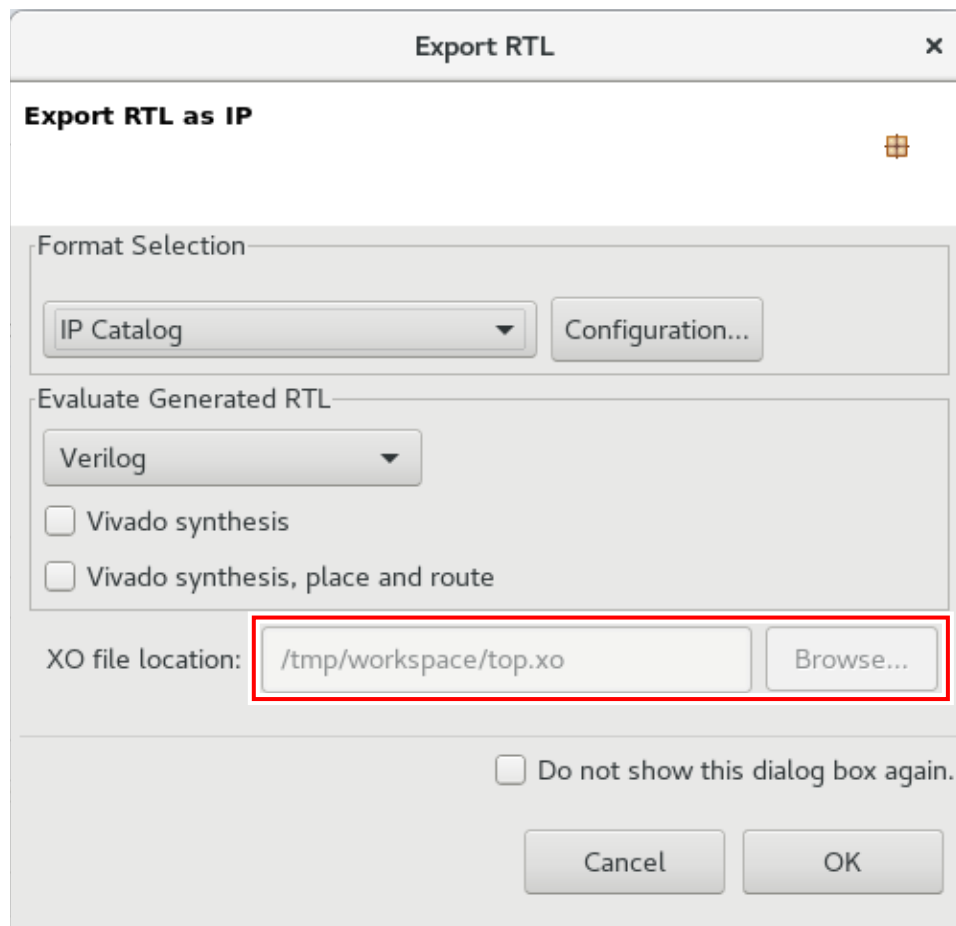
図 60: デバイスの選択



終了したら、最適化プロセスが再開され、最適なインプリメンテーションに到達するまで繰り返されます。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。

最適化したデザインの合成が終了したら、SDAccel ツール チェーンにエクスポートする必要があります。エクスポート コマンドは、メインツールバー → [Solution] → [Export RTL] をクリックすると実行できます。

図 61: RTL を IP としてエクスポート



XO ファイルの位置を確認する必要があります。これで生成された XO-File に名前が付いて、次のセクションで SDAccel にインポートし戻されます。

注記: 前のセクションのオプションのほとんどは、メインメニュー → [Solution] → [Solution Settings] から設定および変更することもできます。[Synthesis] および [Export] セクションには、前述と同じ内容が含まれます。

これで、SDAccel 用の HLS 合成部分は終了です。次のセクションでは、コマンド ライン フローに必要な詳細を示します。

SDAccel 合成用の典型的な Vivado HLS スクリプト

HLS 合成をコマンド ライン スクリプトから実行する場合、前述の GUI フローと同等の Tcl コードは次のようになります。

```
open_project guiProj
set_top krnl_idct
add_files src/krnl_idct.cpp
add_files -tb src/idct.cpp
open_solution "solution1"
set_part {xcu200-fsgd2104-2-e} -tool vivado
create_clock -period 10 -name default
config_sdx -optimization_level none -target xocc
```

```
config_schedule -effort medium -enable dsp_full_reg
config_compile -name_max_length 256 -pipeline_loops 64
#source "../guiProj/solution1/directives.tcl"
csim_design
csynth_design
cosim_design
export_design -rtl verilog -format ip_catalog -xo \
    /wrk/bugs/xoFlow/idct_hls/krnl_idct.xo
```

SDAccel への Vivado HLS カーネル プロジェクトの取り込み

Vivado HLS の出力は、ザイリンクス オブジェクト ファイル (xo) としてエクスポートされるカーネル コードです。このファイルは、入力としてオブジェクト ファイルを選択することにより SDAccel にスムーズに取り込むことができます (詳細は [ソースの追加](#) を参照)。SDAccel に xo ファイルをインポートすると、カーネル名が自動的に抽出され、ホストコードでアクセラレータを適用できるようになります。

SDAccel のコンパイル中、カーネルから複数の計算ユニットを作成はできますが、インプリメンテーションは Vivado HLS の実行で指定したままになります。

SDAccel でこのフローを使用すると、通常のデバッグおよび解析機能が完全にサポートされます。ハードウェア エミュレーション フローをビルドすると、インプリメンテーションを詳細にテストおよびデバッグでき、システムビルドでホストコードのパフォーマンスを調整できます。

注記: 重複したヘッダー ファイルの依存性により問題が発生する可能性があるため、純粋なソフトウェア エミュレーションモードは現時点ではサポートされていません。

既知の制限

このフローには、トップダウンフローにはない制限があります。

- xo ファイルを使用したプロジェクトのソフトウェア エミュレーションはサポートされません (存在しない可能性およびヘッダー ファイルが重複している可能性あり)。
- ハードウェア エミュレーション フローの GDB カーネル デバッグはサポートされません。
- HLS 解析機能は、Vivado HLS プロジェクトでしか使用できず、SDAccel からは使用できません。

サンプル デザインの概要

すべてのザイリンクス SDx™ 環境には、サンプル デザインが含まれています。これらのサンプル デザインには、次の特徴があります。

- SDx IDE および makefile フローなどのコンパイル フローを習得。
- 新規アプリケーション プロジェクトを作成する開始点として使用。
- 便利なコーディング スタイルを表示。
- 重要な最適化手法を表示。

SDx 環境で提供される各プラットフォームには、ユーザーがすばやく開発を開始できるようにするサンプル デザインが含まれており、[アプリケーション プロジェクトの作成](#)で説明されているようにプロジェクトの作成時にアクセスできます。<SDx_Install_Dir>/samples にあるこれらのサンプル デザインには、コマンドラインのみでコードのビルド、エミュレート、および実行を実行できるようにする makefile も含まれています。

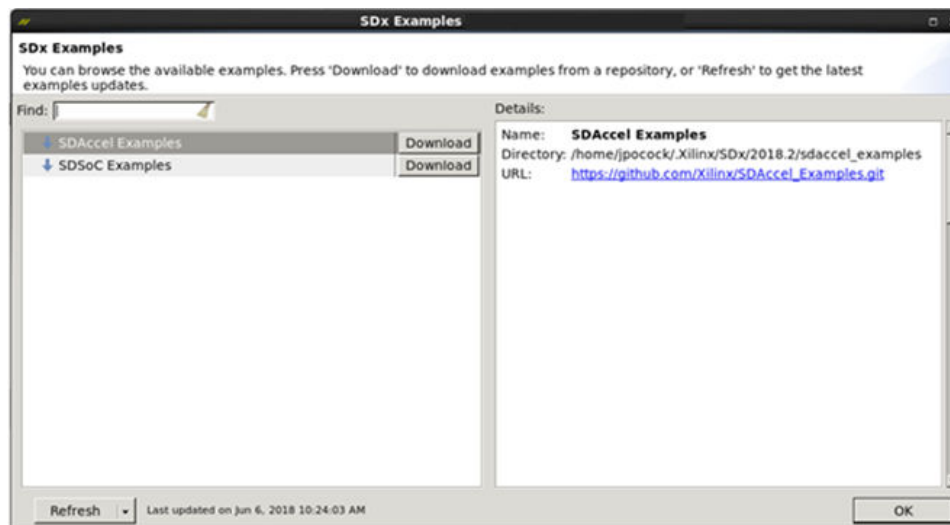
ザイリンクス [GitHub](#) リポジトリから、多数のサンプル デザインおよびチュートリアルをダウンロードできます。[サンプル デザイン リポジトリ](#)には、ザイリンクス PCIe[®] FPGA アクセラレーション ボードをターゲットとするアプリケーションの最適化を開始するための最新のサンプルが含まれます。すべての例は、SDAccel™ でサポートされるボードおよびアクセラレーション クラウド サービスでコンパイルおよび実行できます。

[チュートリアル リポジトリ](#)では、アプリケーションのビルド、エミュレーション、C++ と RTL カーネルの混合、ホストコードの最適化など、さまざまな手順が詳細に説明されています。

サンプル デザインのインストール

New SDx Project ウィザードを使用する場合、新しいプロジェクトのテンプレートを選択します。テンプレート プロジェクトは、[Xilinx] → [SDx Examples] をクリックして、既存のプロジェクトから読み込むこともできます。

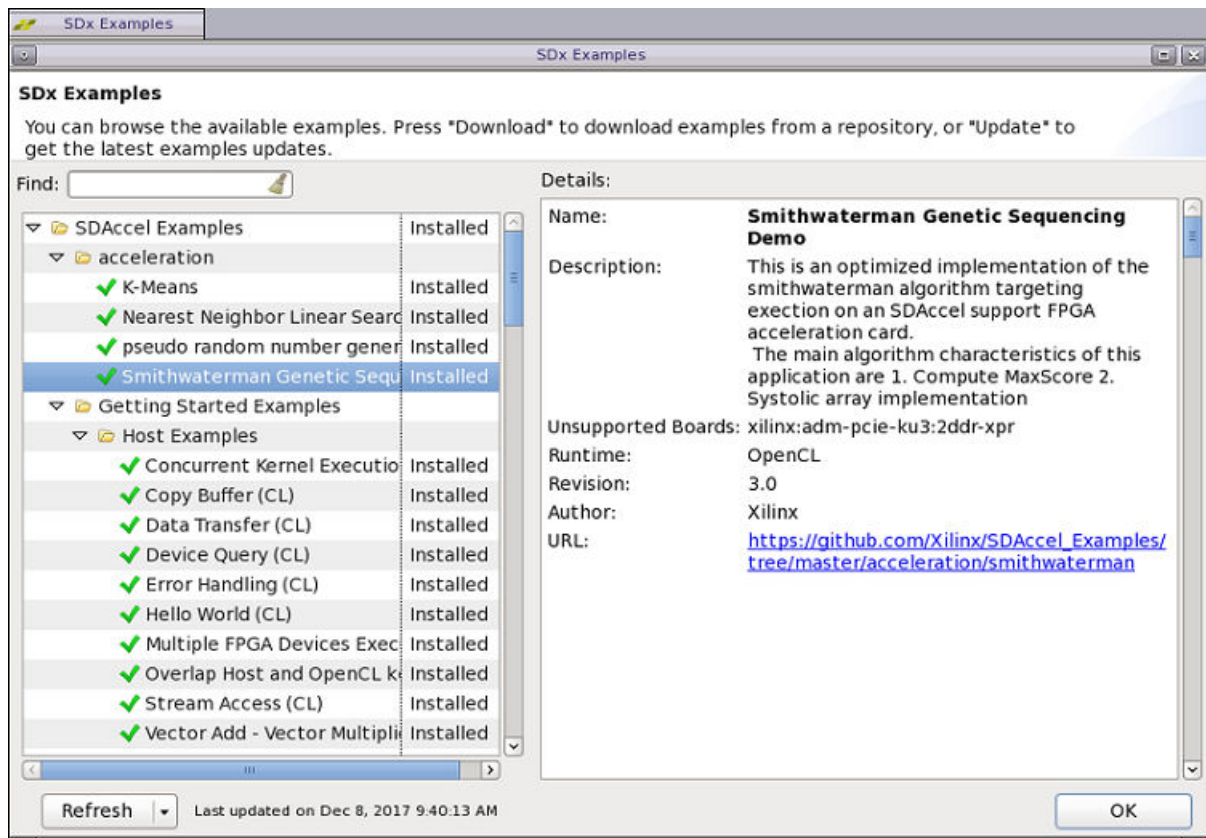
図 62: SDAccel サンプル デザイン - 空の状態



このダイアログ ボックスの左側には [SDAccel™ Examples] が表示され、それぞれに [Download] ボタンがあります。ダイアログ ボックスの右側には、サンプル デザインがダウンロードされるディレクトリとダウンロード元の URL が表示されます。ダウンロード ディレクトリの場所を変更する方法は、[ローカル コピーの使用](#) を参照してください。

[SDAccel Examples] の横の [Download] をクリックすると、サンプル デザインがダウンロードされ、ダイアログ ボックスにリストされます。サンプル デザインをダウンロードすると、次の図に示すように表示されます。

図 63: SDAccel サンプル デザイン - ダウンロード後の表示



[SDx Examples] ダイアログボックスの左下のコマンドメニューには、サンプルデザインのリポジトリを管理する2つのコマンドがあります。

- [Refresh]: ダウンロード済みサンプルデザインを [GitHub](https://github.com/Xilinx/SDAccel_Examples/tree/master/acceleration/smithwaterman) リポジトリからアップデートします。
- [Reset]: .Xilinx フォルダからダウンロードしたサンプルデザインを削除します。

注記: 企業のファイアウォールにより、外部への接続が制限されていることがあります。特定のプロキシ設定が必要なこともあります。

ローカル コピーの使用

新規プロジェクトを作成する際にテンプレートを追加するには、サンプルをダウンロードする必要がありますが、SDx IDE ではサンプルはローカルの .Xilinx/SDx/<version> フォルダにダウンロードされます。

- Linux: ~/.Xilinx/SDx/<version>

ダウンロード ディレクトリは、[SDx Examples] ダイアログ ボックスからは変更できません。サンプル ファイルを .Xilinx フォルダーとは別のディレクトリにダウンロードするのが望ましい場合もあります。これには、コマンド シェルから git コマンドを使用し、ダウンロード ディレクトリを別のディレクトリに変更します。

```
git clone https://github.com/Xilinx/SDAccel_Examples  
<workspace>/examples
```

上記に示すように git コマンドでサンプルをクローンすると、サンプル ファイルをアプリケーションおよびカーネルコードのリソースとしてプロジェクトで使用できるようになります。ただし、ほかのサンプル ファイル (さまざまなサンプルの makefile 内で管理) を含めるために、ファイルの多くに include 文が使用されています。これらのインクルード ファイルは、NewSDxProject ウィザードでテンプレートを追加したときにプロジェクトの src フォルダーに自動的に含まれます。ファイルをローカルにするには、ファイルを手動でプロジェクトのローカルにします。

必要なファイルは、クローンされたリポジトリのディレクトリから検索できます。たとえば、次のコマンドを examples フォルダーから実行すると、vadd サンプル デザインに必要な xcl2.hpp ファイルを検索できます。

```
find -name xcl2.hpp
```

ディレクトリ構造

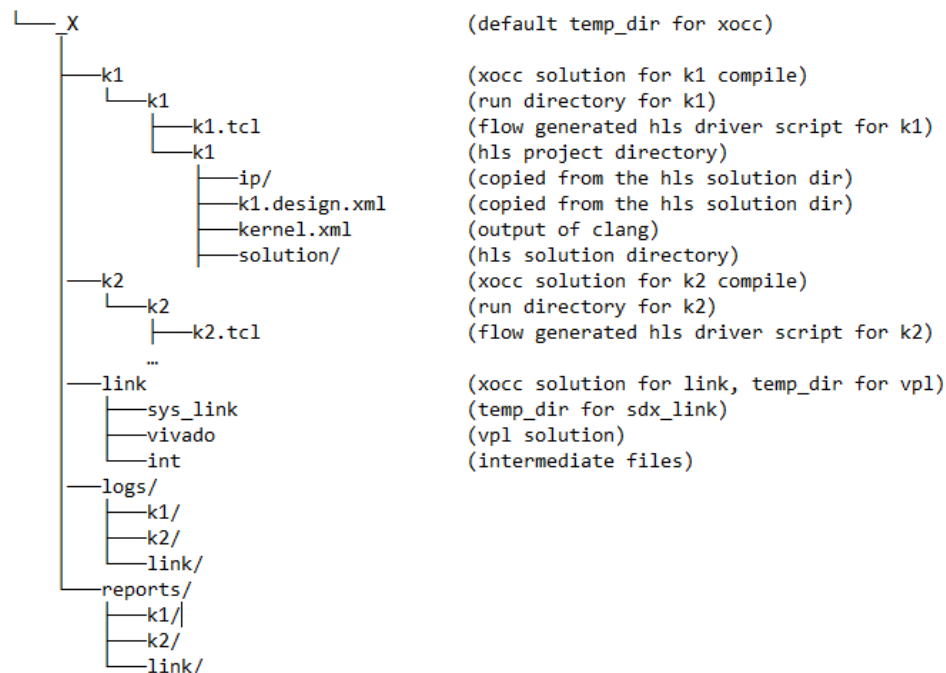
GUI および makefile フローで生成されるディレクトリ構造は、ファイルを簡単に探してアクセスできるように構成されています。compile、link、logs、および reports ディレクトリから、生成されたファイルに簡単にアクセスできます。同様に、各カーネルにもそれぞれディレクトリ構造が作成されます。

コマンド ライン

コマンドラインで xocc を使用すると、コンパイルおよびリンク中にデフォルトでディレクトリ構造が作成されます。_xo および .xclbin は、常に作業ディレクトリに生成されます。すべての中間ファイルは _x ディレクトリ (temp_dir のデフォルト名) に作成されます。

次に、2つの xocc コンパイル run (k1 および k2) と 1つの xocc リンク (design.xclbin) に生成されたディレクトリ構造の例を示します。k1.xo、k2.xo、および design.xclbin ファイルは、作業ディレクトリに含まれます。_x ディレクトリには、関連の k1 および k2 カーネル コンパイル サブディレクトリが含まれます。link、logs、および reports ディレクトリには、ビルドのリンク、ログ、およびレポート情報が含まれます。

図 64: コマンドラインのディレクトリ構造



次の xocc オプションを使用すると、このディレクトリ構造を変更できます。

```
--log_dir <dir_name (full or relative path)>
```

```
--report_dir <dir_name (full or relative path)>
```

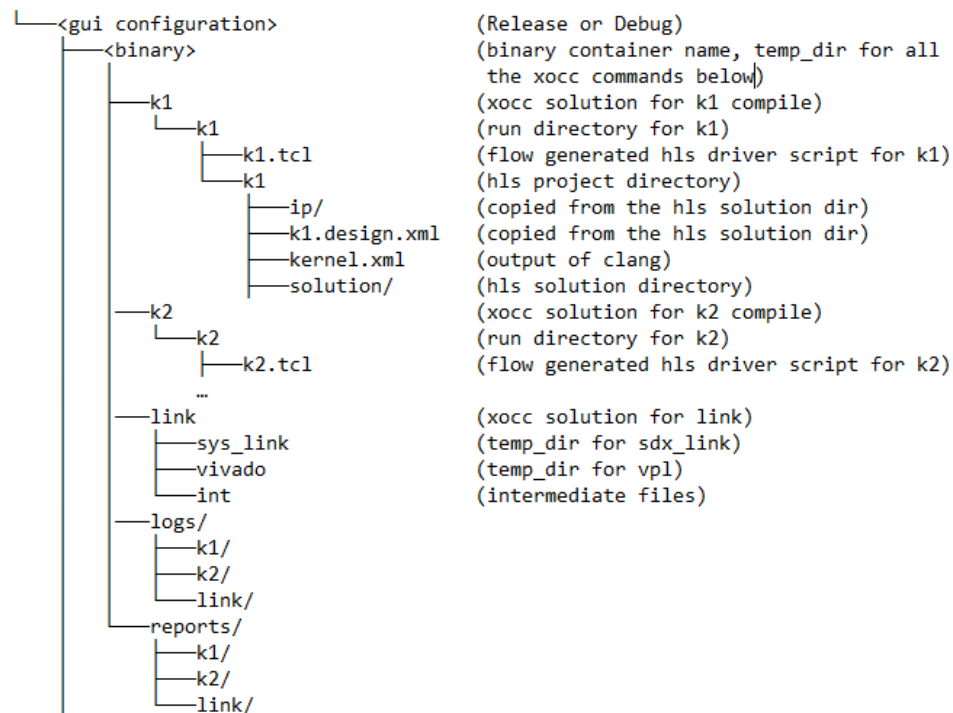
```
--temp_dir <dir_name (full or relative path)>
```

xocc コマンド オプションの詳細は、『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。

GUI

ディレクトリのデフォルトの命名規則と構造は、makefile フローで作成されるものと似ていますが、まったく同じではありません。次に、2つの xocc コンパイル run (k1 および k2) と 1つの xocc リンク (design.xclbin) に生成されたディレクトリ構造の例を示します。k1.xo、k2.xo、および design.xclbin ファイルは、作業ディレクトリに含まれます。_x ディレクトリには、関連の k1 および k2 カーネル コンパイル サブディレクトリが含まれます。link、logs、および reports ディレクトリには、ビルドのリンク、ログ、およびレポート情報が含まれます。

図 65: GUI のディレクトリ構造



GUI では、makefile に含まれる次の `xocc` コマンド仕様を使用してディレクトリ構造が作成されます。

```
--temp_dir
```

```
--report_dir
```

```
--log_dir
```

`xocc` コマンド オプションの詳細は、『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。

便利なコマンド ライン ユーティリティ

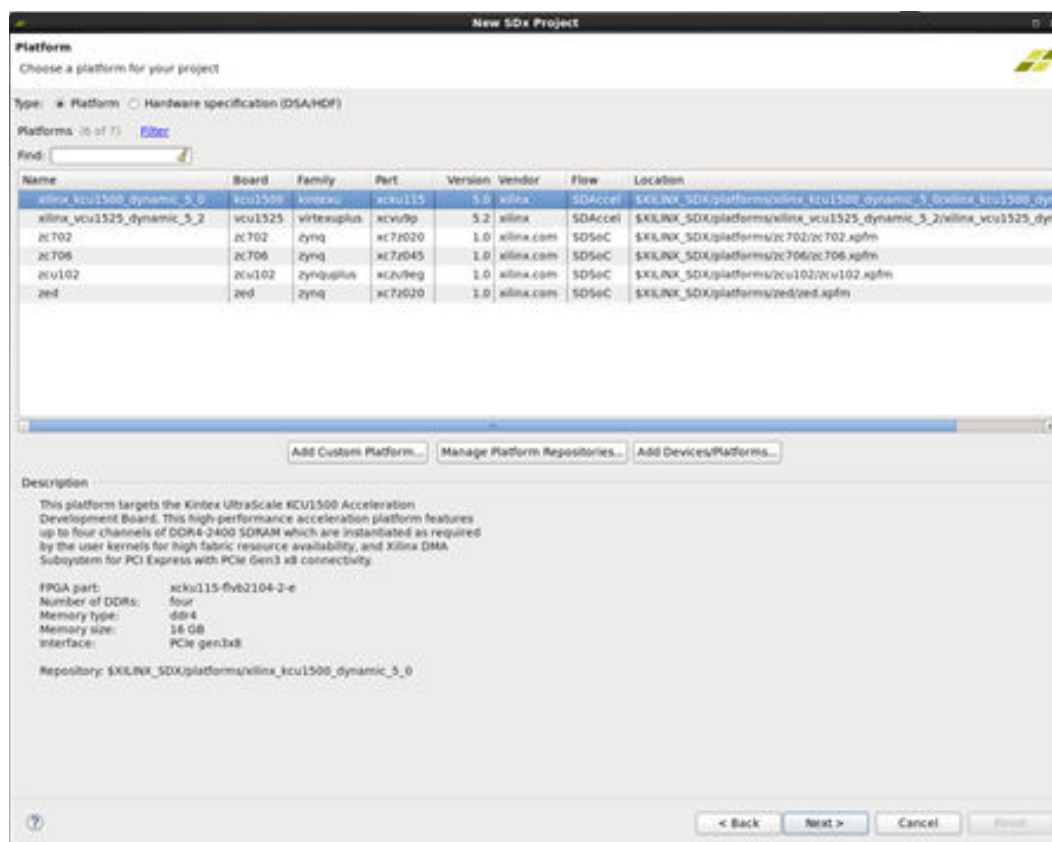
便利なサイリンクス コマンド ライン ユーティリティが複数提供されており、`xocc` コマンド ラインを作成するのに有益な詳細な情報や、使用可能な SLR リソースなどのプラットフォームに関するデータを取得するのに役立ちます。これらには、`platforminfo`、`kernelinfo`、`xclbinutil`、エミュレーション コンフィギュレーション、サイリンクス ボード、および SDSoc™ ユーティリティが含まれます。これらのコマンド ライン ユーティリティの詳細は、『SDx コマンドおよびユーティリティ リファレンス ガイド』 ([UG1279](#)) を参照してください。

プラットフォームおよびリポジトリの管理

SDx™ 環境には、ビルトイン プラットフォームが含まれます。プロジェクトにカスタム プラットフォームを使用する必要がある場合は、アプリケーションのインプリメンテーションでそのプラットフォームを使用できるようにしておく必要があります。

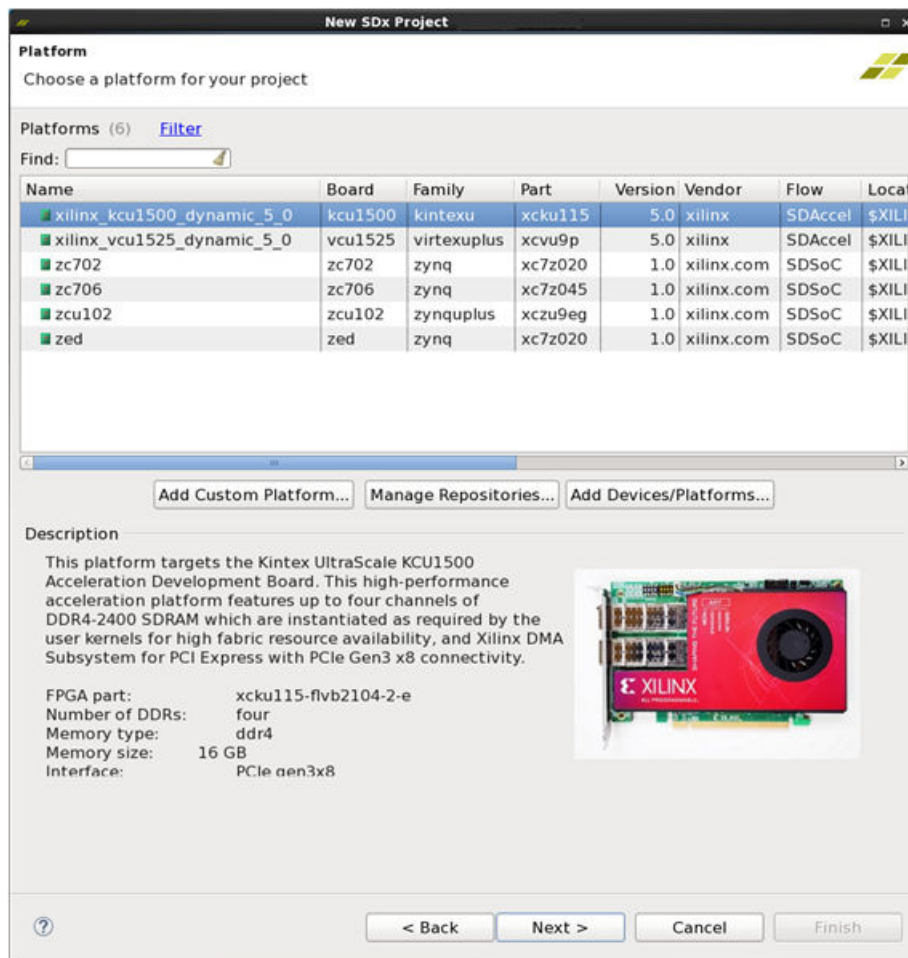
プロジェクトを作成する際、New SDx Project ウィザードの [Platform Selection] ページで、SDAccel™ アプリケーションプロジェクトで使用可能なプラットフォームを管理できます。このページで、プロジェクトの作成時に新しいプラットフォームを追加できます。

図 66: SDAccel プラットフォームの参照



New SDx Project ダイアログ ボックスが開き、使用可能なプラットフォームおよびプラットフォーム リポジトリを管理できます。

図 67: SDAccel プラットフォームの指定



- [Add Custom Platform]: ユーザーのプラットフォームを使用可能なプラットフォームのリストに追加します。カスタムプラットフォームの最上位ディレクトリを選択して、[OK] をクリックすると、新しいプラットフォームが追加されます。カスタムプラットフォームは、すぐに使用可能なプラットフォームのリストから選択できるようになります。[Xilinx] → [Add Custom Platform] をクリックし、ツールにカスタムプラットフォームを直接追加します。
- [Manage Repositories]: 標準プラットフォームとカスタムプラットフォームを追加または削除します。カスタムプラットフォームを追加すると、その新しいプラットフォームへのパスがリポジトリに自動的に追加されます。リポジトリのリストからプラットフォームを削除すると、使用可能なプラットフォームのリストからプラットフォームが削除されます。
- [Add Devices/Platforms]: インストールするザイリンクスデバイスおよびプラットフォームを管理します。インストール時にデバイスまたはプラットフォームを選択しなかった場合、このコマンドを使用して後から追加できます。このコマンドをクリックすると、SDx インストーラーが起動し、追加するデバイスまたはプラットフォームを選択できます。[Help] → [Add Devices/Platforms] をクリックし、ツールにカスタムプラットフォームを直接追加します。

新規ターゲット プラットフォームへの移行

このセクションでは、あるターゲット用にアクセラレーションされた SDAccel™ 環境アプリケーションを別のターゲットに移行する方法を説明します。たとえば、アプリケーションを Virtex® UltraScale+™ VCU1525 アクセラレーション開発ボードから U200 アクセラレーション開発ボードに移行する場合などです。

次のトピックについて説明します。

- FPGA デバイスの物理的な側面を含むデザイン移行プロセスの概要。
- 新規リリースを使用する場合のホスト コードおよびデザイン制約の変更。
- カーネル配置および DDR インターフェイス接続の制御。
- パフォーマンスを達成するためにオプションを追加する必要がある新規シェルでのタイミング問題。

デザインの移行

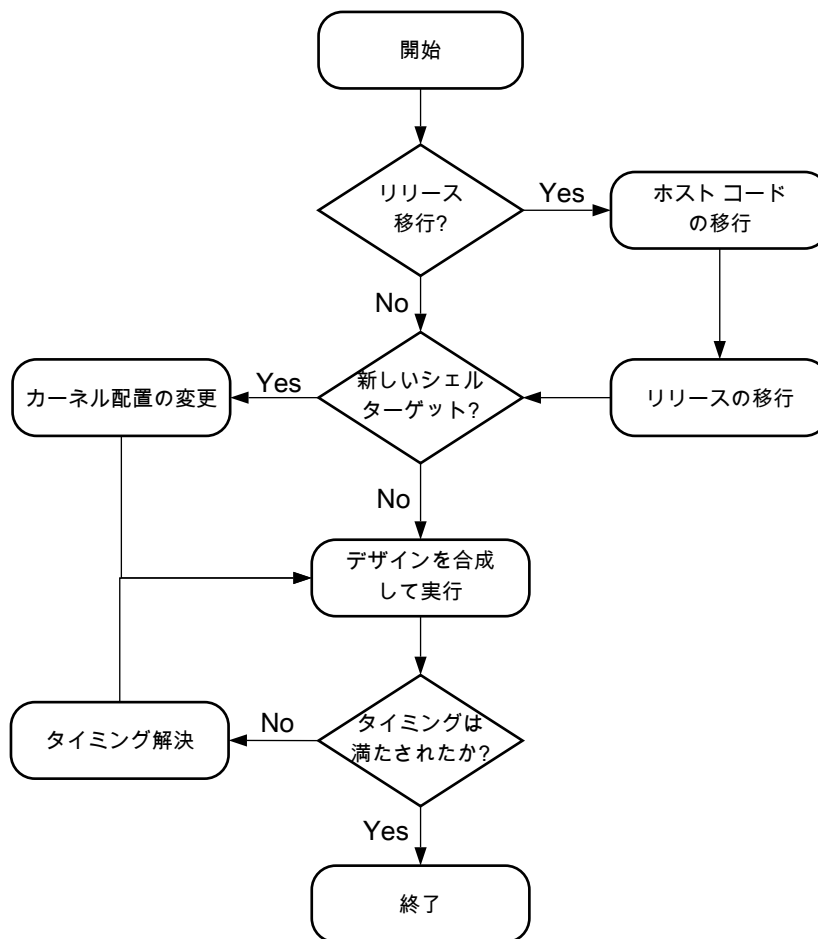
1つのターゲット プラットフォームにインプリメントされたアプリケーションを別のプラットフォームに移行するには、ターゲット プラットフォーム間の違いとその違いがデザインに及ぼす影響について理解することが重要です。

主な注意事項:

- リリースで変更があったか。
- 新規ターゲット プラットフォームに別のシェルが含まれているか。
- カーネルを複数の SLR (Super Logic Region) に再分配する必要があるか。
- デザインが新しいプラットフォームで必要な周波数(タイミング) パフォーマンスを満たすか。

次の図に、このガイドおよびトピックで説明される移行フローを示します。

図 68: シェル移行のフローチャート



X21401-120318



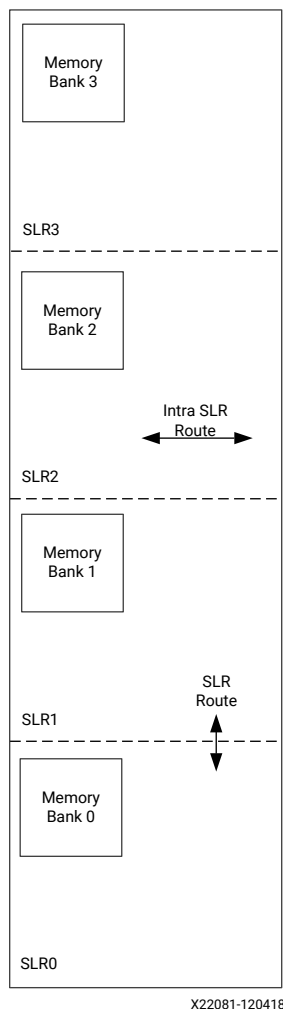
重要: デザインの移行を開始する前に、FPGA およびシェルのアーキテクチャを理解することが重要です。

FPGA アーキテクチャの理解

デザインを新しいターゲット プラットフォームに移行する前に、FPGA アーキテクチャの基礎を理解しておく必要があります。次の図に、ザイリンクス FPGA デバイスのフロアプランを示します。理解する必要のあるコンセプトは、次のとおりです。

- SSI デバイス
- SLR
- SLR の配線リソース
- メモリ インターフェイス

図 69: 4 つの SLR 領域を含むザイリンクス FPGA の物理的な表示



ヒント: 上記の FPGA フロアプランは、各 SLR に DDR メモリ インターフェイスを含む 4 つの SLR がある、SSI デバイスのものです。

スタックド シリコン インターコネクト デバイス

SSI デバイスでは、シリコン インターコネクトを介して複数のシリコン ダイが一緒に接続され、1 つのデバイスにパッケージされます。SSI デバイスを使用すると、かなり多くの接続が提供されるので、複数のダイ間でバンド幅の大きな接続ができるようになります。また、レイテンシもかなり低くなり、消費電力も複数の FPGA またはマルチチップ モジュールのいずれかのアプローチ方法よりもかなり低くなるほか、大量のインターコネクト ロジック、トランシーバー、およびオンチップ リソースを 1 つのパッケージに統合できるようになります。SSI デバイスの利点については、『ザイリンクスのスタックド シリコン インターコネクト テクノロジーで飛躍的な FPGA 容量、帯域幅、電力効率を実現』 (WP380: [英語版](#)、[日本語版](#)) を参照してください。

Super Logic Region

SLR (Super Logic Region) は、SSI デバイスに含まれる 1 つの FPGA ダイ スライスです。複数の SLR コンポーネントがアセンブルされて、SSI デバイスになります。各 SLR には、ほとんどのザイリンクス FPGA デバイスに使用される能動回路が含まれます。この回路には、次が多く含まれます。

- LUT
- レジスタ
- I/O コンポーネント
- ギガビット トランシーバー
- ブロック メモリ
- DSP ブロック

1 つまたは複数のカーネルを SLR 内にインプリメントできます。1 つのカーネルを複数の SLR にまたがってインプリメントすることはできません。

SLR の配線リソース

FPGA にインプリメントされるカスタム ハードウェアは、オンチップ配線リソースを介して接続されます。SSI デバイスの配線リソースには、次の 2 種類があります。

- SLR 間リソース: SLR 間配線リソースは、ハードウェア ロジックの接続に使用される、高速リソースです。SDAccel 環境では、カーネルをインプリメントする際にハードウェア エlement を接続するのに最適なリソースが自動的に選択されます。
- SLL (Super Long Line) リソース: SLL は、1 つの領域から次の領域へロジックを接続するために使用される SLR 間の配線リソースです。これらの配線リソースは、SLR 間配線よりも遅くなります。ただし、カーネルが 1 つの SLR に配置され、そこに接続される DDR が別の SLR にある場合、SDAccel 環境は自動的に専用ハードウェアをインプリメントして、パフォーマンスに影響なく SLL 配線リソースが使用されるようにします。配置の管理に関する詳細は、[カーネル配置の変更](#) を参照してください。

メモリ インターフェイス

SLR にはそれぞれ 1 つまたは複数のメモリ インターフェイスが含まれます。これらのメモリ インターフェイスは DDR メモリの接続に使用され、ホスト バッファのデータがカーネル実行前にコピーされます。カーネルは、それぞれデータを DDR メモリから読み込んで、結果を同じ DDR メモリに書き戻します。メモリ インターフェイスは FPGA のピンに接続し、メモリ コントローラー ロジックを含めます。

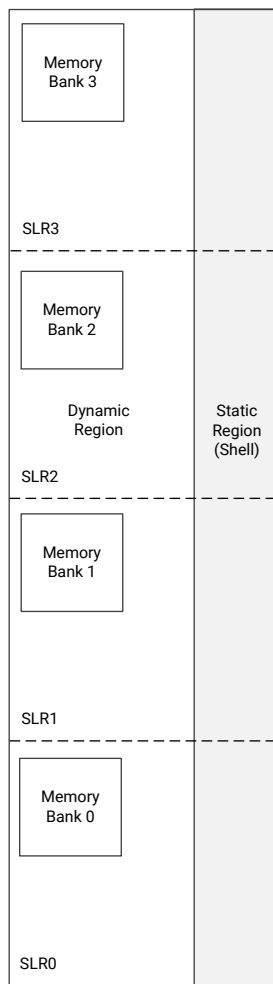
シェルの理解

SDAccel 開発環境では、シェルとはカスタム ロジックまたはアクセラレータが追加される前に FPGA にインプリメントされるハードウェア デザインのことです。シェルではターゲット プラットフォームで使用する FPGA の属性が定義され、次の 2 つの領域が含まれます。

- カーネルおよびデバイス マネージメント ロジックを含むスタティック領域。
- アクセラレーションされたカーネルのカスタム ロジックが配置されるダイナミック領域。

次の図に、シェルが適用された FPGA を示します。

図 70: 4 つの SLR 領域を含む FPGA 上のシェル



X22082-120418

シェル(ユーザーが変更できないスタティック領域)には、FPGAを動作させ、ダイナミック領域とのデータ転送に必要なロジックが含まれます。スタティック領域(図のグレーの部分)は1つのSLR内に含まれることもあれば、上記の例のように複数のSLRにまたがっていることもあります。スタティック領域には、次が含まれます。

- DDR メモリ インターフェイス コントローラー
- PCIe® インターフェイス ロジック
- XDMA ロジック
- ファイアウォール ロジックなど

ダイナミック領域は、上記の図の白色の部分です。この領域には、シェルのリコンフィギュラブル コンポーネントすべてが含まれ、すべてのアクセラレータ カーネルが配置されます。

スタティック領域はデバイスで使用可能なハードウェア リソースの一部を使用するので、ダイナミック領域にインプリメントされるカスタム ロジックではその残りのリソースしか使用できません。上記の例では、シェルはFPGAの使用可能な4つのDDR メモリ インターフェイスすべてを定義しています。これには、DDR インターフェイスで使われるメモリ コントローラー用のリソースが必要になります。

各シェルのダイナミック領域にインプリメントされるロジックの量については、『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)を参照してください。この詳細は、この後のカーネル配置の変更でも説明します。

リリースの移行

新規ターゲット プラットフォームに移行する前に、新しいプラットフォームのターゲットを別の SDAccel 環境リリースに変更する必要があるかどうかも決定する必要があります。ターゲットを新しいリリースに変更する場合は、まず新しいソフトウェアリリースを使用して既存のプラットフォームをターゲットとして、変更が必要ないかどうかを確認してから、新しいターゲット プラットフォームに移行することを強くお勧めします。

既存プラットフォームを使用して新しいリリースをターゲットにする手順は、次のとおりです。

- ホスト コードの移行
- リリースの移行



重要: 新しいリリースに移行する前に、『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)を参照することをお勧めします。

ホスト コードの移行

SDAccel 環境の 2018.3 リリースでは、ザイリンクス ランタイム (XRT) 環境およびシェルのインストール方法は基本的に異なっています。前のリリースでは、XRT 環境とシェルが SDAccel 環境をインストールすると自動的に含まれていました。これは、ホスト コードをコンパイルするのに必要な設定で示されます。

2018.3 のインストールの詳細は、『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)を参照してください。

XILINX_XRT 環境変数は、XRT 環境のディレクトリを指定するのに使用され、ホスト コードをコンパイルする前に設定する必要があります。XRT 環境をインストールしたら、`/opt/xilinx/xrt/setup.csh` または `/opt/xilinx/xrt/setup.sh` ファイルを適宜読み込んで XILINX_XRT 環境変数を設定できます。次に、LD_LIBRARY_PATH 変数も XRT インストール ディレクトリを指定するようにします。

コンパイルして、ホスト コードを実行するには、SDAccel インストールから `<SDX_INSTALL_DIR>/settings64.csh` または `<SDX_INSTALL_DIR>/settings64.sh` ファイルを読み込むようにしてください。

GUI を使用する場合は、新しい XRT ディレクトリが自動的に読み込まれ、プロジェクトを構築すると `makefile` が生成されます。

ただし、カスタマイズした `makefile` を使用する場合は、次のような変更を加える必要があります。

- `makefile` では、前のリリースで使った XILINX_SDX 環境変数は使用しないようにします。
- XILINX_SDX 変数およびパスは、XILINX_XRT 環境変数に変更する必要があります。
 - インクルード ディレクトリは、`-I${XILINX_XRT}/include` および `-I${XILINX_XRT}/include/CL` のように指定します。
 - ライブラリ パスは `-L${XILINX_XRT}/lib` のように指定します。
 - OpenCL™ ライブラリは `libxilinxopencl.so` のように指定します。このため、`makefile` ファイルで `lxilinxopencl` を使用します。

リリースの移行

ホスト コードを移行したら、SDAccel 開発環境の新しいリリースを使用して、既存のターゲット プラットフォームでコードをビルドします。新しいリリースの SDAccel 環境でプロジェクトを実行して、問題なく動作するかどうか、タイミングを満たしているかどうかを確認します。

新しいリリースを使用した場合は、次のような問題が発生する可能性があります。

- C ライブラリまたはライブラリ ファイルの変更。
- カーネルのパス名の変更。
- カーネル コードに埋め込まれた HLS プラグマまたはプラグマ オプションの変更。
- C/C++/OpenCL コンパイラ サポートの変更。
- カーネルのパフォーマンスの変更 (既存カーネル コードのプラグマを調整する必要があることあり)。

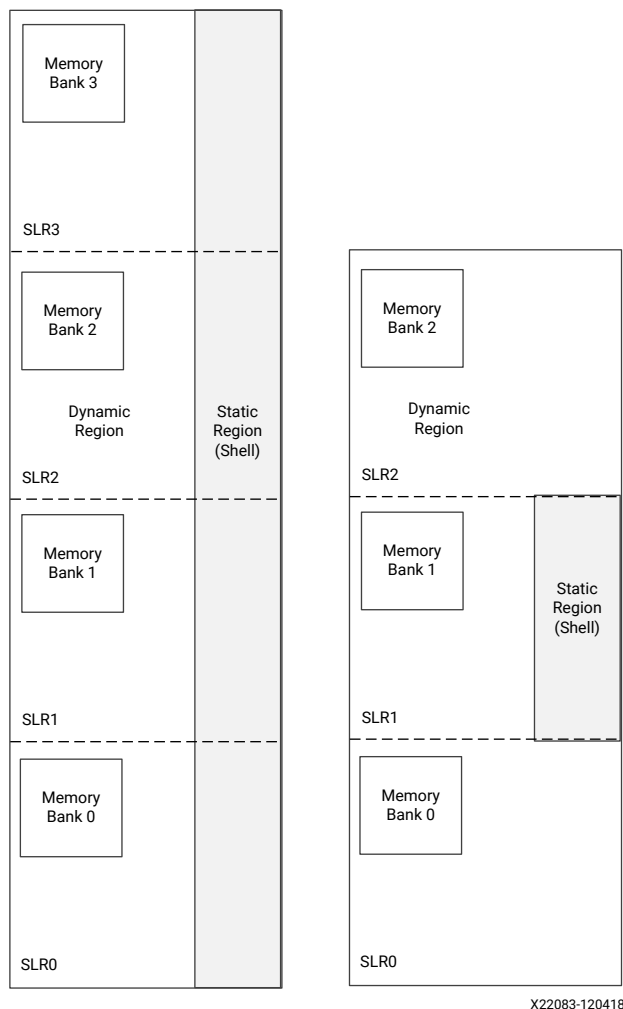
これらの問題は、カーネルの開発で使ったのと同じ手法を使用して解決します。この段階では、新しいリリースを使用してターゲット プラットフォームのスループット パフォーマンスが要件を満たすかどうかを確認します。最終的なタイミング (最大クロック周波数) に変更がある場合は、新しいターゲット プラットフォームに移行してから、問題を解決します。これについては、[タイミングの解決](#)で説明しています。

カーネル配置の変更

新しいプラットフォームにターゲットを変更する際の主な問題は、既存のカーネル配置が新しいターゲット プラットフォームで動作するようにすることです。各ターゲット プラットフォームには、シェルで定義された FPGA が含まれます。次の図に示すように、シェルは異なっていることがあります。

- 左側の元のプラットフォームのシェルには SLR が 4 つあり、スタティック領域が 4 つの SLR すべてにまたがっています。
- 右側のターゲット プラットフォームのシェルには SLR が 3 つしかなく、スタティック領域はすべて SLR1 内に完全に含まれています。

図 71: ハードウェア プラットフォームのシェルの比較



X22083-120418

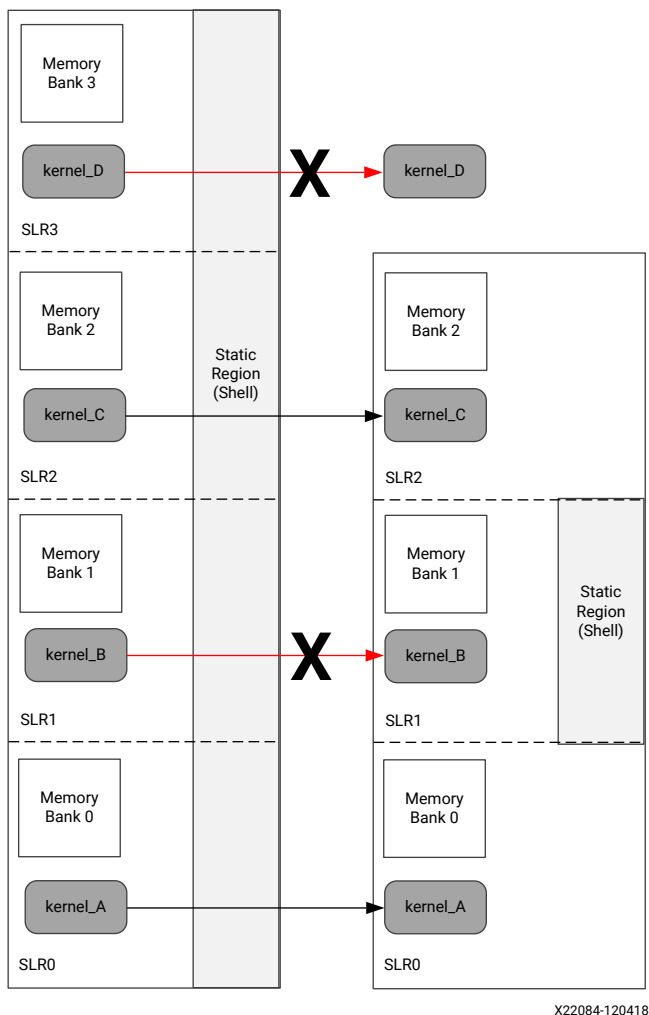
このセクションでは、カーネルの配置前の変更方法を説明します。

新しいハードウェア プラットフォームに移行する際の注意点

次の図に、新規ターゲット プラットフォームまたはシェルに移行する際に発生するカーネル配置の問題を示します。この例では、次のようになっています。

- 既存のカーネルの kernel_B は新しいターゲット プラットフォームの SLR2 には大きすぎて収まりません。これは、この SLR のほとんどがスタティック領域で使用されているからです。
- 新しいターゲット プラットフォームには既存のプラットフォームのように SLR が 4 つないので、既存のカーネルの kernel_D は新しい SLR に移動する必要があります。

図 72: プラットフォームの移行: カーネルの配置



新しいプラットフォームに移行する場合は、次を実行する必要があります。

- 新しいターゲットプラットフォームの各 SLR で使用可能なリソースを理解しておく (『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) を参照)。
- デザイン内の各カーネルに必要なリソースを理解しておく。
- `xocc` リンカー オプション (`--slr` および `--sp`) を使用して、各カーネルが配置される SLR と、各カーネルが接続される DDR を指定。

これらについては、次のセクションで説明します。

カーネルを配置する場所の決定

カーネルを配置する場所を決めるには、次の 2 つの情報が必要です。

- ハードウェア プラットフォーム (.dsa) のシェルの各 SLR で使用可能なリソース。
- 各カーネルに必要なリソース。

これら 2 つの情報を使用すると、シェルの各 SLR にどのカーネルを配置するかを決定できます。

これらを計算する際は、使用可能なリソースの 10% がシステム インフラストラクチャで使用される可能性があることに注意してください。

- インフラストラクチャ ロジックは、カーネルが SLR 境界をまたぐ必要がある場合にカーネルを DDR インターフェイスに接続するのに使用できることがあります。
- FPGA では、リソースが信号配線にも使用されます。信号配線にもリソースが必要なので、FPGA で使用可能なリソースを 100% 使用することはできません。

使用可能な SLR リソース

ザイリンクス で提供される各 SLR で使用可能なリソースについては、『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) を参照してください。次の図は、サンプル シェルです。この例では、次を確認できます。

- SLR 記述は、どの SLR にスタティック領域やダイナミック領域が含まれるかを示しています。
- 各 SLR で使用可能なリソース (LUT、レジスタ、RAM など) がリストされます。

これにより、各 SLR でどのリソースが使用可能かどうかを判断できます。

表 19: ハードウェア プラットフォームの SLR リソース

エリア	SLR 0	SLR 1	SLR 2
SLR 記述	デバイスの最下部、ダイナミック領域専用。	デバイスの中部、ダイナミック領域およびスタティック領域リソースで共有。	デバイスの最上部、ダイナミック領域専用。
ダイナミック領域の Pblock 名	pfa_top_i_dynamic_region_pblock_dynamic_SLR0	pfa_top_i_dynamic_region_pblock_dynamic_SLR1	pfa_top_i_dynamic_region_pblock_dynamic_SLR2
計算ユニット配置構文	set_property CONFIG.SLR_ASSIGNMENTS SLR0[get_bd_cells<cu_name>]	set_property CONFIG.SLR_ASSIGNMENTS SLR1[get_bd_cells<cu_name>]	set_property CONFIG.SLR_ASSIGNMENTS SLR2[get_bd_cells<cu_name>]
ダイナミック領域で使用可能なグローバル メモリ リソース			
メモリ チャンネル、システムポート名	bank0 (16 GB DDR4)	bank1 (16 GB DDR4、スタティック領域) bank2 (16 GB DDR4、ダイナミック領域)	bank3 (16 GB DDR4)
ダイナミック領域で使用可能なファブリック リソースの概算			
CLB LUT	388K	199K	388K
CLB レジスタ	776K	399K	776K
ブロック RAM タイル	720	420	720
UltraRAM	320	160	320
DSP	2280	1320	2280

カーネル リソース

各カーネルのリソースは、システム見積もりレポートからわかります。

システム見積もりレポートは、ハードウェア エミュレーションまたはシステム実行のいずれかが終了したら、[Assistant] ビューに表示されるようになります。次は、そのレポートの例です。

図 73: システム見積もりレポート

Area Information						
Compute Unit	Kernel Name	Module Name	FF	LUT	DSP	BRAM
smithwaterman_1	smithwaterman	smithwaterman	2925	4304	1	10

- FF は、各 SLR のプラットフォーム リソースの箇所に記述された CLB レジスタのことです。
- FF は、各 SLR のプラットフォーム リソースの箇所に記述された CLB レジスタのことです。
- DSP は、各 SLR のプラットフォーム リソースの箇所に記述された DSP のことです。
- ブロック RAM は、各 SLR のプラットフォーム リソースの箇所に記述されたブロック RAM タイルのことです。

この情報は、カーネルごとに適切な SLR を指定するのに役立ちます。

カーネルの SLR への割り当て

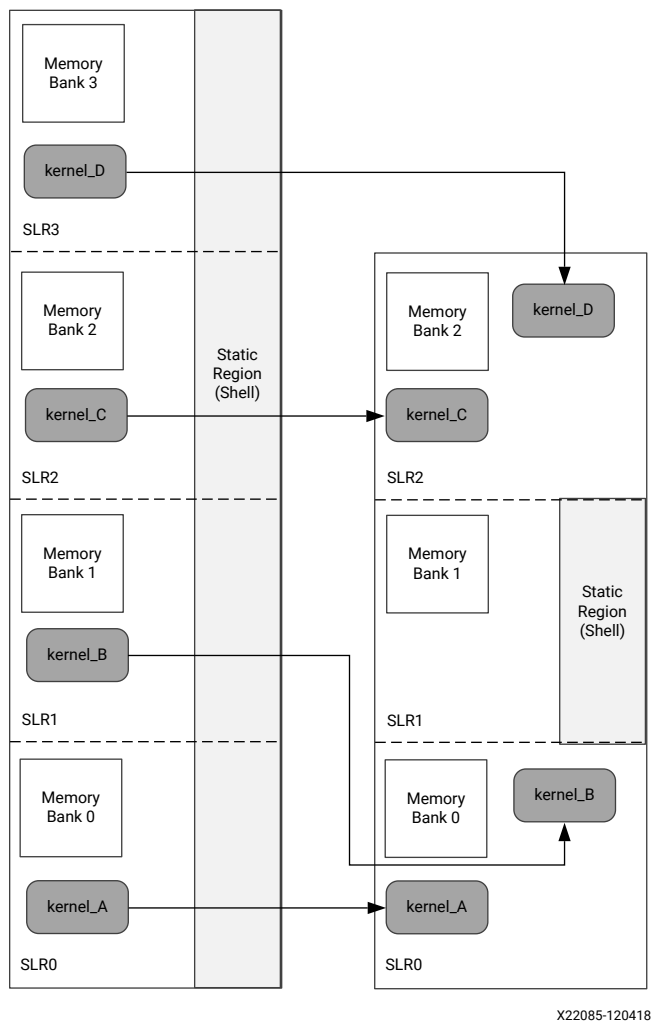
デザインの各カーネルは、配置ファイルを指定する `xocc --slr` コマンドライン オプションを使用して SLR 領域に割り当てることができます。カーネルを配置する場合は、`xocc --sp` コマンドライン オプションを使用して、カーネルが接続される特定の DDR メモリ バンクを割り当てることもお勧めします。次の例では、これらの 2 つのコマンドライン オプションを説明します。

次の図は、既存のターゲット プラットフォームのシェルには 4 つの SLR が含まれ、新規プラットフォームのシェルには 3 つの SLR が含まれ、スタティック領域の構造がターゲット プラットフォーム間で異なる例を示しています。この移行の例の詳細は、次のとおりです。

- Kernel_A は SLR0 にマップされます。
- Kernel_B は SLR1 にはフィットしなくなったので、使用可能なリソースがある SLR0 にリマップされます。
- Kernel_C は SLR2 にマップされます。
- Kernel_D は使用可能なリソースがある SLR2 にリマップされます。

カーネル マップは、次の図に示すようになります。

図 74: カーネルの SLR へのマップ



X22085-120418

カーネル配置の指定

上記の例の場合、カーネルは次の `xocc` コマンド オプションを使用して配置されます。

```
xocc --slr kernel A:SLR0 \  
  --slr kernel B:SLR0 \  
  --slr kernel C:SLR2 \  
  --slr kernel D:SLR2
```

これらのコマンドライン オプションを使用すると、各カーネルが上記の図のように配置されます。

カーネル DDR インターフェイスの指定

カーネル配置を指定する場合は、カーネル DDR メモリ インターフェイスも指定する必要があります。DDR インターフェイスを指定すると、別の SLR にある DDR インターフェイスへのカーネル接続が自動的にパイプライン処理されるようになります。これにより、タイミングが落ちることはないで、最大クロック周波数を削減される可能性があります。

この例では、上記の図のカーネル配置を使用して、次が実行されます。

- Kernel_A はメモリ バンク 0 に接続されます。
- Kernel_B はメモリ バンク 1 に接続されます。
- Kernel_C はメモリ バンク 2 に接続されます。
- Kernel_D はメモリ バンク 1 に接続されます。

次の `xocc` コマンド ラインでこれらの接続が実行されます。

```
xocc --sp kernel_A.arg1:bank0 \
--sp kernel_B.arg1:bank1 \
--sp kernel_C.arg1:bank2 \
--sp kernel_D.arg1:bank1
```



重要: `--sp` オプションを使用してカーネル ポートメモリ バンクに割り当てる際は、カーネルのすべてのインターフェイス/ポートに対して `--sp` オプションを指定する必要があります。詳細は、『SDAccel 環境プログラマ ガイド』(UG1277) の「DDR バンクとカーネルの接続のカスタマイズ」を参照してください。

タイミングの解決

システム実行が違反なく終了すれば、正しく移行されています。

タイミングが満たされなかった場合は、タイミングが満たされるようにカスタム制約を指定する必要があることもあります。タイミングを満たすための詳細は、『UltraFast 設計手法ガイド (Vivado Design Suite 用)』(UG949)を参照してください。

カスタム制約

カスタム制約は、カスタムの配置およびタイミング制約を伝えるために `xocc -xp` オプションを使用して Vivado® に渡されます。カーネルのフロアプラン用のカスタム Tcl 制約は、新しいターゲット プラットフォーム (.dsa) で確認する必要があります。たとえば、カーネルを新しいシェルでは別の SLR に移動されると、そのカーネルの配置制約も変更する必要があります。

通常は、タイミングは異なるターゲット プラットフォーム間 (9P Virtex UltraScale デバイス ベース) で比較可能です。タイミング クロージャ用のカスタム Tcl 制約はすべて、新しいプラットフォーム用に評価して変更する必要があります。

また、デフォルト以外のオプションが `xocc` または Vivado ツール (`xocc --xp` オプションを使用) に渡される場合も、新しいシェル用にアップデートする必要があります。

タイミング クロージャに関する注意事項

SDx™ リリースまたはシェルを移行すると、特に次のいずれかの条件が当てはまる場合は、デザイン パフォーマンスおよびタイミング クロージャが変わってしまうことがあります。

- タイミング クロージャにフロアプラン制約が必要であった。
- デバイスまたは SLR 使用率が通常のガイドラインよりも大きかった。
 - LUT 使用率が 70% を超えていた。

- DSP、RAMB、UltraRAM 使用率が 80% を超えていた。
- FD 使用率が 50% を超えていた。
- タイミング クロージャのためにコンパイル ストラテジに多くのエフォートが必要であった。

使用率ガイドラインには、デザインのコンパイル時間が長くなったか、またはパフォーマンスが初期見積もりよりも低下したかを示すしきい値があります。複数の SLR を必要とする大型のデザインの場合は、`xocc` コマンドラインでカーネル/DDR の関連付けを指定して、フロアプラン制約で次のようになるかどうかを確認します。

- 各 SLR の使用率が推奨ガイドラインよりも低い。
- ハードウェア リソースの 1 つのタイプがガイドラインよりも多く必要な場合に、その使用率が SLR 間にバランスを取って分配されている。

全体的に使用率の高いデザインの場合、カーネルでパイプライン処理の量を増やすと (レイテンシは長くなる)、タイミング クロージャが達成しやすくなり、パフォーマンスも達成しやすくなります。

上記の点をすべてすばやく確認するには、SDx フロー内で次の 2 つのいずれかのオプションを使用してフェイルファースト レポートを生成します。

- `xocc -R 1`
 - `report_failfast` が各カーネルの合成段階の終わりに実行されます。
 - `report_failfast` がデザイン全体の `opt_design` の後に実行されます。
 - `opt_design DCP` が保存されます。
- `xocc -R 2`
 - `-R 1` と同じレポートに加え、次が追加されます。
 - `report_failfast` は各 SLR の配置後に実行されます。
 - その他のレポートおよび中間 DCP が生成されます。

すべてのレポートおよび DCP は、カーネル合成レポートも含め、中間ディレクトリに保存されます。

```
<runDir>/_x/link/vivado/prj/prj.runs/impl_1
```

タイミング クロージャおよびフェイルファースト レポートの詳細は、『UltraFast 設計手法タイミング クロージャ クイックリファレンス ガイド』 ([UG1292](#)) を参照してください。

プライベート デバッグ ネットワーク用の JTAG フォールバック

データセンター環境での RTL カーネルおよびプラットフォームのデバッグには、ボード上の物理的な JTAG コネクタは通常使用できないので、XVC-over-PCIe® 接続が使用されるのが一般的です。XVC-over-PCIe を使用するとリモートでシステムをデバッグできますが、AXI インターコネクットのシステム ハングアップなどのデバッグ シナリオでは、これらの PCIe/AXI 機能を使用するデザインのデバッグ機能は使用できません。これらのシナリオをデバッグするのは、プラットフォーム設計では特に重要です。

JTAG フォールバックは、これまで XVC-over-PCIe でのみアクセス可能であったデバッグ ネットワークにアクセスするために設計された機能です。JTAG フォールバック機能は、プラットフォーム デザインの XVC-over-PCIe ベースのデバッグ ネットワークを変更せずにイネーブルにできます。

ホスト側では、Vivado® ユーザーが `hw_server` を介してテスト中のデバイス (DUT) の物理的な JTAG ピンに接続されている JTAG ケーブルに接続すると、`hw_server` が XVC-over-PCIe の DUT へのパスをディスエーブルにします。JTAG ケーブルへの接続を解除すると、`hw_server` は XVC-over-PCIe の DUT へのパスを再びイネーブルにします。

JTAG フォールバック手順

JTAG フォールバックをイネーブルにするには、次の手順に従います。

1. JTAG アクセスが必要なデバッグ ネットワークのデバッグ ブリッジ (AXI-to-BSCAN モード) マスターの JTAG フォールバック機能をイネーブルにします。これにより、このデバッグ ブリッジ インスタンスの BSCAN スレーブ インターフェイスがイネーブルになります。
2. プラットフォーム デザインのスタティック ロジック パーティションに別のデバッグ ブリッジ (BSCAN プリミティブ モード) をインスタンス化します。
3. 手順 2 のデバッグ ブリッジ (BSCAN プリミティブ モード) の BSCAN マスター ポートを手順 1 のデバッグ ブリッジ (AXI-to-BSCAN モード) の BSCAN スレーブ インターフェイスに接続します。

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 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) ものとします。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 2015-2019 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ックス社の商標です。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。