

SDAccel 環境デバッグ ガイド

UG1281 (v2018.2.xdf) 2018 年 10 月 2 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
2018年10月2日 バージョン 2018.2.xdf	
資料全体	xbsak を xbutil に変更。
2018年7月2日 バージョン 2018.2	
資料全体	マイナーな編集上の変更。
2018年6月6日 バージョン 2018.2	
全体的なアップデート	初版。

目次

改訂履歴.....	2
第 1 章: SDAccel™ でのデバッグの概要.....	4
SDAccel アプリケーションの実行モデル.....	4
SDAccel のビルド プロセス.....	6
SDAccel デバッグ フローの概要.....	7
第 2 章: SDAccel のデバッグ機能.....	9
SDAccel のソフトウェア デバッグ.....	9
ハードウェア デバッグのユーティリティ.....	15
ChipScope を使用したハードウェア デバッグ.....	16
第 3 章: デバッグ手法.....	21
論理検証 (ソフトウェア エミュレーション).....	21
ハードウェア エミュレーションでのデバッグ.....	25
システム検証、ハードウェアのハング.....	28
付録 A: その他のリソースおよび法的通知.....	40
ザイリンクス リソース.....	40
Documentation Navigator およびデザイン ハブ.....	40
参考資料.....	41
お読みください: 重要な法的通知.....	41

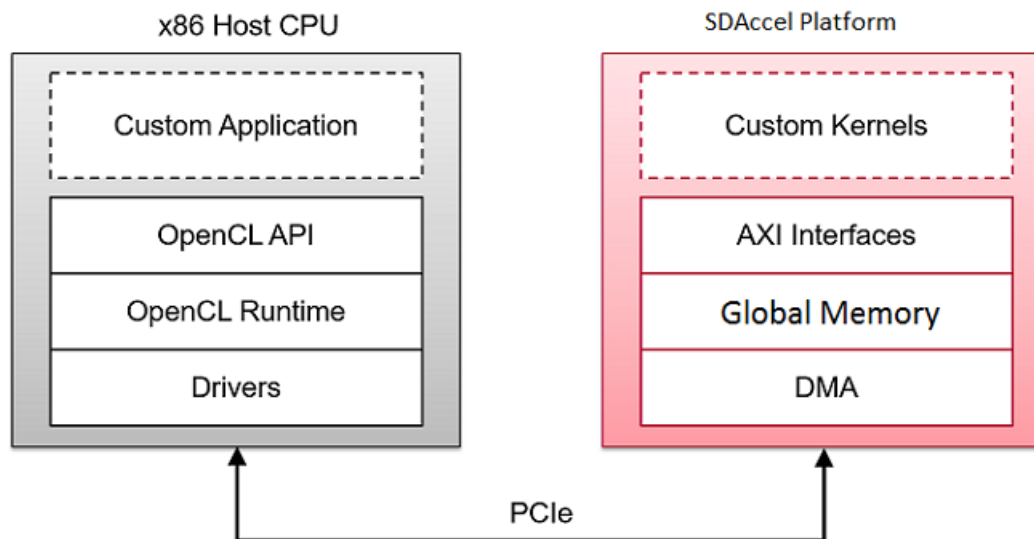
SDAccel™ でのデバッグの概要

この資料では、SDAccel™ のデバッグ機能について説明し、SDAccel フローで発生したエラーを解析する方法を詳細に説明します。ツールの問題がなく、デザインが正しく機能している場合は、『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、デザインのパフォーマンスに改善の余地があるかどうかを調べてみてください。

SDAccel アプリケーションの実行モデル

SDAccel™ 環境では、FPGA ベースのソフトウェア アクセラレーション プラットフォームを簡単に開発できます。次の図に、SDAccel の一般的な構造を示します。

図 1: SDAccel アプリケーションのアーキテクチャ



カスタム アプリケーションはホスト x86 サーバーで実行され、OpenCL™ API 呼び出しを使用して FPGA アクセラレータと通信します。これらの通信は、SDAccel ランタイムで管理されます。アプリケーションは OpenCL を使用して C/C++ で記述されます。カスタム カーネルは、ホスト アプリケーションとアクセラレータ間の通信を管理する SDAccel ランタイムを介して、サイリンクス FPGA 内で実行されます。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。

SDAccel ハードウェア プラットフォームには、グローバル メモリ バンクが含まれます。ホスト マシンとカーネルの間のデータ転送は、これらのグローバル メモリ バンクを介して実行されます。FPGA 上で実行されるカーネルには、1 つまたは複数のメモリ インターフェイスを含めることができます。メモリ バンクからこれらのメモリ インターフェイスへの接続はプログラム可能であり、コンパイラのリンク オプションにより決定されます。

SDAccel 実行モデルでは、次が実行されます。

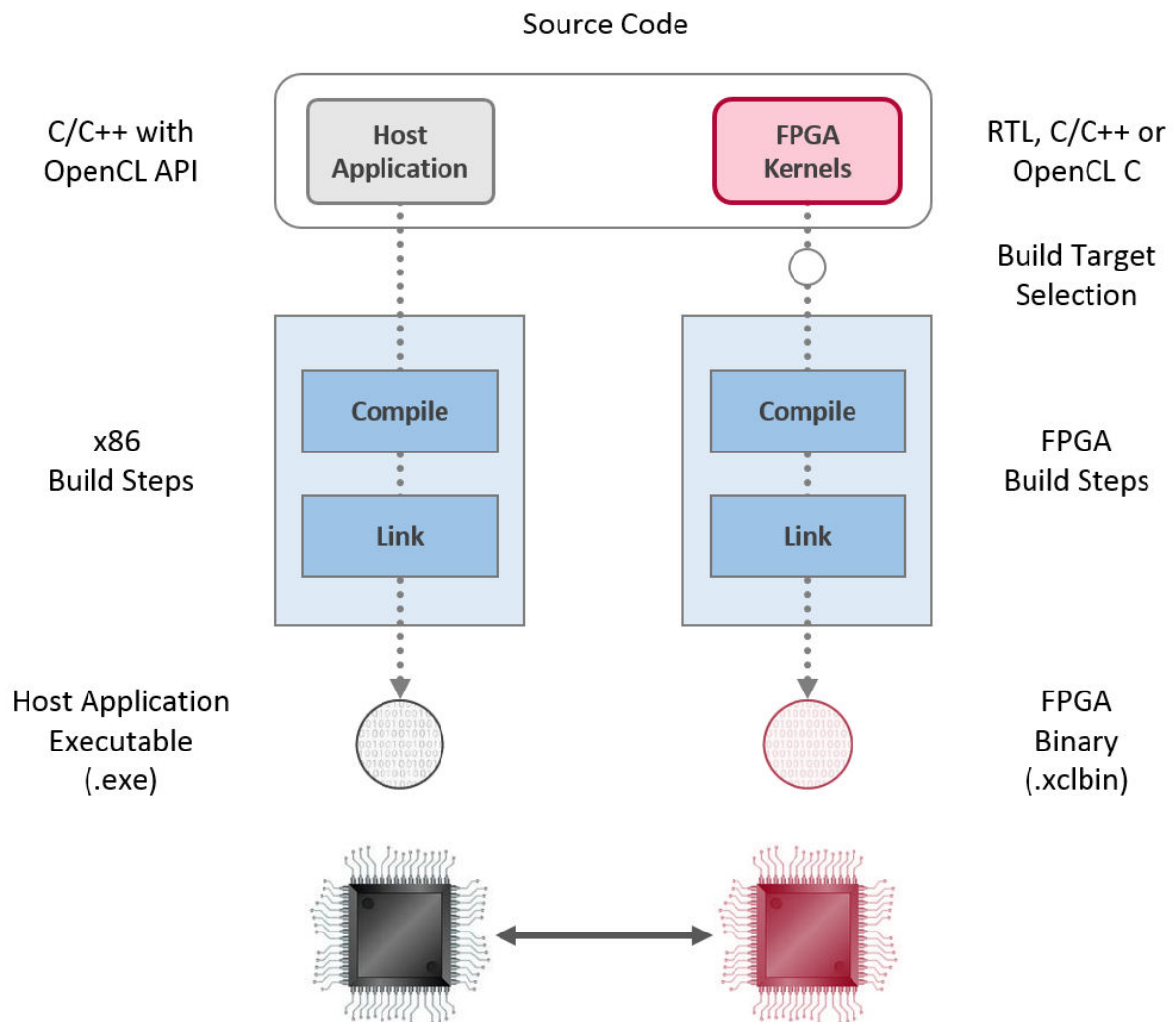
1. ホスト アプリケーションが PCIe を介して、カーネルで必要とされるデータを SDAccel ハードウェア プラットフォームのグローバル メモリに書き込みます。
2. ホストがカーネルをその入力パラメーターを使用してプログラムします。
3. ホスト アプリケーションが FPGA のカーネル関数の実行をトリガーします。
4. カーネルが必要な計算、グローバル メモリからのデータの読み出しおよび書き込みを必要に応じて実行します。
5. カーネルがメモリ バンクにデータを書き込み、ホストにタスクが終了したことを通知します。
6. ホスト アプリケーションがグローバル メモリからホスト メモリ空間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、SDAccel OpenCL ランタイムで管理されます。カーネルのインスタンス数は変数で、ホスト プログラムおよびコンパイル オプションにより決定されず。

SDAccel のビルド プロセス

SDAccel™ 環境には、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 用のクロスコンパイラ、コードの問題を特定して解決するための安定したデバッグ環境、ボトルネックを特定してコードを最適化するためのパフォーマンス プロファイラーなどの標準的なソフトウェア開発環境の機能がすべて含まれています。この環境内の SDAccel 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 を使用した 1 つのプロセスでビルドされ、FPGA バイナリはザイリンクス xocc コンパイラを使用した別のプロセスでビルドされます。

図 2: ソフトウェア/ハードウェアのビルド プロセス



1. GCC を使用したホスト アプリケーションのビルド プロセス:

- ホスト アプリケーションのソース ファイルをそれぞれオブジェクト ファイル (.o) にコンパイルします。
- オブジェクト ファイル (.o) をザイリンクス SDAccel ランタイム共有ライブラリとリンクし、実行ファイル (.exe) を作成します。

2. xocc を使用した FPGA ビルド プロセス:

- 各カーネルを個別にサイリンクス 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 でサポートされるのと同じ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 Kernel 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 ファイルをハードウェア プラットフォーム (.dsa) にリンクし、FPGA バイナリ (.xclbin) を作成します。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またはアーキテクチャ 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。

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 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 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_emulator): ホスト アプリケーション コードとカーネル コードの両方を x86 プロセッサで実行できるようコンパイルします。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、構文の問題を特定し、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 コードをハードウェア モデル (RTL) にコンパイルし、専用シミュレータで実行します。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 コードをハードウェア モデル (RTL) にコンパイルした後 FPGA デバイスにインプリメントし、実際の FPGA で実行されるバイナリを生成します。

SDAccel デバッグ フローの概要

このセクションには、実証済みの開発プロセスの一般的な手順を示して SDAccel™ の一般的なデバッグ フローについて説明します。デザインで発生する可能性のあるエラーをすばやく見つけられるようにします。これにより、開発中にエラーが発生した場合に、どこから始めればよいかのかわかるようになります。

ここで説明するデバッグ フローでは、SDAccel プラットフォーム ボードがインストールされており、初期のセットアップ チェックでは問題がなかったと想定しています。SDAccel 環境をカスタム ハードウェア プラットフォームを使用できるように設定することも可能です。これには、ボードの基本的なコンポーネントを定義するプラットフォーム DSA が必要です。

SDAccel™ 環境は、ホスト コードとカーネル コード、およびそれら 2 つのコード間の相互作用を効率的に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。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、ソフトウェア エミュレーション、ハードウェア エミュレーション、ハードウェア実行の 3 つのレベルがあります。

この 3 段階アプローチにより、ホストとカーネル コード、およびその相互作用を異なる抽象化レベルでデバッグできます。次に示す実行モデルは、SDAccel IDE でサポートされるほか、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たバッチ フローでもサポートされます。

SDAccel のデバッグ機能

この章では、デバッグをサポートする SDAccel™ 環境のさまざまな機能について説明し、プロジェクトの解析およびデバッグの実行に使用できるデバッグ ツールを紹介します。ここに示す機能を使用したデバッグ方法は、この次の章で説明します。

SDAccel のソフトウェア デバッグ

SDAccel™ では、ホストおよびカーネル コード用の典型的なソフトウェアのようなデバッグがサポートされます。このフローはソフトウェアおよびハードウェア エミュレーションでサポートされ、ソフトウェア デバッグでよく実行されるブレークポイントの使用、変数の解析などが可能です。

注記: 実際のハードウェアを実行している場合でも、ホスト コードはこのモードでデバッグできます。

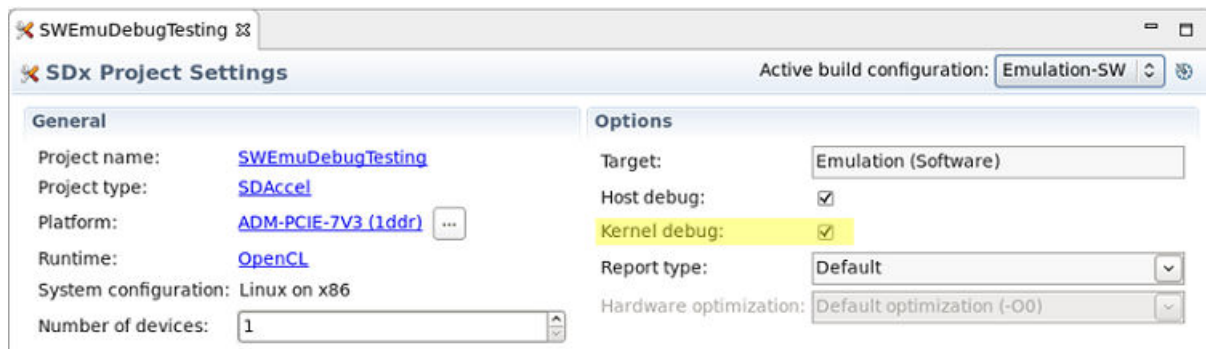
GUI デバッグ フロー

GUI フローで SDAccel™ を実行すると、デバッグ機能に簡単にアクセスできます。[コマンドラインでのデバッグフロー](#)に示すように、実行ファイルをデバッグ用に設定するには、多くの手動手順が必要です。GUI デバッグ フローを使用すると、これらの手順が GUI で処理されます。

注記: SDAccel デバッグ フローでは、デバッグに複数の異なるシェルが使用されます。これには、`.bashrc` または `.cshrc` などのシェル設定スクリプトが `LD_LIBRARY_PATH` などの SDAccel 設定と競合しないようにする必要があります。

デバッグ用に実行ファイルを準備するには、ビルド コンフィギュレーションを変更してデバッグ フラグを適用できるようにする必要があります。これらのオプションは、[SDx Project Settings] で設定できます。[Options] セクションに 2 つのチェック ボックスがあります。[Host debug] をオンにするとホスト デバッグ ビルドがイネーブルになり、[Kernel debug] をオンにするとカーネルのデバッグがイネーブルになります。

図 3: ソフトウェアのプロジェクト設定のオプション



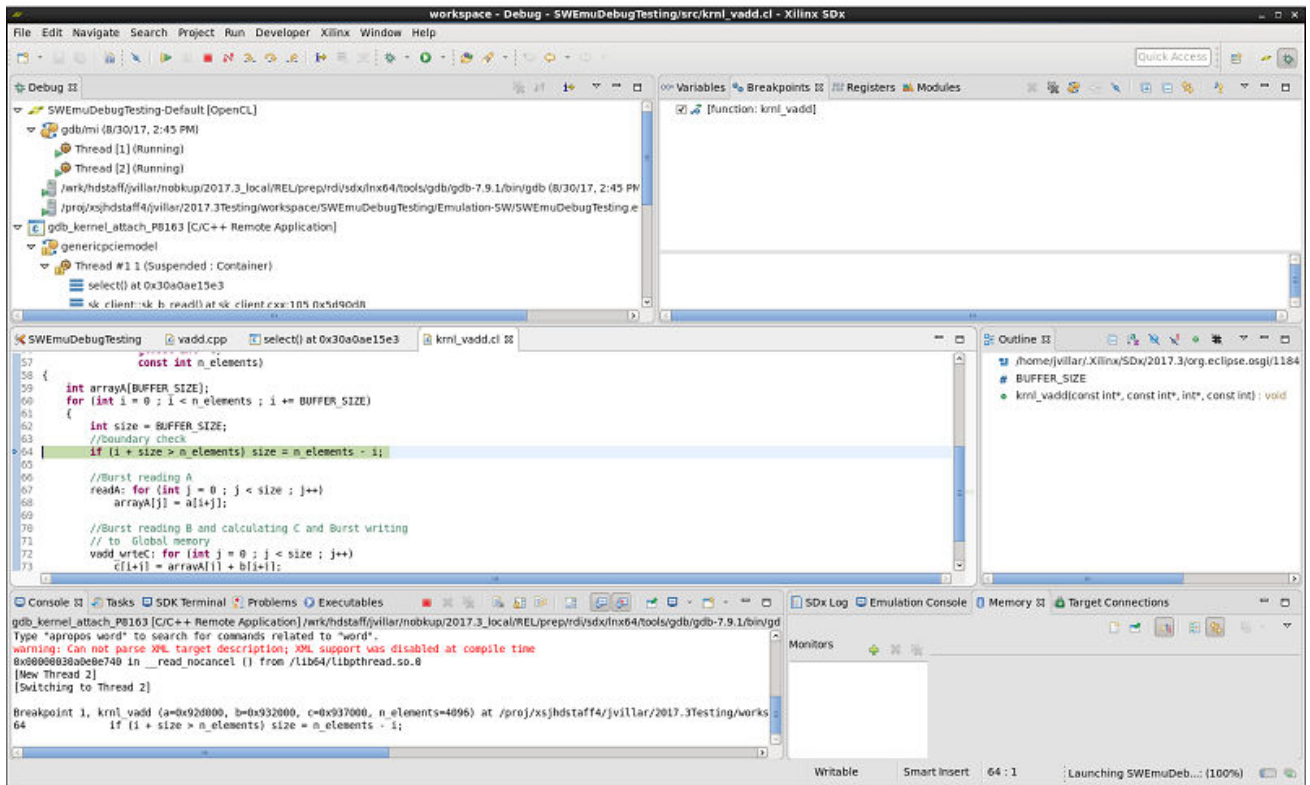
これらのビルド オプションを設定するには、右クリック メニューの [Settings] を使用する方がわかりやすいかも知れません。これには、[Assistant] ビューでビルド コンフィギュレーションを右クリックし、[Settings] をクリックします。または、ビルド コンフィギュレーションをダブルクリックします。同じ 2 つのチェック ボックスが表示されます。ホスト デバッグはすべてのターゲットでイネーブルにできますが、カーネル デバッグはソフトウェア エミュレーションおよびハードウェア エミュレーションビルド ターゲットでしかサポートされません。

これで設定は終了です。ビルド ディレクトリをクリーンアップしてアプリケーションをビルドし直すと、GDB デバッグ環境でプロジェクトを実行する準備が整います。

GUI から GDB セッションを実行すると、必要なものがすべて設定され、ハードウェアまたはソフトウェア エミュレーションの環境設定が自動的に管理されます。ランタイム環境でデバッグがサポートされるようにするため SDAccel ランタイムが設定され、カーネル モデル、ホスト モデル、およびデバッグ サーバーの実行に必要なさまざまなコンソールが管理されます。

デバッグ セッションを開始すると、SDAccel でデバッグ パースペクティブに切り替えるかどうかを尋ねるメッセージが表示されます。デバッグ パースペクティブには、デバッグ コンソールおよびソース コード ウィンドウなどを管理するウィンドウが複数含まれます。

図 4: GDB コンソール



アプリケーションを開始すると、デフォルトではホスト コードの `main` 関数本体の開始部分でアプリケーションが停止します。ほかの GDB グラフィカル フロントエンドと同様、ブレークポイントを設定し、変数を検証できます。SDAccel 環境では、アクセラレーションされたカーネル インプリメンテーションに対しても同じ機能を使用できます。

注記: ハードウェア エミュレーションでは、効率の良いインプリメンテーションのため C/C++/OpenCL™ カーネル コードが変換されるので、すべての文にブレークポイントを配置できるわけではありません。ほとんどの場合、未処理のループおよび関数には配置できます。また、保持されている変数にしかアクセスできません。

SDAccel™ インプリメンテーション構造に簡単にアクセスできるように設計された追加デバッグ機能については、[ザイリンクス OpenCL ランタイム GDB 拡張機能](#)を参照してください。

コマンド ラインでのデバッグ フロー

SDAccel™ 環境のコマンド ライン デバッグフローでは、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、およびハードウェアのすべてのモードで、実行されるホストおよびカーネル アプリケーションをデバッグするためのツールが提供されています。

注記: この機能を使用したホスト コードのデバッグは、ハードウェア実行モードでのみ可能です。

コマンド ライン フローを使用した SDAccel での デバッグには、次の 4 つの手順があります。

1. 一般的な環境設定
2. デバッグ用のホスト コードの準備
3. デバッグ用のカーネル コードの準備

4. デバッグ用の GDB をスタンドアロンで起動



重要: SDAccel 環境では、ホスト プログラムのデバッグはすべてのモードでサポートされますが、カーネルのデバッグは gdb を使用したエミュレーション フローでのみサポートされます。波形解析などのハードウェア中心のデバッグ サポートも、カーネル用に提供されています。

一般的な環境設定

コマンド ラインからソフトウェア エミュレーションまたはハードウェア エミュレーションを実行するには、次の環境変数を設定する必要があります。

表 1: 環境変数および値

環境変数	値
XCL_EMULATION_MODE	sw_emu または hw_emu この環境設定は、必要なエミュレーションを正しく実行するためにランタイム ライブラリで使用されます。特定のエミュレーション フローの実行ファイルをビルドするのに加え、これを設定する必要があります。
XILINX_SDX	SDx™ インストールへのパス。
XILINX_OPENCL	SDx インストールへのパス ({XILINX_SDX} と同じ)。
LD_LIBRARY_PATH	\${LD_LIBRARY_PATH}:\${XILINX_SDX}/lib/lnx64.o:\${XILINX_SDX}/runtime/lib/x86_64:\${XILINX_SDX}/lib/lnx64.o/Default

ホスト コードの準備

ホスト プログラムは、実行ファイルにより生成されたデバッグ情報を使用してコンパイルする必要があります。これには、次に示すように xcpp コマンド ラインに -g を追加します。

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



ヒント: xcpp はシステム コンパイラ (gcc) のラッパーなので、-g オプションを指定すると、コンパイラでデバッグ情報が生成されるようになります。

カーネルの準備

カーネル コードは、ソフトウェア エミュレーションまたはハードウェア エミュレーションでホスト プログラムと共にデバッグできます。デバッグ情報は、まず、xocc コマンド ラインに -g オプションを渡して、バイナリコンテナで生成する必要があります。

```
xocc -g -t [sw_emu | hw_emu | hw] ...
```

-t (または -target) オプションを使用してコンパイル ターゲットをソフトウェア エミュレーション (sw_emu)、ハードウェア エミュレーション (hw_emu)、またはハードウェア実行 (hw) のいずれかに指定します。

ソフトウェア エミュレーション フローでは、OpenCL™ ベースのカーネルに対して追加のランタイム チェックを実行できます。ランタイム チェックには、次のものが含まれます。

- カーネル インターフェイス バッファによる範囲外アクセスをチェック (オプション: address)
- カーネル ローカルからカーネルに対して初期化されていないメモリへのアクセスがあったかどうかをチェック (オプション: memory)

次の例に示すように、これらのオプションは --xp オプションおよび param:compiler.fsanitize 指示子を使用して指定し、リンク段階 (-l) でイネーブルにする必要があります。

```
xocc -l -t sw_emu --xp param:compiler.fsanitize=address -o bin_kernel.xclbin
xocc -l -t sw_emu --xp param:compiler.fsanitize=memory -o bin_kernel.xclbin
xocc -l -t sw_emu --xp param:compiler.fsanitize=address,memory -o
bin_kernel.xclbin
```

オプションをイネーブルにすると、エミュレーションを実行したときにエミュレーション診断メッセージを含むデバッグログが <project_dir>/Emulation-SW/<proj_name>-Default>/emulation_debug.log に出力されます。

GDB ホスト コード デバッグの開始

コードをデバッグ情報を含めてビルド (-g オプションを使用) している場合、GDB スタンドアロンを起動してホストプログラムをデバッグできます。このフローは、GNU から入手可能な DDD (Data Display Debugger) などの GDB 用のグラフィカル フロントエンドを使用した場合でも使用できます。ホストおよびカーネル コードを同時にデバッグする場合は、[ホストおよびカーネル デバッグの開始](#)を参照してください。

GDB を起動するには、次の手順を実行します。

- SDx を実行する環境を設定するため、次のファイルを source コマンドで読み込んで SDx コマンド設定が PATH に含まれるようにします。
 - C シェル: `source <SDX_INSTALL_DIR>/settings64.csh`
 - Bash: `source <SDX_INSTALL_DIR>/settings64.sh`
- XCL_EMULATION_MODE 環境変数が正しいモードに設定されていることを確認します。
- アプリケーション デバッグ機能は sdaccel.ini ファイルの属性を使用してランタイム時にイネーブルにする必要があります。ホスト実行ファイルと同じディレクトリに sdaccel.ini ファイルを作成し、次の行を含めます。

```
[Debug]
app_debug=true
```

これにより、カーネルがデバッグ イネーブルになったことがランタイム ライブラリで認識されるようになります。

- ザイリンクス ラッパーから gdb を開始します。

```
xgdb --args host.exe test.xclbin
```

xgdb ラッパーで次の設定が実行されます。

- ホストプログラムで GDB を起動します。

```
gdb --args host.exe test.xclbin
```

- Python インストールに PYTHONHOME および PYTHONPATH 環境変数を設定します。現時点では、SDx 環境の gdb には Python 2.6 または Python 2.7 が必要です。たとえば、コンピューターにインストールされている Python が Python 2.6 の場合は、次のように環境変数を設定します (Bash シェルの例)。

```
export PYTHONHOME=/usr
export PYTHONPATH=/usr/lib64/python2.6/:/usr/lib64/python2.6/lib-
dynload/
```

- GDB コンソールで Python スクリプトを実行し、ザイリンクス GDB 拡張機能をイネーブルにします。

```
gdb> source ${XILINX_SDX}/scripts/appdebug.py
```

ホストおよびカーネル デバッグの開始

ソフトウェア エミュレーションでは、エミュレートされるハードウェアをよりうまく模倣できるように、カーネルが個別のプロセスとして生成されます。ホスト コードのデバッグに GDB を使用している場合は、カーネル コードがそのプロセス内で実行されないで、カーネル行に設定されているブレークポイントでは停止しません。ホスト コードおよびカーネル コードの同時デバッグをサポートするため、SDAccel 環境には、`sdx_server` を使用して生成されたカーネルに接続するメカニズムがあります。

コマンド ライン フローで、3 つのターミナルを実行する必要があります。

1. 最初のターミナルでは、次のコマンドを実行して `sdx_server` を起動します。

```
${XILINX_SDX}/bin/sdx_server --sdx-url
```

2. 2 つ目のターミナルでは、[GDB ホスト コード デバッグの開始](#)で説明されているように、`xgdb` でホスト コードを実行します。

この時点で、`sdx_server` を実行している最初のターミナルに「GDB listener port NUM」と表示されます。カーネル プロセスをデバッグするのに GDB で GDB リスナー ポートが使用されるので、`sdx_server` で返される数値を継続的にチェックします。GDB リスナー ポートが表示されると、生成されたカーネル プロセスが `sdx_server` に接続され、ユーザーからのコマンド入力を待機します。このプロセスを制御するには、GDB の新しいインスタンスを開始し、`sdx_server` に接続する必要があります。



重要: `sdx_server` が実行中の場合は、デバッグ用にコンパイルされているすべての生成プロセスが接続され、ユーザーからの指示を待機します。GDB が接続されない場合、またはコマンドを供給しない場合は、カーネル コードがハングしているように見えます。

3. 3 つ目のターミナルでは、`xgdb` コマンドを実行し、GDB のプロンプトで次のコマンドを実行します。

- ソフトウェア エミュレーションの場合:

```
"file ${XILINX_SDX}/data/emulation/unified/cpu_em/generic_pcie/model/genericpciemodel"
```

- ハードウェア エミュレーションの場合:

1. `sdx_server` の一時ディレクトリ `/tmp/sdx/$uid` を検索します。
2. このデバッグ セッションの DWARF ファイルを含む `sdx_server` プロセス ID (PID) を検索します。
3. `gdb` コマンド ラインで `file /tmp/sdx/$uid/$pid/NUM.DWARF` を実行します。

- どちらのエミュレーションでも、カーネル プロセスに接続します。

```
target remote :NUM
```

NUM は、GDB リスナー ポートとして `sdx_server` から返される数値です。



ヒント: SDAccel 環境の GUI でソフトウェア/ハードウェア エミュレーションのカーネルをデバッグする場合は、ホスト コードおよびカーネル コードを同時にデバッグするための環境が用意されていれば、これらの手順が自動的に処理され、カーネル プロセスも自動的にデバッグされます。

これらのコマンドを実行したら、必要に応じてカーネルにブレークポイントを設定し、`continue` コマンドを実行してカーネル コードをデバッグできます。すべてのカーネル実行が完了しても、ホスト コードは続行し、`sdx_server` 接続は解除されます。

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) に変換されてデバッグ中にシミュレーションされるので、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。特にハードウェア エミュレーションでは、保持されるループおよび関数など、限られた数のブレークポイントしかサポートされません。このような制限はありますが、カーネル/ホスト インターフェイスのデバッグにはこのモードが便利です。

ハードウェア デバッグのユーティリティ

場合によっては、通常の IDE およびコマンドライン デバッグ機能では問題を見つけられないことがあります。これは特に、ソフトウェア/ハードウェアが何も進捗していない(ハングしている)ように見える場合です。このようなシステム問題は、このセクションで説明するユーティリティを使用して解析することをお勧めします。

Linux dmesg の使用

適切に設計された Linux カーネルおよびモジュールでは、カーネル リング バッファを介して問題がレポートされます。これは SDAccel™ モジュールの場合も同様で、アクセラレータ ボードと通信する最下位 Linux レベルでのデバッグが可能になります。

注記: このユーティリティは、ハードウェア デバッグでのみ使用してください。



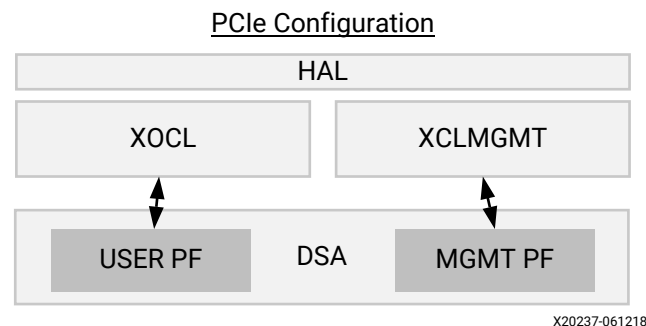
ヒント: ほとんどの場合、詳細度の低い `xbutil` 機能でも問題を検出するには十分です。`xbutil` コマンドの詳細は、『SDx コマンドおよびユーティリティ リファレンス ガイド』(UG1279)を参照してください。

`dmesg` ユーティリティは、カーネル リング バッファを読み出すための Linux ツールです。カーネル リング バッファでは、循環バッファにカーネル情報メッセージが格納されます。リソース要件を制限するため固定サイズの循環バッファが使用されており、1 番古いエントリが次に入ってくるメッセージで上書きされます。

SDAccel ツールでは、情報メッセージは `xocl` モジュールおよび `xclmgmt` ドライバー モジュールによりリング バッファに書き込まれます。このため、アプリケーションのハングやクラッシュが発生したり、予期しない動作(ビットストリームをプログラムできないなど)が見られた場合は、`dmesg` ツールを使用してリング バッファを確認する必要があります。

次の図に、SDAccel ボード プラットフォームに関連付けられたソフトウェア プラットフォームのレイヤーを示します。

図 5: ソフトウェア プラットフォーム レイヤー



Linux ツールからのメッセージを確認するには、まずリング バッファをクリアする必要があります。

```
sudo dmesg -c
```

これにより、すべてのメッセージがリング バッファから消去されるので、`xocl` および `xclmgmt` からのメッセージを見つけやすくなります。その後、アプリケーションを開始して `dmesg` を別のターミナルで実行します。

```
sudo dmesg
```

`dmesg` は、次のモジュール レポートなどの記録を表示します。

```
[ 9902.316729] xclmgmt: AXI Firewall 2 has tripped. Status: 0x00000
[ 9902.316874] xclmgmt: xclmgmt_killall_processes
[ 9902.317007] xclmgmt: Killing pid: 19891
[ 9902.317501] xocl:xdma_xfer_submit: xfer 0xffff8801c1be1018,268435456, s 0x1 timed out, ep 0x10000000.
[ 9902.317911] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e0000) = 0x1fc00006 (id).
[ 9902.318410] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e0040) = 0x00000001 (status).
[ 9902.318895] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e0004) = 0x00f83e1f (control)
[ 9902.319370] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e4080) = 0xa7a30000 (first_desc_lo)
[ 9902.319848] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e4084) = 0x00000000 (first_desc_hi)
[ 9902.320336] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e4088) = 0x0000000f (first_desc_adjacent).
[ 9902.320802] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e0048) = 0x00000000 (completed_desc_count).
[ 9902.321279] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e0090) = 0x00f83e1e (interrupt_enable_mask)
[ 9902.321759] xocl:engine_status_dump: SG engine 0-H2C0-MM status: 0x00000001: BUSY
[ 9902.322233] xocl:transfer_abort: abort transfer 0xffff8801c1be1018, desc 240, engine desc queued 0.
[ 9902.322752] [drm:xdma_migrate_bo [xocl]] *ERROR* DMA failed to device addr 0x0, tid 19897, channel 0
[ 9902.323232] [drm:xdma_migrate_bo [xocl]] *ERROR* Dumping SG Page Table
```

上記の例の場合、AXI Firewall 2 が作動しているので、`xbutil` を使用して検証することをお勧めします。

ザイリンクス `xbutil` の使用

ザイリンクス ボード ユーティリティ (`xbutil`) は、下位レベルのハードウェア/ソフトウェア間の相互作用の問題をデバッグするのに使用可能なスタンドアロンのコマンドライン ユーティリティです。このパラメーターの詳細は、『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。

デバッグには、次の `xbutil` オプションを使用できます。

- `query`: SDAccel™ プラットフォームの全体的なステータスを表示します。
- `program`: バイナリ (`xclbin`) をザイリンクス デバイスのプログラマブル領域にダウンロードします。
- `status`: SDx™ パフォーマンス モニター (`spm`) および軽量 AXI プロトコル チェッカー (`lapc`) のステータスを表示します。

ChipScope を使用したハードウェア デバッグ

最終的なシステム イメージ (`xclbin`) を生成して SDAccel™ プラットフォーム上で実行したら、CPU で実行されるホスト アプリケーションとザイリンクス FPGA でアクセラレーションされたカーネルを含むシステム全体が、実際のハードウェアで正しく実行されるかどうかを確認できます。この段階では、ホスト コードおよびカーネルの機能をターゲット ハードウェアで検証し、検出された問題をデバッグできます。

確認または解析する必要のある問題には、次のものがあります。

- プロトコル違反によるシステム ハングのデバッグ。
 - これらの違反により、システム全体が停止することがあります。

- これらの違反により、カーネルが無効なデータを取得したり、停止することがあります。
- これらの違反がどこでいつ発生しているかを検出するのは困難です。
- このような状況をデバッグするには、AXI プロトコル チェッカーから ILA をトリガーする必要がある場合があります。これは、使用中の SDAccel プラットフォームで設定する必要があります。
- RTL カーネル内の問題のデバッグ。
 - これらは、タイミング問題、レース コンディション、無効なデザイン制約など、インプリメンテーションにより発生することがあります。
 - ハードウェア エミュレーションで表示されなかった論理的な問題。
- パフォーマンス問題のデバッグ。
 - たとえば、フレーム/秒の処理が予測と異なる場合。
 - データ ビートとパイプライン処理を確認できます。
 - トリガー シーケンサー付きの ILA を使用して、バースト サイズ、パイプライン処理、データ幅を確認し、ボトルネックを特定します。

ハードウェア デバッグ サポートのための FPGA ボードのチェック

ハードウェア デバッグをサポートするには、プラットフォームで複数の IP コンポーネント (Debug Bridge など) がサポートされるようにする必要があります。プラットフォーム設計者に、これらのコンポーネントがプラットフォーム DSA に含まれるかどうかを確認してください。

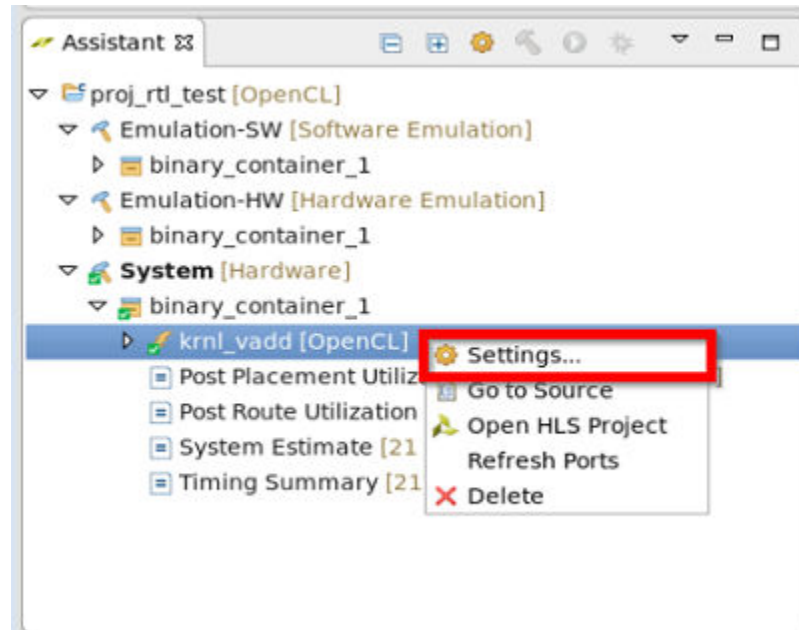
ザイリックスの提供するプラットフォームでは、ハードウェア デバッグが完全にサポートされます。

- xilinx_kcu1500_dynamic_5_0
- xilinx_vcu1525_dynamic_5_1

SDx GUI の ChipScope の有効化

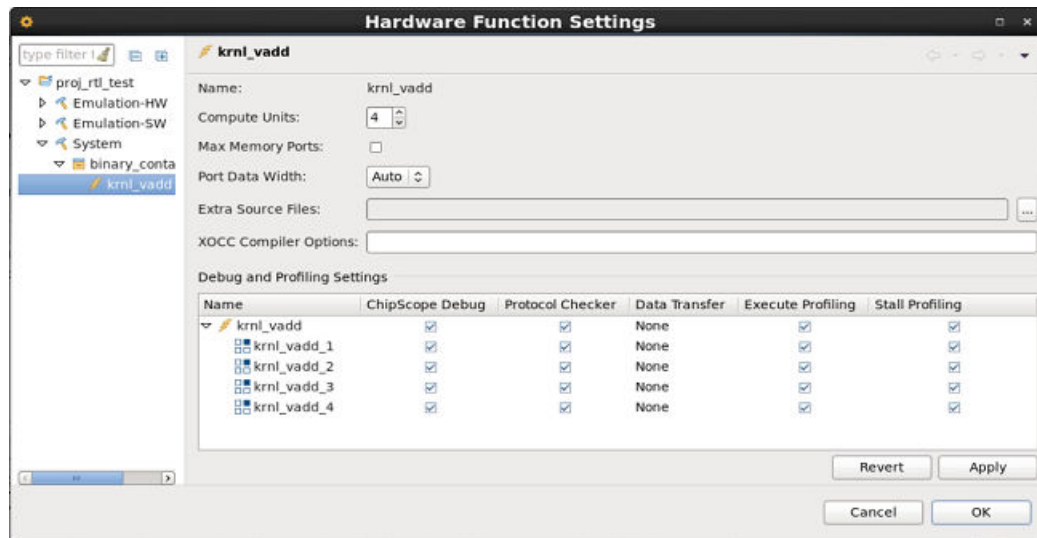
SDx™ GUI には、デザインの計算ユニットのインターフェイスポートで ChipScope デバッグを有効にするオプションがあります。計算ユニットでこのオプションをイネーブルにすると、SDAccel™ コンパイラにより計算ユニットのインターフェイスポートを監視するため System ILA デバッグ コアが追加されます。これにより、カーネルの実行中に SDAccel プラットフォーム ハードウェア上のインターフェイス信号をデバッグできます。このオプションにアクセスするには、[Assistant] ビューでシステムビルドコンフィギュレーションのカーネルを右クリックし、[Settings] をクリックします。

図 6: SDx™ の [Assistant] ビュー



次の図に示す [Hardware Function Settings] ダイアログ ボックスが開きます。このダイアログ ボックスの [Debug and Profiling Settings] の表で、カーネルの特定の計算ユニットの [ChipScope Debug] チェック ボックスをオンにします。その計算ユニットのインターフェイス/ポートの監視が有効になります。

図 7: SDx [Hardware Function Settings] ダイアログ ボックス



ヒント: 複数のカーネルまたは計算ユニットを含む大型のデザインで [ChipScope Debug] をオンにすると、FPGA デバイス リソースが過剰に使用される可能性があります。サイリンクスでは、コマンド ラインで `xocc --dk list_ports` オプションを使用して計算ユニット上のインターフェイスの数とタイプを確認することをお勧めします。デザインをハードウェアで実行中にデバッグ目的で監視する必要のあるポートがわかっている場合は、[コマンド ライン フロー](#) に説明されている `--dk` オプションを使用する方法が推奨されます。

コマンド ライン フロー

System ILA デバッグ コアを使用すると、トランザクション レベルでアクセラレーションされたカーネルやハードウェアで実行される関数などを表示できます。System ILA コアでは、特定の AXI トラフィックをキャプチャして表示させることもできます。

ILA コアは、既存の RTL IP デザインのハードウェア全体にインスタンスエートして、そのデザイン内でデバッグ機能をイネーブルにしたり、コンパイラで自動的に挿入されるようにしたりできます。XOCC コンパイラには、デバッグおよびパフォーマンス監視用にインターフェイスで System ILA コアをカーネルに接続するための `--dk` オプションがあります。

次の構文に示すように、`--dk` オプションを使用して ILA IP コアの挿入をイネーブルにします。

```
--dk <[chipscope|list_ports]<:compute_unit_name><:interface_name>>
```

通常、`<interface_name>` の使用はオプションです。指定しない場合、すべてのポートが解析されます。

`<compute_unit_name>` および `<interface_name>` に対して `chipscope` オプションを使用する場合、計算ユニットに名前を指定する必要があります。

`list_ports` オプションは、現在のデザインの有効な計算ユニットとポート組み合わせのリストを生成します。

注記:

コマンド ライン 1 行に複数の `--dk` オプションを含めて、インターフェイス モニター機能を追加していきます。

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。

デザインをビルドしたら、『Vivado Design Suite ユーザー ガイド: プログラムおよびデバッグ』 ([UG908](#)) に示すように、ハードウェア マネージャーを使用してデザインをデバッグできます。

デバッグ手法

このセクションでは、さまざまなデバッグ手法について説明します。ソフトウェアベースのデバッグ手法とハードウェア重視の手法に分類されています。ソフトウェアベースの手法では、FPGA への最終的なカーネルコードのマップを理解する必要はありません。ただし、この概念が適用できるのはあるレベルの詳細度までで、それ以上はハードウェアベースの詳細な解析を実行する必要があります。

このセクションでは、SDAccel™ の異なるデバッグ段階を順に説明していきます。まず、ソフトウェアエミュレーションでの論理検証です (純粋なソフトウェアベースの手法)。その次はハードウェアエミュレーションで、カーネルコードが実際のハードウェア記述に変換され、最終的なインプリメンテーションの詳細が示されます。ハードウェアデバッグおよびソフトウェアデバッグの概念は、ハードウェアエミュレーション段階のデバッグに適用できます。最後の段階はシステム検証で、実際のハードウェアが実行されます。この段階では、ソフトウェアデバッグの概念を適用できるのはホストのみで、カーネルにはハードウェアデバッグの概念を適用する必要があります。

論理検証 (ソフトウェアエミュレーション)

論理検証は、システムを記述したソフトウェアを最終的なインプリメンテーション目標に従って検証するプロセスです。ソフトウェアが指定したデータで想定どおりに動作するかをこの段階で確認します。これはソフトウェア開発では一般的なタスクで、さまざまな概念があります。

ソフトウェアが想定どおりに動作しない場合は、デバッガーを使用して問題の原因を突き止め、必要であればソフトウェア実行中にデータポイントをダンプします。このセクションでは、SDx™ プロジェクトに適用されるこれらの概念を紹介します。

カーネルデバッグでの printf() の使用

アルゴリズムをデバッグするには、プログラムを実行して主なデータ値を検証するのが、最もシンプルで基本的な方法です。アプリケーション開発では、コード内のチェックポイント値を表示するのが、プログラム実行内で問題を特定するのに実証済みの方法です。この段階ではアルゴリズムの一部が FPGA 上で実行されているので、このデバッグ方法でも追加のサポートが必要です。

SDAccel™ 開発環境では、すべての開発フロー (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、実際のハードウェアでのカーネルの実行) で OpenCL™ の printf() ビルトイン関数がサポートされています。

次に、カーネルで printf() を使用する例と、カーネルを global サイズ 8 で実行した場合の出力を示します。

```
__kernel __attribute__((reqd_work_group_size(1, 1, 1)))
void hello_world(__global int *a)
{
    int idx = get_global_id(0);

    printf("Hello world from work item %d\n", idx);
    a[idx] = idx;
}
```

出力は次のようになります。

```
Hello world from work item 0
Hello world from work item 1
Hello world from work item 2
Hello world from work item 3
Hello world from work item 4
Hello world from work item 5
Hello world from work item 6
Hello world from work item 7
```



重要: `printf()` メッセージはグローバルメモリに格納され、カーネル実行が終了するとアップロードされます。`printf()` を複数カーネルで使用した場合、各カーネルからのメッセージがどの順でホストターミナルに表示されるかはわかりません。特にハードウェアエミュレーションおよびハードウェアで実行している場合、ハードウェアバッファサイズによりキャプチャされる `printf` 出力の量は制限されます。

この機能は、すべての開発フローで OpenCL カーネルでのみサポートされることに注意してください。

C/C++ カーネルモデルに対しては、`printf()` はソフトウェアエミュレーションでのみサポートされるので、Vivado® HLS 合成から除外されるようにする必要があります。この場合、`printf()` 文を次のコンパイラマクロで囲みます。

```
#ifndef __SYNTHESIS__
    printf("text");
#endif
```

GDB ベースのデバッグ

このセクションでは、GDB を使用してホストおよびカーネルデバッグを実行する方法を説明します。このフローはソフトウェア開発者には慣れたフローであるはずなので、このセクションでは FPGA 用のホストコードのデバッグ機能とカーネルベースのハードウェアエミュレーションサポートの現状について説明します。

ホストコードのデバッグ

前の章で説明したデバッグ環境の起動方法を除き、SDAccel™ ホストコードのデバッグと一般的な GDB アプリケーションのデバッグフローおよび機能に違いはありません。

`gdb` を起動したら GDB でコードをステップ実行し、C/C++/OpenCL™ オブジェクトを調べてコードのどの地点でも内容が正しいことを確認できます。

ただし、概要で説明したように、特にハードウェアエミュレーションでは、ホストとカーネル間のプロトコル同期に関する問題を探るのが一般的です。SDAccel 環境には、アプリケーションホストから 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。これらのコマンドの詳細を、次のセクションで説明します。

ザイリンクス OpenCL ランタイム GDB 拡張機能

ザイリンクス OpenCL™ ランタイムデバッグ環境には、ホストアプリケーションから OpenCL ランタイムライブラリを視覚化する新しい GDB コマンドが含まれます。

注記: SDAccel™ 環境外で GDB を実行する場合は、[GDB ホスト コード デバッグの開始](#)で説明されているように、これらのコマンドをイネーブルにする必要があります。

gdb コマンド ラインから、次の 2 種類のコマンドを呼び出すことができます。

- OpenCL ランタイムのデータ構造を可視化するためコマンド (cl_command_queue、cl_event、および cl_mem)。xprint queue および xprint mem への引数はオプションです。アプリケーションのデバッグ環境には、すべての OpenCL オブジェクトが記録され、引数が指定されない場合は有効なすべてのキューおよび cl_mem オブジェクトが自動的に出力されます。さらに、これらのコマンドは、指定されているコマンドの queue、event、cl_mem 引数を検証します。

```
xprint queue [<cl_command_queue>]
xprint event <cl_event>
xprint mem [<cl_mem>]
xprint kernel
xprint all
```

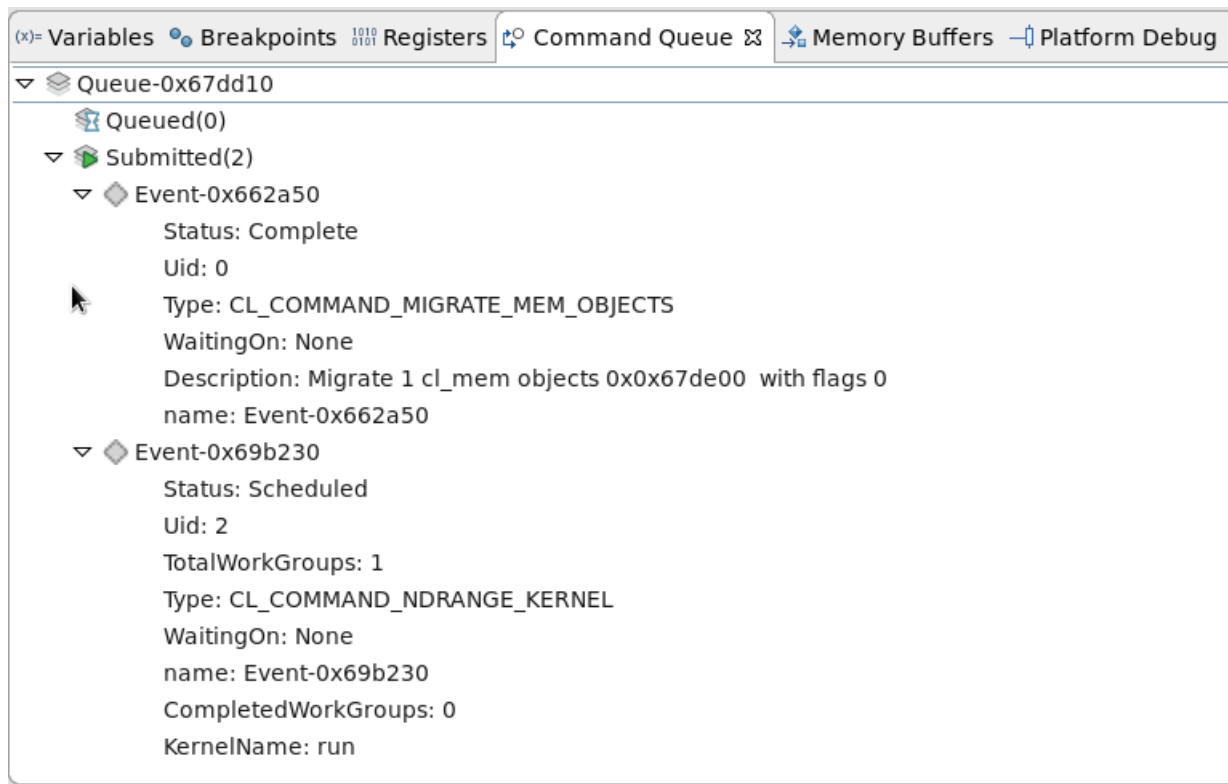
- SDAccel プラットフォームの IP を可視化するコマンド。この機能はシステム フロー (ハードウェア実行) でのみ使用でき、エミュレーション フローでは使用できません。

```
xstatus all
xstatus --<ipname>
```

コマンドの詳細な情報を入手するには、`help <command>` を使用します。

これらのコマンドは、ホストアプリケーションがハングした場合によく使用されます。この場合、ホストアプリケーションがコマンドキューが終了するのを待っているか、またはイベントリストを待機していると考えられます。`xprint` コマンドを使用してコマンドキューを表示すると、どのイベントが終了していないのかわかり、イベント間の依存性を解析できます。

これら両方のコマンドの出力は、SDAccel GUI を使用してデバッグすると自動的に表示されます。[Debug] パースペクティブの左上にある [Variables]、[Breakpoints]、[Registers] という3つのビューの横に、[Command Queue]、[Memory Buffers]、[Platform Debug] というビューが表示され、それぞれ `xprint queue`、`xprint mem`、および `xstatus` の出力を示します。



注記: これらのビューの情報は、ホストコードを実際にデバッグ中の場合にのみ表示されます。このため、このデバッグ手法は、実際のシステム実行 (ハードウェア) でも使用できます。

GDB カーネル ベースのデバッグ

GDB カーネル デバッグは、ソフトウェア エミュレーション フローおよびハードウェア エミュレーション フローでサポートされます。[コマンドラインでのデバッグフロー](#)に示すように、通常のホストコードのデバッグと同様、GDB 実行ファイルが GUI またはコマンドラインフローでカーネルに接続されたら、ブレークポイントを設定し、カーネルの変数の内容をクエリします。

カーネル GDB プロセスは生成されたソフトウェア プロセスに接続するだけなので、これはソフトウェア エミュレーションで完全にサポートされます。

一方、ハードウェア エミュレーションでは、Vivado® HLS でカーネル コードが RTL に変換されてから実行されます。RTL モデルのシミュレーションでは、パフォーマンス最適化および同時ハードウェア実行のすべての変換が適用されます。このため、すべての C/C++/OpenCL™ 行が独自に RTL コードにマップされるわけではないので、一部のブレークポイントのみがサポートされ、特定の変数のみをクエリ可能です。GDB ツールは、要求されたブレークポイント文に基づいて次の可能な行で停止し、RTL 変換のため変数をクエリできない場合はそれを明確に示します。

ハードウェア エミュレーションでのデバッグ

ハードウェア エミュレーションでは、カーネルのインプリメンテーションを詳細に検証できます。SDAccel™ では、このモードで典型的なハードウェアと同様のデバッグが実行できるほか、ハードウェア インプリメンテーションの解析に基づいてソフトウェアと同様の GDB を実行することもできます。

GDB ベースのデバッグ

ソフトウェア ベースの GDB フローを使用したデバッグは、ハードウェア エミュレーションで完全にサポートされています。GDB フローでは RTL がソース コード記述にマップされるので、カーネル コードを表す実際の RTL コードを実行するという点以外は、ユーザーにとっては違いはありません。ただし、RTL 生成 (HLS) 中に変数およびループが分解されることがあり、ブレークポイントの設定および変数の確認が制限される場合があります。

デバッグ機能の詳細は、[第2章: SDAccel のデバッグ機能](#) を参照してください。GDB の拡張機能については、[GDB ベースのデバッグ](#) を参照してください。

波形ベースのカーネル デバッグ

C/C++ および OpenCL™ カーネル コードは Vivado® 高位合成 (HLS) を使用して合成され、ハードウェア記述言語 (HDL) に変換され、FPGA (xclbin) にインプリメントされます。

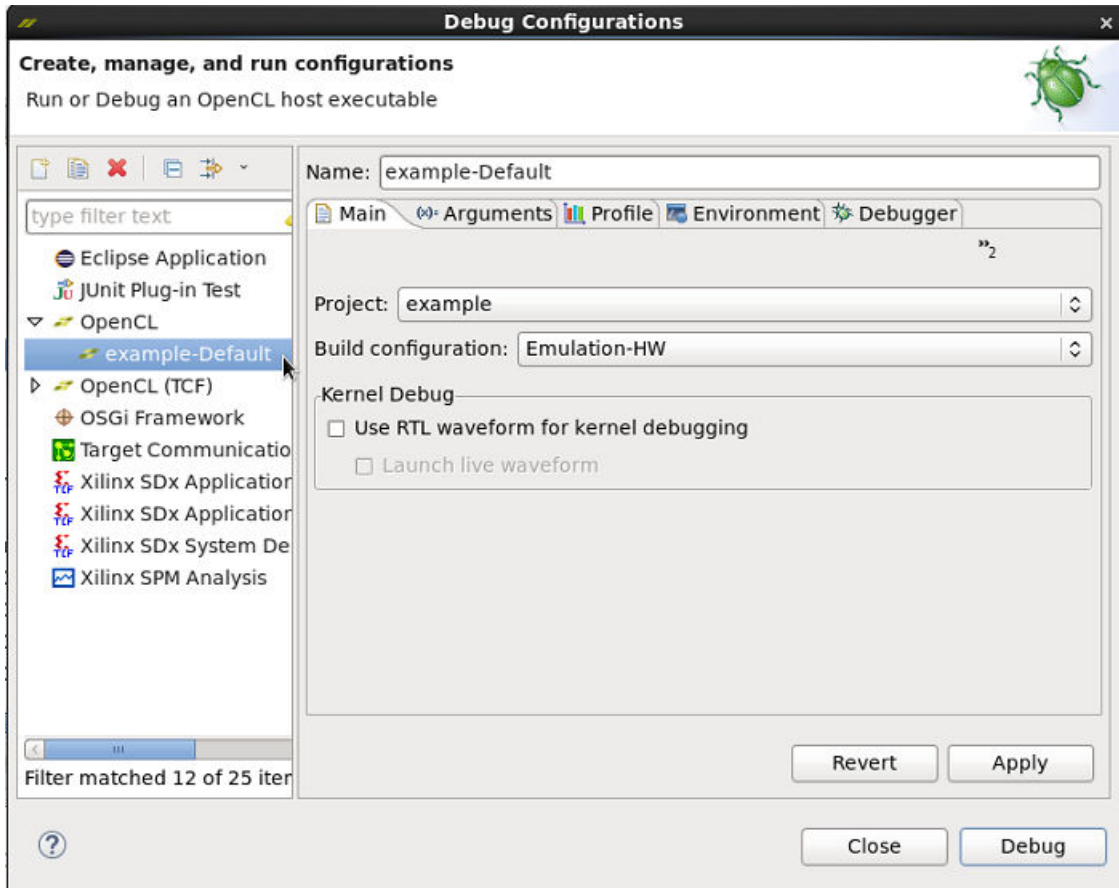
もう1つのデバッグ手法は、シミュレーション波形に基づいており、ハードウェア中心のアルゴリズムのプログラマにとっては、熟知した手法である可能性があります。この波形ベースの HDL デバッグは、ハードウェア エミュレーションで SDAccel™ GUI フローを使用して実行するのが最適です。



ヒント: ほとんどのデバッグで、HDL モデルを解析する必要はありません。波形デバッグは、アドバンス デバッグ機能です。

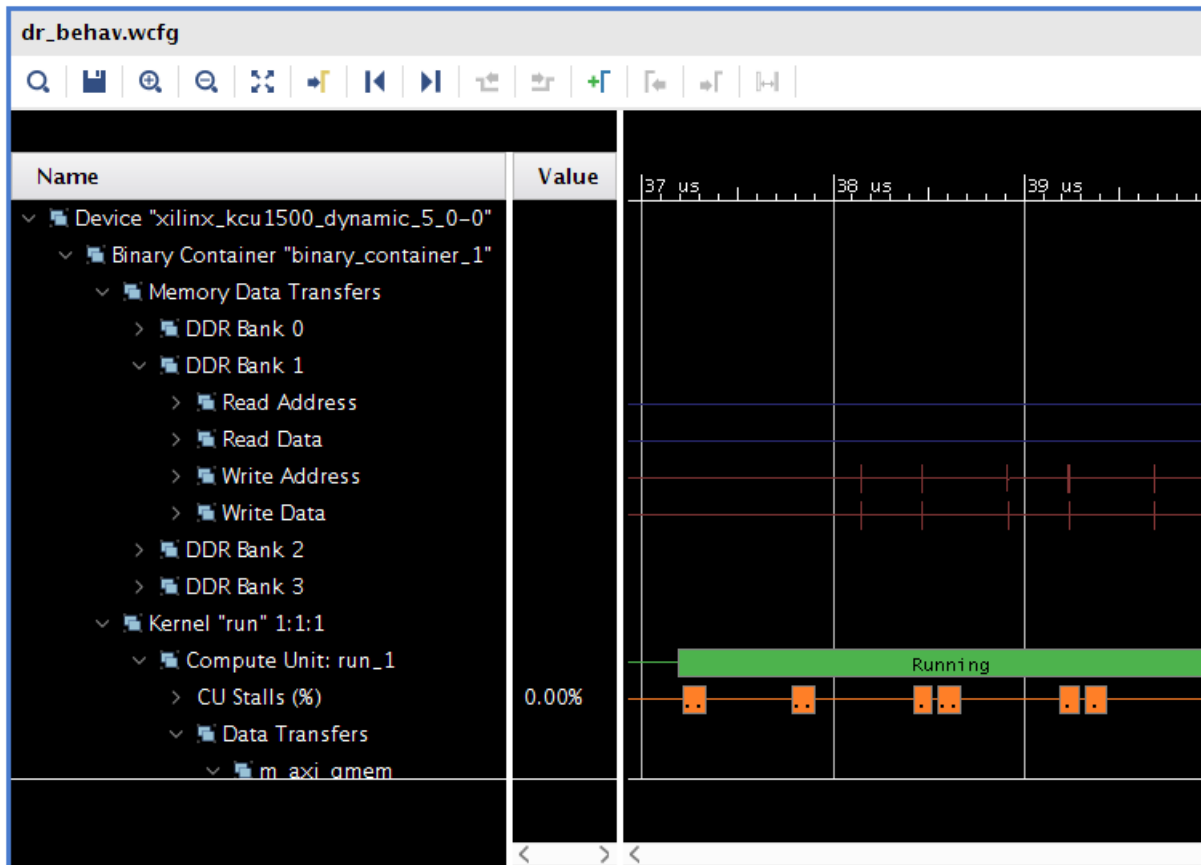
このフローを実行するには、次の手順を実行します。

1. SDx™ を起動して、通常の設定を実行します。
2. [Run] → [Debug Configurations] をクリックして [Debug Configurations] ウィンドウを開きます。
3. [Debug Configurations] ウィンドウで、次の図に示すように、OpenCL リストから現在の起動コンフィギュレーションを選択します。



4. [Main] タブに、[Use RTL waveform for kernel debugging] と [Launch live waveform] の2つのカーネル デバッグ オプションがあります。両方をオンにして [Debug Configurations] ウィンドウを閉じると、デバッグ セッション が自動的に開始します。[Use RTL waveform for kernel debugging] をオンにすると、シミュレーション波形データ ベースが生成され、[Launch live waveform] をオンにすると、実際のシミュレーション中に波形ビューアーが新た に生成されて、シミュレーション エンジンおよび波形表示を完全に制御できるようになります。

ライブ波形ビューアーが生成されるように設定した場合は、実行ファイルを実行したときに波形ビューアーが自動的 に開きます。デフォルトでは、波形ビューアーにすべてのインターフェイス信号と次のデバッグ階層が表示されます。



- [Memory Data Transfers]: すべての計算ユニットからのこれらのインターフェイスを介したデータ転送を表示しま す。



ヒント: これらのインターフェイスのビット幅は、計算ユニットとは異なることがあります。その場合、バースト長が 異なります。たとえば、計算ユニットでの 16 個の 32 ビットワードのバーストは、OCL マスターでは 1 つの 512 ビ ットワードのバーストになります。

- Kernel <kernel name> <workgroup size>Compute Unit<CU name>
 - [CU Stalls (%)] : 計算ユニット (CU) 全体のストールのサマリを表示します。すべての最下位ストール信号のバ スが 1 つ作成されます。このバスは、各時点においてアクティブな信号の割合 (%) として波形に表示されます。
 - [Data Transfers]: CU のすべての AXI マスターのデータ転送を表示します。
 - [User Functions]: CU の階層内の関数すべてをリストします。
 - [Function]: <function name>
 - [Dataflow/Pipeline Activity]: CU の関数レベルのループのデータフロー/パイプライン信号を表示しま す。

- [Function Stalls]: この関数内の3つのストール信号をリストします。
- [Function I/O]: 関数のI/Oをリストします。これらのI/Oは、-m_axi、ap_fifo、ap_memory、またはap_none プロトコルのものです。



ヒント: 波形デバッガーと同様、[Scope] メニューから該当するインスタンスを選択し、[Object] メニューから信号を選択すると、内部信号の追加のデバッグ データを表示できます。また、HDL ブレークポイントなどのデバッグ制御、HDL コード ルックアップ、および波形マーカーもサポートされます。波形ビューアーの詳細は、『Vivado Design Suite ユーザー ガイド: ロジック シミュレーション』 (UG900) を参照してください。

波形デバッグ プロセスは、XOCC コマンド ラインを使用してもイネーブルにできます。イネーブルにするには、次の手順を使用します。

1. カーネル コンパイル中にデバッグ コードの生成をオンにします。

```
xocc -g ...
```

2. ホスト実行ファイルと同じディレクトリに次の内容の `sdaccel.ini` ファイルを作成します。

```
[Emulation]
launch_waveform=batch

[Debug]
profile=true
timeline_trace=true
data_transfer_trace=fine
```

3. ハードウェア エミュレーションを実行します。ハードウェア トランザクション データは、`<hardware_platform>-<device_id>-<xclbin_name>.wdb` という名前のファイルに収集されます。このファイルは、SDAccel GUI から直接開くことができます。



ヒント: エミュレーション セクションで `[Emulation] launch_waveform=gui` のように `launch_waveform` オプションを `gui` に設定すると、ハードウェア エミュレーションの実行中にライブの波形ビューアーが生成されます。

システム検証、ハードウェアのハング

アプリケーションのハング

このセクションでは、ホスト コードとアクセラレーションされたカーネルの間での通信に関連する問題のデバッグについて説明します。これらの問題が発生すると、マシンまたはアプリケーションがハングします。GDB デバッグ環境を使用すると (`xprint` を使用)、ハングが特定のカーネルに関連するなど、エラーを特定するのに役立つことはありますが、このセクションに示すように、`dmesg` および `xbutil` コマンドを使用してデバッグするのが最適です。コマンドおよびすべての関連するオプションは、[ハードウェア デバッグのユーティリティ](#) を参照してください。

次のよく発生する3つの問題を検証します。

- [AXI Firewall が作動する](#)
- [AXI 違反のためカーネルがハングする](#)
- [ホスト アプリケーションが DDR メモリにアクセス中にハングする](#)

このハードウェア デバッグ プロセスで問題を解決できない場合は、ChipScope を使用してハードウェア デバッグを実行する必要があります。[ChipScope を使用したデバッグ セクション](#) を参照してください。

AXI Firewall が作動する

AXI Firewall は、ホストがハングしないようにします。このため、ザイリンクスでは SDAccel プラットフォームに AXI Protocol Firewall IP を含めることをお勧めしています。最初のチェックの1つでファイアウォールが作動した場合、ホスト コードとカーネルが同じメモリ バンクを使用するように設定されているかを確認します。次は、このチェックを実行する最もシンプルな方法の1つです。

1. `xbutil` を使用して FPGA をプログラムします。

```
xbutil program -p <xclbin>
```

2. `xbutil` クエリ オプションを実行してメモリ トポロジをチェックします。

```
xbutil query
```

次の例では、カーネルに関連付けられているメモリ バンクはありません。

図 8: xbutil クエリの結果

```
Xclbin ID: 0x5a341b7a
Mem Topology:
  Bank      Type      Base Address      Size (KB)
  [0] bank0  **UNUSED**      0x0                0x400000
  [1] bank1  **UNUSED**      0x100000000        0x400000
  [2] bank2  **UNUSED**      0x200000000        0x400000
  [3] bank3  **UNUSED**      0x300000000        0x400000
```

3. ホスト コードで DDR バンクが使用されることが要求される場合は、このレポートは問題があることを示しています。この場合、カーネルとホスト コードで何が要求されているかをチェックする必要があります。ホスト コードがザイリンクス OpenCL™ 拡張機能を使用している場合、カーネルでどの DDR バンクが使用されるべきかを確認する必要があります。これらは、指定した `xocc -sp` 引数と同じである必要があります。

AXI 違反のためカーネルがハングする

カーネルとメモリ コントローラー間の AXI トランザクションが無効であるため、カーネルがハングする可能性があります。これらの問題をデバッグするには、カーネルをインストルメント化する必要があります。

1. SDAccel 環境には、XOCC リンク (-1) 中に適用可能なインストルメンテーション オプションが 2 つあります。これらのオプションは両方ともインプリメンテーションにハードウェアが追加されるので、使用率に基づいてインストルメンテーションを制限する必要があることがあります。
 - a. 軽量 AXI プロトコル チェッカー (`lapc`) を追加します。これらのプロトコル チェッカーは、`--dk` オプションを使用して追加します。次の構文を使用します。

```
--dk <[protocol|list_ports]<:compute_unit_name><:interface_name>>
```

通常、`<interface_name>` の使用はオプションです。指定しない場合、すべてのポートが解析されます。

プロトコル チェッカーが挿入されるように定義するには、`protocol` オプションを使用します。このオプションでは、`<compute_unit_name>` および `<interface_name>` のいずれかまたは両方に、キーワード `all` を使用できます。

`list_ports` オプションは、現在のデザインの有効な計算ユニットとポート組み合わせのリストを生成します。

注記: コマンドライン 1 行に複数の `--dk` オプションを含めて、インターフェイス モニター機能を追加していきます。

- b. SDx™ パフォーマンス モニター (`spm`) を追加すると、詳細な通信統計 (カウンター) のリストがイネーブルになります。これはパフォーマンス解析に最も役立ち、未処理のポート アクティビティのデバッグにおいて有益な情報を得ることができます。パフォーマンス モニターを追加するには、`profile_kernel` オプションを使用します。`profile_kernel` オプションの基本的な構文は、次のとおりです。

```
--profile_kernel data:<krnl_name|all>:<cu_name|all>:<intrfc_name|all>:<counters|all>
```

パフォーマンス モニターを追加する正確なインターフェイスを指定するには、3つのフィールドが必要ですが、リソース使用量が問題ではない場合は、`all` キーワードを使用すると、1つのオプションで既存のカーネル、計算ユニット、インターフェイスすべてを監視できるよう設定できます。または、`kernel_name`、`cu_name`、および `interface_name` を明示的に指定してインストールメンテーションを制限します。

最後のオプション `<counters|all>` は、大型デザインで情報の収集を `counters` に制限するか、`all` (デフォルト) を指定して実際のトレース情報が収集されるようにします。

注記: コマンドライン 1 行に複数の `--profile_kernel` オプションを含めて、パフォーマンス機能を追加していきます。

```
--profile_kernel data:kernel1:cu1:m_axi_gmem0
--profile_kernel data:kernel1:cu1:m_axi_gmem1
--profile_kernel data:kernel2:cu2:m_axi_gmem
```

2. アプリケーションをビルドし直したら、追加した SPM IP および LACP IP を含め、`xclbin` を使用してホスト アプリケーションを実行し直します。
3. アプリケーションがハングしたら、`xbutl status` を使用してエラーや異常を確認します。
4. 次のように SPM 出力を確認します。
 - `xbutl status --spm` を数回実行して、動いているカウンターがあるかを確認します。カウンターが動いている場合、カーネルはアクティブです。



ヒント: SPM 出力のテストは、GDB デバッグでも `xstatus spm` コマンドを使用してサポートされます。

- カウンターが止まっている場合、未処理のカウン트가 0 より大きいということは、AXI トランザクションにハングしているものがあることを意味します。

SDx Performance Monitor Counters										
CU Name	AXI Portname	Write Bytes	Write Trans.	Read Bytes	Read Trans.	Outstanding Cnt	Last Wr Addr	Last Wr Data	Last Rd Addr	Last Rd Data
RCMA	N/A	67108864	131872	0	0	0	0x3ffff00	0	0x0	0
kernel_1	m_axi_gmem	417920	409	451136	448	196416	0x2004000	3128415552	0xfc00	58462976
kernel_2	s_axi_gmem	262144	257	262144	272	65552	0x3000000	3284320784	0x1803c00	3284320784

INFO: xbutl status successful.

5. 次のように LACP 出力を確認します。

- `xbutl status --lapc` を実行して、AXI 違反がないかどうかを確認します。



ヒント: LACP 出力のテストは、GDB デバッグでも `xstatus lapc` コマンドを使用してサポートされます。

- AXI 違反がある場合は、カーネル インプリメンテーションに問題があることを意味します。

```
Light Weight AXI Protocol Checkers codes
(U Name: kernel_1 AXI Port: m_axi_gmem)
First violation:
  AXI_ERRS_READY_AW_A slave must not take BVALID HIGH until after the write address handshake is complete
Other violations:
  AXI_RECS_READY_MW_WAIT Recommended that MREADY is asserted within #MREADY cycles of BVALID being asserted
  AXI_RECS_READY_MW_WAIT Recommended that MREADY is asserted within #MREADY cycles of BVALID being asserted
  RECS_WAIT_TO_BVALID_MW_WAIT BVALID should be asserted within #MREADY cycles of AW command transfer or WLAST transfer (whichever is later), or previous B transfer if there are yet more AW and WLAST transfers outstanding.
(U Name: kernel_1 AXI Port: s_axi_gmem)
First violation:
  AXI_ERRS_READY_AW_A slave must not take BVALID HIGH until after the write address handshake is complete
Other violations:
  RECS_CONTINUOUS_TRANSFERS_MW_WAIT BVALID should be asserted within #MREADY cycles of either AW command transfer or previous B transfer while there are outstanding AW commands.
  RECS_CONTINUOUS_TRANSFERS_MW_WAIT BVALID should be asserted within #MREADY cycles of either W command transfer or previous W transfer while there are outstanding AW commands.
```

ホスト アプリケーションが DDR メモリにアクセス中にハングする

ホスト コードからの DMA の転送が完了しない場合にも、アプリケーションがハングまたはクラッシュする可能性があります。これは、ホスト コードが間違っているからとは限りません。カーネルが無効なトランザクションを発行したため、AXI がロックアップすることもあります。

1. プラットフォームに SDAccel™ プラットフォームと同様 AXI ファイアウォールがある場合、ファイアウォールが作動します。ドライバーにより SIGBUS が発行され、アプリケーションが強制終了され、デバイスがリセットされます。これは `xbutil query` を実行するとチェックできます。次に、表示されるエラーの例を示します。

図 9: ファイアウォールのステータス

```

Firewall Last Error Status:
  0:      0x0 (GOOD)
  1:      0x0 (GOOD)
  2:      0x80000 (RECS_WRITE_TO_BVALID_MAX_WAIT). Error occurred on Tue 2017-12-19 11:39:13 PST

Xclbin ID: 0x5a39da87
    
```



ヒント: ファイアウォールが作動しなかった場合は、Linux ツール `dmesg` で追加の情報が示される場合もあります。詳細は、[Linux dmesg の使用](#)を参照してください。

2. ファイアウォールが作動した場合は、DMA タイムアウトの原因を見つけることが重要です。原因には、無効な DMA 転送、カーネルの誤動作などが考えられます。AXI ファイアウォールが作動すると、アプリケーションの強制終了後にドライバーのヘルスチェック機能でボードがリセットされてしまいます。デバイス上にある根本的な原因を見つけるのに役立つ情報はすべて失われます。この問題をデバッグするには、`xclmgmt` カーネル モジュールでヘルスチェック スレッドをディスエーブルにして、エラーがキャプチャされるようにします。これには、一般的な Unix カーネル ツールを次の順で使用します。
 - a. `sudo modinfo xclmgmt`: モジュールの現在の設定をリストし、`health_check` パラメーターがオンかオフかをリストします。`xclmgmt` モジュールへパスも返します。
 - b. `sudo rmmod xclmgmt`: `xclmgmt` カーネル モジュールを削除し、ディスエーブルにします。
 - c. `sudo insmod <path to module>/xclmgmt.ko health_check=0`: `health_check` をディスエーブルにした状態で `xclmgmt` カーネル モジュールを再インストールします。



ヒント: このモジュールへのパスは、`modinfo` への呼び出しの出力にレポートされます。

3. ヘルス チェックをディスエーブルにしたら、アプリケーションを再実行します。[カーネルのハング](#)に関するセクションで説明したように、カーネル インストールメンテーションを使用して問題を特定できます。

ChipScope を使用したデバッグ

ChipScope デバッグ環境と Vivado® ハードウェア マネージャーを使用すると、ホスト アプリケーションとカーネルをすばやく効率的にデバッグできます。これには、次のいずれかの条件が満たされている必要があります。

- SDAccel™ アプリケーション プロジェクトに、`--dk` コンパイラ オプションを使用してデバッグ コアが挿入されている ([ChipScope を使用したハードウェア デバッグ](#)を参照)。
- プロジェクトで使用されている RTL カーネルにデバッグ コアが層インスタンス化されている ([RTL カーネルへのデバッグ IP の追加](#)を参照)。

これらのツールでは、カーネルがハードウェアで実行されている状態で、ロジック レベルからシステム レベルのデバッグまで幅広い機能を使用できます。

XVC およびハードウェア サーバーを実行し、ホスト アプリケーションを実行して、最後に Vivado ハードウェア マネージャーでデバッグ コアをトリガー待機状態にしてトリガーするには、次の手順を実行します。

1. RTL カーネルにデバッグ IP を追加します。これは、RTL カーネルでのみ必要です。[RTL カーネルへのデバッグ IP の追加](#)を参照してください。
2. ホスト アプリケーションをインストルメント化して、ホスト実行でデバッグする適切な位置で一時停止するようにします。[ホスト アプリケーションを使用したデバッグ](#)を参照してください。
3. ハードウェア デバッグのための環境を設定します。これは、手動で実行するか、スクリプトを使用して自動的に実行します。次の手順は、[ハードウェア デバッグの手動設定](#)および[ハードウェア デバッグ用の自動設定](#)で説明されています。
 - a. 必要な XVC およびハードウェア サーバーを実行します。
 - b. ホスト アプリケーションを実行し、ホスト実行の適切な位置で一時停止して、ILA トリガーのセットアップをイネーブルにします。
 - c. Vivado ハードウェア マネージャーを開き、XVC サーバーに接続します。
 - d. デザインの ILA トリガー条件を設定します。
 - e. ホスト アプリケーションの実行を続行します。
 - f. Vivado ハードウェア マネージャーで結果を確認します。
 - g. 必要に応じて、上記の手順 b から繰り返します。

RTL カーネルへのデバッグ IP の追加



重要: このデバッグ手法を使用するには、Vivado® Design Suite および RTL デザインに関する知識が必要です。

カーネル ロジックをデバッグするには、ILA (Integrated Logic Analyzer) および VIO (Virtual Input/Output) などのデバッグ コアを RTL コードにインスタンス化する必要があります。Vivado IDE でほかの IP を使用する場合と同様に、Vivado Design Suite から RTL カーネルを編集して ILA IP のカスタマイズまたは VIO IP を RTL コードにインスタンス化します。ILA またはその他のデバッグ コアを使用する方法、および HDL の generate 文を使用してデバッグ コアの生成をイネーブル/ディスエーブルにする方法については、『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。



ヒント: デバッグ コアは、RTL カーネルを作成したときに追加するのがベストです。詳細は、『UltraFast 設計手法ガイド (Vivado Design Suite 用)』 ([UG949](#)) の「デバッグ」を参照してください。

または、開いている Vivado プロジェクトから Tcl スクリプトを使用して ILA デバッグ コアを追加することもできます。次にコード例を示します。

```
create_ip -name ila -vendor xilinx.com -library ip -version 6.2 -
module_name ila_0
set_property -dict [list CONFIG.C_PROBE6_WIDTH {32} CONFIG.C_PROBE3_WIDTH
{64} \
CONFIG.C_NUM_OF_PROBES {7} CONFIG.C_EN_STRG_QUAL {1}
CONFIG.C_INPUT_PIPE_STAGES {2} \
CONFIG.C_ADV_TRIGGER {true} CONFIG.ALL_PROBE_SAME_MU_CNT {4}
```



```

CONFIG.C_PROBE6_MU_CNT {4} \
CONFIG.C_PROBE5_MU_CNT {4} CONFIG.C_PROBE4_MU_CNT {4}
CONFIG.C_PROBE3_MU_CNT {4} \
CONFIG.C_PROBE2_MU_CNT {4} CONFIG.C_PROBE1_MU_CNT {4}
CONFIG.C_PROBE0_MU_CNT {4}] [get_ips ila_0]
    
```

次に、ILA デバッグ コアを GitHub の [RTL カーネルのデバッグ](#) デザイン例の RTL カーネル ソース ファイルにインスタンシエートした例を示します。ILA は `src/hdl/krnl_vadd_rtl_int.sv` ファイルで指定した組み合わせ加算器の出力を監視します。

```

// ILA monitoring combinatorial adder
ila_0 i_ila_0 (
    .clk(ap_clk), // input wire clk
    .probe0(areset), // input wire [0:0] probe0
    .probe1(rd_fifo_tvalid_n), // input wire [0:0] probe1
    .probe2(rd_fifo_tready), // input wire [0:0] probe2
    .probe3(rd_fifo_tdata), // input wire [63:0] probe3
    .probe4(adder_tvalid), // input wire [0:0] probe4
    .probe5(adder_tready_n), // input wire [0:0] probe5
    .probe6(adder_tdata) // input wire [31:0] probe6
);
    
```

RTL カーネルにデバッグ用に適切なデバッグ コアを挿入したら、前のトピックに示すように、ハードウェアを ChipScope ツールで解析できます。

ホスト アプリケーションを使用したデバッグ

SDAccel™ プラットフォームで実行されるカーネル コードを使用してホスト アプリケーションをデバッグするには、カーネルがデバイスにプログラムされた後、にカーネルを開始前に ILA トリガー条件を設定できるようにアプリケーション ホスト コードを変更する必要があります。

C++ ホスト アプリケーションの一時停止

次のコード例は、GitHub の [RTL カーネル サンプル](#) の `src/host.cpp` からのものです。

```

....
std::string binaryFile = xcl::find_binary_file(device_name, "vadd");

cl::Program::Binaries bins = xcl::import_binary_file(binaryFile);
devices.resize(1);
cl::Program program(context, devices, bins);
cl::Kernel krnl_vadd(program, "krnl_vadd_rtl");

//if (interactive == true)
//wait_for_enter("\nPress ENTER to continue after setting up ILA
trigger...");

//Allocate Buffer in Global Memory
std::vector<cl::Memory> inBufVec, outBufVec;
cl::Buffer buffer_r1(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
    vector_size_bytes, source_input1.data());
...

//Copy input data to device global memory
q.enqueueMigrateMemObjects(inBufVec, 0/* 0 means from host*/);
    
```

```
//Set the Kernel Arguments
...

//Launch the Kernel
q.enqueueTask(krnl_vadd);
```

条件 `if (interactive)` を追加し、`wait_for_enter` 関数を使用してホスト アプリケーションを一時停止すると、ILA が必要なトリガーを設定し、カーネルからのデータをキャプチャできる準備をするための時間が与えられます。Vivado® ハードウェア マネージャーを正しく設定したら、Enter キーを押してホスト アプリケーションの実行を続行します。

GDB を使用したホスト アプリケーションの一時停止

ホスト アプリケーションを変更してカーネル実行の前に一時停止するようにする代わりに、SDx™ IDE から GDB セッションを実行できます。この場合、ホスト アプリケーションのカーネル実行の前にブレークポイントを設定します。ブレークポイントに到達したら、Vivado® ハードウェア マネージャーでデバッグ ILA トリガーを設定してトリガー待機状態にし、GDB でカーネルの実行を再開します。

ハードウェア デバッグ用の自動設定

1. SDx インストール エリアにある適切な `settings64.sh/.csh` ファイルを読み込んで、SDx™ 環境を設定します。
2. `sdx_debug_hw` スクリプトを使用して、`xvc_pcie` および `hw_server` を起動します。`/dev/xvc_*` キャラクター型デバイスは、プラットフォームおよび SDx リリースによって異なります。たとえば、2018.2 SDx リリースの `vcu1525 5.1 DSA` の場合は、ファイルは `/dev/xvc_pub.m1025` です。

```
sdx_debug_hw --xvc_pcie /dev/xvc_pub.m1025 --hw_server
launching xvc_pcie...
xvc_pcie -d /dev/xvc_pub.m1025 -s TCP:::10200
launching hw_server...
hw_server -sTCP:::3121
```

3. SDx IDE で、ホスト コードのカーネルが作成/ダウンロードされた後、カーネル実行が開始する前に一時停止文を追加します。

- C++ ホスト コードでは、`clCreateKernel()` 関数呼び出しをラップする関数の後に一時停止文を追加します。次に、Vector Add テンプレート デザインの C++ コードの例を示します。

```
134 // This call will get the kernel object from program. A kernel is an
135 // OpenCL function that is executed on the FPGA.
136 cl::Kernel krnl_vector_add(program, "krnl_vadd");
137
138 // Add a pause here to prompt user to arm ILA trigger
139 std::cout << "Pausing to allow you to arm ILA trigger. Hit enter here to resume host program..." << std::endl;
140 std::cin.get();
141
142 // These commands will allocate memory on the Device. The cl::Buffer objects can
143 // be used to reference the memory locations on the device.
144 cl::Buffer buffer_a(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
145 size_in_bytes, source a.data());
146 cl::Buffer buffer_b(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
147 size_in_bytes, source b.data());
148 cl::Buffer buffer_result(context, CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY,
149 size_in_bytes, source_results.data());
150
```

- C ホスト コードでは、`clCreateKernel()` 関数呼び出しの後に一時停止文を追加します。

```

// Build the program executable
//
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
if (err != CL_SUCCESS)
{
    size_t len;
    char buffer[2048];

    printf("Error: Failed to build program executable!\n");
    clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
    printf("%s\n", buffer);
    printf("Test failed\n");
    return EXIT_FAILURE;
}

// Create the compute kernel in the program we wish to run
//
kernel = clCreateKernel(program, "vadd", &err);
if (!kernel || err != CL_SUCCESS)
{
    printf("Error: Failed to create compute kernel!\n");
    printf("Test failed\n");
    return EXIT_FAILURE;
}

// PAUSE
wait_for_enter("\nPress ENTER to continue after setting up ILA trigger...");

// Create the input and output arrays in device memory for our calculation
//
d_a = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(int) * LENGTH, NULL, NULL);
d_b = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(int) * LENGTH, NULL, NULL);
d_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(int) * LENGTH, NULL, NULL);
if (!d_a || !d_b || !d_c)
{
    printf("Error: Failed to allocate device memory!\n");
    printf("Test failed\n");
    return EXIT_FAILURE;
}
    
```

4. 変更したホスト プログラムを実行します。

```

vadd_test.exe ./binary_container_1.xclbin
Loading: './binary_container_1.xclbin'
Pausing to allow you to arm ILA trigger. Hit enter here to resume host
program...
    
```

5. SDAccel インストール ディレクトリにある `sdx_debug_hw` スクリプトを使用して Vivado® Design Suite を起動します。

```

> sdx_debug_hw --vivado --host xcoltlab40 --ltx_file ../workspace/
vadd_test/System/pfm_top_wrapper.ltx
    
```

コマンド ウィンドウに次のように表示されます。

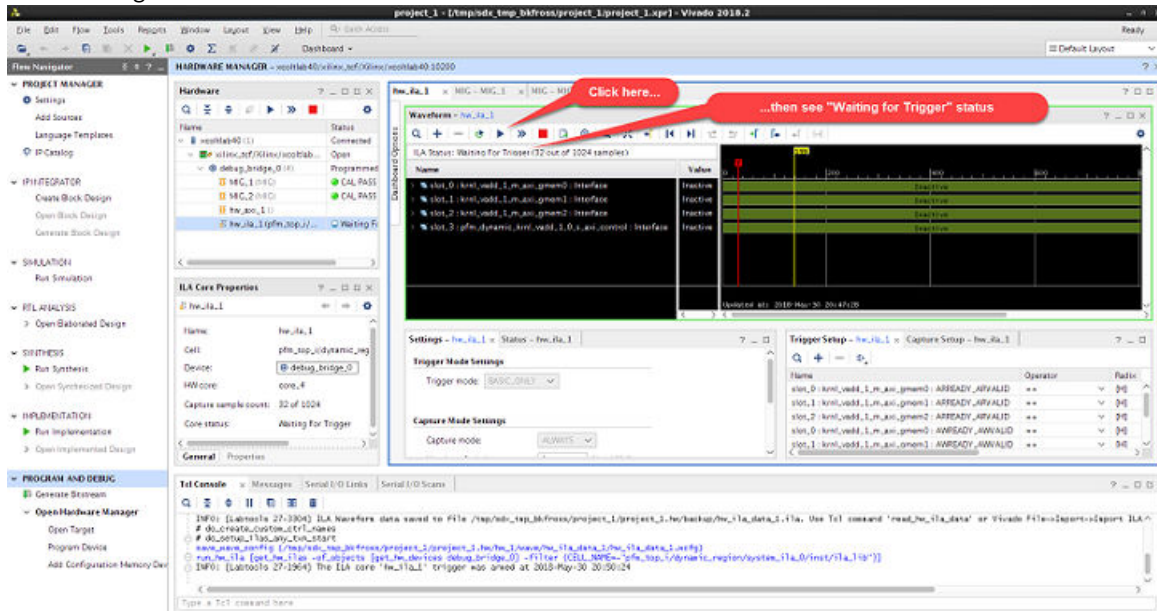
```

launching vivado... ['vivado', '-source', 'sdx_hw_debug.tcl', '-
tclargs', '/tmp/sdx_tmp/project_1/project_1.xpr', 'workspace/vadd_test/
System/pfm_top_wrapper.ltx', 'xcollab40', '10200', '3121']

***** Vivado v2018.2 (64-bit)
**** SW Build 2245749 on Wed May 30 12:36:19 MDT 2018
**** IP Build 2245576 on Wed May 30 15:12:50 MDT 2018
** Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.

start_gui
    
```

6. Vivado Design Suite で ILA トリガーを実行します。



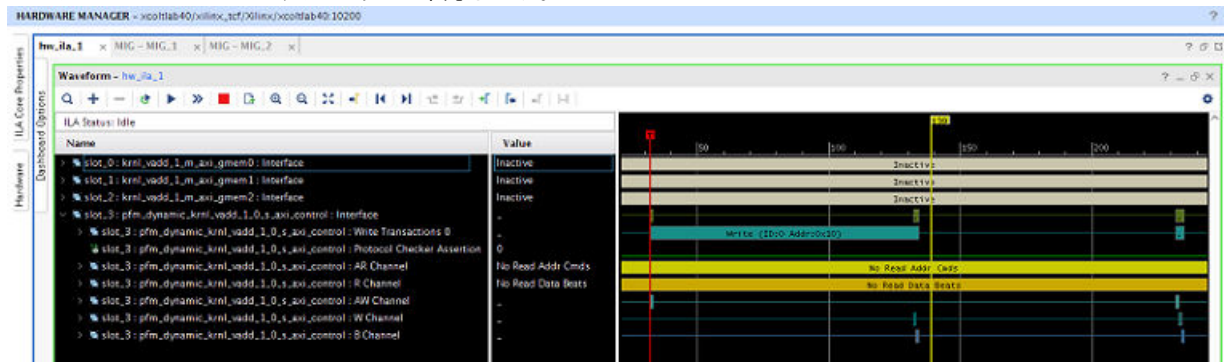
7. Enter キーを押してホスト プログラムの一時停止を解除します。

```

vadd_test.exe ./binary_container_1.xclbin
Loading: './binary_container_1.xclbin'
Pausing to allow you to arm ILA trigger. Hit enter here to resume host
program...

TEST PASSED
    
```

8. Vivado Design Suite の [Waveform] ウィンドウで、カーネル計算ユニットのスレーブ制御インターフェイス上のインターフェイス トランザクションを確認します。



ハードウェア デバッグの手動設定

Amazon F1 インスタンスのデバッグ サーバーを起動します。

Amazon F1 インスタンスのデバッグ サーバーの起動方法は、https://github.com/aws/aws-fpga/blob/master/hdk/docs/Virtual_JTAG_XVC.md を参照してください。

Nimble およびその他のクラウド プラットフォームでのデバッグ サーバーの起動

Vivado® ハードウェア マネージャーでデザインをデバッグする前に、次の 2 つの手順を実行してデバッグ サーバーを起動する必要があります。

1. SDx™ `settings64.csh` または `settings64.sh` 環境セットアップ スクリプトを読み込んで、`xvc_pcie` ドライバーを実行します。`xvc_pcie` に渡すファイル名は、カーネル デバイス ドライバーにインストールされているキャラクター型ドライバー ファイルと同じである必要があります。

```
>xvc_pcie -d /dev/xvc_pub.m1025
```

2. ポート 10201 で XVC サーバーを起動し、ポート 3121 で `hw_server` を起動します。

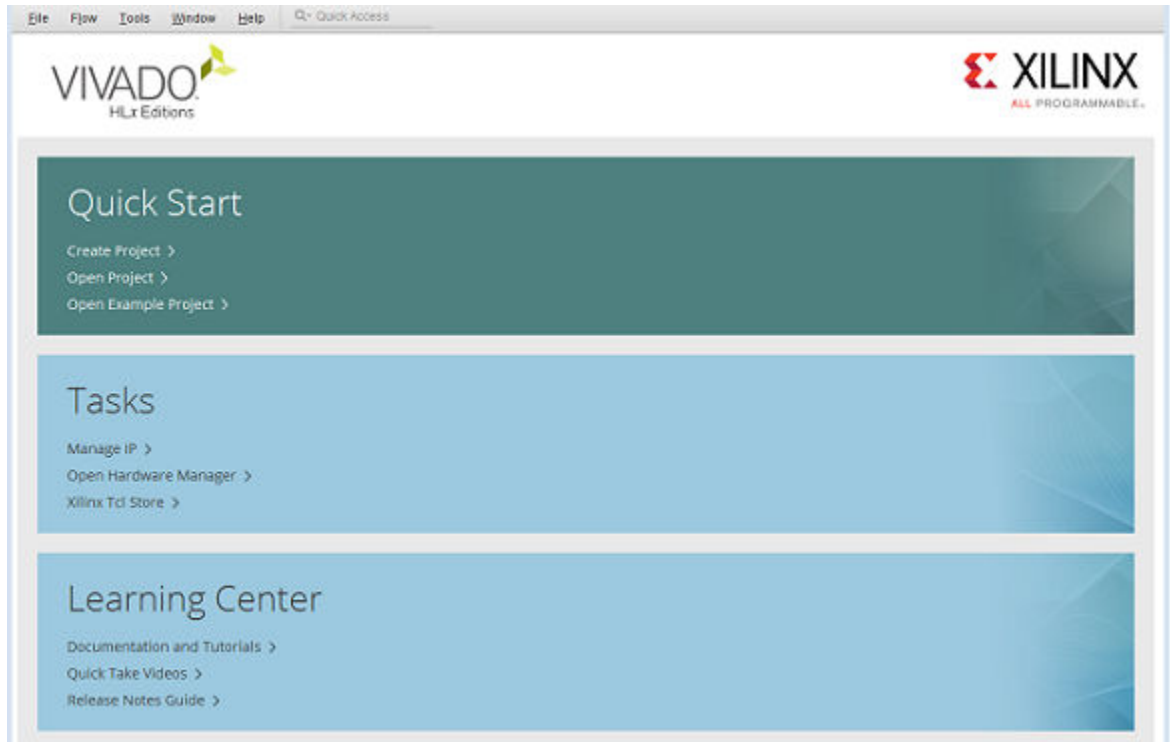
```
>hw_server "set auto-open-servers xilinx-xvc:localhost:10201" -e "set always-open-jtag 1"
```

Vivado ハードウェア マネージャーを使用したデザインのデバッグ

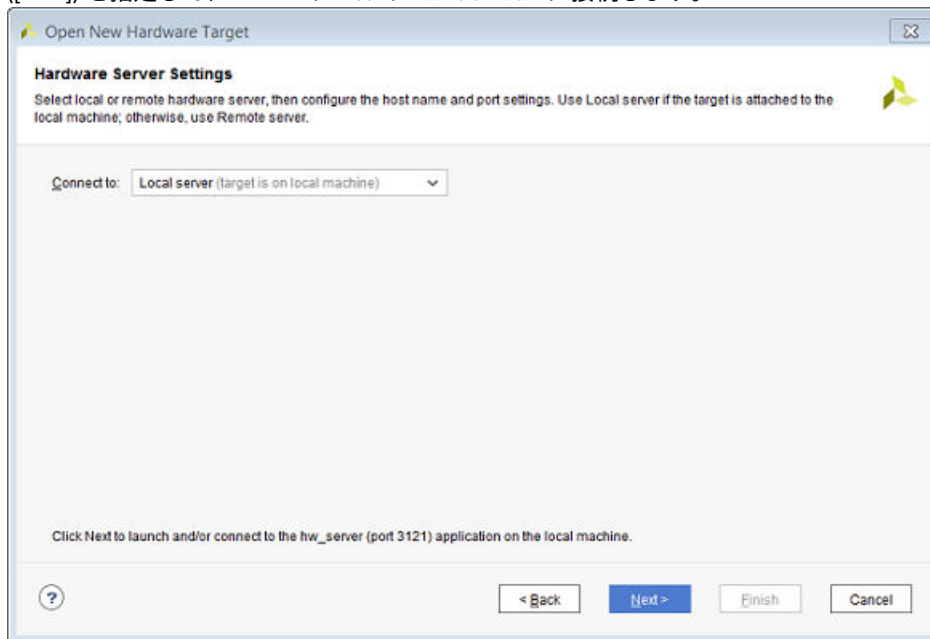
FPGA のデバッグには、従来物理的な JTAG 接続が使用されてきました。SDAccel プラットフォームでは、クラウドでのデバッグを可能にするため、XVC を活用します。この機能を利用するため、SDAccel では XVC サーバーの実行を有効にします。XVC サーバーは、ザイリンクス 仮想ケーブル (XVC) プロトコルのインプリメンテーションで、Vivado® Design Suite をローカルまたはリモート ターゲットの FPGA に接続し、ILA (Integrated Logic Analyzer) や VIO (Virtual Input/Output IP) などのザイリンクス デバッグ コアを使用してデバッグできるようにします。

Vivado ハードウェア マネージャー (Vivado Design Suite または Vivado Lab Edition) は、ターゲット インスタンス上で実行するか、異なるホスト上でリモートで実行できます。XVC サーバーがリスニングしている TCP ポートが、Vivado ハードウェア マネージャーを実行するホストにアクセスできることが必要です。Vivado ハードウェア マネージャーをターゲット上の XVC サーバーに接続するには、Vivado ツールをホストするマシンで次の手順を実行します。

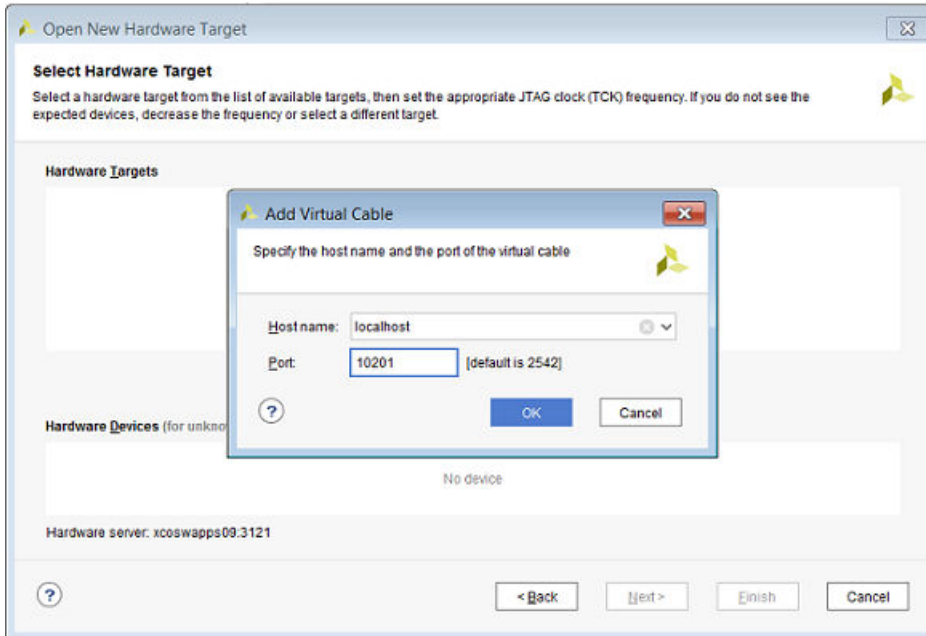
1. Vivado Lab Edition またはフル Vivado Design Suite を起動します。
2. [Tasks] セクションで [Open Hardware Manager] をクリックします。



- 次の図に示すように、ローカルまたはリモート接続 ([Connected to])、ホスト名 ([Host name])、およびポート ([Port]) を指定して、Vivado ツールの `hw_server` に接続します。



- ターゲット インスタンスである仮想 JTAG XVC サーバーに接続します。

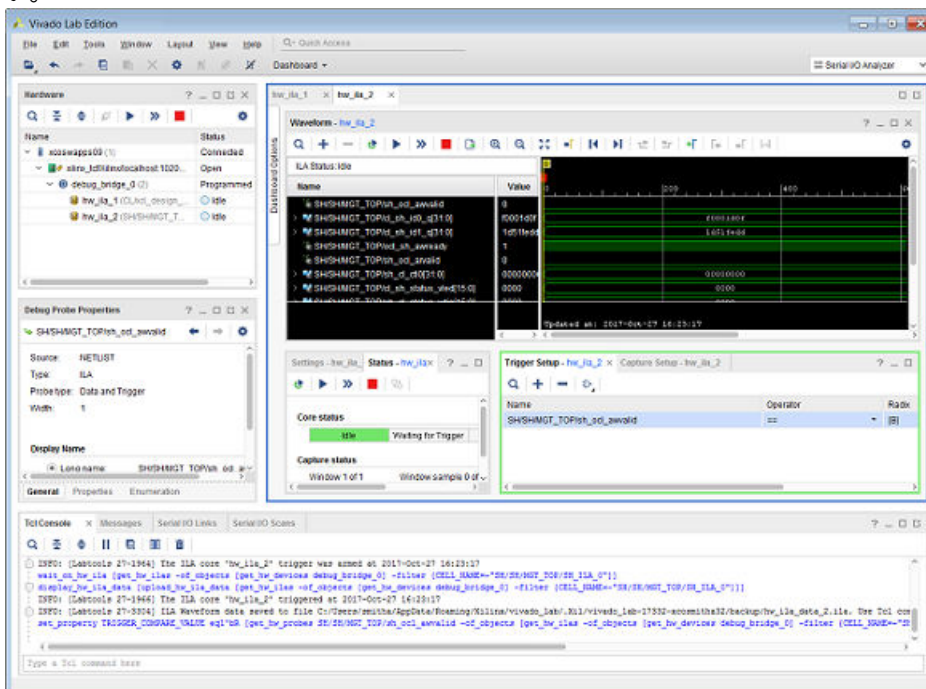


5. ChipScope ツールの [Hardware] ウィンドウでデバッグ ブリッジ インスタンスを選択します。
6. [Hardware Device Properties] ウィンドウで、[Probes file] の横にあるアイコンをクリックしてデザインのプローブファイルを選択し、[OK] をクリックします。ハードウェアデバイスが更新され、デザインに含まれるデバッグ コアが表示されます。



ヒント: プロブ ファイル (.ltx) は、カーネルに [ChipScope を使用したハードウェア デバッグ](#) で指定されているデバッグ コアがある場合は、カーネルのインプリメンテーション中に Vivado ツールにより生成されます。

7. これで、ChipScope ツールを使用して SDAccel プラットフォーム上で実行中のカーネルをデバッグできます。ツールの詳細は、『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。



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。

ソリューションセンター

デバイス、ツール、IP のサポートについては、[ザイリンクス ソリューションセンター](#)を参照してください。デザイン アシスタント、デザイン アドバイザリ、トラブルシューティングのヒントなどが含まれます。

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 環境ユーザー ガイド』 ([UG1023](#))
3. 『SDAccel 環境プロファイリングおよび最適化ガイド』 ([UG1207](#))
4. 『SDAccel 環境チュートリアル: 概要』 (UG1021: [英語版](#)、[日本語版](#))
5. [SDAccel 開発環境ウェブ ページ](#)
6. [Vivado® Design Suite の資料](#)
7. 『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](#))
8. 『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](#))
9. 『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](#))
10. 『Vivado Design Suite ユーザー ガイド: 高位合成』 ([UG902](#))
11. 『UltraFast 設計手法ガイド (Vivado Design Suite 用)』 ([UG949](#))
12. 『Vivado Design Suite プロパティ リファレンス ガイド』 ([UG912](#))
13. [Khronos Group ウェブ ページ](#): OpenCL 規格の資料
14. [ザイリンクス Virtex UltraScale+ FPGA VCU1525 アクセラレーション開発キット](#)
15. [ザイリンクス Kintex UltraScale FPGA KCU1500 アクセラレーション開発キット](#)

お読みください: 重要な法的通知

本通知に基づいて貴殿または貴社 (本通知の被通知者が個人の場合には「貴殿」、法人その他の団体の場合には「貴社」、以下同じ) に開示される情報 (以下「本情報」といいます) は、ザイリンクスの製品を選択および使用することのためにのみ提供されます。適用される法律が許容する最大限の範囲で、(1) 本情報は「現状有姿」、およびすべて受領者の責任で (with all faults) という状態で提供され、ザイリンクスは、本通知をもって、明示、黙示、法定を問わず (商品性、非侵害、特定目的適合性の保証を含みますがこれらに限られません)、すべての保証および条件を負わない (否認する) ものとし、また、(2) ザイリンクスは、本情報 (貴殿または貴社による本情報の使用を含む) に関係し、起因し、関連する、いかなる種類・性質の損失または損害についても、責任を負わない (契約上、不法行為上 (過失の場合を含む)、その他のいかなる責任の法理によるかを問わない) ものとし、当該損失または損害には、直接、間接、特別、付随的、結果的な損失または損害 (第三者が起こした行為の結果被った、データ、利益、業務上の信用の損失、その他あらゆる種類の損失や損害を含みます) が含まれるものとし、それは、たとえ当該損害や損失が合理的に予見可能であったり、ザイリンクスがそれらの可能性について助言を受けていた場合であったとしても同様です。ザイリンクスは、本情報に含まれるいかなる誤りも訂正する義務を負わず、本情報または製品仕様のアップデートを貴殿または貴社に知らせる義務も負いません。事前の書面による同意のない限り、貴殿または貴社は本情報を再生産、変更、頒布、または公に展示してはなりません。一定の製品は、ザイリンクスの限定的保証の諸条件に従うこととなるので、<https://japan.xilinx.com/legal.htm#tos> で見られるザイリンクスの販売条件を参照してください。IP コアは、ザイリンクスが貴殿または貴社に付与したライセンスに含まれる保証と補助的条件に従うこととなります。ザイリンクスの製品は、フェイルセーフとして、または、フェイルセーフの動作を要求するアプリケーションに使用するために、設計されたり意図されたりしていません。そのような重大なアプリケーションにザイリンクスの製品を使用する場合のリスクと責任は、貴殿または貴社が単独で負うものです。<https://japan.xilinx.com/legal.htm#tos> で見られるザイリンクスの販売条件を参照してください。

自動車用のアプリケーションの免責条項

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

商標

© Copyright 2018 Xilinx, Inc. Xilinx、Xilinx のロゴ、Alveo、Artix、ISE、Kintex、Spartan、Versal、Virtex、Vivado、Zynq、およびこの文書に含まれるその他の指定されたブランドは、米国およびその他の各国のサイリンクス社の商標です。OpenCL および OpenCL のロゴは Apple Inc. の商標であり、Khronos による許可を受けて使用されています。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。