SDAccel プロファイリングおよび最適化ガイド

UG1207 (v2018.2.xdf) 2018 年 10 月 2 日

XILINX®

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	
ホスト コードの DDR バンクの割り当て	DDR 割り当ておよびカーネルの SLR 配置を揃える必要があることを示す注記を追加。
2018 年 7 月 2 日 バージョン 2018.2	
資料全体	2018.2 用にマイナーな編集上の変更。
2018 年 6 月 6 日 バージョン 2018.2	
	全体的に大幅な並べ替えと書き直し。プロファイリングおよび最適化に重点を置くため、背景的な情報を削除。
[Guidance] ビュー	プロジェクト パフォーマンスを改善する [Guidance] ビューに関する説明を追加。
波形ビューおよびライブ波形ビューアー	SDAccel™ 環境の波形ビューアーに関する説明を追加。
インプリメンテーション ツールの使用	Vivado バックエンド最適化の説明を追加。
インターフェイス最適化	全 AXI データ幅の使用
計算並列処理の最適化	ループの並列処理とタスクの並列処理
	計算ユニットの最適化
	メモリ アーキテクチャの最適化
2018 年 4 月 4 日 バージョン 2018.1	
資料全体	2018.1 用にマイナーな編集上の変更。
コマンド ライン	profile_kernel オプションの変更。
ホスト コードの DDR バンクの割り当て	RTL Kernel ウィザードのカーネル命名規則。

目次

第 1 章

概要

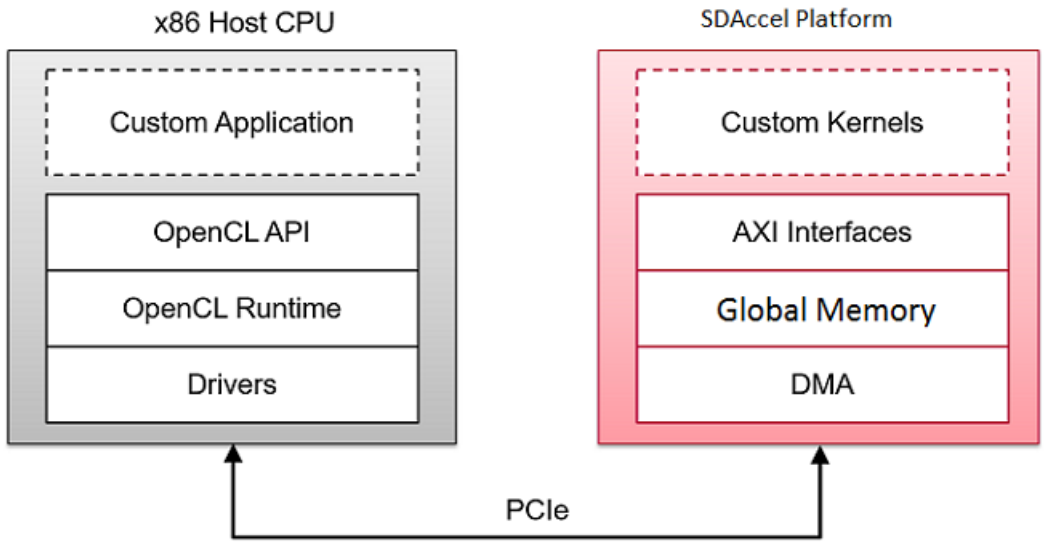
この文書には、デザインのパフォーマンス解析に関連する SDx™ 機能すべての説明が含まれ、実際にパフォーマンスを改善する方法が論理的な構造で記述されています。SDAccel™ パフォーマンスのボトルネック (アクセラレータ、PCIe® バス転送、およびホスト コードなど) の主なコンポーネントについてそれぞれセクション別に説明します。これらのセクションは、システムの全体的なパフォーマンスを高めるために、ボトルネックを把握して回避策を見つけやすいように分かれています。

注記: パフォーマンス最適化では、まず問題なく動作するデザインがあると想定して、パフォーマンスを改善していきます。問題が発生した場合は、『SDAccel 環境デバッグ ガイド』 (UG1281) を参照してください。同様に、ホスト コードまたはアクセラレータ カーネルのコード記述に関する一般的な概念についてはここでは説明しません。これについては『SDAccel 環境プログラマ ガイド』 (UG1277) を参照してください。

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、ザイリンクス 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。

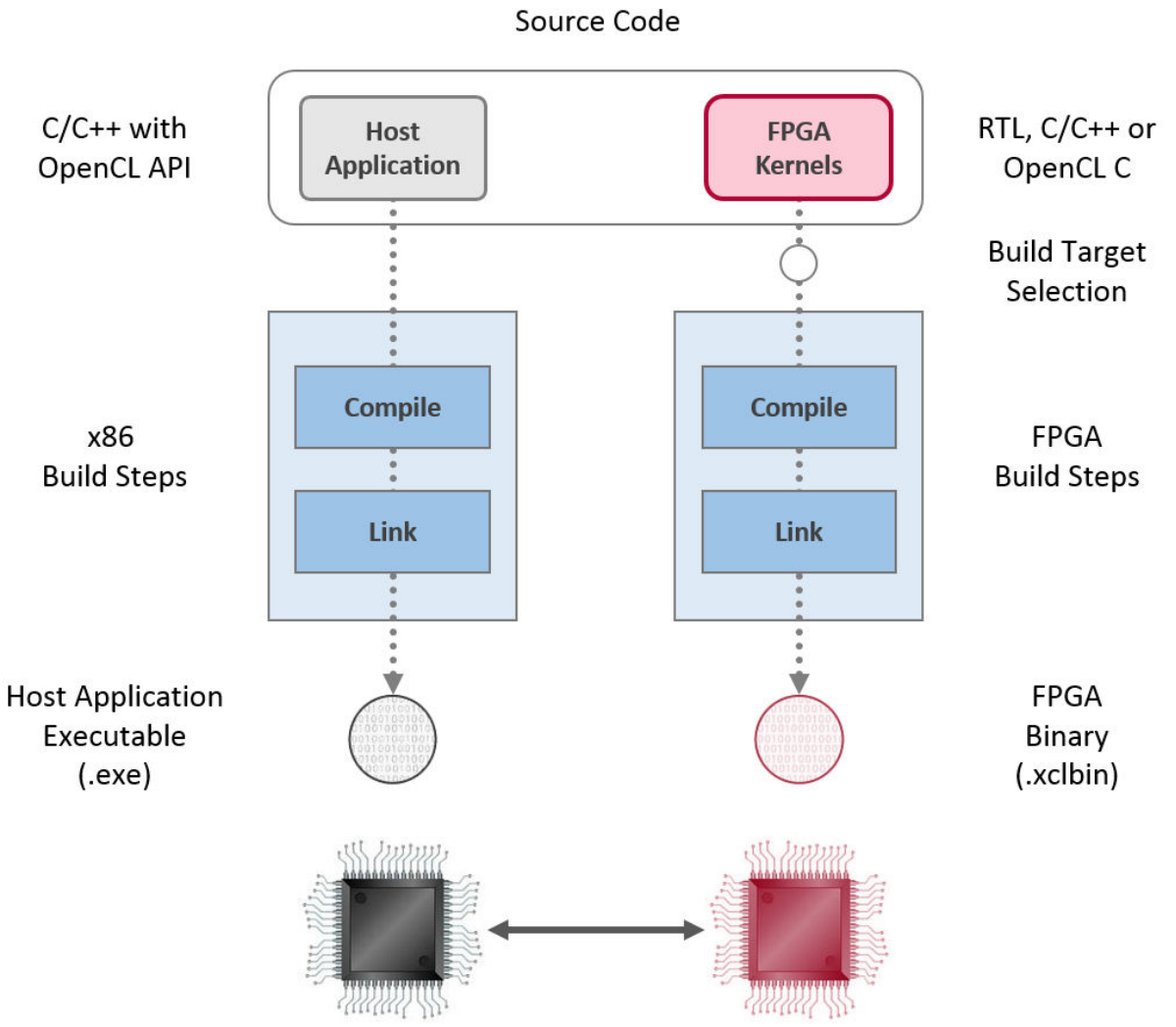
1. ホスト アプリケーションが PCIe を介して、カーネルで必要とされるデータを SDAccel ハードウェア プラットフォームのグローバル メモリに書き込みます。
2. ホストがカーネルをその入力パラメーターを使用してプログラムします。
3. ホスト アプリケーションが FPGA のカーネル関数の実行をトリガーします。
4. カーネルが必要な計算、グローバル メモリからのデータの読み出しおよび書き込みを必要に応じて実行します。
5. カーネルがメモリ バンクにデータを書き込み、ホストにタスクが終了したことを通知します。
6. ホスト アプリケーションがグローバル メモリからホスト メモリ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ます。

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 バイナリの特性を定義します。

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 最適化フローの概要

SDAccel™ 環境は、C/C++/OpenCL™ アプリケーションがザイリンクス FPGA でアクセラレーションされるようにするためのソフトウェア開発環境です。次の図は、SDAccel 環境でアプリケーションを最適化するのに推奨されるフローを示しています。

図 3: SDAccel の推奨フロー

機能とパフォーマンスのベースライン

最適化を開始する前に、まずアプリケーションのパフォーマンスを理解しておくことが重要です。これは、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ることで理解できます。

まず、既存プラットフォームで実行している現在のアプリケーションのボトルネックを見つけます。もっとも効果的なのは、`valgrind/callgrind` および GNU `gprof` などのプロファイリング ツールを使用してアプリケーションを実行する方法です。これらのツールで生成されたプロファイリング データには、すべての関数に対する呼び出し回数と実行時間を示すコールグラフが表示されます。実行時間が最も長い関数は、FPGA でのアクセラレーションの良い候補となります。

ターゲット関数を選択したら、それらを最適化なしで OpenCL™ CL カーネルまたは C/C++ カーネルに変換します。これらのカーネルを呼び出すアプリケーション コードも、データ移動およびタスクのスケジューリングに OpenCL API が使用されるように変換する必要があります。この段階ではすべてをできるだけ簡素にし、既存コードへの変更は最小限にして、FPGA で動作するデザインをすばやく生成し、ベースラインのパフォーマンスとリソース数を把握できるようにします。

次に、CPU およびハードウェア エミュレーションを実行して、関数に問題がないかどうかを検証し、ホスト コードおよびカーネルでプロファイリング データを生成します。カーネル コンパイル レポート、プロファイル サマリ、タイムライン トレース、およびデバイス ハードウェア トランザクションを解析して、タイミング、間隔、レイテンシ、DSP や BRAM などのリソース使用量といったパフォーマンス見積もりからのベースラインを理解します。

ベースライン作成の最後に、アプリケーションをビルドして FPGA アクセラレーション カードで実行します。システム コンパイルからのレポートおよびアプリケーション実行からのプロファイリング データを解析し、実際のパフォーマンスおよびリソース使用量を確認します。

ベースラインのレポートはすべて保存して、最適化を設定していく際に参照したり比較したりできるようにしておきます。

データ移動の最適化

OpenCL™ プログラミング モデルでは、すべてのデータがまずホスト メモリからデバイスのグローバル メモリに転送されて、グローバル メモリからカーネルに転送されて計算されます。この計算結果はカーネルからグローバル メモリに書き戻されて、最後にグローバル メモリからホスト メモリに送信されます。このプログラミング モデルでどうデータを効率的に動かすかがカーネル計算最適化のストラテジを決める重要な要素なので、計算を最適化するよりも前にアプリケーションでデータ移動を最適化しておくことをお勧めします。

データ移動の最適化では、計算が非効率的であるとデータ移動がストールしてしまうことがあるので、データ転送コードを計算コードとは分けておくことが重要です。データ転送コードを含むホスト コードとカーネルを変更するのは、この最適化段階でのみにしてください。目的は、PCIe® 帯域幅および DDR 帯域幅を最大限に活用することにより、システム レベルのデータ スループットを最大にすることにあります。この目的を達成するには、通常 CPU エミュレーション、ハードウェア エミュレーションの実行を何度か繰り返し、FPGA で実行する必要があります。

カーネル計算の最適化

FPGA を使用する利点の 1 つは、特定のアプリケーション用のカスタム ロジックを作成できるところにあります。カーネル計算最適化は、カーネル インターフェイスにデータが到達したらすぐにすべてのデータを消費できるプロセッシング ロジックを作成することを目的としています。この段階の重要なメトリクスは、開始間隔 (II) です。これは通常、関数のパイプライン処理、ループ展開、配列分割、データフローなどの手法を使用してデータパスを一致させるようにプロセッシング コードを展開すると達成されます。SDAccel™ 環境では、最適化のため、ハードウェア エミュレーションおよびシステム実行中にさまざまなコンパイル レポートおよびプロファイリング データが生成されます。コンパイルおよびプロファイリング レポートの詳細は、2: SDAccel プロファイリングおよび最適化機能を参照してください。

XILINX

第 2 章

SDAccel プロファイリングおよび最適化機能

SDAccel™ 環境では、コンパイル中に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。また、エミュレーション モードおよび FPGA アクセラレーション カードでアプリケーション実行中のプロファイリング データも収集されます。レポートおよびプロファイリング データには、アプリケーションのパフォーマンス ボトルネックを特定し、パフォーマンスを向上するために使用可能な最適化手法などの情報が含まれます。この章では、レポートを生成して、プロファイリング結果を収集して SDAccel 環境に表示する方法を説明します。

システム見積もり

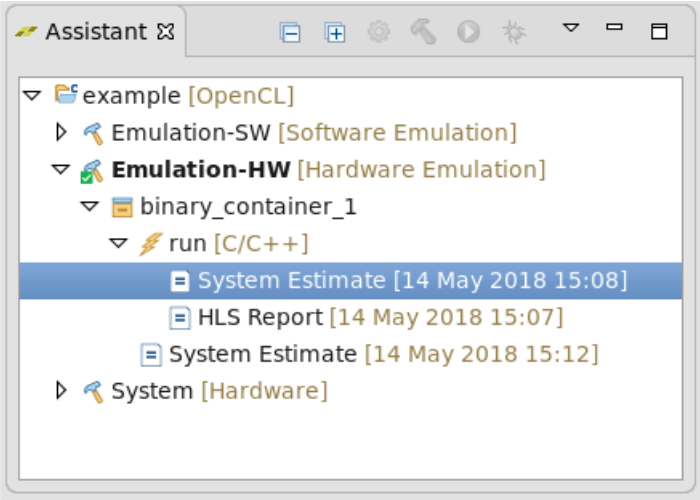
SDAccel™ 開発環境で FPGA プログラミング ファイルを生成するのは、最も実行時間がかかります。この実行時間は、ターゲット ハードウェア デバイスおよび FPGA ファブリック上の計算ユニット数によってかなり変わります。このため、アプリケーション プログラマがアプリケーションをターゲット デバイスで実行する前にパフォーマンスをすばやく理解しておくことが重要となります。これで FPGA プログラミング ファイルの生成を待たずに、より時間をかけてアプリケーションをイテレーションおよび最適化できるようになります。

SDAccel 開発環境のシステム見積もりには、ターゲット ハードウェア デバイスとアプリケーションの各計算ユニットが考慮されます。実際のパフォーマンス メトリクスはターゲット デバイスでのみ測定できますが、SDAccel 環境の見積もりレポートには予測されるビヘイビアーが正確に表示されます。

GUI フロー

このレポートは、ハードウェア エミュレーション フロー中に自動的に生成されます。カーネルごとに 1 つのレポートが、バイナリ コンテナー全体に対して最上位レポートが生成されます。レポートには、[Assistant] ビューの [Emulation-HW] フォルダーから簡単にアクセスできます。

次の図は、[Assistant] ビューに `binary_container_1` の `run` という名前のカーネルのシステム見積もりレポートを表示したところを示しています。



コマンド ライン

次のコマンドでは、`kernel.cl` のすべてのカーネルのシステム パフォーマンス見積もりレポート `system_estimate.xtxt` が生成されます。

```
xocc -c -t hw_emu --platform xilinx:adm-pcie-7v3:1ddr:3.0 --report
estimate kernel.cl
```

`xocc -report estimate` で生成されるパフォーマンス見積もりレポートには、アプリケーションのバイナリ コンテナー別およびデザインの計算ユニット別に情報が含まれます。レポートには、次が含まれます。

- ターゲット デバイス情報
- アプリケーションのカーネルごとのサマリ
- バイナリ コンテナーごとの詳細な情報

データの説明

次のレポート ファイル例は、見積もりレポートの情報を示しています。

```
================================================================
Design Name:              _xocc_compile_kernel_bin.dir
Target Device:            xilinx:adm-pcie-ku3:2ddr-xpr:3.3
Target Clock:             200MHz
Total number of kernels: 1
================================================================

Kernel Summary
Kernel Name    Type  Target              OpenCL Library  Compute Units
-------------  ----  ------------------  --------------  -------------
smithwaterman  clc   fpga0:OCL_REGION_0  xcl_xocc        1


================================================================
OpenCL Binary:     xcl_xocc
Kernels mapped to: clc_region
```

```
Timing Information (MHz)
Compute Unit     Kernel Name    Module Name    Target Frequency
---------------  -------------  -------------  ----------------
smithwaterman_1  smithwaterman  smithwaterman  200

Estimated Frequency
-------------------
202.020203

Latency Information (clock cycles)
Compute Unit     Kernel Name    Module Name    Start Interval
---------------  -------------  -------------  --------------
smithwaterman_1  smithwaterman  smithwaterman  29468

Best Case  Avg Case  Worst Case
---------  --------  ----------
29467      29467     29467

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

デザインおよびターゲット デバイスのサマリ

すべてのデザイン見積もりレポートの最初に、ターゲット ハードウェアに関するアプリケーションのサマリと情報が表示されます。デバイス情報は、レポートのその後のセクションに含まれます。

```
--------------------------------------------------------------------------------
Design Name:              _xocc_compile_kernel_bin.dir
Target Device:            xilinx:adm-pcie-ku3:2ddr-xpr:3.3
Target Clock:             200MHz
Total number of kernels: 1
--------------------------------------------------------------------------------
```

デザイン サマリのためにユーザーが提供する情報は、デザイン名とターゲット デバイスだけです。このセクションには、そのほかターゲット ボードおよびクロック周波数も含まれます。

ターゲット ボードは、SDAccel™ 開発環境でコンパイルされたアプリケーションを実行するボードの名前になります。クロック周波数は、FPGA ファブリックにマップされた計算ユニットのためにどれくらいロジックが速く実行されるかを定義したものです。これらのパラメーターは、どちらもデバイス開発者が変更するもので、SDAccel 環境内からは変更できません。

カーネル サマリ

カーネル サマリのセクションには、現在の SDAccel™ ソリューションに対して定義されたカーネルすべてがリストされます。次は、カーネル サマリの例です。

```
Kernel Summary
Kernel Name    Type  Target                OpenCL Library  Compute Units
-------------  ----  --------------------  --------------  -------------
smithwaterman  clc   fpga0:OCL_REGION_0    xcl_xocc        1
```

カーネル名と共に、実行ターゲットおよび入力ソースのタイプが表示されます。OpenCL™、C、C/C++ ソース ファイルではコンパイルおよび最適化手法が違うので、カーネル ソース ファイルのタイプが指定されます。

[Kernel Summary] セクションは、レポートの最後に表示されるサマリ情報で、この後には各計算ユニットのバイナリ コンテナーに関する詳細な情報が含まれます。

タイミング情報

各バイナリ コンテナーの詳細セクションには、まずすべての計算ユニットの実行ターゲットが表示されます。また、計算ユニットごとにタイミング情報も表示されます。通常、見積もられた周波数がデバイス ターゲットのものよりも多ければ、計算ユニットはハードウェアで実行できます。見積もられた周波数がターゲット周波数よりも少なければ、その計算ユニットのカーネル コードをさらに最適化して、計算ユニットが FPGA ファブリックで正しく実行されるようにする必要があります。次は、その例です。

```
OpenCL Binary:      xcl_xocc
Kernels mapped to: clc_region

Timing Information (MHz)
Compute Unit      Kernel Name     Module Name     Target Frequency
---------------   -------------   -------------   ----------------
smithwaterman_1   smithwaterman   smithwaterman   200

Estimated Frequency
-------------------
202.020203
```

タイミング情報で重要なのは、ターゲット周波数と見積もられた周波数の違いです。計算ユニットは FPGA ファブリック内で隔離して配置はされません。計算ユニットは、アプリケーションのクラスをサポートするために有効な FPGA デザイン (デバイス開発者の定義したその他のコンポーネントを含めることが可能) の一部として配置されます。

計算ユニット カスタム ロジックは一度にカーネルを 1 つずつ生成するので、デバイス ターゲットよりも見積もられた周波数が多ければ、SDAccel™ 環境を使用した開発者は FPGA プログラミング ファイルの作成中に問題が発生しないと判断できます。

レイテンシ情報

レイテンシ情報には、バイナリ コンテナーの各計算ユニットの実行プロファイルが含まれます。このデータを解析する際は、すべての値がカスタム ロジックを使用した計算ユニット境界から計測されていることに注意してください。グローバル メモリへのデータ転送に関連するインシステム レイテンシは、これらの値の一部としてはレポートされません。また、このレイテンシ数は FPGA ファブリックでターゲットとなる計算ユニットに対してのみレポートされます。次はレイテンシ レポートの例です。

```
Latency Information (clock cycles)
Compute Unit      Kernel Name     Module Name     Start Interval   Best Case
---------------   -------------   -------------   --------------   ---------
smithwaterman_1   smithwaterman   smithwaterman   29468            29467

Avg Case   Worst Case
--------   ----------
29467      29467
```

レイテンシ レポートは、次のフィールドに分けられます。

- 開始間隔
- ベスト ケース レイテンシ
- 平均ケース レイテン
- ワースト ケース レイテンシ

開始間隔は、任意のカーネルの計算ユニットの起動間にかかる時間を定義します。

ベスト、平均、ワースト ケース レイテンシの数は、計算ユニットがそのカーネルの 1 つの NDRange データ タイルの結果を生成するのにかかる時間を示します。カーネルにデータ依存の計算ループがない場合、レイテンシの値は同じになります。ループのデータ依存実行があると、データごとにレイテンシが変わります。これらはレイテンシ レポートに含まれます。

間隔およびレイテンシの数は、次のいずれかまたは複数の条件の場合、カーネルに対して「undef」 (未定義) とレポートされます。

- OpenCL™ カーネルに明示的な `reqd_work_group_size(x,y,z)` がない
- 可変境界付きのループがある

注記: レイテンシ情報は、ループの変更とそのモデルの並列処理の解析に基づいた見積もりを反映します。パイプライン処理およびデータフローなどの高度な変更があると、実際のスループット数がかなり変わります。このため、レイテンシは run 間の相対的なガイドとしてのみ使用してください。

エリア情報

FPGA はある種の空白の計算キャンバスと考えることもできますが、各 FPGA で使用可能な基本的な構築ブロック数には限りがあります。これらの基本的な構築ブロック (FF、LUT、DSP、ブロック RAM) は、SDAccel™ 開発環境でデザイン内の計算ユニットごとにカスタム ロジックを生成するために使用します。FPGA ファブリックに同時に読み込むことができる計算ユニット数は、計算ユニット内のカスタム ロジックにインプリメントする必要のある各基本リソースの数によって決まります。次の例は、計算ユニットに対してレポートされるエリア情報を示しています。

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

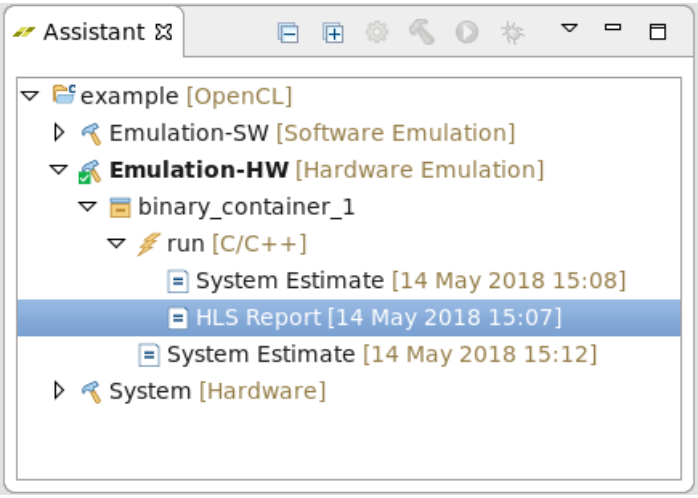
HLS レポート

SDx™ GUI または XOCC コマンド ラインを使用してカーネルをコンパイルしたら、HLS (高位合成) レポートが生成されます。HLS レポートには、ユーザー カーネル コードから生成されたカスタム ハードウェア ロジックのパフォーマンスおよび使用量についての詳細が含まれます。カーネル コンパイル結果の詳細がわかるので、カーネル最適化をしやすくなります。

GUI フロー

SDx™ GUI を使用してカーネルをコンパイルしたら、[Assistant] ビューで高位合成 (HLS) を表示できます。レポートは、[Emulation-HW] または [System] ビルド コンフィギュレーションの下の *binary container* および *kernel* 名が付いたものです。次の [Assistant] ビューはそれを示しています。

図 4: [Assistant] ビュー



コマンド ライン

HLS レポートは、SDAccel™ GUI で表示するものですが、コマンド ラインユーザー用にテキスト記述でも生成されます。このレポートは HLS ソリューション ディレクトリのカーネル合成ディレクトリの下のレポート ディレクトリ内に含まれます。

`xocc` コマンドを使用すると、この合成ディレクトリよりも上の階層に複数のディレクトリができるので、単にファイル名で見つけることをお勧めします。

```
find . -name <module>_csynth.rpt
```

`<module>` は、HLS 合成レポートに表示されるモジュールの名前です。

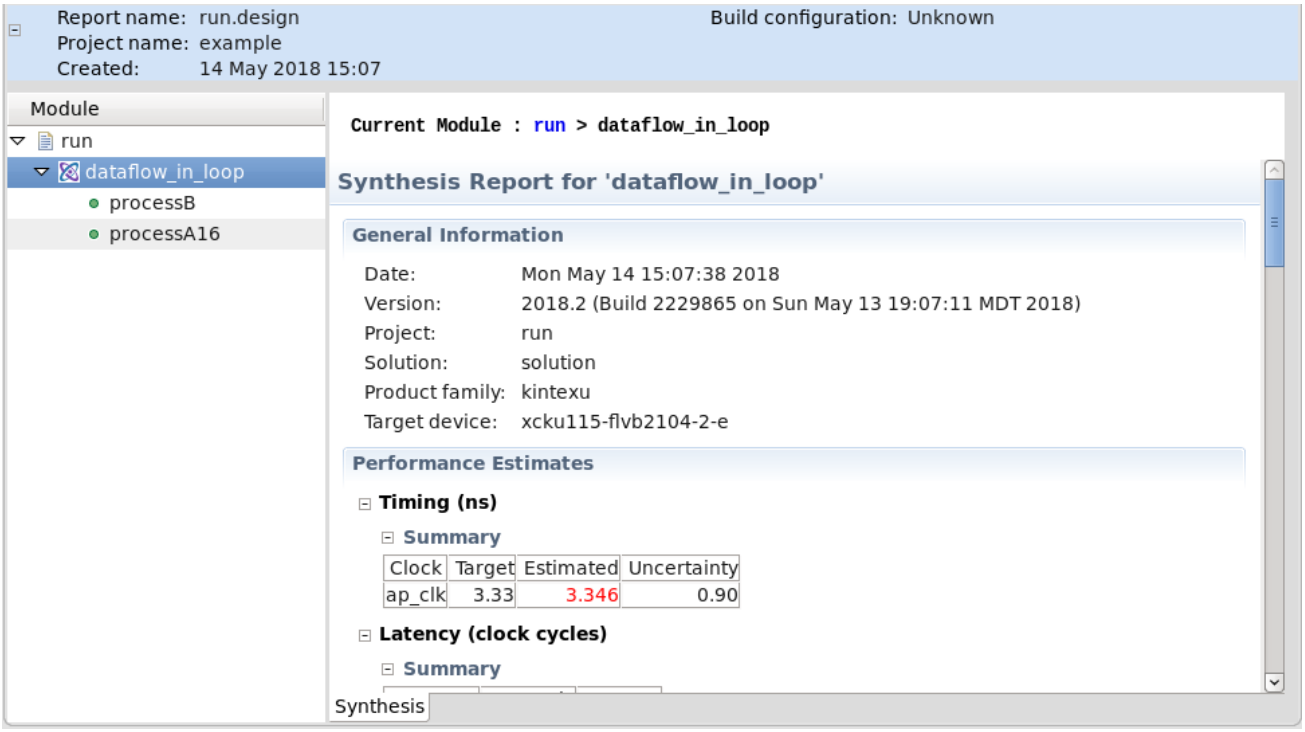
注記: `find` コマンドでは、次のコマンドのようにワイルドカードを使用した検索がサポートされており、下位ディレクトリのすべての合成レポートを見つけることができます。

```
find . -name "*_csynth.rpt"
```

データの説明

[HLS Report] の左側にはモジュール階層が表示されます。この階層には、高位合成 run の一部として生成された各モジュールが含まれます。SDAccel™ 環境では、これらのモジュールのいずれかを選択すると、右側の [Synthesis Report] タブでそのモジュールの合成の詳細が表示されます。

図 5: HLS レポート ウィンドウ



[Synthesis] レポートには [General Information]、[Performance Estimates] (タイミング & レイテンシ)、[Utilization Estimates]、[Interface Information] などの複数のセクションが含まれます。この情報が階層ブロックの一部である場合は、その階層に含まれるブロックの情報すべてがまとめられます。どのインスタンスがデザイン全体の何に対して影響しているのかがわかっているのであれば、階層はレポート内からも確認できます。

注意: サイクル/レイテンシの絶対数に関しては、これらの数が合成中にわかった見積もりに基づいているので、パイプラインおよびデータフローなどの高度な変更を使用する場合は特に最終的な結果を正確に反映していないことがあります。レポートにクエスチョン マーク (?) が表示される場合、原因は可変境界ループにある可能性があります。このようなループにはトリップカウントを設定して、相対的な見積もりがレポートに表示されるようにすることをお勧めします。

プロファイル サマリ レポート

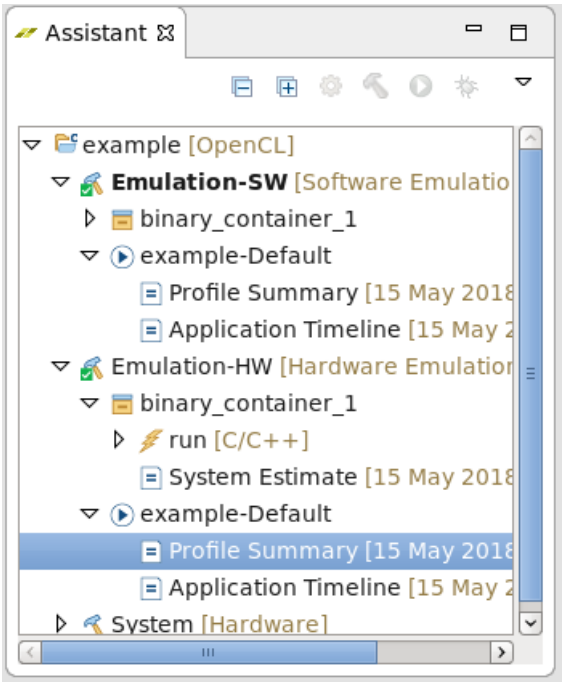
SDAccel™ ランタイムはホスト アプリケーションで自動的にプロファイリング データを収集します。アプリケーションが実行を終了すると、ソリューション レポート ディレクトリまたは作業ディレクトリ内にプロファイル サマリが HTML、`.csv`、および Google Protocol Buffer フォーマットで保存されます。これらのレポートはウェブ ブラウザー、スプレッドシート ビューアー、または SDAccel の [Profile Summary] ビューで表示できます。プロファイル レポートは SDAccel GUI および XOCC コマンド ライン フローの両方で生成されます。

GUI フロー

SDAccel™ 環境からアプリケーションをコンパイルおよび実行すると、プロファイル サマリが自動的に生成されます。プロファイル情報の生成を制御するには、[Run]→[Run Configurations] をクリックして、ビルド コンフィギュレーションのコンテキスト メニューから run コンフィギュレーションを編集します。

実行したら [Assistant] ビューの [Run Configuration] の下の項目からレポートを表示できるようになります。[Run Configuration] を一度実行しておくと、[Assistant] ビューの run コンフィギュレーションのコンテキスト メニューから直接コンフィギュレーションを変更できるようになります。

図 6: SDAccel GUI フローでのプロファイル サマリ



レポートは、ダブルクリックすると開きます。

コマンド ライン

コマンド ライン ユーザーの場合は、SDAccel™ 環境外でスタンドアロン アプリケーションを実行します。プロファイル サマリ データを生成する場合は、その他のオプションなしにデザインをコンパイルできます。ビットストリーム ファイル (xclbin) のリンクには、`--profile_kernel` オプションが必要です。

`--profile_kernel` オプションで指定した引数は、データ収集の制限 (大型システムに必要な場合あり) に使用できます。プロファイル サマリ レポートに関する profile_kernel オプションの一般的な構文は、次のとおりです。

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

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`、`compute_unit_name`、および `interface_name` を明示的に指定してインスト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: `profile_kernel` オプションは追加していくことができるので、リンク ライン上で複数回使用できます。

`profile = true` を `sdaccel.ini` ファイルで指定した場合、プログラムを実行すると `sdaccel_profile_summary.csv` ファイルが作成されます。

```
[Debug]
profile = true
```

`.csv` ファイルは、プロファイリング結果を [Profile Summary] ビューで表示する前に、手動で Google Protocol Buffer 形式 (`.xprf`) に変換する必要があります。次は、`.csv` 入力ファイルから `.xprf` ファイルを生成するコマンド ライン例です。

```
sdx_analyze profile sdaccel_profile_summary.csv
```

プロファイル サマリの表示

コマンド ラインから作成した SDAccel™ プロファイル サマリを表示する方法は、次のとおりです。

ウェブ ブラウザー

HTML プロファイル サマリをウェブ ブラウザーに表示にするには、まず次のコマンドを使用してデータを含む HTML ファイルを作成する必要があります。

```
sdx_analyze profile -i
sdaccel_profile_summary.csv -f html
```

これで HTML ファイルが作成されたので、任意のウェブ ブラウザーで開くことができるようになりました。次は、FPGA のシステム run からのプロファイリング結果を示しています。

SDAccel Profile Summary

Application: host

Created: 2018-06-25 19:34:22

Devices: xilinx_vcu1525_dynamic_5_2-0

Msec: 1529976862881

Report name: Profile Summary

Target: System Run

Tool version: 2018.2

OpenCL API Calls

API Name	Number Of Calls	Total Time (ms)	Minimum Time (ms)	Average Time (ms)	Maximum Time (ms)
clCreateProgramWithBinary	1	1954.600	1954.600	1954.600	1954.600
clReleaseProgram	1	12.004	12.004	12.004	12.004
clWaitForEvents	6	4.989	0.013	0.831	1.437
clCreateBuffer	2	3.020	1.428	1.510	1.592
clEnqueueReadBuffer	1	1.308	1.308	1.308	1.308
clEnqueueNDRangeKernel	5	0.810	0.149	0.162	0.192
clEnqueueWriteBuffer	1	0.617	0.617	0.617	0.617
clReleaseKernel	1	0.460	0.460	0.460	0.460
clReleaseEvent	6	0.077	0.010	0.013	0.015
clGetPlatformIDs	2	0.061	0.006	0.030	0.054
clSetKernelArg	4	0.045	0.008	0.011	0.016
clCreateKernel	1	0.041	0.041	0.041	0.041
clCreateContext	1	0.028	0.028	0.028	0.028
clReleaseMemObject	2	0.024	0.005	0.012	0.019
clGetDeviceInfo	3	0.023	0.007	0.008	0.009
clCreateCommandQueue	1	0.022	0.022	0.022	0.022
clGetDeviceIDs	1	0.014	0.014	0.014	0.014
clReleaseContext	1	0.012	0.012	0.012	0.012
clGetPlatformInfo	1	0.012	0.012	0.012	0.012
clReleaseCommandQueue	1	0.011	0.011	0.011	0.011

Kernel Execution

Kernel	Number Of Enqueues	Total Time (ms)	Minimum Time (ms)	Average Time (ms)	Maximum Time (ms)
median	5	4.380	0.857	0.876	0.893

Compute Unit Utilization

Device	Compute Unit	Kernel	Global Work Size	Local Work Size	Number Of Calls	Total Time (ms)	Minimum Time (ms)	Average Time (ms)	Maximum Time (ms)	Clock Frequency (MHz)
xilinx_vcu1525_dynamic_5_2-0	median_1	median	1:1:1	1:1:1	5	2.998	0.599	0.600	0.600	300

Compute Units: Stall Information

Compute Unit	Execution Count	Running Time (ms)	Intra-Kernel Dataflow Stalls (ms)	External Memory Stalls (ms)	Inter-Kernel Pipe Stalls (ms)
median_1	5	2.998	0.000	0.327	0.000

Data Transfer: Host and Global Memory

Context:Number of Devices	Transfer Type	Number Of Transfers	Transfer Rate (MB/s)	Average Bandwidth Utilization (%)	Average Size (KB)	Total Time (ms)	Average Time (ms)
context0:1	READ	1	919.705891	9.580270	1048.580	1.140121	1.140121
context0:1	WRITE	1	2708.924724	28.217966	1048.580	0.387082	0.387082

Data Transfer: Kernels and Global Memory

Device	Compute Unit/ Port Name	Kernel Arguments	DDR Bank	Transfer Type	Number Of Transfers	Transfer Rate (MB/s)	Average Bandwidth Utilization (%)	Average Size (KB)	Average Latency (ns)
xilinx_vcu1525_dynamic_5_2-0	median_1/M_AXI_GMEM	input_r/output_r	1	READ	5130	1752.250	15.210	1.024	373.771
xilinx_vcu1525_dynamic_5_2-0	median_1/M_AXI_GMEM	input_r/output_r	1	WRITE	5120	1748.840	15.181	1.024	230.277

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)	Total Transfer Rate (MB/s)
xilinx_vcu1525_dynamic_5_2-0	median_1	10250	1024.000	25.000	10.496	5.243	5.253	3501.090

Top Kernel Execution

Kernel Instance Address	Kernel	Context ID	Command Queue ID	Device	Start Time (ms)	Duration (ms)	Global Work Size	Local Work Size
0x156efa0	median	0	0	xilinx_vcu1525_dynamic_5_2-0	2615.060	0.893	1:1:1	1:1:1
0x156efa0	median	0	0	xilinx_vcu1525_dynamic_5_2-0	2617.300	0.892	1:1:1	1:1:1
0x156efa0	median	0	0	xilinx_vcu1525_dynamic_5_2-0	2616.200	0.872	1:1:1	1:1:1
0x156efa0	median	0	0	xilinx_vcu1525_dynamic_5_2-0	2619.490	0.867	1:1:1	1:1:1
0x156efa0	median	0	0	xilinx_vcu1525_dynamic_5_2-0	2618.420	0.857	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)
0x4400000000	0	0	2614.610	0.387082	1048.580	2708.924724

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)
0x4400100000	0	0	2620.590	1.140121	1048.580	919.705891

Profile Rule Checks (16 met, 6 warnings)

Rule	Threshold Value	Actual Value	Conclusion	Details	Guidance
Average Read Size (KB)	> 0.512	1.024	Met	Average kernel read transfer size was adequate.	
Average Write Size (KB)	> 0.512	1.024	Met	Average kernel write transfer size was adequate.	
Read Bandwidth (%)	> 5.000	15.210	Met	Kernel read transfers were efficient from off-chip global memory.	
Write Bandwidth (%)	> 5.000	15.181	Met	Kernel write transfers were efficient to off-chip global memory.	
Read Amount - Minimum (MB)	> 0.250	5.010	Met	Compute units on all devices used adequate data from host.	
Read Amount - Maximum (MB)	< 2.000	5.010	Not Met	Total kernel read of 5.253 MB on xilinx_vcu1525_dynamic_5_2-0 was 500.975% of host data.	Re-use data with local or private memories. SDAccel Profiling and Optimization Guide
Port Data Width	= 512	512	Met	All kernel ports utilized the full memory data width.	
DDR Bank Connections	> 0	0	Not Met	DDR bank 0 was not used.	Utilize all DDR banks to maximize performance. SDAccel Profiling and Optimization Guide
DDR Bank Connections	> 0	1	Met	DDR bank 1 was used.	
DDR Bank Connections	> 0	0	Not Met	DDR bank 2 was not used.	Utilize all DDR banks to maximize performance. SDAccel Profiling and Optimization Guide
DDR Bank Connections	> 0	0	Not Met	DDR bank 3 was not used.	Utilize all DDR banks to maximize performance. SDAccel Profiling and Optimization Guide
DDR Bank Read Bandwidth (%)	> 5.000	15.210	Met	DDR bank read transfers were efficient from off-chip global memory.	
DDR Bank Write Bandwidth (%)	> 5.000	15.181	Met	DDR bank write transfers were efficient from off-chip global memory.	
Migrate Memory API Calls	> 0	0	Not Met	Migrate Memory OpenCL APIs were not used.	Utilize the Migrate Memory APIs (e.g., clEnqueueMigrateMemObjects). SDAccel Profiling and Optimization Guide
Average Read Size (KB)	> 4.096	1048.580	Met	Host read transfers were efficient from off-chip global memory.	
Average Write Size (KB)	> 4.096	1048.580	Met	Host write transfers were efficient to off-chip global memory.	
Compute Unit Calls - Minimum	> 0	5	Met	Compute unit median_1 was used.	
Compute Unit Utilization (%)	> 20.000	68.441	Met	Compute unit median_1 had sufficient utilization.	
Compute Unit Calls - Maximum	< 16	1	Met	Kernel median was used an adequate amount.	
Kernel Utilization (%)	= 100.000	100.000	Met	Kernel median utilized correct amount of workgroups.	
Compute Unit Count	> 1	1	Not Met	Kernel median was executed 5 time(s) with 1 compute unit(s).	Ensure kernel utilizes multiple compute units. SDAccel Profiling and Optimization Guide
Device Utilization (ms)	> 0	5.297	Met	Device xilinx_vcu1525_dynamic_5_2-0 was used.	

[Profile Summary] ウィンドウ

[Profile Summary] ウィンドウには ライン フローで生成されたプロファイル サマリを表示できます。

次の手順に従って、[Profile Summary] ウィンドウにプロファイル サマリを開きます。

1. `.csv` データ ファイルを protobuf 形式に変換します。

```
sdx_analyze profile -i
sdaccel_profile_summary.csv -f protobuf
```

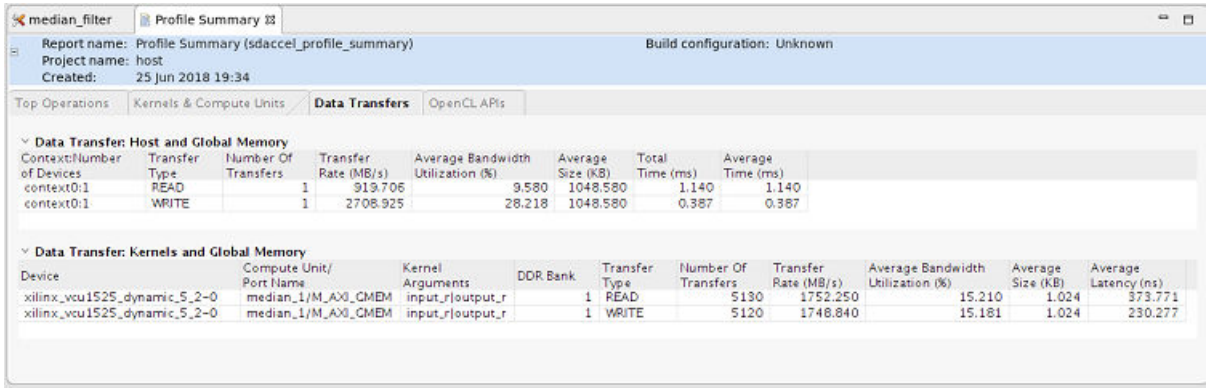
2. SDAccel GUI を `sdx` コマンドで開始します。

```
$sdx
```

3. デフォルトのワークスペースを選択します。
4. [File]→[Open File] をクリックし、手順 1 の `sdx_analyze` コマンドで作成した `.xprf` ファイルを開きます。

次は、OpenCL™ API 呼び出し、カーネル実行、データ転送、およびプロファイル ルール チェック (PRC) を表示する [Profile Summary] ウィンドウを示しています。

図 7: [Profile Summary] ウィンドウ



データの説明

プロファイル サマリには、OpenCL™ アプリケーションに役立つさまざまな統計が含まれ、アプリケーションの機能的なボトルネックの概要が表示されます。プロファイル サマリには、次の表が含まれます。

- [Top Operations]
  - [Top Data Transfer: Kernels and Global Memory]: 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。
    - [Device]: デバイス名
    - [Compute Unit]: 計算ユニット名
    - [Number of Transfers]: デバイスで監視される書き込みおよび読み出し AXI トランザクションの合計
    - [Average Bytes per Transfer]: (読み出しバイト合計 + 書き込みバイト合計) / (読み出し AXI トランザクション合計 + 書き込み AXI トランザクション合計)

- [Transfer Efficiency (%)]: (転送ごとの平均バイト) / min(4K, (メモリ ビット幅/8 * 256))

  AXI4 仕様では、最大バースト長が 256、最大バースト サイズが 4K バイトに制限されています。
- [Total Data Transfer (MB)]: (読み出しバイト合計 + 書き込みバイト合計) / 1.0e6
- [Total Write (MB)]: (書き込みバイト合計) / 1.0e6
- [Total Read (MB)]: (読み出しバイト合計) / 1.0e6
- [Transfer Rate (MB/s)]: (データ転送合計) / (計算ユニットの合計時間)

◦ [Top Kernel Execution]

- [Kernel Instance Address]: カーネル インスタンスのホスト アドレス (16 進数)
- [Kernel]: カーネル名
- [Context ID]: ホストのコンテキスト ID
- [Command Queue ID]: ホストのコマンド キュー ID
- [Device]: カーネルが実行されたデバイス名 (フォーマット: `<device>-<ID>`)
- [Start Time (ms)]: 実行の開始時間 (ms)
- [Duration (ms)]: 実行期間 (ms)
- [Global Work Size]: カーネルの NDRange
- [Local Work Size]: カーネルのワーク グループ サイズ

◦ [Top Memory Writes: Host and Device Global Memory]

- [Buffer Address]: バッファーのホスト アドレス (16 進数)
- [Context ID]: ホストのコンテキスト ID
- [Command Queue ID]: ホストのコマンド キュー ID
- [Start Time (ms)]: 書き込み転送の開始時間 (ms)
- [Duration (ms)]: 書き込み転送期間 (ms)
- [Buffer Size (KB)]: 書き込み転送サイズ (KB)
- [Writing Rate (MB/s)]: 書き込みレート = (バッファー サイズ) / (期間)

◦ [Top Memory Reads: Host and Device Global Memory]

- [Buffer Address]: バッファーのホスト アドレス (16 進数)
- [Context ID]: ホストのコンテキスト ID
- [Command Queue ID]: ホストのコマンド キュー ID
- [Start Time (ms)]: 読み出し転送の開始時間 (ms)
- [Duration (ms)]: 読み出し転送期間 (ms)
- [Buffer Size (KB)]: 読み出し転送サイズ (KB)
- [Reading Rate (MB/s)]: 読み出しレート = (バッファー サイズ) / (期間)

- [Kernels & Compute Units]

  ◦ [Kernel Execution (includes estimated device times)]: スケジュールおよび実行されたカーネルすべてのプロファイル データ サマリを表示します。

- [Kernel]: カーネル名
- [Number of Enqueues]: カーネルが待機キューに追加される回数
- [Total Time (ms)]: すべての待機キューのランタイム合計 (OpenCL 実行モデルで START から END まで測定)
- [Minimum Time (ms)]: すべての待機キューの最小ランタイム
- [Average Time (ms)] (合計時間) / (待機キュー数)
- [Maximum Time (ms)]: すべての待機キューの最大ランタイム

◦ [Compute Unit Utilization (includes estimated device times)]: FPGA のすべての計算ユニットのサマリ プロファイル データを表示します。

- [Device]: デバイス数 (フォーマット: `<device>-<ID>`)
- [Compute Unit]: 計算ユニット名
- [Kernel]: 計算ユニットが関連付けられるカーネル
- [Global Work Size]: カーネルの NDRange (フォーマットは x:y:z)
- [Local Work Size]: ローカル ワーク グループ サイズ (フォーマットは x:y:z)
- [Number of Calls]: 計算ユニットが呼び出される回数
- [Total Time (ms)]: すべての呼び出しのランタイム合計
- [Minimum Time (ms)]: すべての呼び出しの最小ランタイム
- [Average Time (ms)]: (合計時間) / (ワーク グループ数)
- [Maximum Time (ms)]: すべての呼び出しの最大ランタイム
- [Clock Frequency (MHz)]: 該当アクセラレータに使用するクロック周波数 (MHz)

• [Data Transfers]

◦ [Data Transfer: Host and Global Memory]: PCI Express® リンクを介したホストおよびデバイス メモリ間の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。

- [Context:Number of Devices]: コンテキスト ID およびコンテキスト内のデバイス数
- [Transfer Type]: [READ] または [WRITE]
- [Number of Transfers]: ホスト データ転送数 (注記: `printf` 転送を含む可能性あり)
- [Transfer Rate (MB/s)] (送信バイト合計) / (合計時間 (usec))

  合計時間にはソフトウェア オーバーヘッドが含まれます。
- [Average Bandwidth Utilization (%)]: (転送レート) / (最大転送レート)

  最大転送レート = (256/8 バイト) * (300 MHz) = 9.6 GBps
- [Average Size (KB)]: (送信 KB 合計) / (転送数)
- [Total Time (ms)]: 転送時間合計
- [Average Time (ms)]: (合計時間) / (転送数)

◦ [Data Transfer: Kernels and Global Memory]: FPGA およびデバイス メモリ間の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。

- [Device]: デバイス名
- [Compute Unit/Port Name]: <計算ユニット名>/<ポート名>

    - [Kernel Arguments]: このポートに接続される引数のリスト
    - [DDR Bank]: このポートが接続される DDR バンク数
    - [Transfer Type]: [READ] または [WRITE]
    - [Number of Transfers]: デバイスで監視される AXI トランザクション数 (注記: `printf` 転送を含む可能性あり)
    - [Transfer Rate (MB/s)]: (送信バイト合計) / (計算ユニット合計時間)
        - 計算ユニット合計時間 = 計算ユニットの合計実行時間
        - 送信バイト合計 = すべてのトランザクションのバイト合計
    - [Average Bandwidth Utilization (%)]: (転送レート) / (0.6 *最大転送レート)

      最大転送レート = (512/8 bytes) * (300 MHz) = 19200 MBps
    - [Average Size (KB)]: (送信 KB 合計) / (AXI トランザクション数)
    - [Average Latency (ns)]: (全トランザクションのレイテンシ合計) / (AXI トランザクション数)
- [`OpenCL API Calls`]: ホスト アプリケーションで実行されるすべての OpenCL ホスト API 関数呼び出しのプロファイル データを表示します。
    - [API Name]: API 関数名 (例: clCreateProgramWithBinary、clEnqueueNDRangeKernel)
    - [Number of Calls]: この API への呼び出し数
    - [Total Time (ms)]: すべての呼び出しのランタイム合計
    - [Minimum Time (ms)]: すべての呼び出しの最小ランタイム
    - [Average Time (ms)]: (合計時間) / (呼び出し数)
    - [Maximum Time (ms)]: すべての呼び出しの最大ランタイム

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。
- AXI トランザクションの開始/停止、カーネルの開始/停止を含むデバイス トレース データ。

タイムラインおよびデバイス トレース データはデフォルトでは収集されません。これは、FPGA からのトレース データを定期的にアンロードする必要があるために、全体的なアプリケーション実行に余計な時間がかかるからです。ただし、デバイス データは、データ コレクションが FPGA のカーネル機能に影響しないように、FPGA 内の専用ハードウェアを使用して収集されます。次のセクションでは、時間とデバイスのデータ収集を有効にするために必要な設定について説明します。

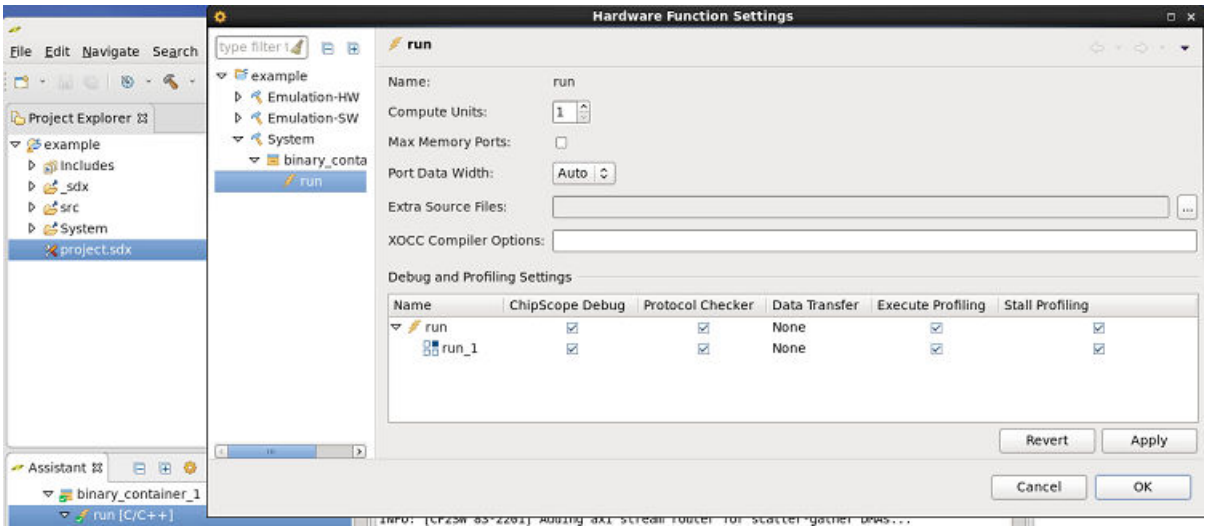
デバイスのプロファイリングは介入的であり、全体的なパフォーマンスに悪影響を与える可能性があります。この機能は、システム パフォーマンスのデバッグにのみ使用してください。

注記: デバイス プロファイリングは、ハードウェア エミュレーションで悪影響なく使用できます。

GUI フロー

タイムラインおよびデバイス トレース データ コレクションは、SDAccel™ 統合環境から作成された SDAccel プロジェクトの run コンフィギュレーションの一部です。イネーブルにするには、次の手順を使用します。

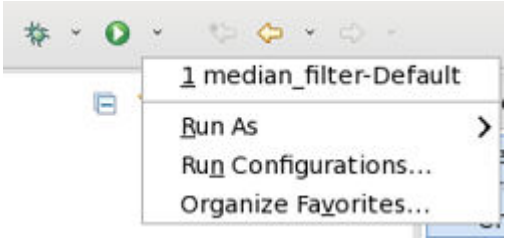
1. システム実行には、コードを必ず含める必要があります。これは、[Hardware Function Settings] ダイアログ ボックス ([Assistant] → [Settings]) から指定できます。



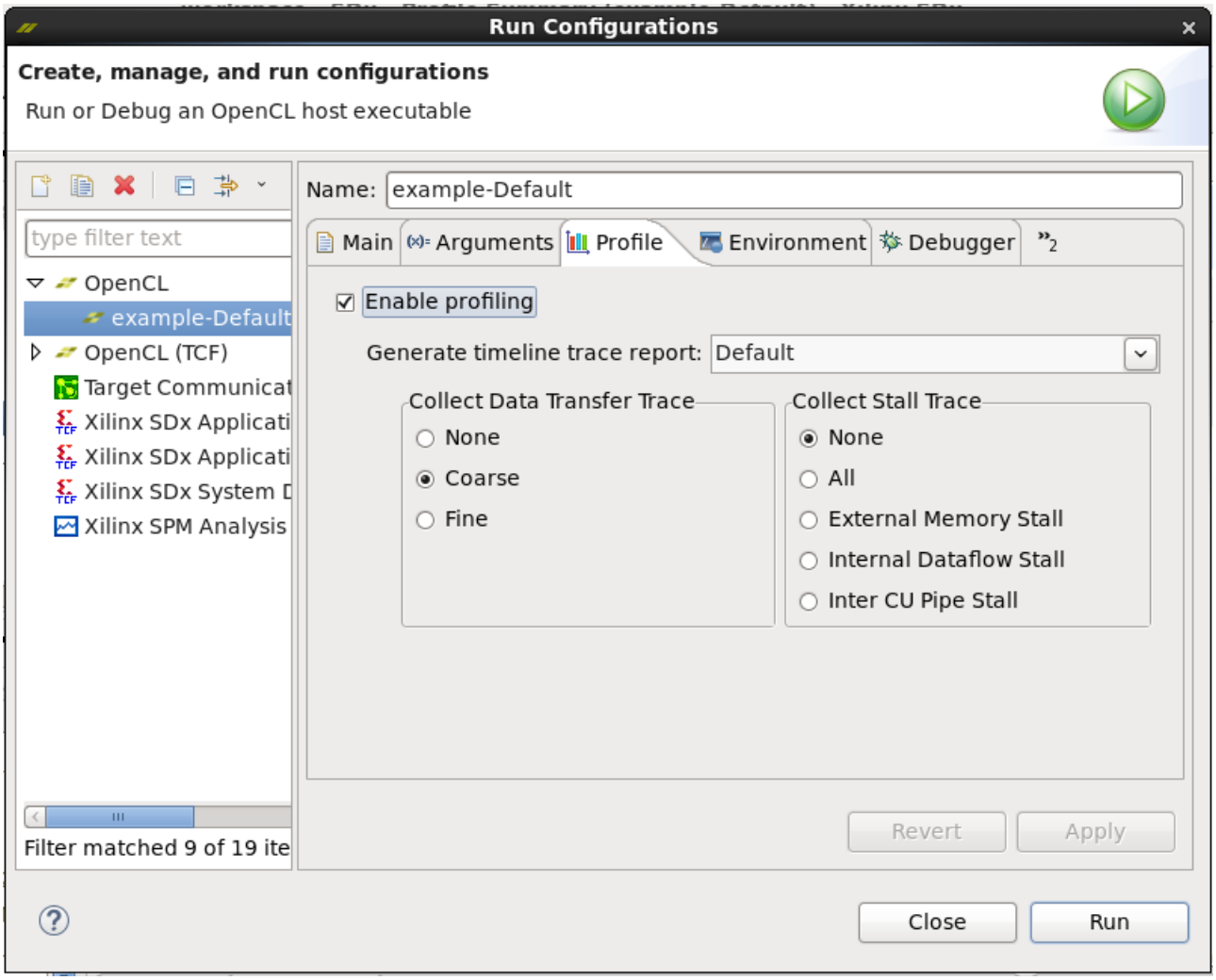
アプリケーション タイムライン機能では、[Data Transfer]、[Execute Profile]、および [Stall Profiling] をオンにできます。これらのオプションでは、カーネルの各インスタンスのポートすべてが使用されます。その他のハードウェアが挿入されるので、すべてのポートを使用すると多すぎることがあります。後の方では、コマンド ライン オプションを使用してさらに制御ができるようになります。詳細は、コマンド ライン セクションを参照してください。これらのオプションは、システム run に対してのみ有効です。ハードウェア エミュレーション時には、このデータはデフォルトで生成されます。

- `Data Transfer`: データ ポートの監視が有効になります。
- `Execute Profiling`: システム run 中に最小限のポート データ コレクションが提供されます。計算ユニットの実行時間が記録されます。データおよびストール プロファイリング用にデフォルトでオンになっています。
- `Stall Profiling`: ビットストリームにストール監視ロジックを含めます。

2. どの情報が実際に run でレポートされるのかを指定します。システム実行中のハードウェアからの実際の情報のみがレポートされます。レポートを設定するには、[Debug] または [Run] ボタンの横の下向き矢印をクリックし、[Run Configurations] を選択して [Run Configurations] ウィンドウを開きます。



3. [Run Configurations] ウィンドウで [Profile] タブをクリックし、[Enable profiling] がオンになっていることを確認します。これにより基本的なプロファイリングがサポートされるようになります。データをトレースするには、[Generate timeline trace report] を指定して、ビルド コンフィギュレーションの情報が実際に収集されるようにする必要があります。[Default] の場合、トレース データ キャプチャはシステム実行でサポートされませんが、ハードウェア エミュレーションではイネーブルになります。



また、ランタイムに収集する情報量は選択できます。トレース データ コレクションの粒度を [Data Transfer]、[Trace]、[Stall Trace] に対して個別に選択できます。

- [Collect Data Transfer Trace] のオプションは次のとおりです。
  - [Coarse]: 計算ユニットの転送アクティビティを最初の転送の初めから最後の転送の終わり (計算ユニットの転送終了前) まで表示します。
  - [Fine]: すべての AXI レベルのバースト データ転送を表示します。
  - [None]: 読み出しをオフにして、ランタイム中のデバイス レベルのトレースがレポートされないようにします。
- [Collect Stall Trace] のオプションは次のとおりです。
  - [None]: ストール トレース情報の収集をオフにします。

- [All]: すべてのストール トレース情報を記録します。
- [Internal data flow stall]: カーネル ストリーム間 (例: データフロー ブロック間のフル FIFO へ書き込み)。
- [Internal memory stall]: DDR に対するメモリ ストール (例: DDR からの AXI-MM 読み出し)。
- [Inter compute unite pipe stall]: 内部カーネル パイプ (例: カーネル間のフル OpenCL パイプへの書き込み)。

同じプロジェクトに対して複数の run 設定がある場合は、run 設定ごとにプロファイル設定を変更する必要があります。

4. コンフィギュレーションを実行したら、[Assistant] ビューで [Application Timeline] をダブルクリックし、[Application Timeline] ウィンドウを開きます。

コマンド ライン

コマンド ライン フローでタイムラインおよびデバイス トレース データ コレクションをイネーブルにするには、次の手順に従います。

1. この手順は、SDx™ アクセル モニター (SAM) および SDx パフォーマンス モニター (SPM) を含めた FPGA ビットストリーム計装に使用します。これは 3 つのオプション (`data`、`stall`、`exec`) を含む `--profile_kernel` を使用して実行します。

注記: `--profile_kernel` オプションは、システム コンパイルおよびリンクの場合を除いて無視されます。ハードウェア エミュレーション時には、このデータはデフォルトで生成されます。

`--profile_kernel` オプションには、モニターの適用されるインターフェイスを指定するのに必要な 3 つのフィールドが含まれますが、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`、`compute_unit_name`、および `interface_name` を明示的に指定してインストルメンテーションを制限します。最後のオプション `<counters|all>` では、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: `--profile_kernel` オプションは追加していくことができるので、リンク ライン上で複数回使用できます。

- `data`: SAM および SPM IP を介してデータ ポートの監視を有効にします。このオプションは、リンク中にのみ設定する必要があります。

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

- `stall`: コンパイル中に適用する必要があります。

```
-c --profile_kernel <[stall]:<[kernel_name|all]:[compute_unit_name|
all]:[counters|all]>
```

また、リンク中にも設定する必要があります。

```
-l --profile_kernel <[stall]:<[kernel_name|all]:[compute_unit_name|
all]:[counters|all]>
```

ビットストリームにストール監視ロジック (SAM IP を使用) を含めます。ただし、カーネル インターフェイスにストール ポートが存在している必要があります。このため、C/C++/OpenCL™ カーネル モジュールのコンパイルにはこのオプションが必要です。

- `exec`: システム run 中に最小限のポート データ コレクションを提供します。SAM IP を使用して単にカーネルの実行時間を記録します。この機能は、データまたはストール データ コレクションを使用するどのポートでもデフォルトでイネーブルになります。このオプションは、リンク中にのみ提供する必要があります。

```
-l --profile_kernel <[exec]:<[kernel_name|all]:[compute_unit_name|
all]>:[counters|all]
```

2. カーネルがインストルメントされたら、ランタイム実行中にデータ収集をイネーブルにする必要があります。これには、ホスト実行ファイルと同じディレクトリにある `sdaccel.ini` ファイルを使用します。次の `sdaccel.ini` ファイルを使用すると、ランタイム中に情報が最大限に収集されます。

```
[Debug]
profile=true
timeline_trace=true
data_transfer_trace=coarse
stall_trace=all
```

- `profile=<true|false>`: true に指定すると、基本的なプロファイル モニタリングがイネーブルになります。オプションに何も指定しなくても、ホスト ランタイム ログ プロファイル サマリはイネーブルになりますが、false の場合、モニタリングはまったく実行されません。
- `timeline_trace=<true|false>`: データを収集したタイムライン トレース情報が含まれるようになります。FPGA (データ) にプロファイル IP を追加しない場合は、ホスト情報のみが表示されます。より多くの計算ユニットの開始および終了実行時間をタイムライン トレースに表示するには、最低でも `--profile_kernel exec` を使用して計算ユニットをリンクする必要があります。
- `data_transfer_trace=<coarse|fine|off>`: デバイス レベルの AXI データ転送トレースをイネーブルにします。
  - `coarse`: 計算ユニットの転送アクティビティを最初の転送の初めから最後の転送の終わり (計算ユニットの転送終了前) まで表示します。
  - `fine`: すべての AXI レベルのバースト データ転送を表示します。
  - `off`: 読み出しをオフにして、ランタイム中のデバイス レベルのトレースがレポートされないようにします。
- `stall_trace=<dataflow|memory|pipe|all|off>`: タイムライン トレースに取り込んでレポートするストールのタイプを指定します。デフォルトは `off` です。
  - `off`: ストール トレース情報の収集をオフにします。

注記: ストールをオンにすると、デザインの大部分で FIFO バッファーがいっぱいになる可能性があります。これを回避するには、`trace_stall=off` を設定します。

  - `all`: すべてのストール トレース情報を記録します。
  - `dataflow`: カーネル ストリーム間 (例: フロー ブロック間のフル FIFO へ書き込み)。
  - `memory`: 外部メモリ ストール (例: DDR からの AXI-MM 読み出し)。
  - `pipe`: 内部カーネル パイプ (例: カーネル間のフル OpenCL パイプへの書き込み)。

3. コマンド ライン モードでは、CSV ファイルが生成されてトレース データが取り込まれます。これらの CSV レポートを SDAccel GUI で開いて表示するには、`sdx_analyze` ユーティリティを使用してアプリケーション タイムライン フォーマットに変換する必要があります。

```
sdx_analyze trace sdaccel_timeline_trace.csv
```

これにより、GUI で開くことができる sdaccel_timeline_trace.wdb (デフォルト) というファイルが作成されます。

4. アプリケーション実行中にホストおよびデバイスのイベントのタイムライン レポートを表示する方法は、次のとおりです。

   a. 次のコマンドを実行して SDx IDE を開始します。

   ```
   $sdx
   ```

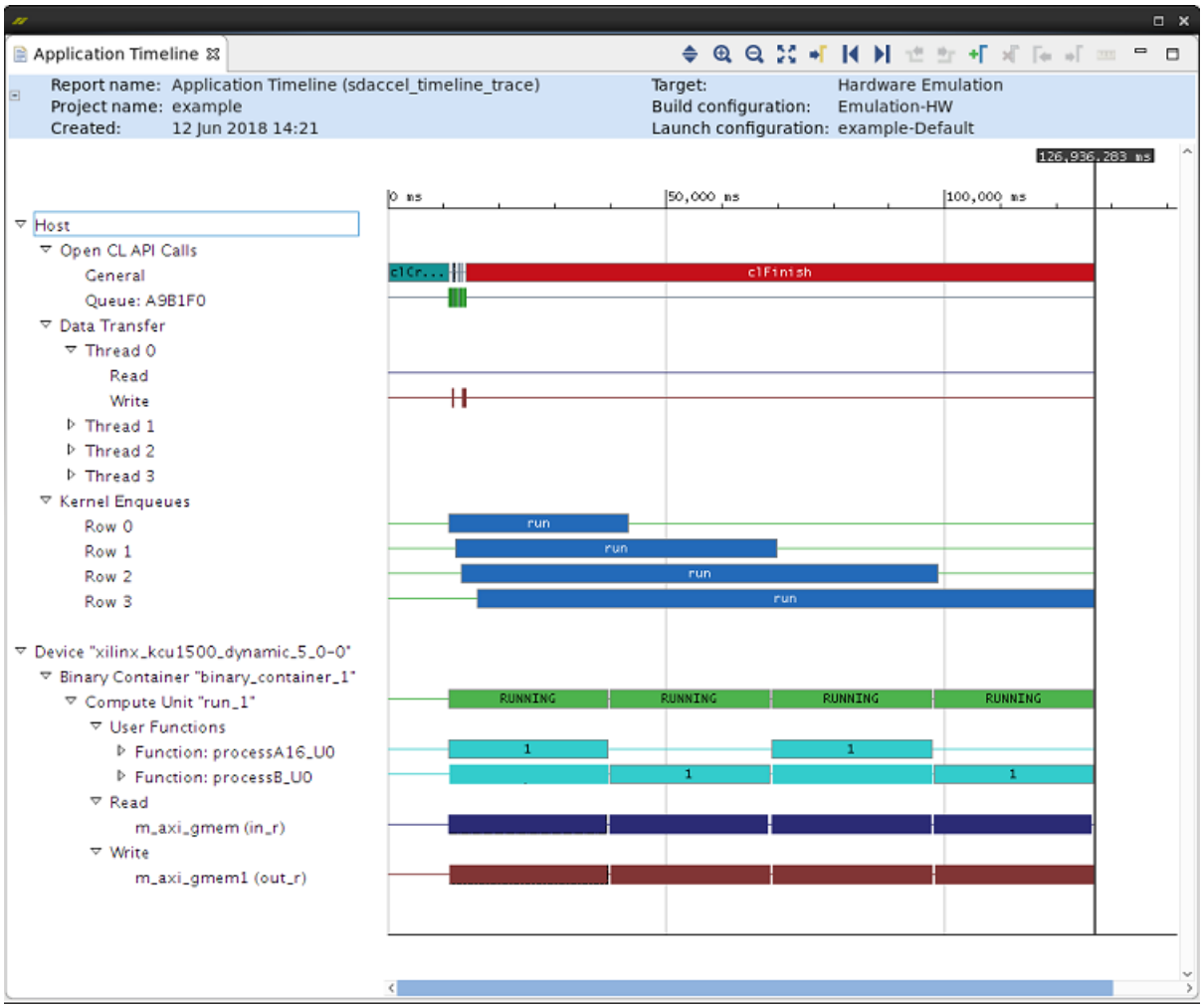
   b. プロンプトが表示されたら、ワークスペースを選択します。

   c. [File]→[Open File] をクリックし、ハードウェア エミュレーションまたはシステム実行時に生成された .wdb ファイルを選択して開きます。

データの説明

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。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 8: [Application Timeline] ウィンドウ



注記: 関数レベルのアクティビティは、ハードウェア エミュレーションでのみ可能です。

アプリケーション タイムライン トレースには、[Host] と [Device] の 2 つのセクションがあります。[Host] セクションには、ホスト側から始まるアクティビティすべてのトレースが表示され、[Device] セクションには、FPGA の計算ユニットのアクティビティが表示されます。

[Host] アクティビティの下は OpenCL™ API 呼び出し、データ転送、およびカーネルに分類されます。

計算ツリーの構造は次のようになります。

- [Host]
  - [OpenCL API Calls]: すべての OpenCL API 呼び出しがここでトレースされます。アクティビティ時間はホスト パースペクティブから測定されます。
    - [General]: clCreateProgramWithBinary()、clCreateContext()、clCreateCommandQueue などの一般的な OpenCL API 呼び出しがここでトレースされます。

- [Queue]: 特定のコマンド キューに関連する OpenCL API 呼び出しがここでトレースされます。これには、clEnqueueMigrateMemObjects、clEnqueueNDRangeKernel などのコマンドが含まれます。ユーザー アプリケーションが複数のコマンド キューを作成すると、このセクションにも同じ数のキューおよびアクティビティが表示されます。

  - [Data Transfer]: このセクションでは、ホストからデバイス メモリまでの DMA 転送がトレースされます。OpenCL ランタイムにインプリメントされる DMA スレッドは複数あり、通常は同数の DMA チャネルがあります。DMA 転送は clEnqueueMigrateMemObjects などの OpenCL API を呼び出して、ユーザー アプリケーションにより開始されます。これらの DMA リクエストがランタイムに転送されて、スレッドの 1 つにデリゲートされます。ホストからデバイスまでのデータ転送は [Write] の下、デバイスからホストまでのデータ転送は [Read] の下に表示されます。

    - [Thread 0]

      - [Read]

      - [Write]

    - [Thread 1]

    - [Thread 2]

    - [Thread 3]

  - [Kernel Enqueues]: アクティブなカーネル実行がここに表示されます。このカーネルは、デバイスのユーザー カーネル/計算ユニットと混乱しないようにしてください。このカーネルは、NDRangeKernels および API で作成されたタスク (clEnqueueNDRangeKernels() and clEnqueueTask()) のことで、ホストのパースペクティブから計測された時間でスケジュールされます。複数のカーネルを同時に実行されるようにスケジュールでき、実行されるようにスケジュールされた点からカーネル実行の終わりまでがトレースされます。複数エントリがあるのは、このためです。行数は、カーネル実行がオーバーラップする回数によって異なります。

注記: カーネルのオーバーラップは、プロセスが実際には即座に実行できる準備ができていないので、デバイス上の実際の並列実行とは異なるものです。

    - [Row 0]

    - [Row 1]

    - [Row 2]

    - [Row 3]

- [Device "name"]

  - [Binary Container "name"]

    - [Accelerator "name"]: これは FPGA の計算ユニット (別名 Accelerator) の名前です。

      - [User Functions]: HLS カーネルの場合は、データフロー プロセスとしてインプリメントされたユーザー関数がここでトレースされます。これらの関数のトレースは、現在並列で実行されているこれらの関数のアクティブなインスタンス数を示します。これらの名前は、波形がイネーブルの場合にハードウェア エミュレーションで生成されます。

注記: 関数レベルのアクティビティは、ハードウェア エミュレーションでのみ可能です。

        - [Function: "name a"]

        - [Function: "name b"]

      - [Read]: AXI-MM ポートを使用して DDR から読み込まれる計算ユニット。計算ユニットで読み込まれるデータのトレース データがここに表示されます。アクティビティはトランザクションとして表示され、各トランザクションのツール ヒントに詳細な AXI トランザクションが表示されます。これらの名前は、`--profile_kernel data` が使用されると生成されます。

        - [m_axi_<bundle name>(port)]

- [Write]: AXI-MM ポートを使用して DDR に書き込まれる計算ユニット。計算ユニットで書き込まれるデータ トレースがここに表示されます。アクティビティはトランザクションとして表示され、各トランザクションのツール ヒントに詳細な AXI トランザクションが表示されます。これは、`--profile_kernel data` が使用されると生成されます。
  - [m_axi_<bundle name>(port)]

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とライブ波形が生成されます。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。

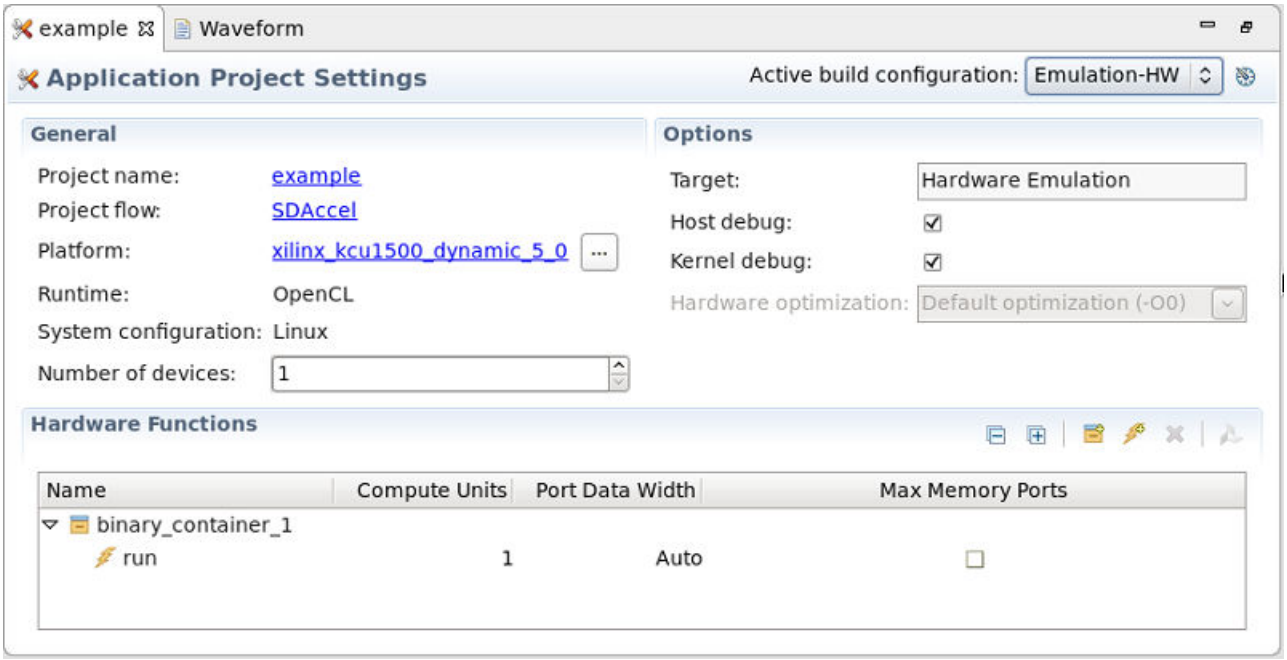
波形ビューおよびライブ波形はデフォルトでは表示されません。表示するようにすると、ハードウェア エミュレーション中にシミュレーション波形を生成する必要があり、時間もディスク容量も消費するからです。次のセクションでは、データ収集をイネーブルにするために必要な設定について説明します。

注記: 波形ビューでは、SDx™ 開発環境内からデバイス トランザクションを直接確認できます。ライブ波形の場合は、ハードウェア トランザクションを視覚化するシミュレーション波形を生成するだけでなく、ユーザーが選択した内部信号も視覚化できることがあります。

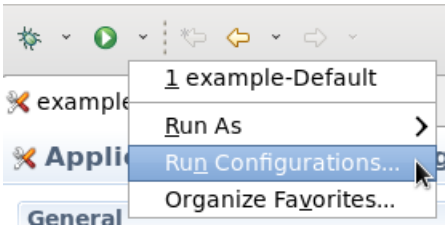
GUI フロー

波形データ コレクションをイネーブルにしてビューアーで開く手順は、次のとおりです。

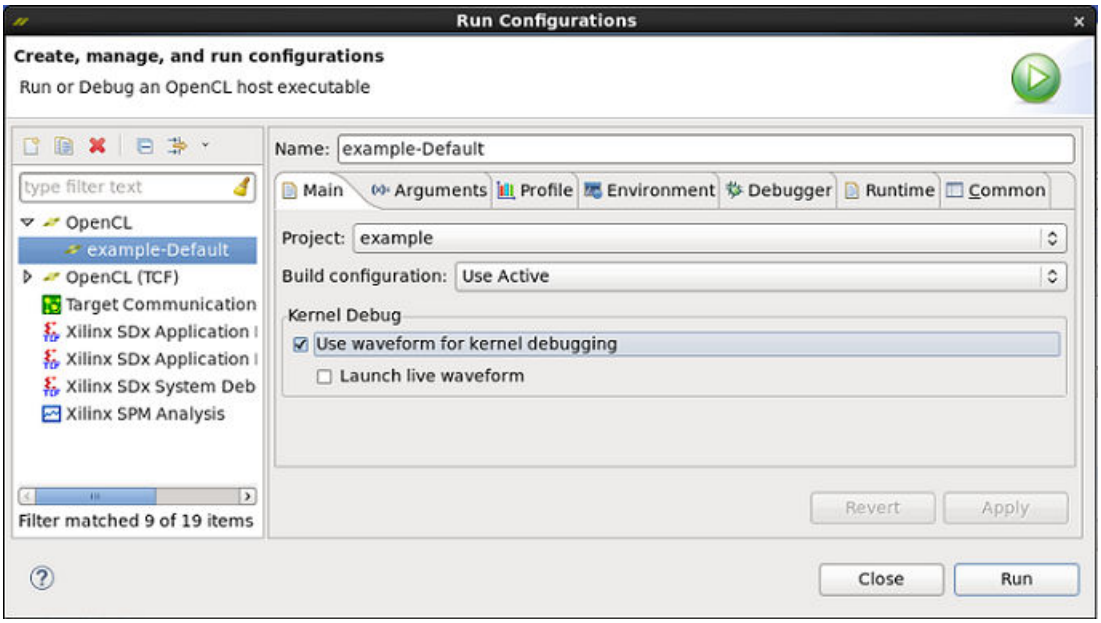
1. [Application Project Settings] ウィンドウを開いて、[Kernel debug] をオンにします。



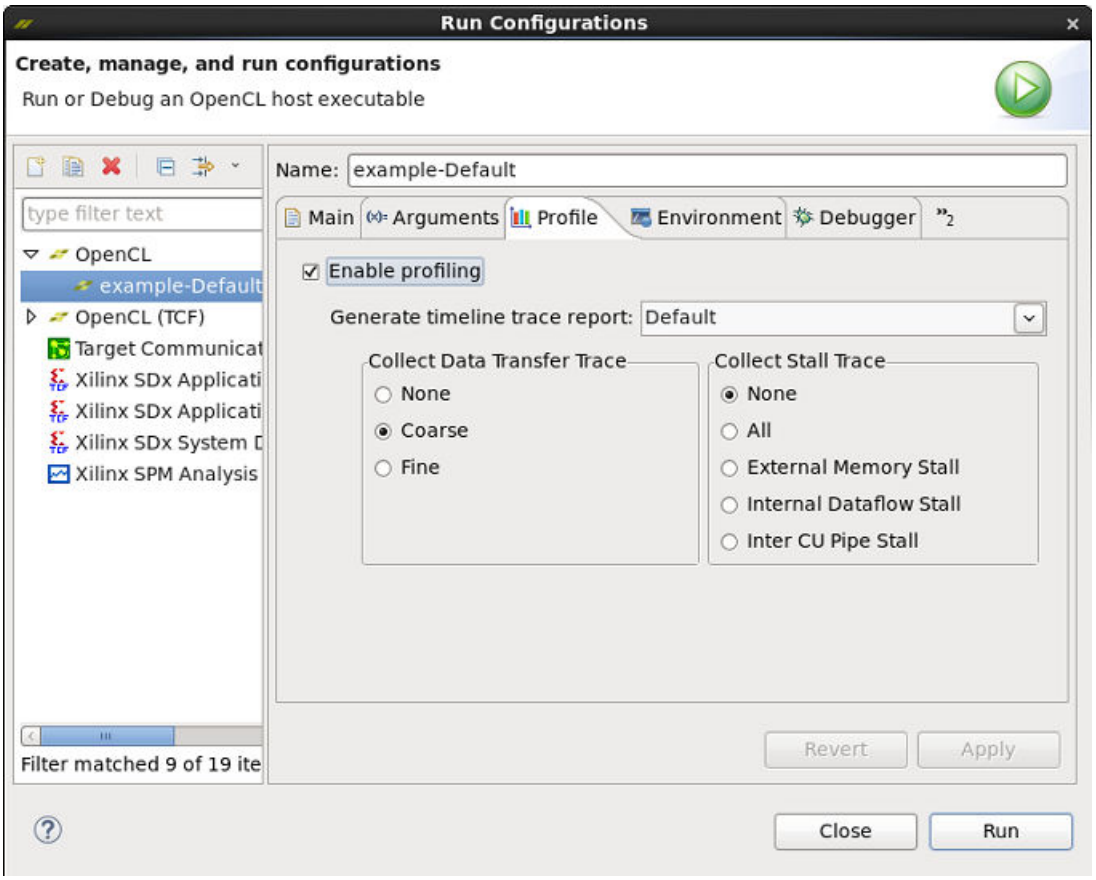
2. [Run] ボタンの横の下向き矢印をクリックし、[Run Configurations] を選択して [Run Configurations] ウィンドウを開きます。



3. [Run Configurations] ウィンドウで [Main] タブをクリックし、[Use waveform for kernel debugging] をオンにします。オプションで [Launch live waveform] をオンにして、ハードウェア エミュレーション実行中に [Simulation] ウィンドウを起動してライブ波形を表示させることもできます。



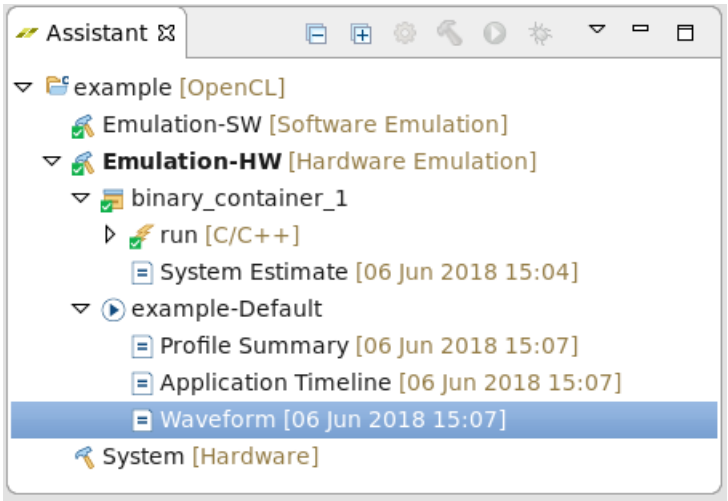
4. [Run Configurations] ウィンドウで [Profile] タブをクリックし、[Enable profiling] がオンになっていることを確認します。これにより基本的なプロファイリングがサポートされるようになります。



同じプロジェクトに対して複数の run 設定がある場合は、run 設定ごとにプロファイル設定を変更する必要があります。

5. ライブ波形が自動的に起動されるようにしていない場合は、SDx™ 開発環境から波形ビューを開きます。

   SDx 開発環境の [Assistant] ウィンドウで [Waveform] をダブルクリックして、[Waveform] ビュー ウィンドウを開きます。



コマンド ライン

ハードウェア エミュレーション中にコマンド ラインから波形データ コレクションをイネーブルにしてビューアーで開く手順は、次のとおりです。

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 -t hw_emu ...
```

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。

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

これにより、可観測性は最大になります。このオプションの詳細は次のとおりです。

- profile=<true|false>: true に設定すると、プロファイル モニタリングがイネーブルになります。オプションに何も指定しなくても、ホスト ランタイム ロギング プロファイル サマリはイネーブルになりますが、false の場合、モニタリングはまったく実行されません。
- timeline_trace=<true|false>: データを収集したタイムライン トレース情報が含まれるようになります。

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` ファイルに収集されます。
4. ライブ波形およびその他のシミュレーション波形を表示するには、次を `sdaccel.ini` ファイルのエミュレーション セクションに追加します。

```
[Emulation]
launch_waveform=gui
```

ライブ波形ビューアーは、ハードウェア エミュレーションの実行中に生成され、ここから波形を詳細に検証できます。

5. ライブ波形ビューアーを開くように設定しなかった場合は、次の手順で波形ビューを開くことができます。
   a. 次のコマンドを実行して SDx™ IDE を開始します。

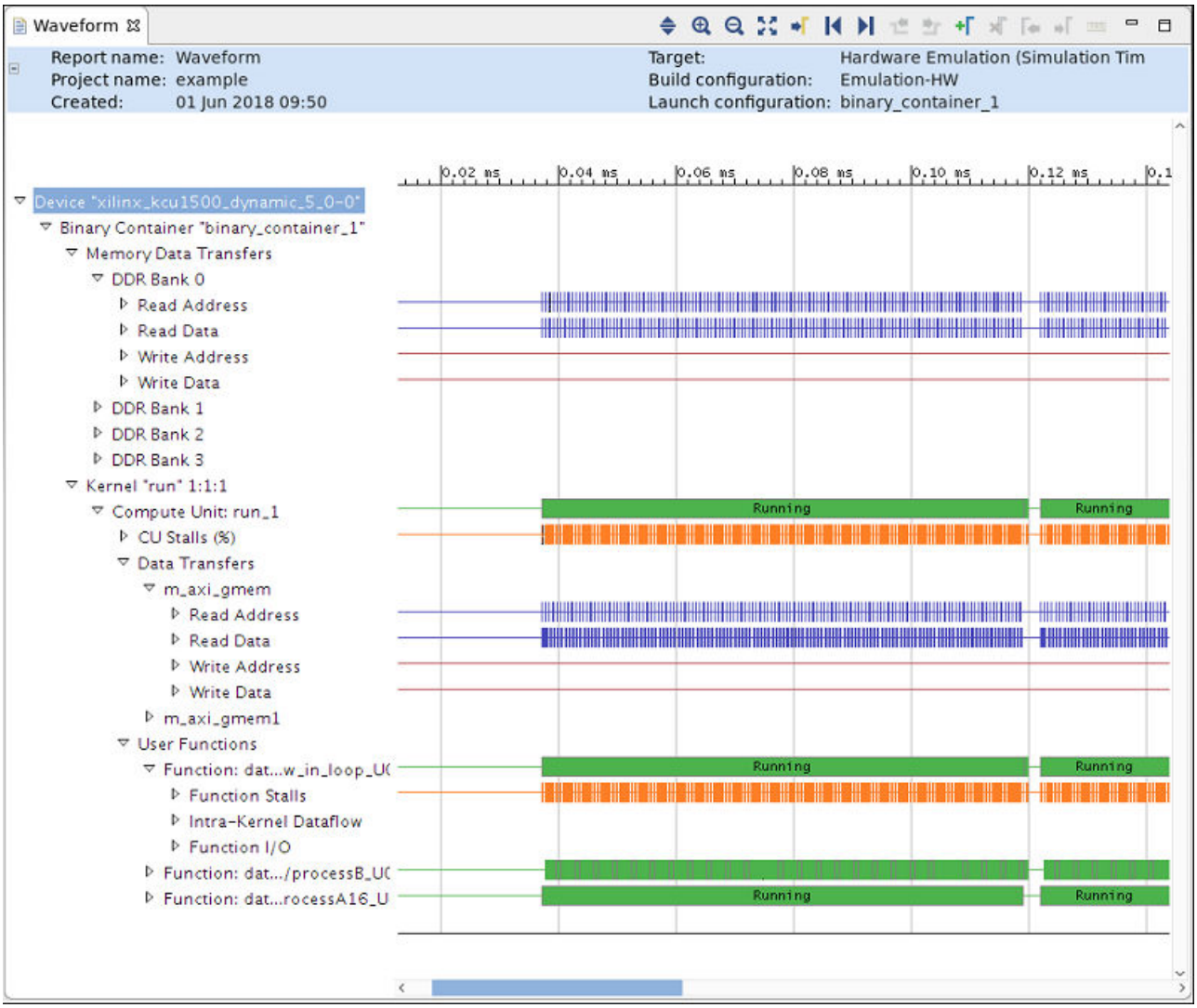
```
$sdx
```

   b. プロンプトが表示されたら、ワークスペースを選択します。
   c. [File]→[Open File] をクリックし、ハードウェア エミュレーション中に生成された `.wdb` ファイルを選択します。

[Waveform] ビューのデータ

次は、[Waveform] ビューのスナップショットです。

図 9: [Waveform] ビュー



[Waveform] ビューは、ナビゲーションしやすいように階層別に表示されます。このビューは、ハードウェア エミュレーション (カーネル トレース) 中に実際に生成された波形に基づいているので、抽象化されたデータの元である下位の個別信号まで詳細に表示できます。ただし、データは後処理されるので、その他の信号は追加できず、DATAFLOW トランザクションなど、ランタイム解析の中には表示されないものもあります。次は、階層ツリーとその説明です。

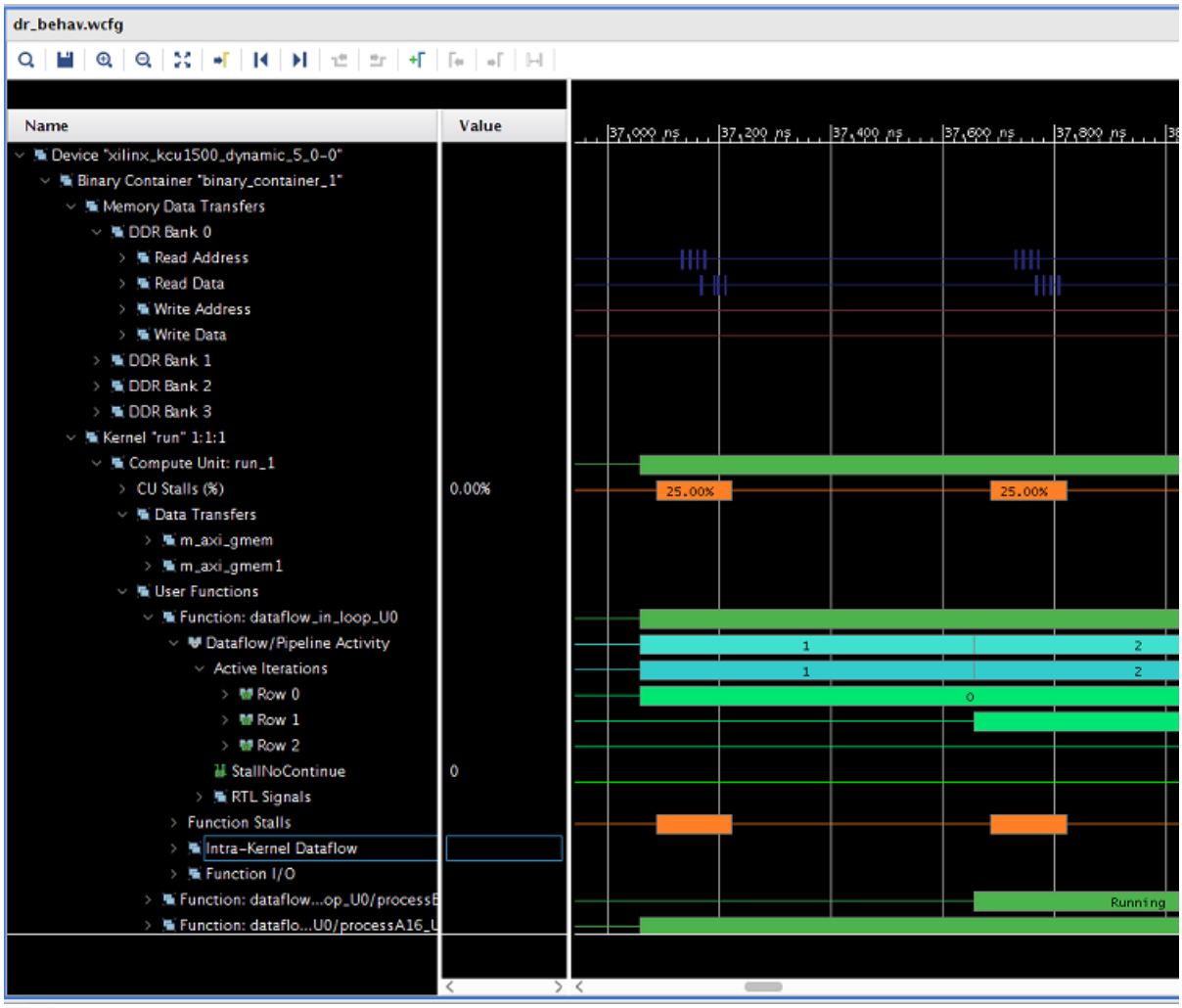
- [Device “name”]: ターゲット デバイス名。
  - [Binary Container “name”]: バイナリ コンテナー名。
    - [Memory Data Transfers]: DDR バンクごとに、ホストからバンクに到着する読み出しおよび書き込みトランザクションすべてのトレースを表示します。
      - [DDR Bank 0]
        - [Read Address]
        - [Read Data]
        - [Write Address]

        - [Write Data]
    - [DDR Bank 1]
    - [DDR Bank 2]
    - [DDR Bank 3]
- [Kernel “name” 1:1:1]: このセクションは、各カーネルとそのカーネルの計算ユニットごとに、計算ユニットからのアクティビティを分割します。
    - [Compute Unit: “name”]: 計算ユニット名。
        - [CU Stalls (%)]: ストール信号が HLS から提供され、外部メモリ アクセス、内部ストリーム (例: データフロー)、または外部ストリーム (例: OpenCL™ パイプ) のために回路の一部がストールした場合にユーザーに知らせます。詳細なカーネル トレースに示されるストール バスは、最下位レベルのストール信号をすべてコンパイルし、ストールしている割合 (%) をレポートします。これにより、どれくらいのカーネルがシミュレーションでストールしているかがわかります。

          たとえば、100 個の最下位レベルのストール信号があり、10 個が任意のクロック サイクルでアクティブであれば、[CU Stall (%)] の割合は 10% です。そのうちの 1 つがアクティブでなくなれば、割合は 9% になります。
        - [Data Transfers]: 計算ユニットの各マスター AXI ポートから DDR への読み出し/書き込みデータ転送アクセスを示します。
            - [m_axi_<bundle name>]
                - [Read Address]
                - [Read Data]
                - [Write Address]
                - [Write Data]
            - [m_axi_<bundle name>]
        - [User Functions]: HLS カーネル用に表示される情報で、ユーザー関数を示します。
            - [Function: “name”]
                - [Function Stalls]: プロセスで発生したさまざまなタイプのストールを示します。外部メモリ ストールおよび内部カーネル パイプ ストールなどが含まれます。行数は現在の実行を表示するためにダイナミックに増加していきます。
                - [Intra-Kernel Dataflow]: 内部からカーネルへの FIFO アクティビティ。
                - [Function I/O]: 実際のインターフェイス信号。
            - [Function: “name”]
            - [Function: “name”]

ライブ波形のデータ

次は、ハードウェア エミュレーションを実行中の [Live Waveform] ビューのスナップショットです。



[Lvie Waveform] ビューは、ナビゲーションしやすいように階層別に表示されます。次は、階層ツリーとその説明です。

注記: [Live Waveform] は実際のハードウェア シミュレーション run (xsim) の結果の一部としてのみ表示されるので、その他の信号および RTL 内部が同じビューにアノテーションされます。また、すべてのグループおよびまとめられたグループは、実際に含まれる信号に展開できます。

- [Device “name”]: ターゲット デバイス名。
  - [Binary Container “name”]: バイナリ コンテナー名。
    - [Memory Data Transfers]: DDR バンクごとに、ホストからバンクに到着する読み出しおよび書き込みトランザクションすべてのトレースを表示します。
      - [DDR Bank 0]
        - [Read Address]
        - [Read Data]
        - [Write Address]
        - [Write Data]

    - [DDR Bank 1]
    - [DDR Bank 2]
    - [DDR Bank 3]
- [Kernel “name” 1:1:1] このセクションでは、各カーネルとそのカーネルの計算ユニットごとに、計算ユニットからのアクティビティを分割します。
    - [Compute Unit: “name”]: 計算ユニット名。
    - [CU Stalls (%)]: ストール信号が HLS から提供され、外部メモリ アクセス、内部ストリーム (例: データフロー)、または外部ストリーム (例: OpenCL™ パイプ) のために回路の一部がストールした場合にユーザーに知らせます。詳細なカーネル トレースに示されているストール バスは、最下位レベルのストール信号をすべてコンパイルし、いずれかの時点でストールしている割合 (%) をレポートします。これにより、どれくらいのカーネルがシミュレーションでストールしているかがわかります。

      たとえば、100 個の最下位レベルのストール信号があり、10 個が任意のクロック サイクルでアクティブであれば、[CU Stall (%)] の割合は 10% です。そのうちの 1 つがアクティブでなくなれば、割合は 9% になります。
    - [Data Transfers]: 計算ユニットの各マスター AXI ポートから DDR への読み出し/書き込みデータ転送アクセスを示します。
        - [m_axi_<bundle name>]
            - [Read Address]
            - [Read Data]
            - [Write Address]
            - [Write Data]
        - [m_axi_<bundle name>]
    - [User Functions]: HLS カーネル用に表示される情報で、ユーザー関数を示します。
        - [Function: “name”]
            - [Dataflow/Pipeline Activity]: 関数がデータフロー プロセスとしてインプリメントされる場合、並列実行される関数の数を示します。
                - [Active Iterations]: 現在のアクティブなデータフローのイテレーションを示します。行数は現在の実行を表示するためにダイナミックに増加していきます。
                    - [Row 0]
                    - [Row 1]
                    - [Row 2]
                - [StallNoContinue]: これは、データフローで出力のストールがあるかどうかを伝えるストール信号です (関数は完了しますが、隣接するデータフロー プロセスからの続行信号は受信しません)。
                - [RTL Signals]: データフロー プロセスの上記のトランザクション ビューを解釈するのに使用された RTL 制御信号です。
            - [Function Stalls]: プロセスで発生したさまざまなタイプのストールを示します。
                - [External Memory]: DDR メモリにアクセス中に発生したストール。
                - [Internal-Kernel Pipe]: 計算ユニットがパイプを使用して相互通信する場合に、関連するストールを示します。

- [Intra-Kernel Dataflow]: 内部からカーネルへの FIFO アクティビティ。
- [Function I/O]: 実際のインターフェイス信号。

- [Function: “name”]
- [Function: “name”]

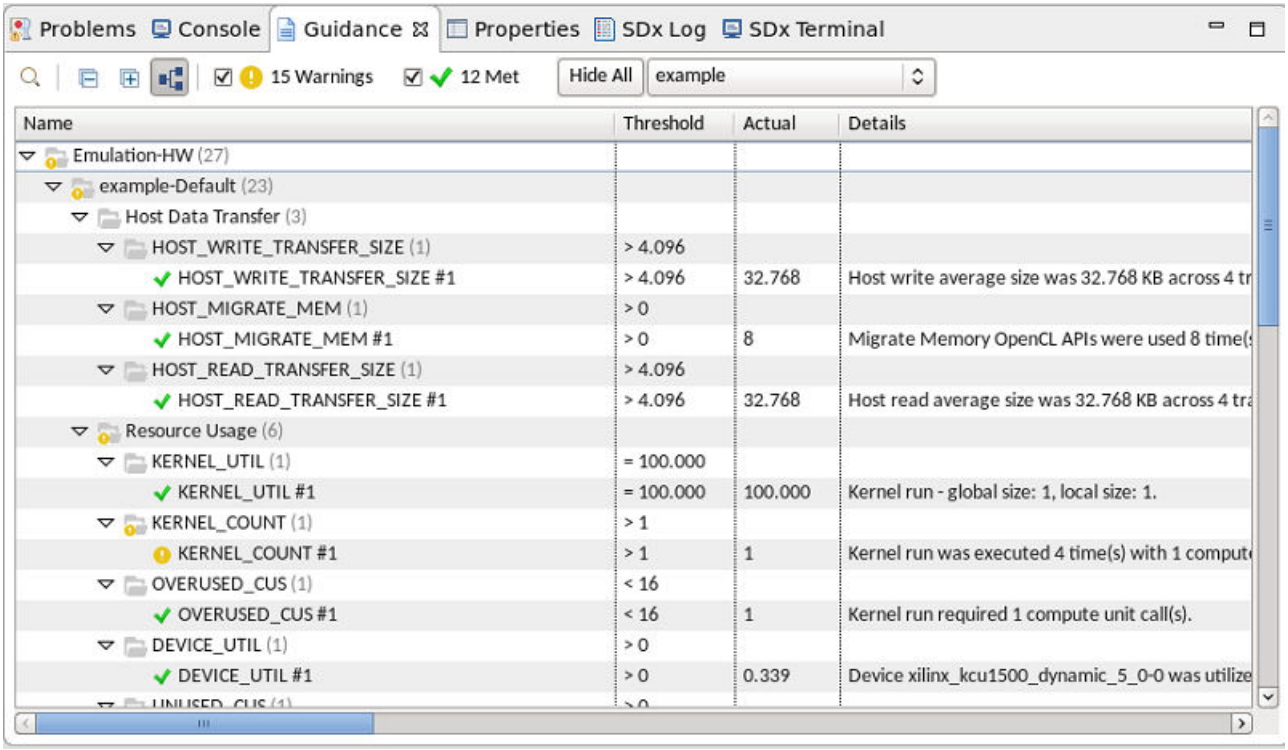
[Guidance] ビュー

[Guidance] ビューは、SDAccel™ 実行フロー全体でユーザーへフィードバックを提供するためのもので、実際のデザインを構築してランタイム解析までに発生した問題すべてが 1 つのページにまとめられます。

[Guidance] ビューは、発生する可能性のある問題を見つけやすくするためのもので、問題の原因はソース コードに関連していたり、ツールの最適化にある場合があります。また、このルールはさまざまな基準デザイン セットからの経験に基づいた汎用ルールですが、特定デザインには使用できないことがあります。このため、特定のガイダンス ルールを理解して、特定のアルゴリズムおよび要件に基づいて適切な操作を実行するようにしてください。

GUI フロー

[Guidance] ビューは自動的に生成され、中央下部に表示されます。ハードウェア エミュレーションを実行すると、次のように [Guidance] ビューが表示されます。



注記: ガイダンスは HLS コンパイル後にも生成できますが、プロファイル ルール チェックは表示されません。

GUI フローではガイダンス情報の表示を簡素にするため、[Guidance] ビューを検索/フィルターして特定のガイダンス ルール入力を検出できるようになっています。ツリー ビューの展開を開いたり閉じたり、階層ツリーを非表示にして、ガイダンス ルールをシンプルに表示させることもできます。また、[Guidance] ビューで表示されるものは選択できます。警告は、規則に従っている限り表示/非表示を切り替えることができるので、ビルドおよびエミュレーションなどのメッセージのソースに基づいて特定の内容を限定して表示することもできます。

デフォルトでは [Guidance] ビューには、ドロップダウンで選択したプロジェクトのガイダンス情報がすべて表示されます。個別のビルドまたは run 段階の内容のみを表示するには、[Window]→[Preferences] で [ザイリンクス SDx]→[Guidance] カテゴリを選択し、[Group guidance rule checks by project] をオフにします。

コマンド ライン

ガイダンス データは、フローのガイダンス情報すべてをまとめたもので、GUI から解析するのが最適ですが、どちらにしても、ガイダンス情報を含む HTML ファイルが自動的に生成されます。複数のガイダンス ファイルがツール フロー全体を通して生成されます。ガイダンス レポートを見つけるには、`guidance.html` ファイルを検索するのが最もシンプルな方法です。

```
find . -name "*guidance.html" -print
```

このコマンドを実行すると、生成されたすべてのガイダンス ファイルがリストされます。これらのファイルは、どのウェブ ブラウザーでも開くことができます。

データの説明

[Guidance] ビューには入力項目が行ごとに表示されます。各行には、ガイダンス ルール名、しきい値、実際の値などの後に、そのルールの説明が簡単に表示されます。最後のフィールドには、ルール違反を理解して回避するための参照資料へのリンクが含まれます。

GUI の [Guidance] ビューでは、[Name] 列にカテゴリ別に分けられたガイダンス ルールと ID が重要度を示すシンボルと共に表示されます。これらは、HTML レポートにそれぞれリストされます。また、HTML レポートにはヒントは表示されませんが、[Full Name] 列が含まれます。

次は、HTML ガイダンス レポートに含まれるすべてのフィールドとその目的です。

[Id]

ガイダンス ルールにはそれぞれ独自の ID が割り当てられます。この ID は、ガイダンス レポートから特定のメッセージを見つけるために使用します。

[Name]

[Name] 列には、ガイダンス ルールを識別するためのニーモニック名が表示されます。これらの名前は、特定のガイダンス ルールを記憶しやすくすることを目的に付けられます。

[Severity]

[Severity] 列からは、ガイダンス ルールの重要度が簡単にわかるようになっています。

[Full Name]

[Full Name] には [Name] 列のニーモニック名よりも暗号度の低い名前が記述されます。

[Categories]

ほとんどのメッセージがカテゴリ別に分けられています。[Guidance] ビューの共通ツリー ノードの下にメッセージが論理的なカテゴリ別にグループ分けされます。

[Threshold]

[Threshold] 列には、基準となるしきい値が表示され、この値からルールが満たされているかどうかがわかります。しきい値は、良いデザインおよびコーディング プラクティスに従った多くのアプリケーションを元に決めらます。

[Actual]

[Actual] 列には、特定のデザインでの実際の値が表示されます。この値を基準値と比較すると、ルールが満たされたかどうかを判断できます。

[Details]

[Details] 列には、現在のルール仕様を簡単に説明した特定のメッセージが含まれます。

[Resolution]

[Resolution] 列には、現在のルールを満たすために変更可能なモデル ソース コードまたはツール変更によく使用される方法へのリンクが含まれます。リンクをクリックすると、ポップアップ メニューまたは特定の問題に使用できるヒントやコードを含む資料が開きます。

インプリメンテーション ツールの使用

Vivado HLS を使用したカーネル最適化

OpenCL™ または C/C++ を使用すると、カーネル最適化のすべてが SDAccel™ 環境から実行できるようになります。この章で説明したような主な最適化制約 (関数およびループのパイプライン処理、関数およびループ間で同時処理を増やすことのできるデータフローの適用、ループの展開など) は、ザイリンクス FPGA デザイン ツールの Vivado® HLS で実行されます。

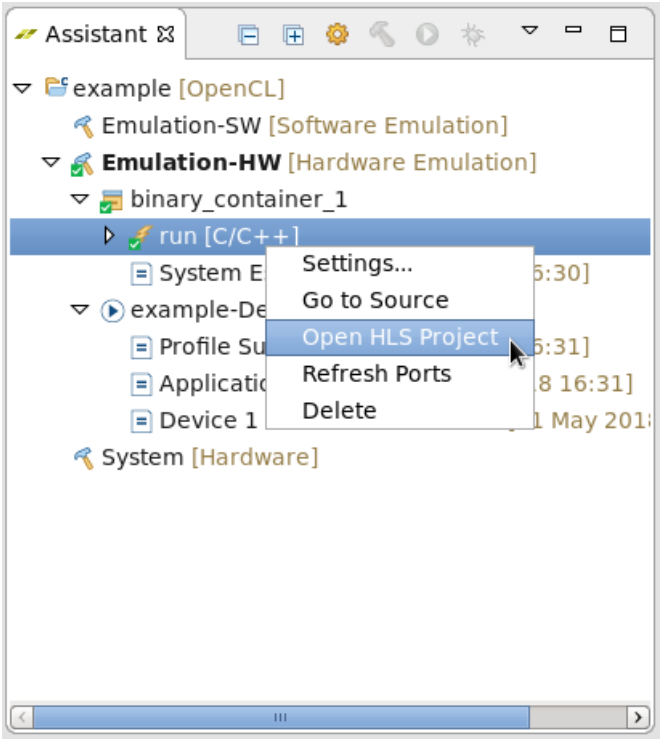
Vivado HLS は SDAccel 環境から自動的に起動されますが、SDAccel 環境内から直接 Vivado HLS を起動するオプションもあります。スタンドアロン モードで Vivado HLS を使用すると、最適化手法が次のように改善できます。

- カーネル最適化のみに集中できます。エミュレーションを実行する必要はありません。
- 複数のソリューションを作成し、結果を比較し、ソリューション スペースを確認して最適なデザインを見つけることができます。
- インタラクティブな [Analysis] パースペクティブを使用してデザイン パフォーマンスを解析できます。

重要: カーネル ソース コードのみを SDAccel 環境に戻すことができます。最適化スペースを確認したら、すべての最適化がカーネル ソース コードに OpenCL 属性または C/C++ プラグマとして適用されるようにします。

Vivado HLS をスタンドアロン モードで開くには、[Assistant] ビューでハードウェア関数オブジェクトを右クリックし、[Open HLS Project] をクリックします (次の図を参照)。

図 10: HLS プロジェクト を開く



Vivado バックエンド最適化

SDx™ には OpenCL™/C/C++ モデルから FPGA アクセラレーション インプリメンテーションまでのスムーズなフローが提供されています。このフローでは、ほとんどの場合 FPGA のプログラマブル領域がカーネル機能をインプリメントするようにコンフィギュレーションされないので、配線遅延およびカーネル配置などの典型的なハードウェア制約については考慮しなくてすみます。ただし、特に大型デザインをインプリメントする場合などは注意が必要なこともあります。フロー後半では、SDx で Vivado® Design Suite バックエンド ツールを完全に制御できるようになります。

SDAccel™ 環境は Vivado を呼び出して、RTL カーネルの合成とインプリメンテーションを自動的に実行します。SDAccel 環境内で Vivado を直接起動することもできます。SDAccel 環境で Vivado をスタンドアロン モードで起動すると、Vivado 合成プロジェクトまたは Vivado インプリメンテーション プロジェクトを開くことができます。

Vivado プロジェクトは、ビルド コンフィギュレーションを [System] に設定してビルドをしたら、SDAccel 環境で開くことができます。Vivado IDE をスタンドアロン モードで開くには、[Xilinx] ドロップダウン リストから [Vivado Integration] → [Open Vivado Project] をクリックします。合成かインプリメンテーションを指定して Vivado プロジェクトを選択し、[OK] をクリックします。

Vivado をスタンドアロン モードで使用すると、さまざまな Vivado 合成およびインプリメンテーション オプションを試して、パフォーマンスおよびエリアを改善するようにカーネルをさらに最適化できます。これらのパラメーターを最大限に使用するには、Vivado ツールに精通していることが推奨されます。

重要: スタンドアロンの Vivado プロジェクトで適用した最適化オプションは、SDAccel 環境には自動的に適用されません。最適化オプションを適用したら、`xocc` に `--xp` オプションを使用して、すべての最適化パラメーターが SDAccel 環境に渡されるようにする必要があります。次に例を示します。

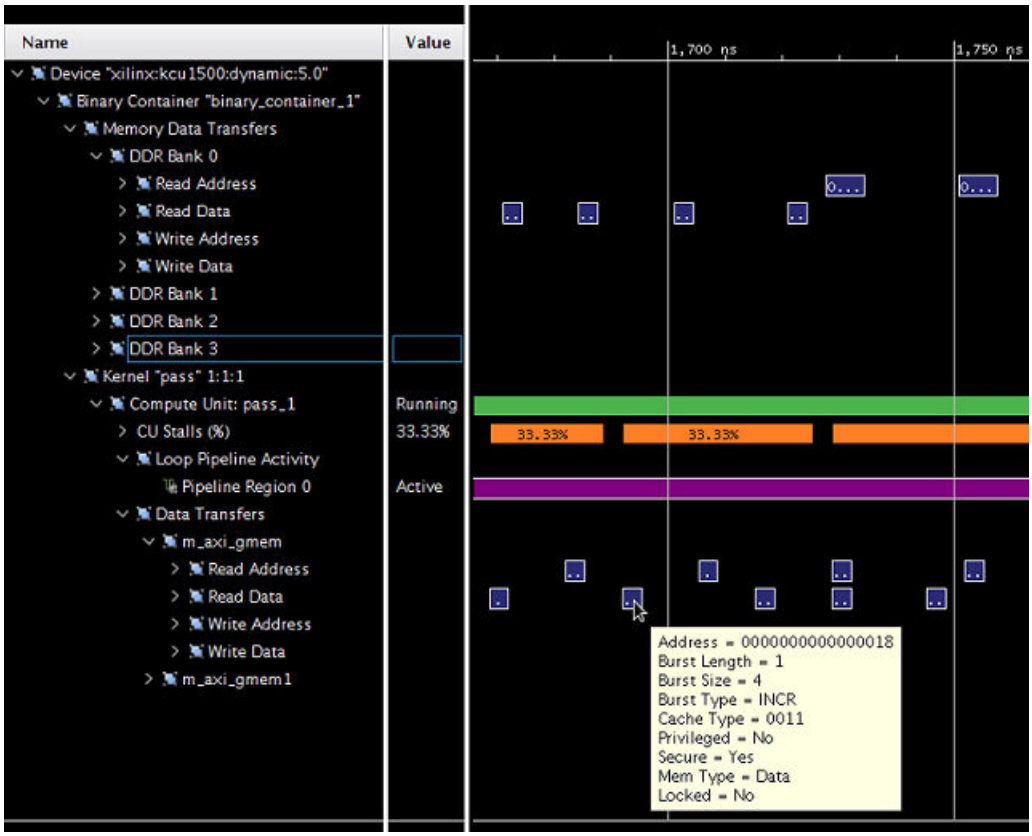
```
 --xp "vivado_prop:run.impl_1.{STEPS.PLACE_DESIGN.ARGS.TCL.POST}={<File
and path>}"
```

インターフェイス最適化

このセクションでは、インターフェイス最適化について説明します。インプリメントされたインターフェイスの属性について説明し、パフォーマンス全体を改善するための推奨事項を示します。

インターフェイス属性 (詳細なカーネル トレース)

詳細なカーネル トレースには、AXI トランザクションおよびそのプロパティが表示されます。AXI トランザクションは DDR 側 ([Memory Data Transfers]) と AXI インターコネクトのカーネル側 ([Kernel "pass" 1:1:1]) に対して表示されます。次の図は、新しくアクセラレーションされたアルゴリズムの典型的なカーネル トレースを示しています。



パフォーマンスに関して注意するフィールドは、次のとおりです。

- [Burst Length]: 1 つのトランザクションで送信されるパッケージ数を示します。
- [Burst Size]: 1 つのパッケージの一部として転送されるバイト数を示します。

たとえば、[Burst Length] が 1 でパッケージごとに 4 バイトだけだとすると、妥当な量のデータを転送するのに個別の AXI トランザクションが多く必要になります。SDx™ では、4 バイト未満のバースト サイズはそれより小さいデータが送信されても作成されません。この場合、後に続くものが AXI バーストをイネーブルせずにアクセスされると、同じアドレスに対する複数の AXI 読み出しが生成されます。

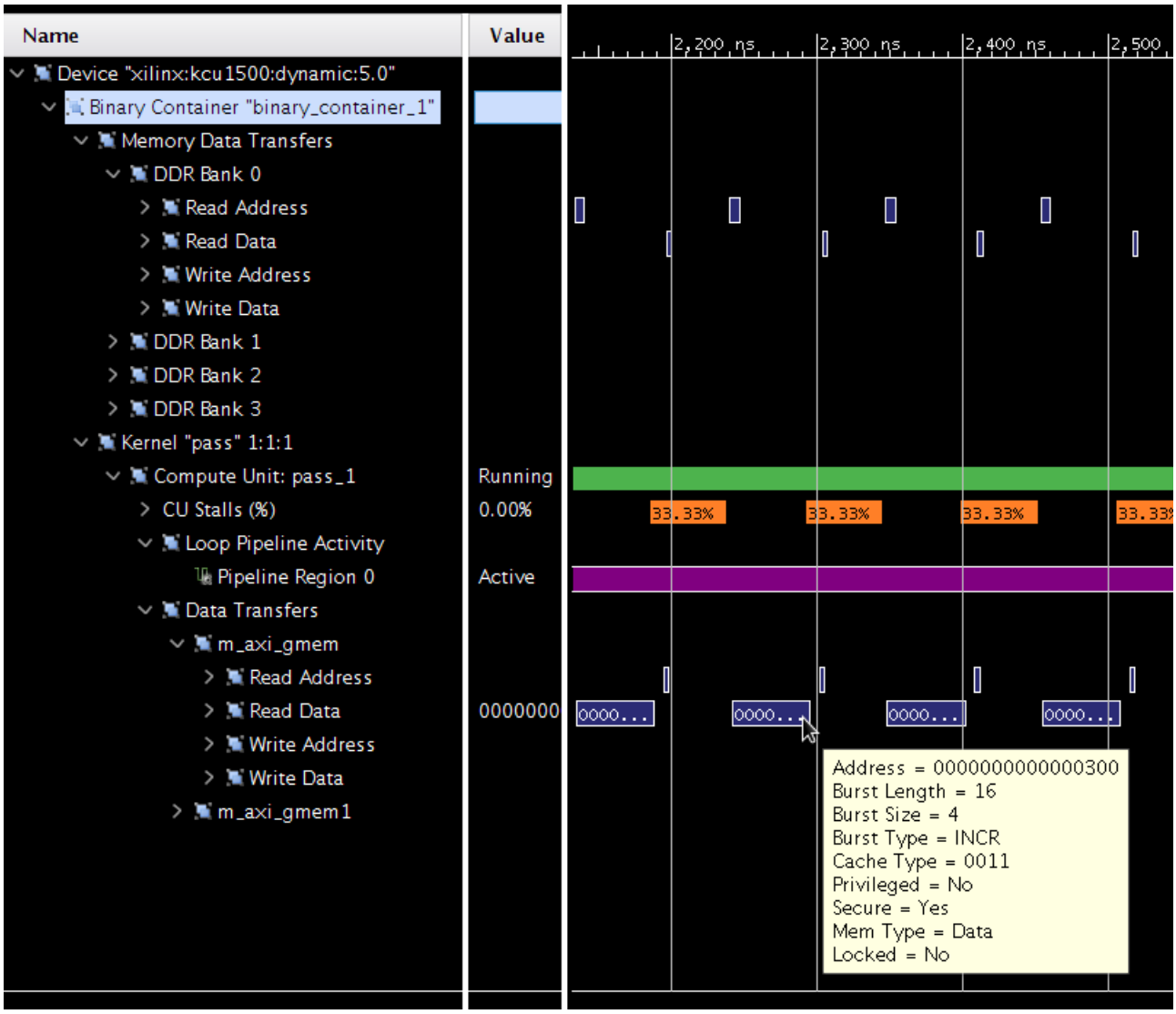
このため、インターフェイス パフォーマンスを最適化するには、[Burst Length] を短くし、[Burst Size] を 512 ビットよりもかなり低く抑えることをお勧めします。次のセクションは、インプリメンテーションの改善方法を示しています。

- バースト データ転送の使用
- メモリ帯域幅の全ユーザー データ幅の使用

バースト データ転送の使用

データをバースト転送すると、メモリ アクセスのレイテンシは表示されず、帯域幅の使用およびメモリ コントローラーの効率が改善されます。バースト転送は、連続したアドレス位置からのデータの連続リクエストから推論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) の「グローバル メモリのバースト転送の推論」を参照してください。

バースト転送が発生すると、詳細なカーネル トレースに表示されるバースト率がより高くなり、バースト長の数も増加します。



この図では、メモリ データ転送の後、AXI インターコネクトが別の方法 (トランザクション時間は短縮) で実際にインプリメントされています。これらのトランザクション上にカーソルを置くと、AXI インターコネクトが 16x4 バイト トランザクションを 1 つの 1x64 バイトのパッケージ トランザクションにパックしたことがわかります。この方が、全帯域幅がより効率的に使用されます。次のセクションでは、この最適化手法について詳細に説明します。

バースト インターフェイスはコーディング スタイルとアクセス パターンによってかなり異なります。問題のあるコード記述の詳細は、コーディング スタイル ガイドに記述されていますが、バースト検出をしやすくしてパフォーマンスを改善するための一般的なルールがあります。次のコード例に示すように、データ転送と計算を分離します。

```
void kernel(T in[1024], T out[1024]) {
    T tmpIn[1024];
    T tmpOu[1024]t;
    read(in, tmpIn);
    process(tmpIn, tmpOut);
    write(tmpOut, out);
}
```

つまり、`read` 関数で AXI 入力から内部変数 (`tmpIn`) に読み込んで、内部変数 `tmpIn` および `tmpOut` で動作する `process` 関数で実際の計算がインプリメントされ、`write` 関数でその生成された出力を取り込んで AXI 出力に書き出します。

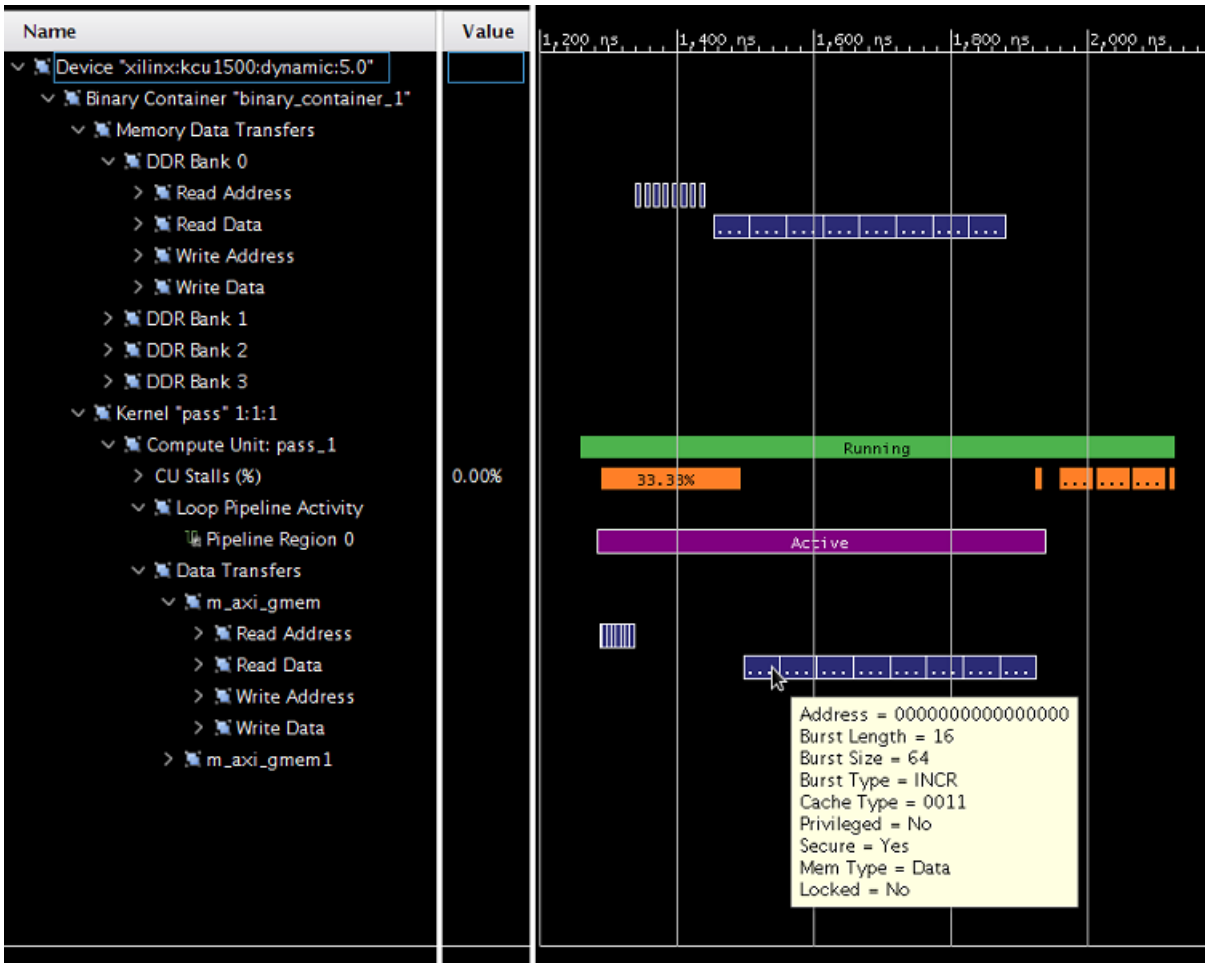
計算結果からの read および write 関数を分離すると、次のようになります。

- read/write 関数の制御構造 (ループ) をシンプルにすると、バースト検出もシンプルになります。
- AXI インターフェイスから計算関数を分離すると、潜在的なカーネル最適化が簡素化されます。詳細は、カーネル最適化を参照してください。
- 内部変数はオンチップ メモリにマップされるので、AXI トランザクションよりも速くアクセスできます。SDAccel™ 環境でサポートされるアクセラレーション プラットフォームには最大で 10MB のオンチップ メモリを含めることができ、これらはパイプ、ローカル、およびプライベート メモリとして使用できます。これらのリソースを効率的に使用することで、アプリケーションの効率およびパフォーマンスをかなり向上できます。

全 AXI データ幅の使用

SDAccel™ コンパイラでは、カーネル引数のデータ型に基づいてカーネルおよびメモリ コントローラー間のユーザー データ幅が設定できます。ザイリンクスではデータ スループットを最大にするために、ユーザーがメモリ コントローラーの全データ幅にマップするデータ型を選択することをお勧めしています。サポートされるアクセラレーション カードすべてのメモリ コントローラーで 512 ビットのユーザー インターフェイスがサポートされており、これらは `int16` または C/C++ 任意精度データ型 `ap_int<512>` などの OpenCL™ ベクター データ型にマップできます。

次の図では、バースト AXI トランザクション (バースト長 16) およびパッケージ サイズ 512 ビット (バースト サイズ 64 バイト) になっています。



この例は、AXI データ幅を最大にした良いインターフェイス コンフィギュレーションと、実際のバースト トランザクションを示しています。

インターフェイスを宣言するのに使用される複雑な構造体またはクラスがあると、メモリ レイアウトやデータ パッケージの違いにより、ハードウェア インターフェイスがかなり複雑になってしまうことがあります。これにより、複雑なシステムをデバッグするのがかなり困難になってしまう可能性があります。ザイリンクスでは、カーネル引数に常に 32 ビット境界へパック可能なシンプルな構造体を使用することをお勧めしています。構造体の使用に推奨される方法については、ザイリンクス オンボーディング例 (GitHub) の kernel_to_gmem カテゴリの「Custom Data Type Example」を参照してください。

OpenCL 属性

OpenCL には、AXI データ幅がより自動的に増加されるのをサポートする属性があります。上記に示したインターフェイス データ幅の変更は OpenCL でもサポートされますが、アルゴリズムを C/C++ と同じようにコード変更して、より大きな入力ベクターに対応するようにする必要があります。

手動でコードを変更しないようにするには、データ幅を広げてアルゴリズムをベクター化する次のような OpenCL 属性を使用します。詳細は、『SDx プラグマ リファレンス ガイド』 (UG1253) を参照してください。

- `vec_type_hint`

- `reqd_work_group_size`
- `xcl_zero_global_work_offset`

次はこれらの使用例です。

```
__attribute__((reqd_work_group_size(64, 1, 1)))
__attribute__((vec_type_hint(int)))
__attribute__((xcl_zero_global_work_offset))
__kernel void vector_add(__global int* c, __global const int* a, __global
const int* b) {
    size_t idx = get_global_id(0);
    c[idx] = a[idx] + b[idx];
}
```

この場合、ハード コード化されたインターフェイスは 32 ビット データ幅のデータパス `(int *c, int* a, int *b)` なので、直接インプリメントするとメモリ スループットがかなり制限されてしまいますが、3 つの属性の値に基づいて自動的に幅を広げて変換する機能が適用されています。

- `__attribute__((vec_type_hint(int)))` では、`int` が計算およびメモリ転送 (32 ビット) に主に使用されるデータ型であることが宣言されています。これにより、AXI インターフェイスのターゲット帯域幅 (512 ビット) に基づいて、ベクター化/幅拡張される係数を計算できます。この例の場合、係数は 16 = 512 ビット / 32 ビットになり、理論上はベクター化が適用される際に 16 個の値が処理されることになります。
- `__attribute__((reqd_work_group_size(X, Y, Z)))` の `X`、`Y`、`Z` は正の定数で、作業項目の合計を定義します。`X*Y*Z` は作業項目の最大数なので、最大可能ベクター化係数を定義すると、メモリ帯域幅が飽和する可能性があります。この例の場合、作業項目の合計は `64*1*1=64` です。

適用される実際のベクター化係数は、実際にコード記述されたデータ型または vec_type_hint で定義されるベクター化係数の最大公約数、および `reqd_work_group_size` で定義された最大可能ベクター係数になります。

実際のベクター化係数で除算される最大可能ベクター化係数の商からは、OpenCL 記述の残りのループ カウントが算出されます。このループはパイプライン処理されるので、複数のループ反復が残っている場合に、パイプライン処理されたインプリメンテーションの利点を生かすことができます。これは、特にベクター化された OpenCL コードに長いレイテンシがある場合に役立ちます。

次のオプションのパラメーターを OpenCL インターフェイスでパフォーマンス最適化に指定することをお勧めします。

- `__attribute__((xcl_zero_global_work_offset))` では、ランタイムで使用されるグローバル オフセット パラメーターがなく、すべてのアクセスをアライメントするようにコンパイラに命令しています。これにより、作業グループのアライメントに関する貴重な情報がコンパイラに伝わり、通常メモリ アクセスのアライメントに伝搬されます (ハードウェアがより少ない)。

これらの変換により、合成される実際のデザインが変わることに注意してください。部分的に展開されるループの場合、データが格納されるローカル配列の形状を変更する必要があります。これは通常問題なく動作しますが、まれに悪影響のあることがあります。

次に例を示します。

- 配列がパーティションされる場合、展開/ベクター化係数でパーティション係数が除算できません。
  - このため、マルチプレクサーが多く必要となり、スケジューラで問題となるので (メモリ使用量およびコンパイル時間がかなり増加する可能性あり)、ザイリンクスでは、2 のべき乗のパーティション係数 (ベクター係数が常に 2 のべき乗であるため) の使用をお勧めしています 。
- ベクター化されるループに関係のないリソース制約がある場合、スケジューラは II が満たされないことを示すメッセージを表示します。

- II は展開されたループで計算される (反復ごとに乗算されたスループットが使用される) ので、パフォーマンスを落としてまで直す必要はありません (通常はそれでもパフォーマンスは向上します)。
- スケジューラからは可能性のあるリソース制約が示され、こういった問題を回避することでパフォーマンスはさらに改善できます。
- ローカル配列は、通常ベクター化をしない方法でコードの後のセクションでアクセスされるので、自動的に再形成されないことに注意してください。

OpenCL パイプを使用したカーネル間通信のレイテンシの削減

OpenCL™ 2.0 仕様には、パイプと呼ばれる新しいメモリ オブジェクトが導入されています。パイプには、FIFO として構成されたデータが格納されます。パイプ オブジェクトには、パイプから読み出してパイプに書き込むビルドイン関数を使用してのみアクセスできます。パイプ オブジェクトはホストからはアクセスできません。パイプを使用すると、データを外部メモリなしで FPGA 内の 1 つのカーネルから別のカーネルにストリーミングでき、全体的なシステム レイテンシを大幅に向上できます。

SDAccel 開発環境では、パイプはすべてのカーネル関数の外部でスタティックに定義する必要があります。`OpenCL 2.x clCreatePipe` API を使用したダイナミック パイプ割り当ては現在のところサポートされていません。パイプの深さは、パイプ宣言内で `xcl_reqd_pipe_depth` 属性を使用して指定する必要があります。有効な値は 16、32、64、128、256、512、1024、2048、4096、8192、16384、32768 です。

```
pipe int p0 __attribute__((xcl_reqd_pipe_depth(32)));
```

1 つのパイプは、異なるカーネル内に 1 つのプロデューサーおよびコンシューマーのみを持つことができます。

パイプには、ノンブロッキング モードの標準 OpenCL `read_pipe()` および `write_pipe()` ビルトイン関数またはブロッキング モードのザイリンクスの拡張 `read_pipe_block()` および `write_pipe_block()` 関数を使用してアクセス可能です。パイプのステータスは、OpenCL `get_pipe_num_packets()` および `get_pipe_max_packets()` ビルトイン関数を使用してクエリできます。これらのビルトイン関数の詳細は、Khronos Group の『The OpenCL C Specification, Version 2.0』を参照してください。

次は、現在サポートされているパイプ関数の関数シグネチャで、`gentype` はビルトイン OpenCL C スカラー整数または浮動小数点データ型を示しています。

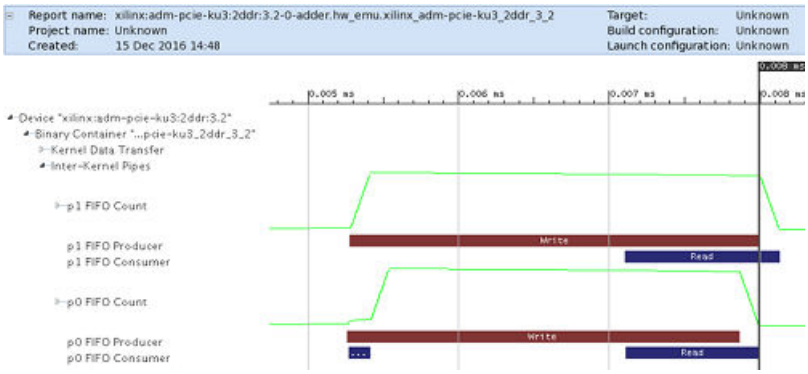
```
int read_pipe_block (pipe gentype p, gentype *ptr)
int write_pipe_block (pipe gentype p, const gentype *ptr)
```

次はザイリンクス オンオンボーディング例 (GitHub) の「Blocking Pipes Example」からの例で、`blocking read_pipe_block()` および `write_pipe_block()` 関数を使用してデータを 1 つの処理段階から次の処理段階に渡しています。

```
pipe int p0 __attribute__((xcl_reqd_pipe_depth(32)));
pipe int p1 __attribute__((xcl_reqd_pipe_depth(32)));
// Input Stage Kernel : Read Data from Global Memory and write into Pipe P0
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void input_stage(__global int *input, int size)
{
    __attribute__((xcl_pipeline_loop))
    mem_rd: for (int i = 0 ; i < size ; i++)
    {
        //blocking Write command to pipe P0
        write_pipe_block(p0, &input[i]);
    }
}
```

```
// Adder Stage Kernel: Read Input data from Pipe P0 and write the result
// into Pipe P1
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void adder_stage(int inc, int size)
{
    __attribute__((xcl_pipeline_loop))
    execute: for(int i = 0 ; i < size ;  i++)
    {
        int input_data, output_data;
        //blocking read command to Pipe P0
        read_pipe_block(p0, &input_data);
        output_data = input_data + inc;
        //blocking write command to Pipe P1
        write_pipe_block(p1, &output_data);
    }
}
// Output Stage Kernel: Read result from Pipe P1 and write the result to
Global
// Memory
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void output_stage(__global int *output, int size)
{
    __attribute__((xcl_pipeline_loop))
    mem_wr: for (int i = 0 ; i < size ; i++)
    {
        //blocking read command to Pipe P1
        read_pipe_block(p1, &output[i]);
    }
}
```

デバイス トレースライン表示には、ハードウェア エミュレーション実行後の OpenCL パイプの詳細なアクティビティおよびストールが表示されます。この情報を使用して正しい FIFO サイズを選択し、最適なアプリケーションのエリアおよびパフォーマンスを達成してください。



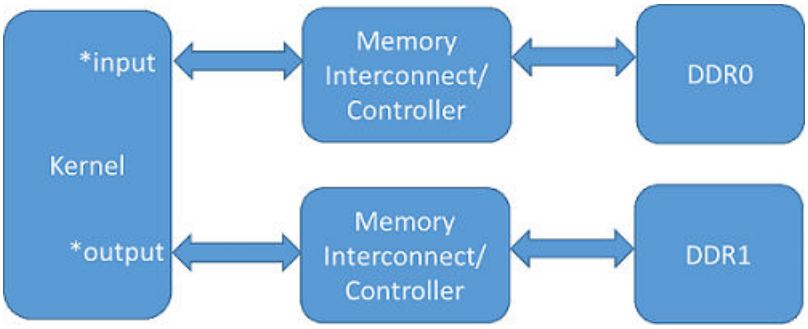
複数 DDR バンクの使用

SDAccel™ 環境でサポートされるアクセラレーション カードには最大 80GB/s の生 DDR 帯域幅の 1、2、または 4 つの DDR バンクが含まれます。FPGA および DDR 間で大容量のデータを移動するカーネルの場合、ザイリンクスでは SDAccel コンパイラおよびランタイム ライブラリで複数の DDR バンクを使用するように指示することを推奨しています。

複数の DDR バンクを使用すると、ホスト コードで CL メモリ バッファーを異なるバンクに割り当て、XCL バイナリ ファイルを `xocc` コマンド ラインでのバンク割り当てと同じになるように設定できます。

ザイリンクス オンボーディング例 (GitHub) の kernel_to_gmem カテゴリの「Global Memory Two Banks Example」のブロック図では、入力ポインターが DDR バンク 0 に、出力ポインターが DDR バンク 1 に接続されています。

図 11: グローバル メモリの 2 つのバンクの例



ホスト コードでの DDR バンクの割り当て

ホスト コードでのバンク割り当ては、ザイリンクス ベンダー拡張でサポートされています。次のコード例は、必要なヘッダー ファイルのほか、入力バッファーを DDR バンク 0 に、出力バッファーをバンク 1 にそれぞれ割り当てたところを示しています。

```
#include <CL/cl_ext.h>
…
int main(int argc, char** argv)
{
…
    cl_mem_ext_ptr_t inExt, outExt;  // Declaring two extensions for both
buffers
    inExt.flags  = XCL_MEM_DDR_BANK0; // Specify Bank0 Memory for input
memory
    outExt.flags = XCL_MEM_DDR_BANK1; // Specify Bank1 Memory for output
Memory
    inExt.obj = 0   ; outExt.obj = 0; // Setting Obj and Param to Zero
    inExt.param = 0 ; outExt.param = 0;

    int err;
    //Allocate Buffer in Bank0 of Global Memory for Input Image using
Xilinx Extension
    cl_mem buffer_inImage = clCreateBuffer(world.context, CL_MEM_READ_ONLY
| CL_MEM_EXT_PTR_XILINX,
            image_size_bytes, &inExt, &err);
    if (err != CL_SUCCESS){
        std::cout << "Error: Failed to allocate device Memory" <<
std::endl;
        return EXIT_FAILURE;
    }
    //Allocate Buffer in Bank1 of Global Memory for Input Image using
Xilinx Extension
    cl_mem buffer_outImage = clCreateBuffer(world.context,
CL_MEM_WRITE_ONLY | CL_MEM_EXT_PTR_XILINX,
            image_size_bytes, &outExt, NULL);
    if (err != CL_SUCCESS){
        std::cout << "Error: Failed to allocate device Memory" <<
```

```
std::endl;
        return EXIT_FAILURE;
    }
…
}
```

cl_mem_ext_ptr_t は struct で、次のように定義されます。

```
typedef struct{
    unsigned flags;
    void *obj;
    void *param;
  } cl_mem_ext_ptr_t;
```

- flags に使用できる値は、XCL_MEM_DDR_BANK0、XCL_MEM_DDR_BANK1、XCL_MEM_DDR_BANK2、XCL_MEM_DDR_BANK3 です。
- obj は CL_MEM_USE_HOST_PTR フラグが clCreateBuffer API に渡される際にのみ CL メモリ バッファーに割り当てられるホスト メモリへのポインターです。それ以外の場合は NULL に設定します。
- param: 今後の使用のために予約されています。常に 0 または NULL に設定します。

ホスト コードの DDR バンクの割り当て

複数の AXI インターフェイスの作成

カーネル コードには、複数の DDR バンクに接続する前に複数の AXI4 インターフェイスが必要です。OpenCL カーネル、C/C++ カーネル、および RTL カーネルでは、AXI インターフェイスへのポートの割り当て方法は異なります。

- OpenCL™ カーネルの場合、カーネル引数の各グローバル ポインターごとに AXI4 インターフェイスを 1 つ生成するのに --max_memory_ports オプションが必要です。AXI4 インターフェイス名は、引数リストのグローバル ポインターの順番に基づいて付けられます。

  次のコード例は、GitHub の SDAccel 入門サンプルの kernel_to_gmem カテゴリの gmem_2banks_ocl 例からのものです。

  ```
  __kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
  void apply_watermark(__global const TYPE * __restrict input,
  __global TYPE * __restrict output, int width, int height) {
   ...
  }
  ```

  この例では、1 つ目のグローバル ポインター input が AXI4 名 M_AXI_GMEM0 を、2 つ目のグローバル ポインター output が M_AXI_GMEM1 を割り当てています。

- C/C++ カーネルの場合、異なるグローバル ポインターの HLS INTERFACE プラグマに異なる bundle 名を指定することで、複数の AXI4 インターフェイスが生成されます。詳細は、『SDAccel 環境プログラマ ガイド』 (UG1277) を参照してください。

次の gmem_2banks_c からのコード例では、input ポインターをバンドル gmem0 に、output ポインターがバンドル gmem1 に割り当てられています。バンドル名はどの有効な C 文字列にでもでき、生成される AXI4 インターフェイス名は M_AXI_<bundle_name> になります。この例の場合、入力ポインターの AXI4 インターフェイス名は M_AXI_gmem0 で、出力ポインター名は M_AXI_gmem1 になります。

```
#pragma HLS INTERFACE m_axi port=input  offset=slave bundle=gmem0
#pragma HLS INTERFACE m_axi port=output offset=slave bundle=gmem1
```

- RTL カーネルに対しては、RTL Kernel ウィザードでのインポート プロセスでポート名が生成されます。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。

DDR バンクへの AXI インターフェイスの割り当て

重要: DDR インターフェイスを複数使用する場合は、--sp オプションを使用してカーネル/CU ごとに DDR メモリ バンクを指定し、どこに SLR カーネルを配置するかを必要があります。--sp 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) の XOCC コマンドを、SLR 配置の詳細は 『SDAccel 環境ユーザー ガイド』 (UG1023) を参照してください。

AXI4 インターフェイスは --sp オプションを使用して DDR バンクに接続します。--sp オプション値のフォーマットは <kernel_instance_name>.<interface_name>:<DDR_bank_name> です。--sp オプションの有効な DDR バンク名は、バンクが 4 つの場合、DDR[0]、DDR[1]、DDR[2]、DDR[3] になります。

次は、入力ポインター (M_AXI_GMEM0) を DDR bank0 に、出力ポインター (M_AXI_GMEM1) を DDR bank1 に接続するコマンド ライン例です。

```
xocc --max_memory_ports apply_watermark
--sp apply_watermark_1.m_axi_gmem0:bank0
--sp apply_watermark_1.m_axi_gmem1:bank1
```

[Device Hardware Transaction] ビューを使用すると、実際の DDR バンク通信を確認して DDR の使用量を解析できます。

図 12: [Device Hardware Transaction] ビューの DDR バンクのトランザクション

XILINX

第 4 章

カーネル最適化

FPGA を使用する利点の 1 つは、特定のアルゴリズム用にカスタマイズしたデザインを作成できる柔軟性と機能です。これにより、アルゴリズムのスループットと消費電力をトレードオフするさまざまなインプリメンテーション選択肢が使用できるようになります。カスタム ロジックを作成する場合の問題点は、従来の FPGA デザイン フローをすべて使用する必要のあるところです。

次のガイドラインを使用すると、デザインの複雑性を管理して、必要なデザイン目標を達成しやすくなります。

計算並列処理の最適化

デフォルトでは、C/C++ ではアルゴリズムは常に順番に実行されるので、計算並列処理が記述できません。一方で、OpenCL™ ではワーク グループに対して計算並列処理は記述できますが、アルゴリズム記述内で並列処理を追加できません。ただし、FPGA のような完全にコンフィギャラブルな計算エンジンの場合は、より柔軟性があるので、この計算並列処理を試してみることができます。

データ並列処理のコード記述

FPGA のアルゴリズムのインプリメンテーション中に計算並列処理をする前に、まずソース コードの計算並列処理が合成ツールで認識されるようにしておく必要があります。ループおよび関数は、ソース記述で計算並列処理および計算ユニットを反映する主な候補ですが、ソース コードの構造によっては SDx™ で必要な変換が適用できないことがあるので、インプリメンテーションで計算並列処理の利点が生かされるかどうかを検証することが重要となります。

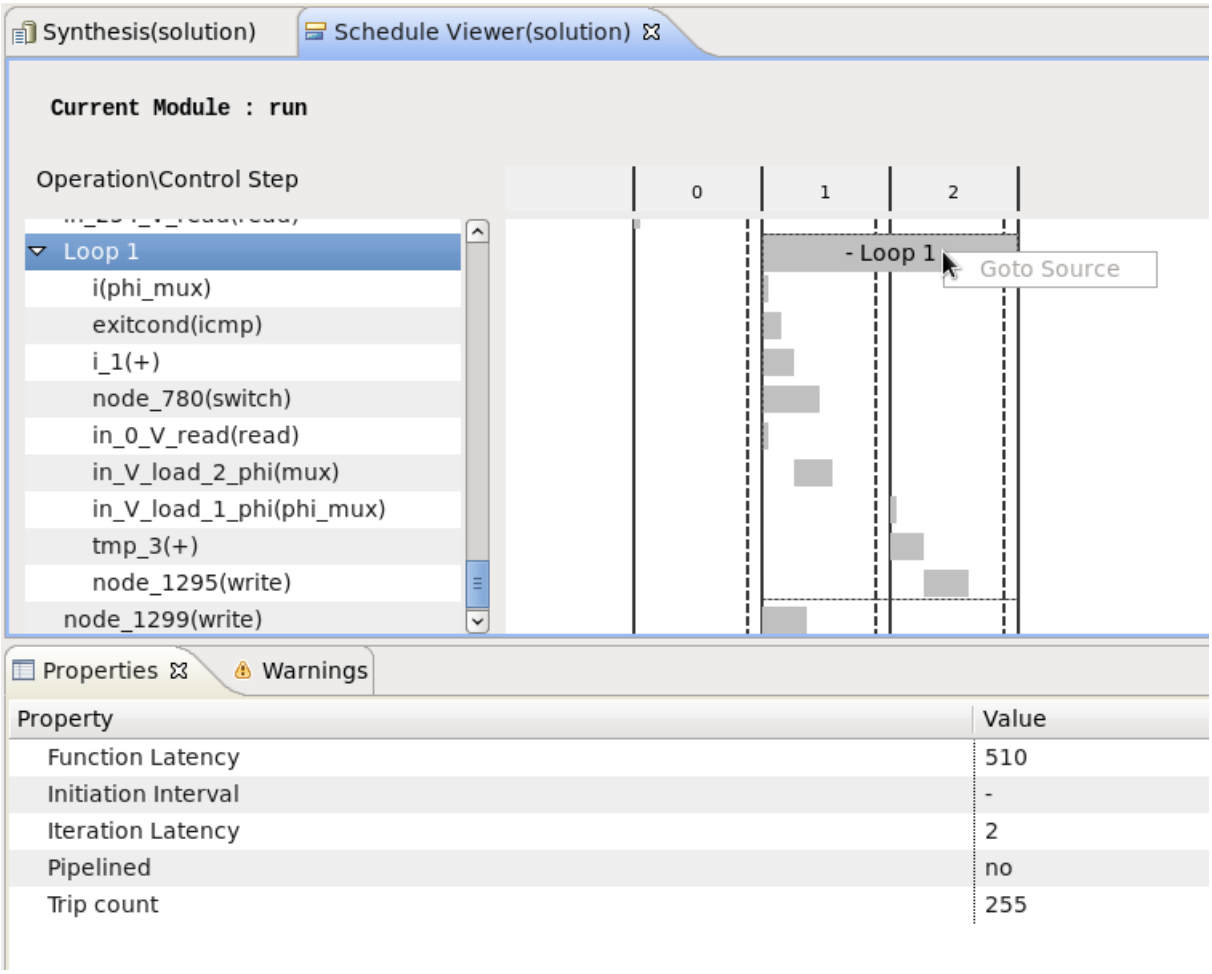
計算並列処理の中には、まずソース コードに反映されないものもあり、ソース コードに追加する必要があることもあります。たとえば、1 つの入力値で動作するようにカーネルが記述されているのに、FPGA インプリメンテーションでは複数値でさらに効率的に並列で計算されることがよくあります。このような並列記述については、全 AXI データ幅セクションを参照してください。512 ビットの インターフェイスは、`int16` または C/C++ 任意精度データ型 `ap_int<512>` などの OpenCL™ ベクター データ型を使用して作成できます。これらのベクター型、たとえば `int16` の場合、並列で動作する最大 16 のデータパスを使用して、カーネル内でデータ並列処理を記述することもできます。ベクター型の使用に推奨される方法については、ザイリンクス オンボーディング例 (GitHub) の vision カテゴリの「Median Filter Example」を参照してください。

ループの並列処理

ループは、繰り返しのアルゴリズム コードを示す基本的な C/C++/OpenCL™ 手法です。次の例は、ループ構造のさまざまな側面を示しています。

```
for(int i = 0; i<255; i++) {
  out[i] = in[i]+in[i+1];
}
out[255] = in[255];
```

このコードは、最後の値の場合を除き、配列の値を反復して増加していきます。このループがそのままインプリメントされると、ループの各反復ではインプリメンテーションに 2 サイクルかかるので、合計 510 サイクルかかります。詳細は、HLS プロジェクトのスケジュール ビューアーから確認できます。



または、合成結果からの合計とレイテンシは次のようになります。

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	5.00	3.123	0.62

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
511	511	511	511	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	47
FIFO	-	-	-	-
Instance	-	-	0	1362
Memory	-	-	-	-
Multiplexer	-	-	-	1145
Register	-	-	52	-
Total	0	0	52	2554
Available	4320	5520	1326720	663360
Available SLR	2160	2760	663360	331680
Utilization (%)	0	0	~0	~0
Utilization SLR (%)	0	0	~0	~0

ここで重要なのは、レイテンシ数と LUT 使用合計です。たとえば、コンフィギュレーションによっては、レイテンシ合計 511 で LUT 使用合計は 47 個になることがあります。これらの値は、インプリメンテーションの選択肢によってかなり異なります。このインプリメンテーションの場合、必要なエリアはかなり少なく、レイテンシは長めになります。

展開ループ

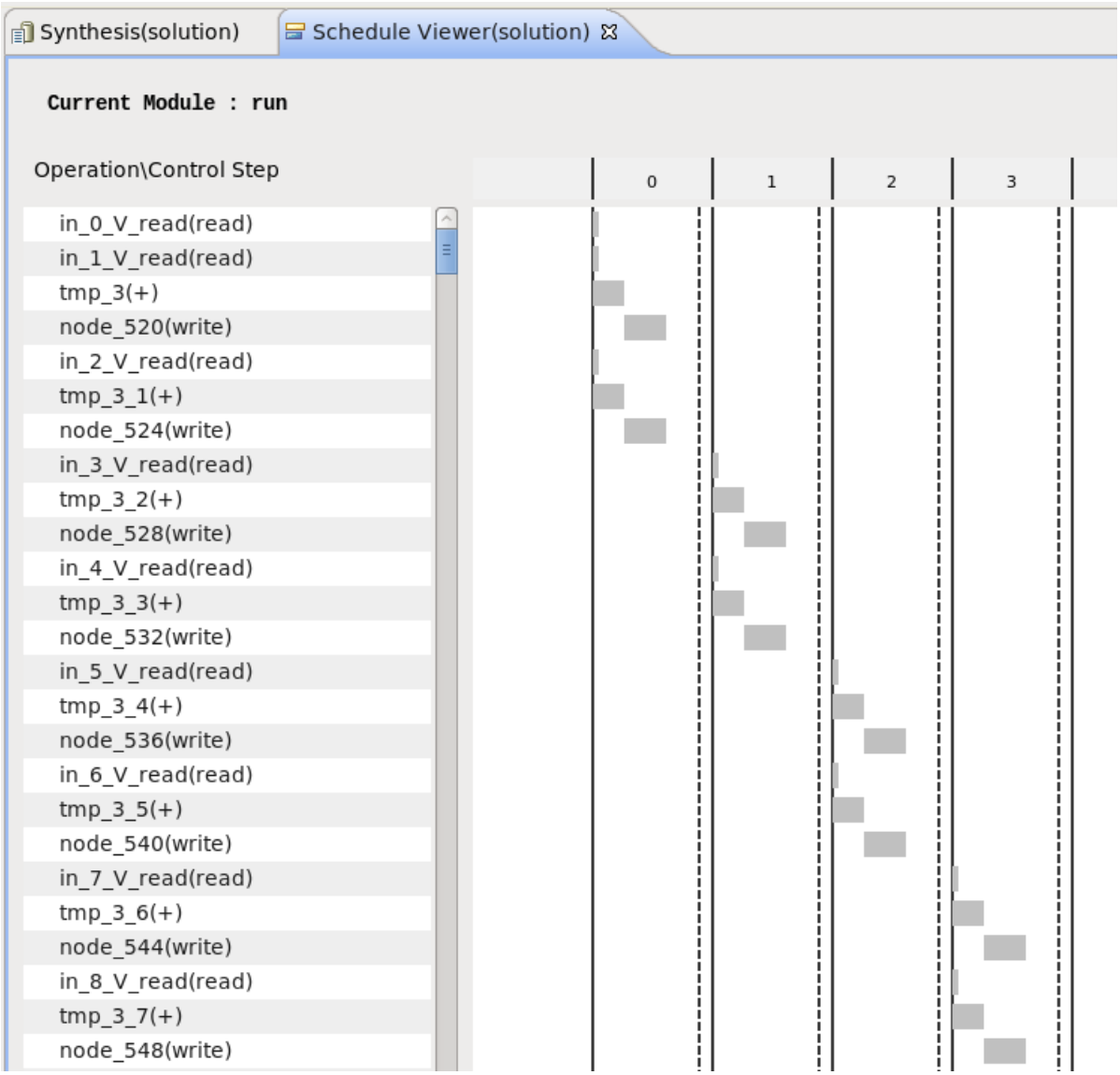
ループを展開すると、モデルを完全に並列処理できるようになります。単に展開するループをマークしておくと、ツールで並列処理を最大限にできるようにしたインプリメンテーションが作成されます。展開するループをマークするには、unroll 属性を使用して OpenCL ループをマークします。

```
__attribute__((opencl_unroll_hint))
```

または、C/C++ ループで unroll プラグマを使用します。

```
#pragma HLS UNROLL
```

特定の例に適用する場合は、HLS プロジェクトのスケジュール ビューアーを使用します。



見積もられたパフォーマンスが表示されます。

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	5.00	3.123	0.62

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
127	127	127	127	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	4845
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	2905
Register	-	-	129	-
Total	0	0	129	7750
Available	4320	5520	1326720	663360
Available SLR	2160	2760	663360	331680
Utilization (%)	0	0	~0	1
Utilization SLR (%)	0	0	~0	2

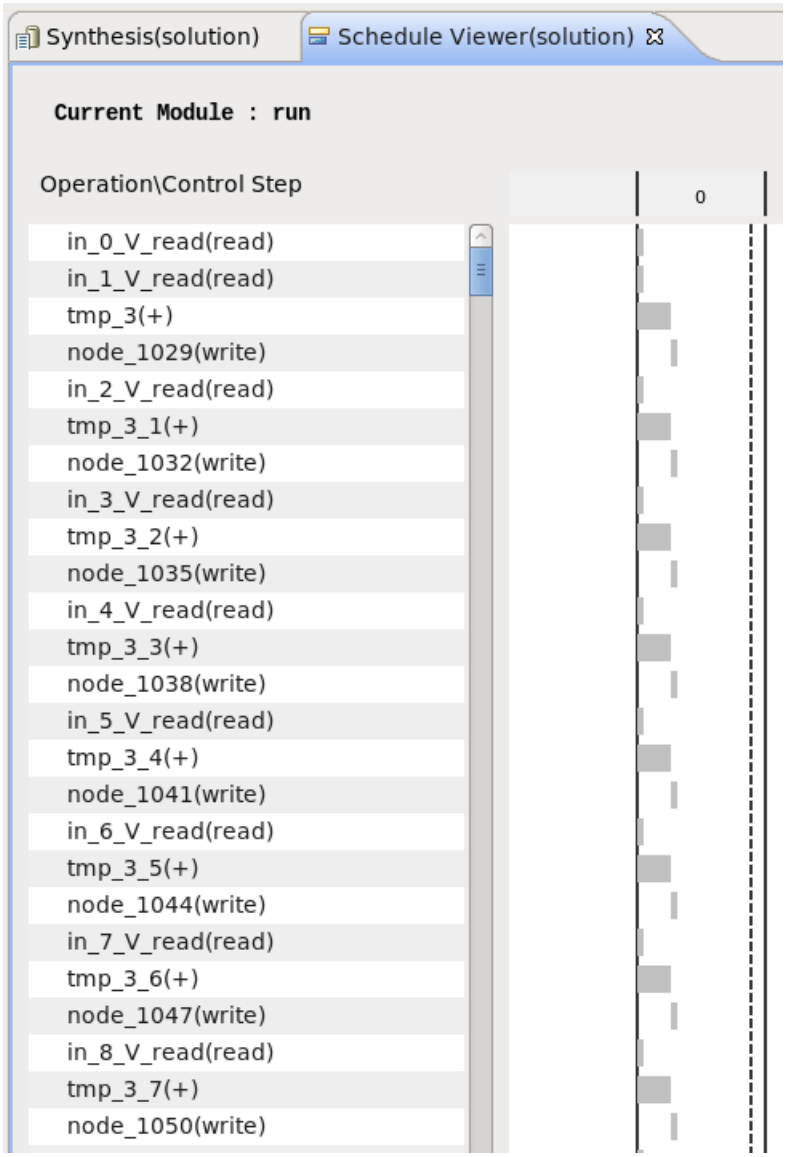
レイテンシ合計はかなり改善されて 127 サイクルになり、同じ計算を並列で実行するため、計算ハードウェアが 4845 個の LUT に増加しています。

ただし、for ループを確認すると、各加算が前のループ反復とは完全に別なので、このアルゴリズムが 1 サイクルでインプリメントできないことがわかります。これは、`out` 変数にメモリ インターフェイスが使用されるからです。SDx™ では、配列に対してデュアル ポート メモリがデフォルトで使用されます。つまり、最大 2 つの値を各サイクルでメモリに書き込むことができます。このため、完全に並列のインプリメンテーションにするには、次のように `out` 変数をレジスタに維持する必要があります。

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

詳細は、『SDx プラグマ リファレンス ガイド』(UG1253) の pragma HLS array_partition セクションを参照してください。

この変更の結果は、スケジュール ビューアーで確認できます。



この場合、見積もりは次のようになります。

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	5.00	1.352	0.62

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
0	0	0	0	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	4845
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	0	0	0	4845
Available	4320	5520	1326720	663360
Available SLR	2160	2760	663360	331680
Utilization (%)	0	0	0	~0
Utilization SLR (%)	0	0	0	1

このコードは、組み合わせ関数としてインプリメントでき、何分の 1 かのサイクルで完了できます。

パイプライン ループ

パイプライン ループを使用すると、ループの反復を時間内にオーバーラップできます。反復を同時に実行できるようにすると、リソースを反復間で共有でき (使用量が少なくてすむ)、展開されないループと比較して必要な実行時間が少なくてすみます。

パイプラインは、次のプラグマを使用して C/C++ でイネーブルにします。

```
#pragma HLS PIPELINE
```

OpenCL では、次の属性を使用します。

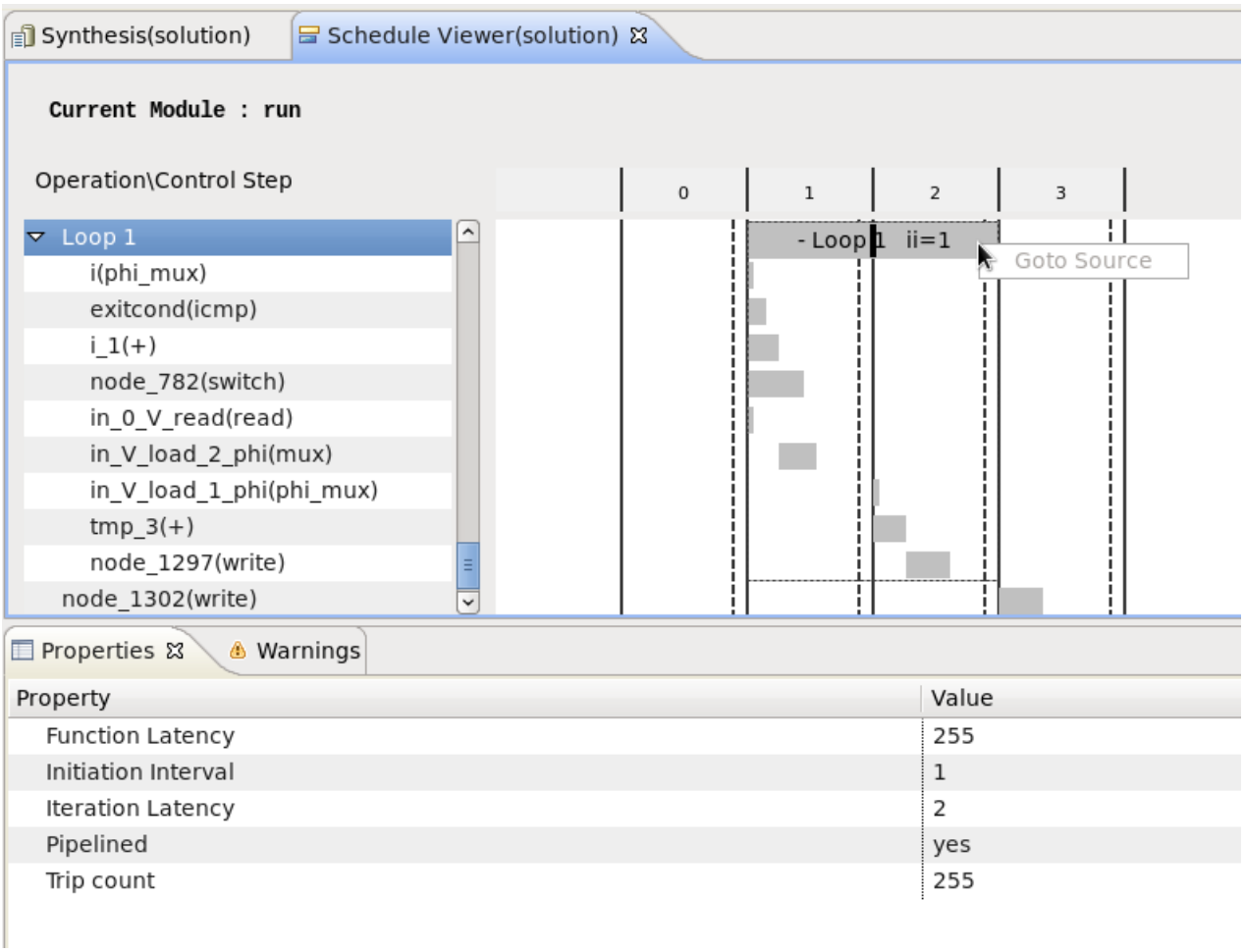
```
__attribute__((xcl_pipeline_loop))
```

OpenCL には、ループ パイプラインを指定する方法がほかにもあります。これは、作業項目のループが明示的に記述されないことと関連しています。これらのループをパイプラインするには次の属性を指定する必要があります。

```
__attribute__((xcl_pipeline_workitems))
```

これらの仕様の詳細は、『SDx プラグマ リファレンス ガイド』 (UG1253) および『SDAccel 環境プログラマ ガイド』 (UG1277) を参照してください。

この例では、HLS プロジェクトのスケジュール ビューアーは次のように表示されます。



全体的な見積もりは次のようになります。

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	5.00	3.123	0.62

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
257	257	257	257	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	53
FIFO	-	-	-	-
Instance	-	-	0	1362
Memory	-	-	-	-
Multiplexer	-	-	-	1173
Register	-	-	47	-
Total	0	0	47	2588
Available	4320	5520	1326720	663360
Available SLR	2160	2760	663360	331680
Utilization (%)	0	0	~0	~0
Utilization SLR (%)	0	0	~0	~0

ループの各反復には 2 サイクルのレイテンシのみがかかるので、オーバーラップする反復は 1 つだけです。これにより、元に比べてレイテンシ合計が半分に抑えられるので、合計レイテンシは 257 サイクルになります。ただし、レイテンシを削減した場合、展開に比べると、リソースがわずかに増加してしまいます。

ほとんどの場合、ループ パイプラインだけで全体のパフォーマンスを改善できますが、パイプラインがどれだけ効率的かはループの構造によって異なります。よくある制限事項は次のとおりです。

- メモリ ポートまたはプロセス チャネルなどのようにリソースに限りがある場合、反復のオーバーラップ (開始間隔) が制限されます。
- 同様に、ループ運搬依存 (1 つの反復内で計算された変数/条件が次の反復に影響する) によりパイプラインの開始間隔が増加することもあります。

これらは、高位合成中にレポートされ、スケジュール ビューアーから確認できます。パフォーマンスを最大にするには、コードを修正してこれらの制限要素を取り除くか、依存性を取り除く (配列のメモリ インプリメンテーションを再構築するか依存をすべてなくす) ようにツールに命令する必要があります。

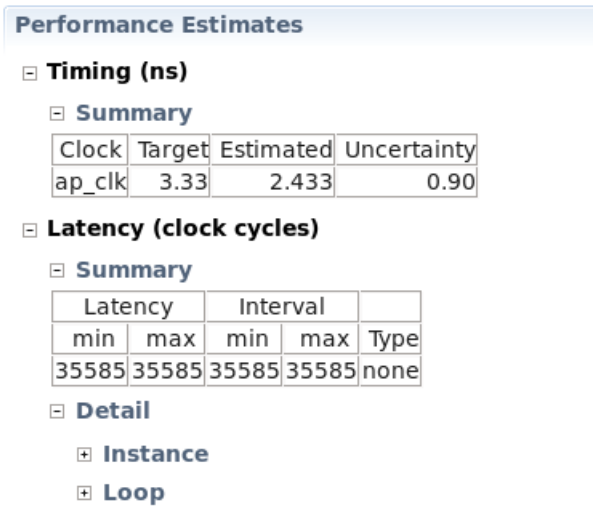
タスクの並列処理

タスクの並列処理を使用すると、データフロー並列処理の利点を生かすことができます。ループの並列処理と違い、タスクの並列処理ではタスク間で発生するバッファリングの利点を生かして、全実行ユニット (タスク) が並列で実行できます。

次の例を参照してください。

```
void run (ap_uint<16> in[1024],
      ap_uint<16> out[1024]
      ) {
  ap_uint<16> tmp[128];
  for(int i = 0; i<8; i++) {
    processA(&(in[i*128]), tmp);
    processB(tmp, &(out[i*128]));
  }
}
```

このコードが実行されると、`processA` および `processB` 関数が順番に 128 回実行されます。ループ内の `processA` および `processB` のレイテンシが合わせて 278 だとすると、レイテンシ合計は次のように見積もられます。



余分なサイクルはループ設定が原因で、これはスケジュール ビューアーで確認できます。

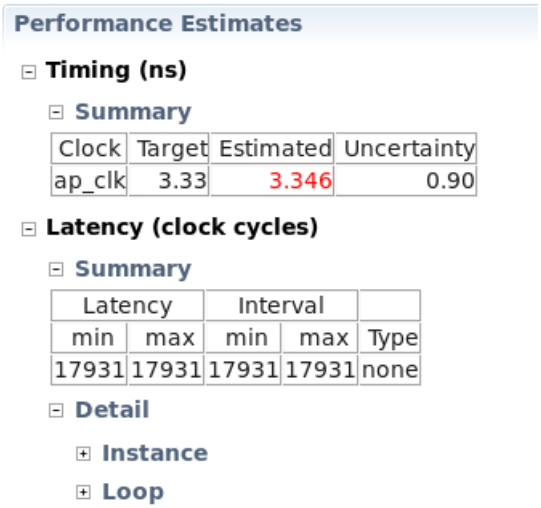
次に、C/C++ モデルで for ループにプラグマを追加してタスクの並列処理を実行します。

```
#pragma HLS DATAFLOW
```

OpenCL™ モデルで for ループ前に属性を追加します。

```
__attribute__ ((xcl_dataflow))
```

これらの仕様および制限の詳細は、『SDx プラグマ リファレンス ガイド』 (UG1253) および『SDAccel 環境プログラマ ガイド』 (UG1277) を参照してください。HLS レポートの見積もりで示したように、タスク間にダブル (ピンポン) バッファリングを使用すると、全体的なパフォーマンスが効率的にかなり改善できます。


Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	3.33	3.346	0.90

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
17931	17931	17931	17931	none

Detail

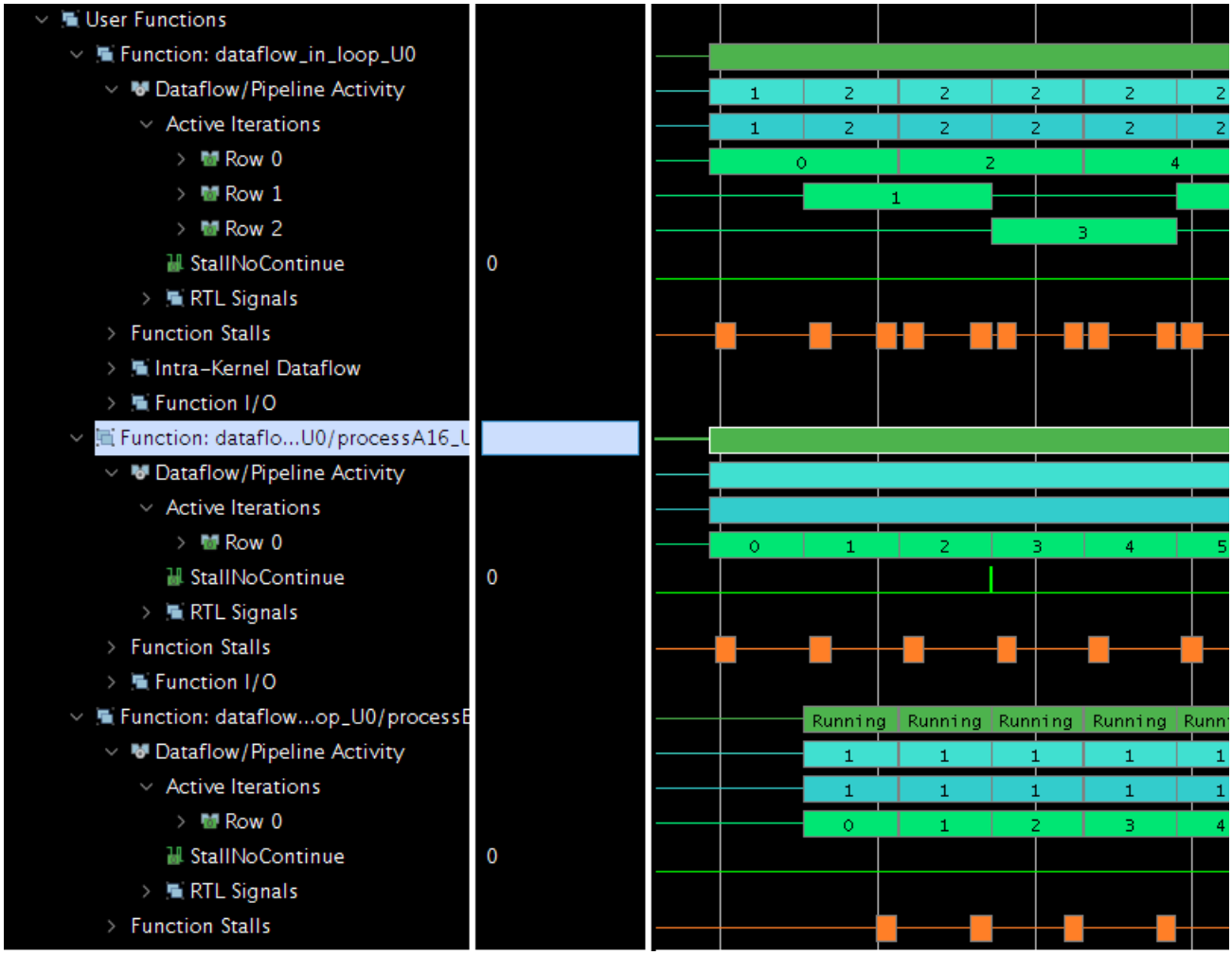
Instance

Loop

この場合、別の反復で別のタスクが同時に実行されるので、デザインの全体的なレイテンシがほぼ半分になります。各関数の処理に 139 サイクルかかり、128 反復が完全にオーバーラップしているので、レイテンシ合計は次のようになります。

```
(1x only processA + 127x both processes + 1x only processB) * 139 cycles =
17931 cycles
```

タスクの並列処理は、インプリメンテーションでパフォーマンスをかなり改善できる手法ですが、DATAFLOW プラグマを特定の任意のコード部分に適用するのがどれくらい効率的かは、状況によってかなり異なることがあります。DATAFLOW を効率的に適用するコーディング ガイドラインは、『SDx プラグマ リファレンス ガイド』 (UG1253) および『SDAccel 環境プログラマ ガイド』 (UG1277) を参照してください。ただし、各タスクの実行パターンを実際に確認して、最終的な DATAFLOW プラグマのインプリメンテーションについて理解しておく必要のあることが多くあります。SDAccel 環境では、終わりの方で詳細なカーネル トレース (Detailed Kernel Trace) が提供され、同時実行について示されます。



この詳細なカーネル トレースでは、データフローされたループの開始が表示されます。プロセス A がループの最初に即座に開始され、プロセス B はプロセス A の終了を待って、最初の反復を開始しますが、プロセス B がループの最初の反復を終了している間に、プロセス A は 2 回目の反復の演算を開始します。

より抽象的な表示は、[Application Timeline] (Host & Device) および [Device Hardware Transaction] ビュー (ハードウェア エミュレーション中のデバイスのみ) に示されます。

計算ユニットの最適化

データ幅

パフォーマンスの改善には、インプリメンテーションに必要なデータ幅が重要となります。ツールはアルゴリズム全体でポート幅を伝搬します。アルゴリズム記述から開始した場合は特に、C/C++/OpenCL™ コードが整数型などの大きなデータ型のみを (デザインのポートでも) 使用することがあります。ただし、アルゴリズムが完全にコンフィギャラブルなインプリメンテーションにマップされていくと、10 または 12 ビットなどのより小さなデータ型で十分なこともあります。このため、最適化中に HLS 合成レポートで基本的な演算のサイズを確認しておくことをお勧めします。通常は、SDx™ がアルゴリズムを FPGA にマップする際、C/C++/OpenCL 構造を把握して動作依存を抽出するのに多くの処理が必要になります。このため、SDx は通常このマップを実行するために、ソース コードを演算ユニットに分割します。これが FPGA にマップされます。これらの演算ユニット (ops) の数およびサイズは、さまざまな要因によって変わります。

次の表では、基本的な演算とそのビット幅がレポートされています。

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	102
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	24	12
Multiplexer	-	-	-	80
Register	-	-	51	-
Total	0	0	75	194
Available	4320	5520	1326720	663360
Available SLR	2160	2760	663360	331680
Utilization (%)	0	0	~0	~0
Utilization SLR (%)	0	0	~0	~0

Detail

Instance

DSP48

Memory

FIFO

Expression

Variable Name	Operation	DSP48E	FF	LUT	Bitwidth P0	Bitwidth P1
i_1_fu_124_p2	+	0	0	11	3	1
i_2_fu_148_p2	+	0	0	15	7	1
i_3_fu_179_p2	+	0	0	15	7	1
sum_i9_fu_194_p2	+	0	0	15	8	8
sum_i_fu_158_p2	+	0	0	15	8	8
exitcond_fu_118_p2	icmp	0	0	9	3	4
exitcond_i6_fu_173_p2	icmp	0	0	11	7	8
exitcond_i_fu_142_p2	icmp	0	0	11	7	8
Total	8	0	0	102	50	39

Multiplexer

Register

アルゴリズムの詳細でよく使用される典型的なビット幅 (16、32、64 ビット) を探して、C/C++/OpenCL ソースからの関連する演算にこれほどの大きさが実際に必要なのかどうかを検証します。演算が小さいほど計算時間も少なくてすむので、これによりアルゴリズムのインプリメンテーションがかなり改善する可能性があります。

固定小数点の演算

アプリケーションの中には、ほかのハードウェア アーキテクチャ用に最適化されているというだけの理由で、浮動小数点計算が使用されているものもあります。『ザイリンクス デバイスでの INT8 に最適化した深層学習の実装』(WP486) で説明するように、深層学習のようなアプリケーションに固定小数点演算を使用すると、同じレベルの精度を保ちつつ消費電力とエリアを大幅に節約できます。浮動小数点演算を使用する前に、アプリケーションに固定小数点演算を使用することを検討してみることをお勧めします。

マクロ演算

より大きな計算エレメントについて考慮した方が良い場合もあります。ツールはソース コードを残りのソース コードとは別に実行し、周囲の演算を気にせずアルゴリズムを FPGA に効率的にマップします。この場合、SDx は演算境界を維持したままで、特定コードに対してマクロ演算を効率的に作成します。これには、次の原則が使用されます。

- マッピング プロセスに対する演算局所性
- 経験則のための複雑性の削減

これにより、結果がかなり異なることがあります。C/C++ マクロ演算は次を使用して作成します。

```
#pragma HLS inline off
```

OpenCL では、関数を定義する場合、同様のマクロは属性を指定しなくても生成できます。

```
__attribute__((always_inline))
```

最適化済みライブラリの使用

OpenCL 仕様には、多くの数学ビルトイン関数が含まれます。native_ が前に付いた数学ビルトイン関数はすべて 1 つまたは複数のネイティブ デバイス命令にマップされ、通常は該当する関数 (native_ 接頭語なし) よりもかなり優れたパフォーマンスになります。これらの関数の正確性と入力範囲 (場合による) はインプリメンテーションで定義されます。SDAccel™ 環境では、これらの native_ ビルトイン関数で Vivado® HLS Math ライブラリ (エリアおよびパフォーマンスに関してザイリンクス FPGA 用に既に最適済み) と同等の関数が使用されます。ザイリンクスでは、精度がアプリケーション要件を満たす場合は、native_ ビルトイン関数または HLS Math ライブラリを使用することをお勧めしています。

メモリ アーキテクチャの最適化

メモリ アーキテクチャはインプリメンテーションの重要な側面です。帯域幅のアクセスには制限があり、全体的なパフォーマンスにかなり影響することがあります。次の例を参照してください。

```
void run (ap_uint<16> in[256][4],
          ap_uint<16> out[256]
         ) {
  ...
  ap_uint<16> inMem[256][4];
  ap_uint<16> outMem[256];

  ... Preprocess input to local memory

  for( int j=0; j<256; j++) {
    #pragma HLS PIPELINE OFF
    ap_uint<16> sum = 0;
    for( int i = 0; i<4; i++) {

      sum += inMem[j][i];
    }
```

```
    outMem[j] = sum;
  }

  ... Postprocess write local memory to output
}
```

このコードでは、2 次元入力配列の内部次元に関連する 4 つの値が追加されます。これ以上変更をしないでインプリメントすると、次のような見積もりになります。

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	3.33	2.433	0.90

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
5908	5908	5908	5908	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	1034	1034	12	1	1	1024	yes
- Loop 2	4608	4608	18	-	-	256	no
+ Loop 2.1	16	16	4	-	-	4	no
- Loop 3	257	257	3	1	1	256	yes

全体的なレイテンシが 4608 (Loop 2) なのは、18 サイクル (内部ループ 16 サイクル + 合計のリセット + 書き出される出力) が 256 回反復されているためです。これは、HLS プロジェクトのスケジュール ビューアーで確認できます。見積もりは、内部ループを展開するとかなり改善されます。

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	3.33	2.433	0.90

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
2580	2580	2580	2580	none

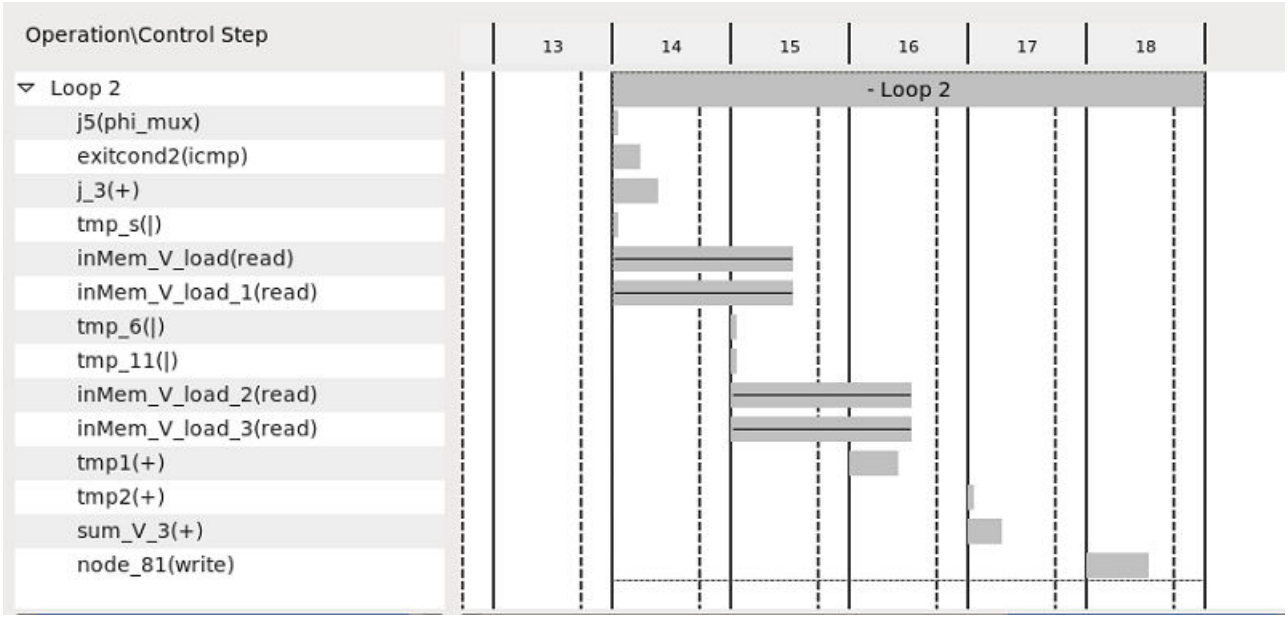
Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	1034	1034	12	1	1	1024	yes
- Loop 2	1280	1280	5	-	-	256	no
- Loop 3	257	257	3	1	1	256	yes

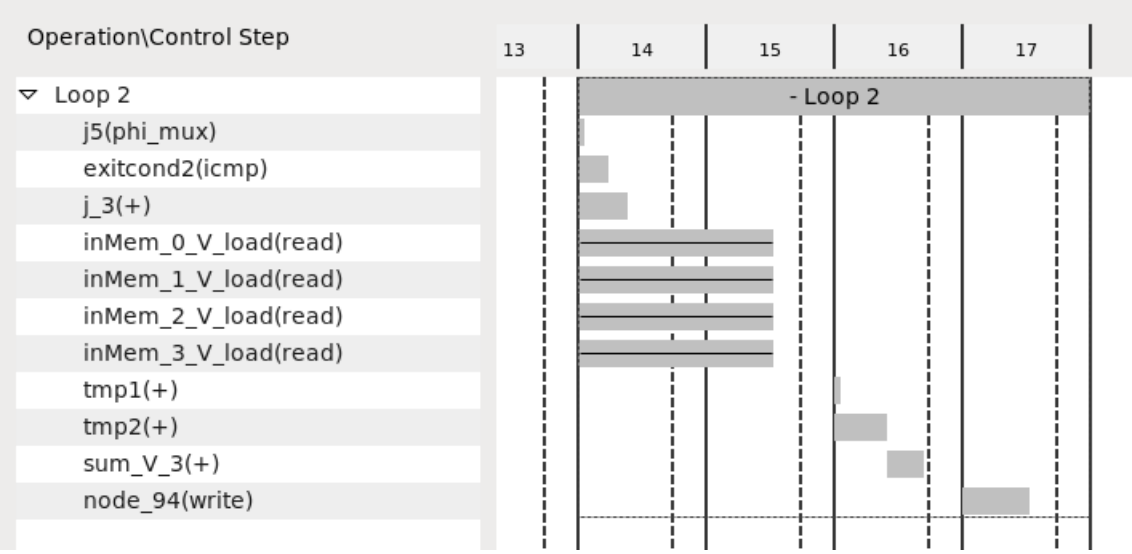
ただし、このように改善されるのは、主にプロセスがデュアル ポート メモリの両方のポートを使用しているためです。これは、プロジェクトのスケジュール ビューアーから確認できます。



メモリからのすべての値にアクセスして合計を計算するために、2 つの読み出しがサイクルごとに実行されています。これにより、メモリへのアクセスが完全にブロックされてしまうので、望ましくない結果になることがよくあります。結果をさらに改善するには、2 次元を使用してメモリを 4 つの小さなメモリに分割します。

```
#pragma HLS ARRAY_PARTITION variable=inMem complete dim=2
```

これにより、4 つの配列読み出しになり、すべてが 1 つのポートを使用して異なるメモリで実行されます。



合計 256 * 4 サイクルを使用 = Loop 2 に 1024 サイクル

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	3.33	2.433	0.90

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
2324	2324	2324	2324	none

Detail

Instance

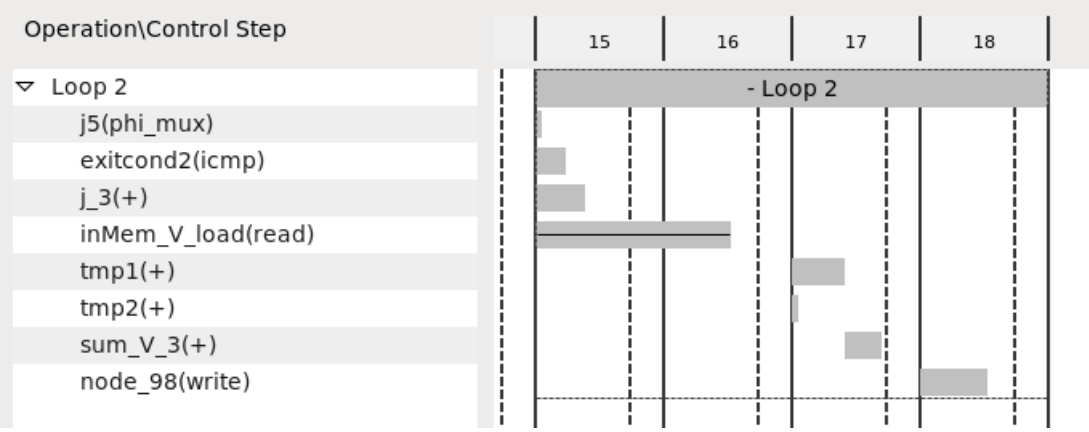
Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	1034	1034	12	1	1	1024	yes
- Loop 2	1024	1024	4	-	-	256	no
- Loop 3	257	257	3	1	1	256	yes

または、メモリを 4 ワードの 1 つのメモリに再形成します。これには、プラグマを使用します。

```
#pragma HLS array_reshape variable=inMem complete dim=2
```

これにより、配列パーティションの場合と同じレイテンシになりますが、この場合は 1 つのポートを使用した 1 つのメモリになります。



どちらのソリューションでも全体的なレイテンシおよび使用量は同じようになりますが、配列を再形成した方がインターフェイスがきれいで、配線の密集は少なくなります。これで配列最適化は終わりですが、実際のデザインでは、ループの並列処理をするとレイテンシがさらに改善できることがあります (詳細はループの並列処理を参照)。

```
void run (ap_uint<16> in[256][4],
      ap_uint<16> out[256]
      ) {
  ...

  ap_uint<16> inMem[256][4];
  ap_uint<16> outMem[256];
  #pragma HLS array_reshape variable=inMem complete dim=2

  ... Preprocess input to local memory

  for( int j=0; j<256; j++) {
    #pragma HLS PIPELINE OFF
    ap_uint<16> sum = 0;
    for( int i = 0; i<4; i++) {
      #pragma HLS UNROLL
      sum += inMem[j][i];
    }
    outMem[j] = sum;
  }

  ... Postprocess write local memory to output

}
```

XILINX

第 5 章

ホスト最適化

このセクションでは、ホスト コード最適化について説明します。ホスト コードでは OpenCL™ API を使用して、個別計算ユニット実行および FPGA とのデータ転送がスケジュールされます。このため、OpenCL キューを使用した同時実行について考慮する必要があります。このセクションでは、よくあるミスの詳細とそれらの見つけ方と解決方法について説明します。

カーネルのキュー追加のオーバーヘッドの削減

OpenCL™ 実行モデルでは、データ並列とタスク並列のプログラミング モデルがサポートされます。カーネルは通常 OpenCL ランタイムで複数回キューに追加されてから、デバイスで実行されるようにスケジュールされます。次のいずれかの方法でカーネルを開始するコマンドを送信する必要があります。

- データ並列の場合 clEnqueueNDRange API を使用
- タスク並列の場合 clEnqueueTask を使用

この送信プロセスはホスト プロセッサで実行されます。実際のコマンドおよびカーネル引数は PCIe® リンクを介して FPGA に送信する必要があります。SDAccel™ 環境の OpenCL ランタイム ライブラリでは、FPGA へのコマンドおよび引数の送信のオーバーヘッドは、カーネルの引数の数によって 30us ～ 60us になります。このオーバーヘッドの影響は、カーネルを実行する必要のある回数を最小限に抑えると減らすことができます。

データ並列の場合、ザイリンクスではホスト コードとカーネルのサイズに合わせてグローバルおよびローカル ワーク サイズを注意して選択して、グローバル ワーク サイズがローカル ワーク サイズの少ない倍数になるようにすることをお勧めしています。理想的なのは、次のコード例のようにグローバル ワーク サイズとローカル ワーク サイズを同じにすることです。

```
size_t global = 1;
size_t local = 1;
clEnqueueNDRangeKernel(world.command_queue, kernel, 1, nullptr,
                       &global, &local, 2, write_events.data(),
                       &kernel_events[0]));
```

タスク並列の場合、ザイリンクスでは clEnqueueTask の呼び出しを最小限にすることをお勧めしています。理想的なのは、すべてのワークロードを clEnqueueTask の呼び出し 1 つで終了させるようにすることです。

データ転送とカーネル計算のオーバーラップ

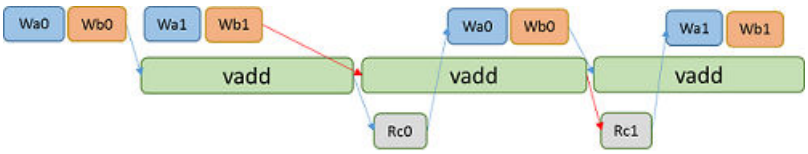
データベース分析のようなアプリケーションには、アクセラレーションされたデバイスで使用可能なメモリより大きなデータ セットが含まれ、完全なデータが転送されてブロックで処理される必要があります。これらのアプリケーションで優れたパフォーマンスを達成するには、データ転送と計算をオーバーラップさせる手法が重要となります。

次は、ザイリンクス オンボーディング例 (GitHub) の host カテゴリの「OpenCL™ Overlap Data Transfers with Kernel Computation Example」からのベクター加算カーネルです。

```
kernel __attribute__((reqd_work_group_size(1, 1, 1)))
void vadd(global int* c,
          global const int* a,
          global const int* b,
          const int offset,
          const int elements)
{
    int end = offset + elements;
    vadd_loop: for (int x=offset; x<end; ++x) {
        c[x] = a[x] + b[x];
    }
}
```

この例の場合、バッファー a の書き込み (Wa)、バッファー b の書き込み (Wb)、vadd カーネルの実行、バッファー c の読み出し (Rc) の 4 つのタスクがホスト アプリケーションで実行されます。OpenCL データ転送とカーネル実行 API は非同期なので、次の図に示すようにデータ転送とカーネル実行をオーバーラップできます。この例では、すべてのバッファーに対してダブル バッファリングが使用されるので、計算ユニットが 1 セットのバッファーを処理している間に、ホストがもう 1 つのバッファーのセットで動作できます。OpenCL イベント オブジェクトを使用すると、簡単に複雑な動作依存を設定して、ホスト スレッドとデバイス動作を同期できます。次の図の矢印は、最適なパフォーマンスを達成するためにイベント トリガーをどのように設定するかを示しています。

図 13: イベント トリガーの設定



次のホスト コード例は、ループ内で 4 つのタスクを待機キューに追加しています。また、異なるタスク間のイベント同期を設定することで、各タスクのデータ依存が達成されるようにもなっています。ダブル バッファリングは、異なるメモリ オブジェクト値を `clEnqueueMigrateMemObjects` API に渡すと設定されます。イベント同期は、各 API がほかのイベントを待ち、その API が終了してから自身のイベントをトリガーするようにすると達成できます。

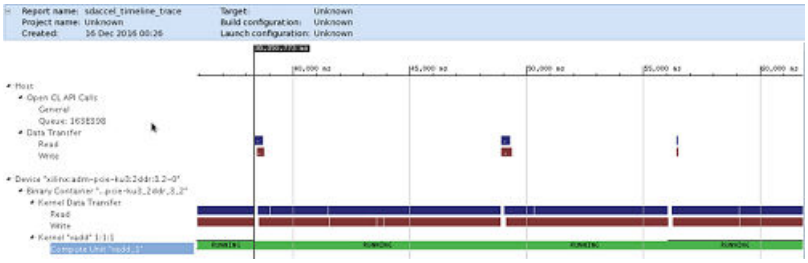
```
  for (size_t iteration_idx = 0; iteration_idx < num_iterations;
iteration_idx++) {
    int flag = iteration_idx % 2;

    if (iteration_idx >= 2) {
        clWaitForEvents(1, &map_events[flag]);
        OCL_CHECK(clReleaseMemObject(buffer_a[flag]));
        OCL_CHECK(clReleaseMemObject(buffer_b[flag]));
        OCL_CHECK(clReleaseMemObject(buffer_c[flag]));
        OCL_CHECK(clReleaseEvent(read_events[flag]));
        OCL_CHECK(clReleaseEvent(kernel_events[flag]));
    }

    buffer_a[flag] = clCreateBuffer(world.context,
            CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
          bytes_per_iteration, &A[iteration_idx *
elements_per_iteration], NULL);
    buffer_b[flag] = clCreateBuffer(world.context,
            CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
          bytes_per_iteration, &B[iteration_idx *
elements_per_iteration], NULL);
```

```
    buffer_c[flag] = clCreateBuffer(world.context,
            CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
           bytes_per_iteration, &device_result[iteration_idx *
elements_per_iteration], NULL);
    array<cl_event, 2> write_events;
    printf("Enqueueing Migrate Mem Object (Host to Device) calls\n");
    // These calls are asynchronous with respect to the main thread
because we
    // are passing the CL_FALSE as the third parameter. Because we are
passing
    // the events from the previous kernel call into the wait list, it
will wait
    // for the previous operations to complete before continuing
    OCL_CHECK(clEnqueueMigrateMemObjects(
        world.command_queue, 1, &buffer_a[iteration_idx % 2],
        0 /* flags, 0 means from host */,
        0, NULL,
        &write_events[0]));
    set_callback(write_events[0], "ooo_queue");

    OCL_CHECK(clEnqueueMigrateMemObjects(
        world.command_queue, 1, &buffer_b[iteration_idx % 2],
        0 /* flags, 0 means from host */,
        0, NULL,
        &write_events[1]));
    set_callback(write_events[1], "ooo_queue");

    xcl_set_kernel_arg(kernel, 0, sizeof(cl_mem), &buffer_c[iteration_idx
% 2]);
    xcl_set_kernel_arg(kernel, 1, sizeof(cl_mem), &buffer_a[iteration_idx
% 2]);
    xcl_set_kernel_arg(kernel, 2, sizeof(cl_mem), &buffer_b[iteration_idx
% 2]);
    xcl_set_kernel_arg(kernel, 3, sizeof(int), &elements_per_iteration);

    printf("Enqueueing NDRange kernel.\n");
    // This event needs to wait for the write buffer operations to complete
    // before executing. We are sending the write_events into its wait
list to
    // ensure that the order of operations is correct.
    OCL_CHECK(clEnqueueNDRangeKernel(world.command_queue, kernel, 1,
nullptr,
                                     &global, &local, 2 ,
write_events.data(),
                                     &kernel_events[flag]));
    set_callback(kernel_events[flag], "ooo_queue");

    printf("Enqueueing Migrate Mem Object (Device to Host) calls\n");
    // This operation only needs to wait for the kernel call. This call
will
    // potentially overlap the next kernel call as well as the next read
    // operations
    OCL_CHECK( clEnqueueMigrateMemObjects(world.command_queue, 1,
&buffer_c[iteration_idx % 2],
                CL_MIGRATE_MEM_OBJECT_HOST, 1, &kernel_events[flag],
&read_events[flag]));

    set_callback(read_events[flag], "ooo_queue");
    clEnqueueMapBuffer(world.command_queue, buffer_c[flag], CL_FALSE,
CL_MAP_READ, 0,
            bytes_per_iteration, 1, &read_events[flag], &map_events[flag],
0);
```

```
    set_callback(map_events[flag], "ooo_queue");

    OCL_CHECK(clReleaseEvent(write_events[0]));
    OCL_CHECK(clReleaseEvent(write_events[1]));
  }
```

次の [Application Timeline] ビューでは、データ転送時間は完全に非表示になっていますが、計算ユニット vadd_1 が常に実行されるところを示しています。

図 14: データ転送時間が非表示になった [Application Timeline] ビュー



複数計算ユニットの使用

ターゲット デバイスで使用可能なリソースによって、同じカーネルまたは異なるカーネルの複数の計算ユニットを作成して並列で実行することで、システムの処理時間とスループットを改善できます。次に例を示します。

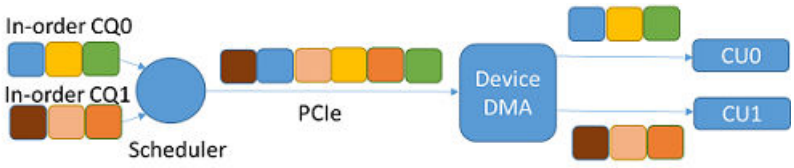
```
xocc -l --nk
<kernel_name:number(:compute_unit_name1.compute_unit_name2...)>
```

アプリケーションは、複数の順番どおりのコマンド キューまたは単一の順番外のコマンド キューを作成することで、ターゲット デバイスで複数の計算ユニットを使用できるようになります。

複数の順番どおりのコマンド キュー

次の図は、2 つの順番どおりのコマンド キュー (CQ0 および CQ1) の例を示しています。スケジューラは各キューからのコマンドを順番どおりに実行しますが、CQ0 および CQ1 からのコマンドをどの順序でも取り出すことができます。必要な場合は、CQ0 および CQ1 間の同期を管理する必要があります。

図 15: 2 つの順番どおりのコマンド キューの例



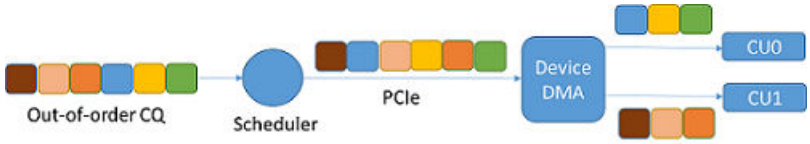
次は、ザイリンクス オンボーディング例 (GitHub) の host カテゴリの「Concurrent Kernel Execution Example」からのコード例で、複数の順番どおりのコマンド キューを設定して、各キューにコマンドを追加しています。

```
cl_command_queue ordered_queue1 = clCreateCommandQueue(
      world.context, world.device_id, CL_QUEUE_PROFILING_ENABLE, &err)

cl_command_queue ordered_queue2 = clCreateCommandQueue(
world.context, world.device_id, CL_QUEUE_PROFILING_ENABLE, &err);

clEnqueueNDRangeKernel(ordered_queue1, kernel_mscale, 1, offset,
                       global, local, 0, nullptr,
                       &kernel_events[0]));

clEnqueueNDRangeKernel(ordered_queue1, kernel_madd, 1, offset,
                       global, local, 0, nullptr,
                       &kernel_events[1]);

clEnqueueNDRangeKernel(ordered_queue2, kernel_mmult, 1, offset,
                       global, local, 0, nullptr,
                       &kernel_events[2]);
```

単一の順番外コマンド キュー

次の図は、単一の順番外のコマンド キュー CQ の例を示しています。スケジューラは、CQ からのコマンドをどの順番でも実行できます。必要であれば、イベント依存および同期を設定します。

図 16: 単一の順番外コマンド キューの例



次は、ザイリンクス オンボーディング例 (GitHub) の「Concurrent Kernel Execution Example」からのコード例で、単一の順番外コマンド キューを設定して、コマンドを各キューに追加しています。

```
cl_command_queue ooo_queue = clCreateCommandQueue(
      world.context, world.device_id,
      CL_QUEUE_PROFILING_ENABLE | CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE,
&err);

clEnqueueNDRangeKernel(ooo_queue, kernel_mscale, 1, offset, global,
                       local, 0, nullptr, &ooo_events[0]);

clEnqueueNDRangeKernel(ooo_queue, kernel_madd, 1, offset, global,
                      local, 1,
                      &ooo_events[0], // Event from previous call
                      &ooo_events[1]);

clEnqueueNDRangeKernel(ooo_queue, kernel_mmult, 1, offset, global,
                       local, 0,
                       nullptr, // Does not depend on previous call
                       &ooo_events[2])
```

次の [Application Timeline] ビューは、複数の順番キュー手法と単一の順番外キュー手法の両方を使用して、計算ユニット `mmult_1` が計算ユニット `mscale_1` および `madd_1` と並列で実行されているところを示しています。

図 17: mult_1 が mscale_1 と madd_1 と一緒に実行されていることを示す [Application Timeline] ビュー

clEnqueueMigrateMemObjects を使用したデータ転送

OpenCL™ には、ホストとデバイス間でデータを転送するための API が多く含まれます。通常は、`clEnqueueWriteBuffer` および `clEnqueueReadBuffer` などのデータ移動 API でメモリ オブジェクトがキューに追加された後に暗示的にデバイスへ移動されますが、必ずデータが転送されるわけではないので、ホスト アプリケーションでデバイスのメモリ オブジェクトの配置とカーネルで実行される計算がオーバーラップしにくくなることがあります。

OpenCL 1.2 に含まれる新しい API の `clEnqueueMigrateMemObjects` を使用すると、メモリ移動が依存コマンドよりも前に明示的に実行されるようにできます。これにより、アプリケーションが通常のコマンド キューのスケジュールを使用して次の別のコマンドの準備ができるように、前もってメモリ オブジェクトの関連付けを変更できます。また、メモリ オブジェクトが必要となる前にその配置とその他の関連しない演算をオーバーラップさせて、発生する可能性のある転送レイテンシを削減することもできます。`clEnqueueMigrateMemObjects` に関連付けられたイベントが CL_COMPLETE とマークされたら、mem_objects で指定されたメモリ オブジェクトは問題なく command_queue に関連付けられたデバイスに移動できます。

`clEnqueueMigrateMemObjects` API は、メモリ オブジェクト作成後の最初の配置を指定するためにも使用でき、最初のキュー追加コマンドでオブジェクトをインスタンシエートする際の開始オーバーヘッドをなくすことができることもあります。

また、`clEnqueueMigrateMemObjects` には、複数のメモリ オブジェクトを 1 つの API 呼び出しに移動できるという利点もあります。これにより、メモリ オブジェクトが複数ある場合のデータ転送のスケジューリングおよび関数呼び出しのオーバーヘッドが削減します。

次は、ザイリンクス オンボーディング例 (GitHub) の host カテゴリの「Vector Multiplication for XPR Device」からの `clEnqueueMigrateMemObjects` を使用したコード例です。

```
int err = clEnqueueMigrateMemObjects(
                        world.command_queue,
                        1,
                        &d_mul_c,
                        CL_MIGRATE_MEM_OBJECT_HOST,
                        0,
                        NULL,
                        NULL);
```

XILINX

付録 A

オンボーディング例

SDAccel™ 環境を始めるユーザーのため、ザイリンクス オンボーディング例 (GitHub) には、良いデザイン プラクティス、コーディング ガイドライン、よくあるアプリケーションのデザイン パターンなどの例が含まれ、アプリケーション パフォーマンスを最大限にするために最も重要な最適化手法が示されています。オンボーディング例は、複数の主なテゴリに分類されています。カテゴリごとにコンセプトが異なり、個別の例が OpenCL™ C および C/C++ (適用可能な場合) で記述されています。すべての例に、ソフトウェア エミュレーション、ハードウェア エミュレーションを実行したり、ハードウェアで実行するのに必要な makefile と、例を詳細に説明する `README.md` が含まれます。

付録 B

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

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ます。

商標

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。