SDAccel 环境剖析和最优化指南

UG1207 (v2019.1) 2019 年 6 月 5 日

查看所有版本

XILINX

修订历史

下表列出了本文档的修订历史。

章节	修订总结
2019 年 6 月 5 日 2019.1 版	
配置信息摘要报告	更新了 数据解读 以反映当前视图。
“Waveform”视图	更新了 “Waveform”视图的数据解读 和 实时波形数据 中的图像和文本以反映当前波形视图。
存储器数据传输类型	添加了有关数据传输类型的信息。
顶层数据流	添加了有关顶层数据流最优化使用的讨论。
常规更新	更新了图和少量的编辑修改。

目录

XILINX

第 1 章

引言

本指南介绍了与设计的性能分析相关的所有 SDx™ 开发环境功能。它还按逻辑进行了组织，以助于实际性能提升。为 SDAccel™ 环境性能瓶颈的主要组件提供了专门章节，即加速器、PCIe® 总线传输和主机代码。每个章节的组织都是为了指导开发者，内容从如何识别瓶颈到提高整体系统性能的各种解决方法。

注释: 性能最优化以旨在实现性能提升的工作设计作为假设起点。如果出现错误行为，请查找《SDAccel 环境调试指南》 (UG1281) 中的指南。

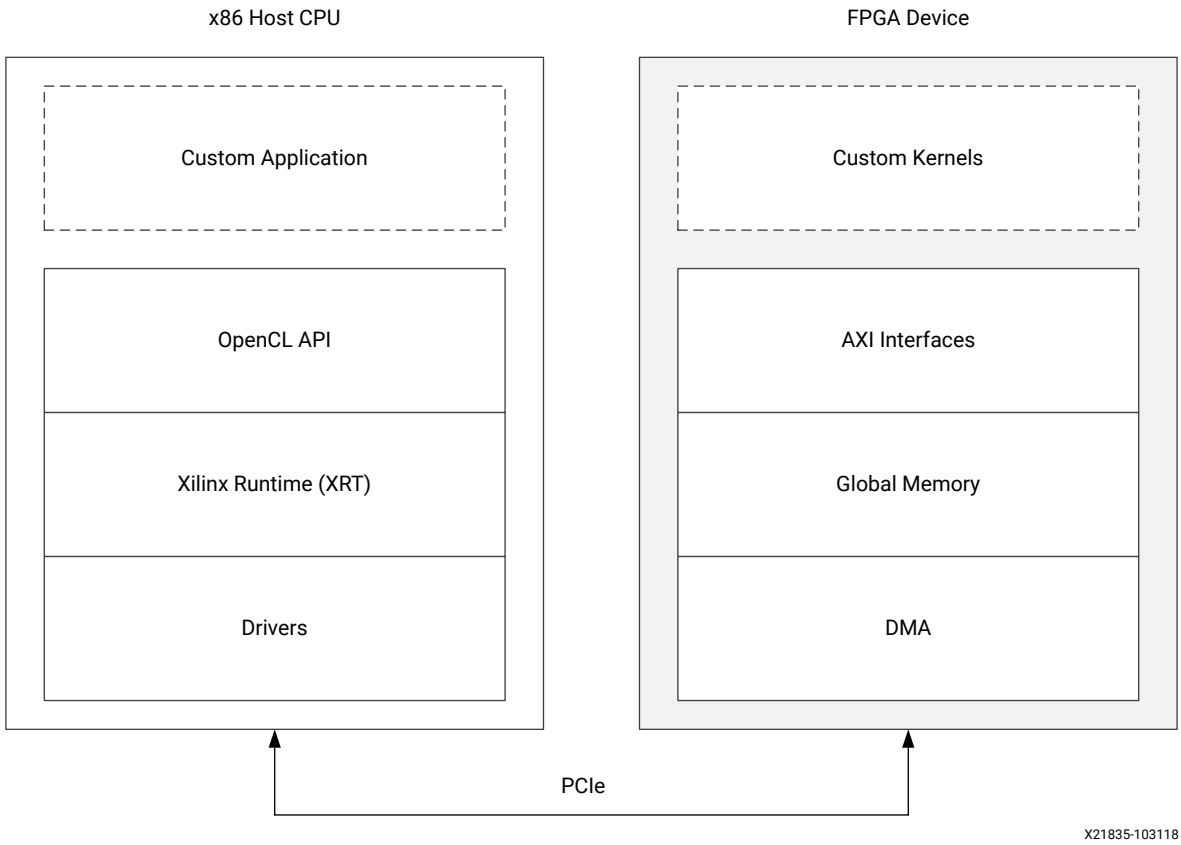
同理，此处并未对与主机代码编码或加速器内核相关的一般概念进行解释；这些概念将在《SDAccel 环境编程指南》(UG1277) 中介绍。

SDAccel 可执行模型

在 SDAccel 框架中，应用程序在主机应用程序和硬件加速的内核之间分配，它们之间具有通信通道。使用 C/C++ 编写并使用 API 抽象（如 OpenCL）的主机应用程序在 x86 服务器上运行，而硬件加速的内核在赛灵思 FPGA 内运行。由赛灵思运行时 (XRT) 管理的 API 调用用于与硬件加速器通信。主机 x86 机器和加速器电路板之间的通信，包括控制和数据传输，通过 PCIe 总线进行。当控制信息在硬件中的特定存储器位置之间传输时，全局存储器用于在主机应用程序和内核之间传输数据。主机处理器和硬件加速器都可以访问全局存储器，而主机存储器只能由主机应用程序访问。

例如，在典型的应用程序中，主机首先将内核操作的数据从主机存储器传输到全局存储器。内核随后将对数据进行操作，将结果存储回全局存储器。内核完成后，主机会将结果传回主机存储器。主机和全局存储器之间的数据传输引入了时延，这对于整体加速来说可能是代价昂贵的。要在实际系统中实现加速，硬件加速内核所获得的收益必须超过数据传输的额外时延。此加速平台的常见结构如下图所示。

图 1: **SDAccel 应用程序的架构**



右侧的 FPGA 硬件平台包含硬件加速的内核、全局存储器以及用于存储器传输的 DMA。内核是可编程的，可以拥有一个或者多个全局存储器接口。SDAccel 可执行模型可以分解为以下步骤：

1. 主机应用程序通过 PCIe 接口将内核所需的数据写入连接器件的全局存储器中。
2. 主机应用程序使用其输入参数设置内核。
3. 主机应用程序触发 FPGA 上内核函数的执行。
4. 必要时，内核在从全局存储器中读取数据时执行所需的计算。
5. 内核将数据写入全局存储器并通知主机它已完成任务。
6. 主机应用程序将数据从全局存储器读取到主机存储器，并根据需要继续处理。

FPGA 可以同时容纳多个内核实例，这可能发生在不同类型的内核或同一内核的多个实例之间。XRT 透明地协调主机应用程序和加速器中的内核之间的通信。内核的实例数由编译选项决定。

SDAccel 构建进程

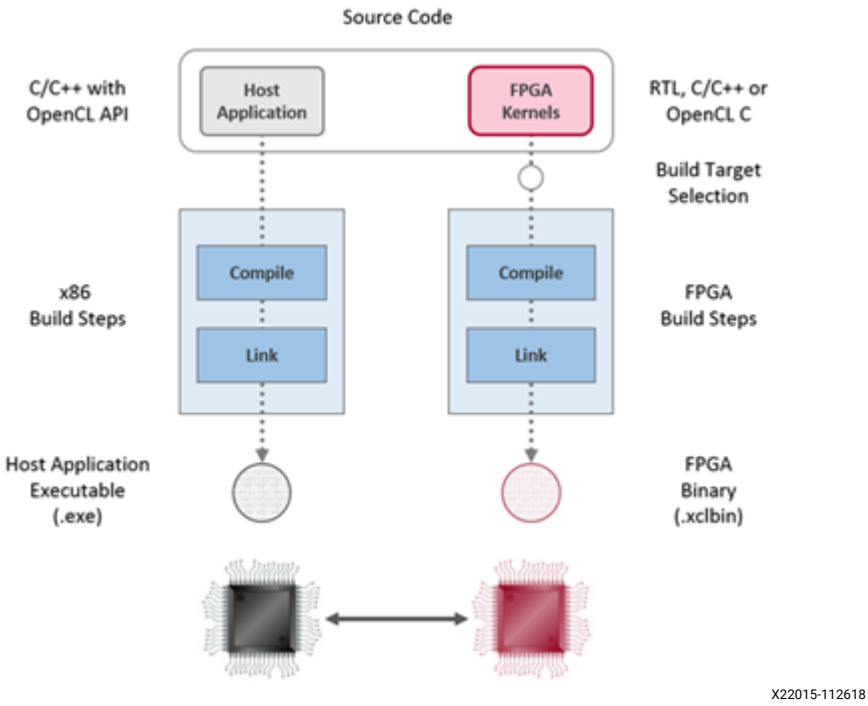
SDAccel 环境提供标准软件开发环境的所有特性：

- 用于主机应用的最优化编译器

- 用于 FPGA 的交叉编译器
- 稳健的调试环境有助于发现并解决代码问题
- 性能分析工具可确定瓶颈并最优化代码

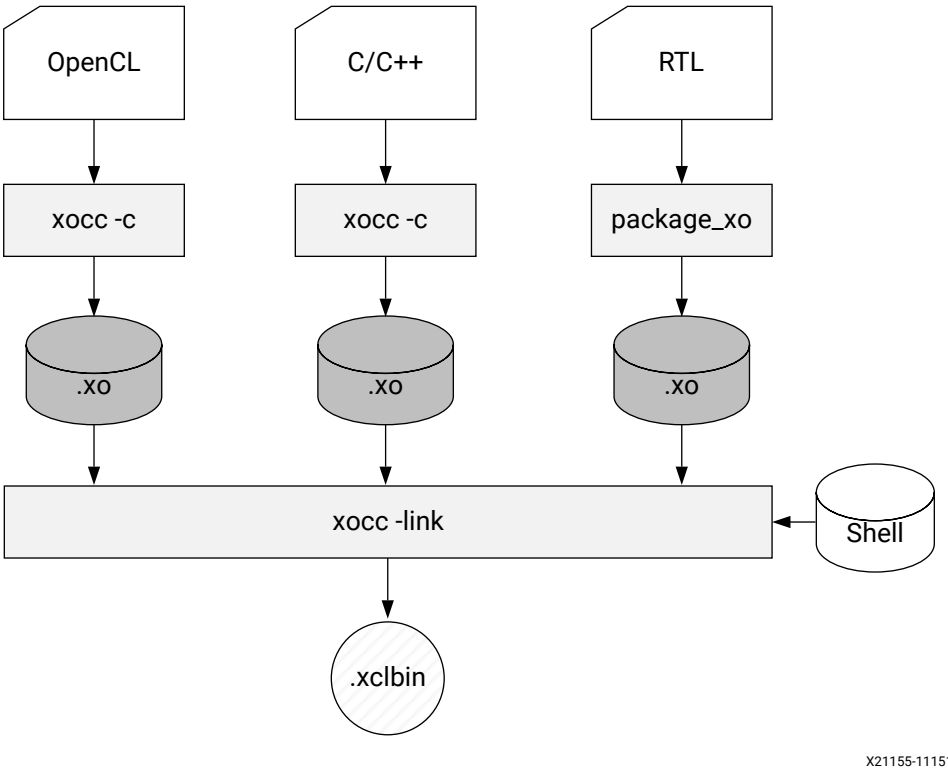
在此环境中，构建进程针对项目的软硬件元素使用的标准编译和链接进程。如下图所示，通过使用标准 GCC 编译器的流程构建主机应用，通过使用赛灵思 `xocc` 编译器的独立流程构建 FPGA 二进制文件。

图 2: **软件和硬件构建进程**



1. 使用 GCC 的主机应用构建进程：
   - 每个主机应用源文件都被编译至一个对象文件 (`.o`)。
   - 对象文件 (`.o`) 与赛灵思 SDAccel 运行时间 (runtime) 共享库链接来创建可执行程序 (`.exe`)。
2. FPGA 构建进程突出显示在下图中：
   - 每个内核都被独立地编译至赛灵思对象 (`.xo`) 文件。
     - 使用 `xocc` 编译器将 C/C++ 和 OpenCL C 内核编译用于 FPGA 上实现。该步骤利用 Vivado® HLS 编译器。由 Vivado HLS 支持的编译指示和属性可用于 C/C++ 和 OpenCL C 内核源代码，以指定想要的内核微架构并控制编译进程的结果。
     - 使用 `package_xo` 工具编译 RTL 内核。在 SDAccel 环境中的 RTL Kernel Wizard 可用于简化进程。
   - 内核 `.xo` 文件与硬件平台 (shell) 链接，以创建 FPGA 库 (`.xclbin`)。在链接步骤中决定重要的架构要素。特别是，这是内核端口与全局内存 bank 建立连接的地方，也是指定每个内核实例数量的地方。
     - 当构建目标为软件或硬件仿真时，如下所述，`xocc` 生成器件内容的仿真模型。
     - 当构建目标为系统（实际硬件）时，`xocc` 生成器件 FPGA 二进制文件，利用 Vivado Design Suite 运行综合和实现。

图 3: **FPGA 构建进程**



注释: `xocc` 编译器自动使用 Vivado HLS 和 Vivado Design Suite 工具构建内核，在 FPGA 平台上运行。它使用这些具有预定义设置、经过证实能够提供优质结果的工具。使用 SDAccel 环境和 `xocc` 编译器无需了解这些工具；但是，了解硬件的开发者可以充分利用这些工具，并使用他们全部的可用功能来实现内核。

构建目标

SDAccel 工具构建进程生成主机应用可执行程序 (`.exe`) 和 FPGA 二进制文件 (`.xclbin`)。SDAccel 构建目标定义了构建过程生成的 FPGA 二进制文件的性质。

SDAccel 工具提供三个不同的构建目标，其中两个仿真目标用于调试和验证，而默认的硬件目标则用于生成实际的 FPGA 二进制文件：

- 软件仿真 (`sw_emu`): 编译主机应用代码和内核代码在 x86 处理器上运行。这样允许通过快速构建和运行循环进行迭代算法精细化。此目标可用于确定语法错误，进行与应用程序一起运行的内核代码的源代码级调试，并验证系统行为。
- 硬件仿真 (`hw_emu`): 将内核代码编译至硬件模型 (RTL) 并在专用仿真器上运行。该构建和运行循环需要更长时间，但是提供详细的、周期准确的内核活动视图。此目标可用于检测 FPGA 内逻辑的功能正确性以及获取初始性能评估。
- 系统 (`hw`): 将内核代码编译至硬件模型 (RTL)，然后在 FPGA 器件上执行，得到的二进制文件将在实际 FPGA 上运行。

SDAccel 最优化流程的简介

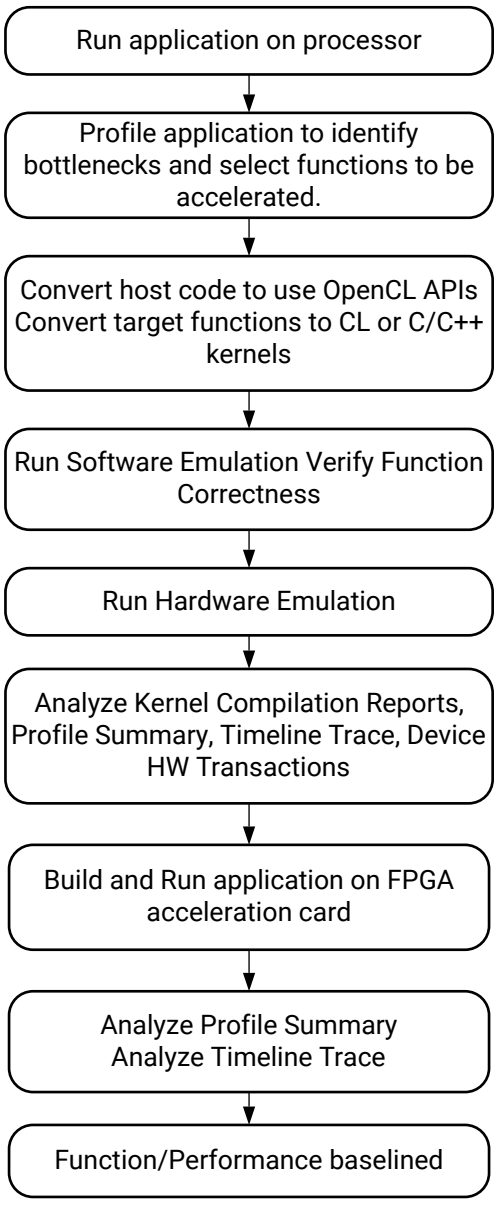
SDAccel 环境是创建、编译和最优化将在赛灵思 FPGA 上加速的 C/C++/OpenCL 应用程序的完整软件开发环境。SDAccel 环境包括用于最优化应用程序所推荐的三个流程。如需了解有关每个流程的信息，请参阅以下资料：

- 功能和性能基线检查
- 最优化数据移动
- 最优化内核计算

功能和性能基线检查

开始进行任何最优化之前，了解您应用的性能非常重要。这可从对应用进行功能和性能的基线检查来实现。

图 4: **功能和性能流程基线检查**



识别瓶颈

第一步是识别现有平台上正在运行的当前应用的瓶颈。最有效的方法是使用剖析工具运行应用，如 `valgrind`、`callgrind` 和 GNU `gprof`。这些工具生成的剖析数据可显示调用图形，包括所有函数的调用数量以及它们的执行时间。消耗最多执行时间的函数就是需要在 FPGA 上进行卸载和加速的合适候选对象。

转换目标函数

选择目标函数后，在不进行任何最优化的情况下将它们转换为 OpenCL C 内核或 C/C++ 内核。调用这些内核的应用代码也需要被转换，以便将 OpenCL API 用于数据移动和任务调度。

提示: 尽可能简化各个事项并对现有代码做最小限度的更改，以便能够在 FPGA 上快速生成工作设计，并获取基线性能和资源数量。

运行软件和硬件仿真

接下来，运行软件和硬件仿真来检查函数的正确性并生成有关主机代码和内核的剖析数据。分析内核编译报告、配置信息摘要、时间线追踪和器件硬件传输事务，以便了解基线性能估算，如时序、间隔以及时延和资源利用，如 DSP 和块 RAM。

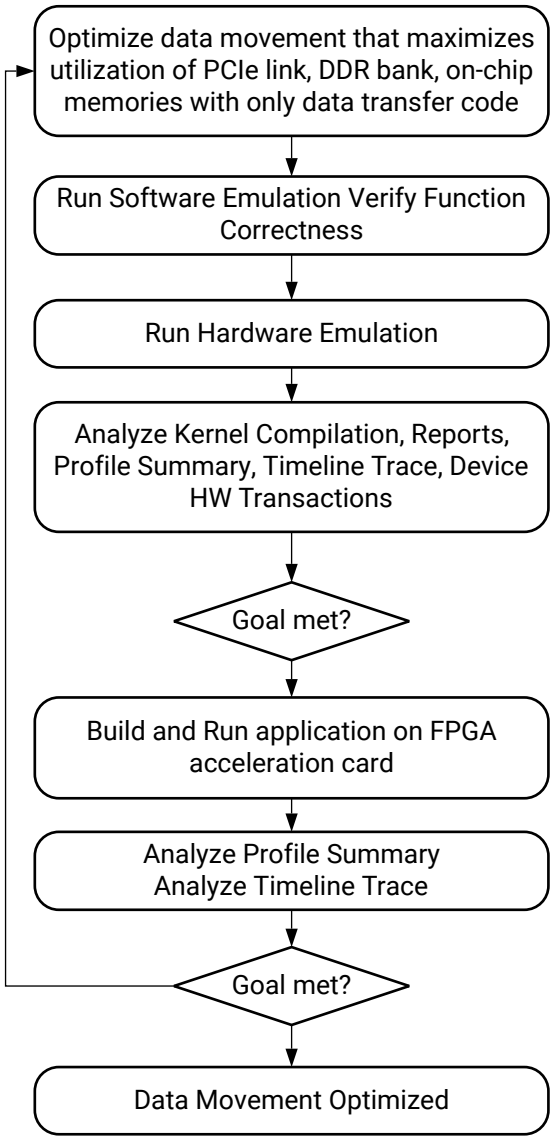
构建和运行应用

基线检查的最后一步是在 FPGA 加速卡上构建和运行应用。从系统编译分析报告并从应用执行分析剖析数据，以便了解实际性能和资源利用情况。

提示: 在基线检查过程中保存所有报告，这样您可以在最优化时参考和比较结果。

最优化数据移动

图 5: **最优化数据移动流程**



在 OpenCL API 中，所有数据首先从主机存储器传输到器件上的全局存储器，然后再从全局存储器传输到内核进行计算。计算结果从内核写入回全局存储器，然后再从全局存储器写入回主机存储器。确定内核最优化策略的关键因素是了解如何才能有效地移动数据。

建议: 在最优化计算之前，请先最优化应用中的数据移动。

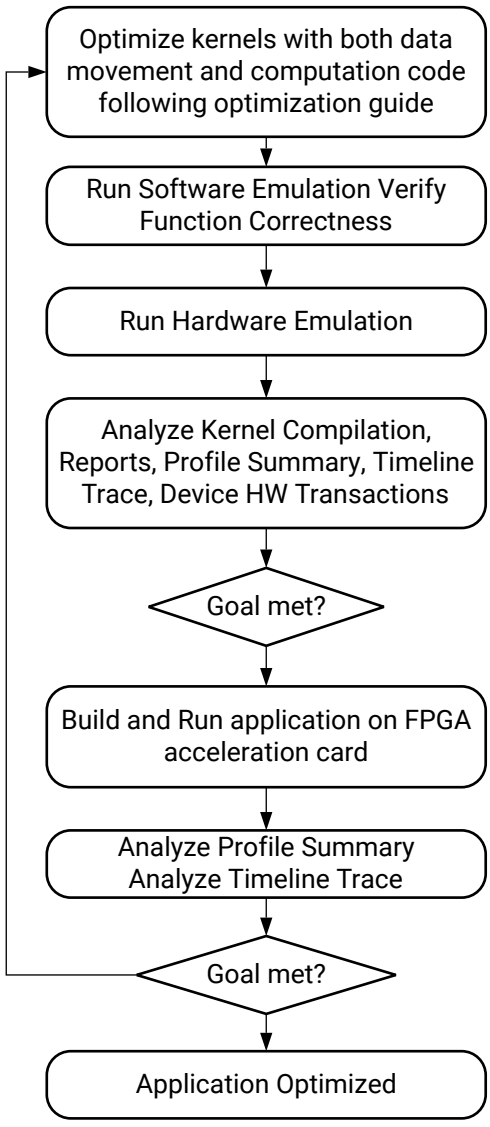
数据移动最优化过程中，将数据传输代码与计算代码隔离非常重要，因为计算效率低下可能会导致数据移动停止。

建议: 赛灵思推荐您仅针对此最优化步骤修改主机代码和具有数据传输代码的内核。

目的是通过最大化 PCIe 带宽使用和 DDR 带宽使用来最大化系统级数据吞吐量。要实现此目的，通常需要多个正在运行的软件仿真和硬件仿真迭代，以及在 FPGA 上执行。

最优化内核计算

图 6: **最优化内核计算流程**



FPGA 的其中一个主要好处是您可以为自己的特定应用创建定制逻辑。内核计算最优化的目标是创建其到达内核接口处时可尽快消耗所有数据的处理逻辑。此步骤中的关键指标是初始化间隔 (II)。这一般通过使用函数流水线化、循环展开、阵列分区、数据流动等技术展开处理代码以匹配技术数据路径来实现。SDAccel 环境在硬件仿真和系统运行过程中产生各种编译报告和剖析数据来帮助您完成最优化。如需了解有关编译和剖析报告的详情，请参阅 第 2 章: SDAccel 剖析和最优化的功能。

第 2 章

SDAccel 剖析和最优化的功能

SDAccel™ 环境在编译过程中生成有关内核资源和性能的各种报告。它还收集仿真模式中和 FPGA 加速卡上应用执行过程中的剖析信息。这些报告和剖析数据为您提供了有关应用中性能瓶颈的信息和可用于提升性能的最优化技术。本章将描述了如何在 SDAccel 环境中生成报告以及收集、显示和读取剖析结果。

系统估算

在 SDAccel 开发环境中，生成 FPGA 二进制文件是具有最长执行时间的步骤。执行时间受到 FPGA 架构和 FPGA 结构上放置的计算单元的数量影响最大。因此，在硬件上运行应用之前，通过更快的方式了解该应用的性能非常重要，这样您可以花费更多的时间来迭代和最优化您的应用，而无需等待 FPGA 编程文件来生成。

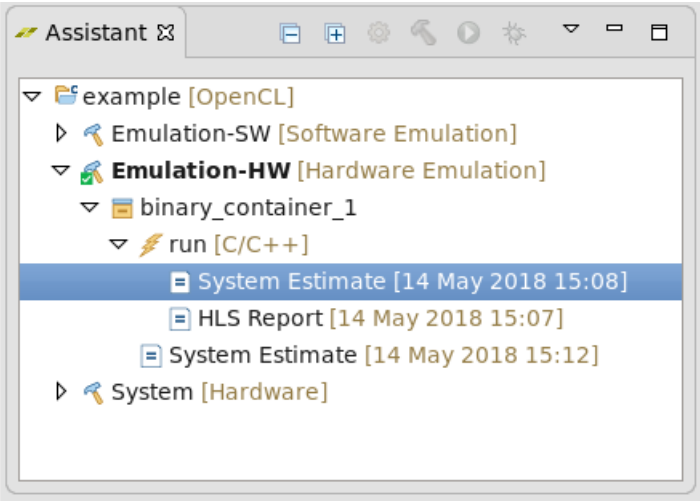
SDAccel 开发环境中的系统估算考虑了目标硬件器件和应用中的每个计算单元。虽然只能通过在 FPGA 上运行应用来准确测量性能指标，但是开发环境中的估算报告能够提供预期行为的精确表示。

GUI 流程

此报告在硬件仿真流程中自动生成。每个内核生成一个报告，整个二进制容器生成一个上层报告。从“Emulation-HW”文件夹内的“Assistant”视图可以方便地访问该报告。

下图显示了 `binary_container_1` 和具有名称为 `run` 的内核的系统估算报告的“Assistant”视图。

图 7: **“Assistant”视图中的“System Estimate”报告**



命令行

以下命令行为 `kernel.cl` 中的所有内核生成系统性能估算报告 `system_estimate.xtxt`：

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

由 `xocc -report estimate` 选项生成的性能估算报告提供了有关应用程序中每个二进制容器以及设计中每个计算单元的信息。该报告的结构如下：

- 目标器件信息
- 应用程序中每个内核的摘要
- 有关解决方案中每个二进制容器的详细信息

数据解读

以下示例报告文件代表了为估算报告生成的信息：

```
--------------------------------------------------------------------
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
--------------------------------------------------------------------
```

设计和目标器件摘要

所有设计估算报告均以应用总结和有关目标器件的信息开始。器件信息提供在报告的以下部分中：

```
--------------------------------------------------------------------
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
--------------------------------------------------------------------
```

对于设计概要，提供的唯一信息就是设计名称和目标器件的选择。本节中提供的其他信息还有目标电路板和时钟频率。

- Target Device: 运行该应用的电路板名称由 SDAccel 开发环境编译。
- Target Clock: 定义被映射到 FPGA 结构的计算单元的逻辑运行速度。这两个参数都由器件开发者决定。

这些参数不能从 SDAccel 环境内进行修改。

内核摘要

“Kernel Summary”章节列出了为当前 SDAccel 解决方案定义的所有内核。以下示例显示了内核摘要：

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

除了内核名称，该摘要还提供了执行目标和输入源的类型。由于 OpenCL™、C 和 C/C++ 源文件的编译和最优化方法存在差异，因此指定了内核源文件的类型。

“Kernel Summary”章节是报告中的最后摘要信息。此后，介绍了有关每个计算单元二进制容器的详细信息。

时序信息

对于每个二进制容器，详细部分从所有计算单元的执行目标开始。它还提供了每个计算单元的时序信息。作为通用规则，如果估算频率高于器件的目标频率，则计算单元将能够在该器件中运行。如果估算频率低于目标频率，则需要进一步最优化计算单元的内核代码，以便计算单元能够在 FPGA 结构上正确运行。此信息显示在以下示例中：

```
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
```

了解目标频率和估算频率之间的差别非常重要。计算单元并非被隔离布局到 FPGA 结构中。计算单元作为有效 FPGA 设计的一部分进行布局，该设计可包括由器件开发者定义的其他组件，以便支持各类应用。

由于一个内核一次生成一个计算单元定制逻辑，高于器件目标频率的估算频率向使用 SDAccel 环境的开发者表明，在 FPGA 编程文件创建过程中不存在任何时序问题。

时延信息

时延信息表示二进制容器中每个计算单元的执行配置信息。分析这些数据时，重要的是记住所有值都是通过定制逻辑从计算单元边界测量而得。与数据传输到全局存储器相关的系统内时延并未作为这些值的一部分进行报告。此外，所报告的时延数量仅用于以 FPGA 结构为目标的计算单元。以下是时延报告的示例：

```
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
```

时延报告分为以下字段：

- 起始间隔
- 最佳情况时延
- 平均情况时延
- 最坏情况时延

起始间隔定义了给定内核的计算单元各次调用之间必须经历的时钟周期数。

最佳、平均和最坏情况时延数量是指计算单元需要多少时间来生成一个内核的 ND 范围数据块结果。对于内核没有依赖数据的计算循环的情况，时延值将是相同的。依赖数据的循环的执行会导致数据特定的时延变化，该变化将被时延报告捕获。

间隔或时延数量将被报告为具有以下所列一个或多个条件的内核的“undef”：

- 没有明确 `reqd_work_group_size(x,y,z)` 的 OpenCL 内核
- 具有带变量绑定循环的内核

注释: 如果具有未定义的计数器，请考虑使用 TRIPCOUNT 编译指示。

注释: 时延信息根据对循环变换和已使用模型并行度的分析来反映估算值。这些高级变换，如流水线和数据流，可以极大改变实际吞吐量数值。因此，时延仅可用作不同运行之间的相关指南。

区域信息

每个 FPGA 中提供的基本构建块的数量是有限的。SDAccel 开发环境使用这些基本块（FF、LUT、DSP、块 RAM）来为设计中的每个计算单元生成定制逻辑。实现计算单元中定制逻辑所需的每种基本资源的数量将决定能够同时将多少个计算单元加载到 FPGA 结构中。以下示例显示了针对计算单元所报告的区域信息：

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

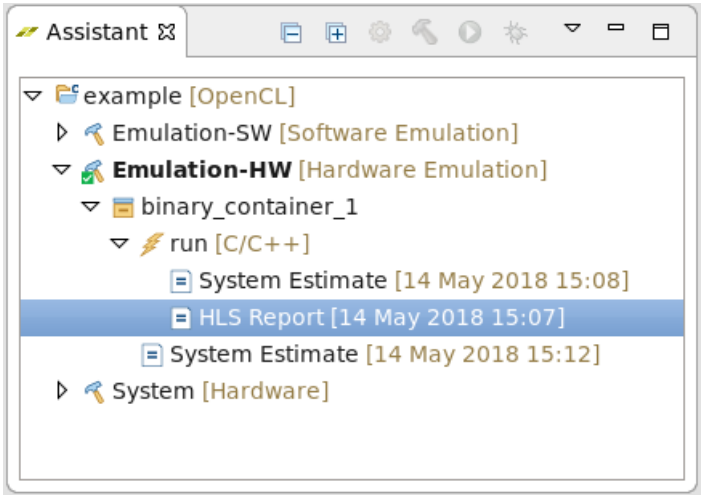
HLS 报告

在使用 SDx™ 开发环境 GUI 或 XOCC 命令行编译内核后，Vivado® 高层次综合 (HLS) 工具 HLS 报告即可使用。HLS 报告包括有关从用户内核代码中定制生成的硬件逻辑的性能和逻辑使用情况的详细信息。这些详细信息为高级用户提供了许多深入了解内核编译结果的机会，从而知道进行内核最优化。

GUI 流程

使用 SDx 环境 GUI 编译内核后，您可以在“Assistant”视图中查看 HLS 报告。该报告位于“Emulation-HW”或“System”构建配置下方，并具有 <binary container> 名称和 <kernel> 名称。这在以下“Assistant”视图中进行了说明：

图 8: **“Assistant”视图**



命令行

HLS 报告的设计目的要通过 SDAccel 环境 GUI 查看。但是，对于命令行用户，还发布了此报告的文本表述。可在位于 Vivado HLS 工具解决方案目录中的内核综合目录下查找此报告。

由于 `xocc` 命令在此综合目录之上生成了几个其他层级的目录，因此最好使用名称来定位该文件：

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

其中 <module> 是内核的名称。

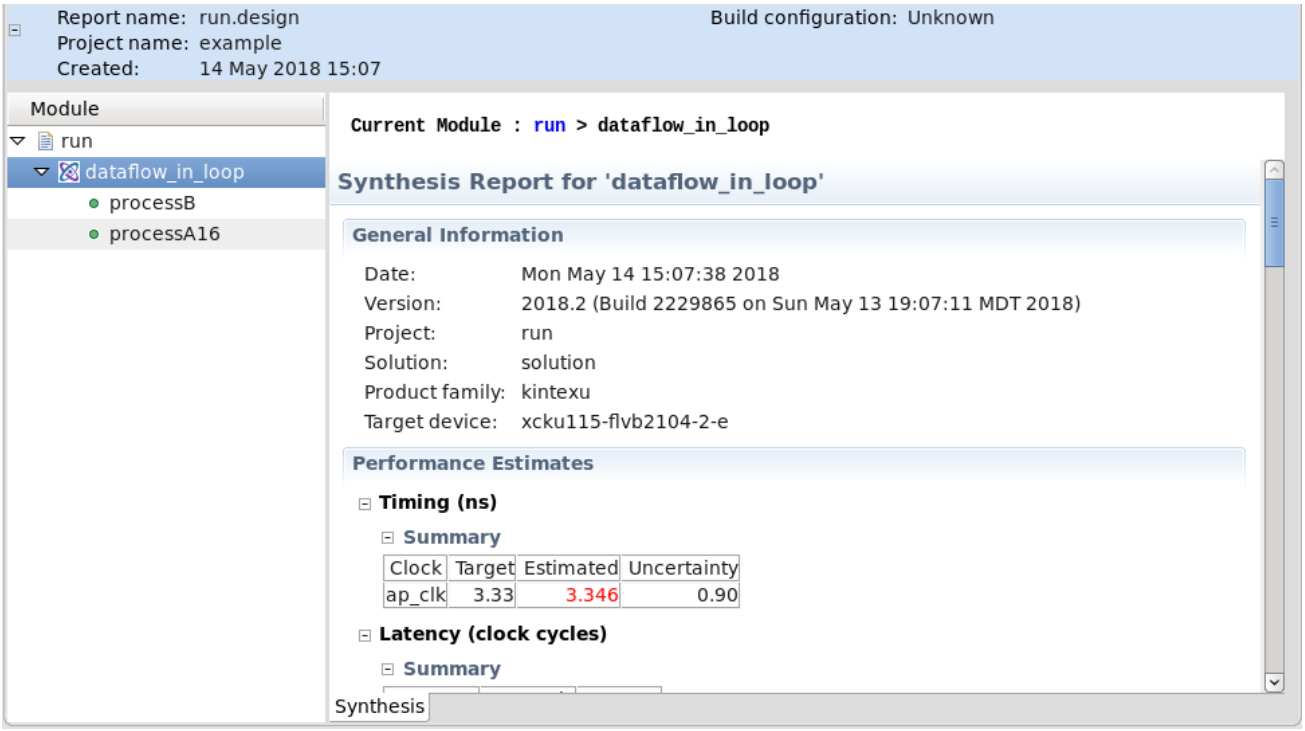
注释: 该 `find` 命令也支持使用通配符进行查找，因此以下命令将查找任何子目录中的所有综合报告：

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

数据解读

HLS 报告的左侧窗格显示模块层级。作为高层次综合运行一部分而生成的每个模块均显示在此层级中。您可以选择这些模块中的任何一个，以在“综合报告”窗口右侧显示该模块的综合详情。

图 9: **HLS 报告窗口**



“综合报告”被分为几个部分，即：

- “General Information”
- “Performance Estimates”（时序和时延）
- “Utilization Estimates”
- “Interface Information”

如果此信息是层级块的一部分，则它将汇总层级中包含的块的信息。鉴于此事实，当层级清楚表明实例对整体设计做出的贡献时，也可以从报告内部导航层级。

注意! 关于周期和时延的绝对计数，这些数字基于综合过程中识别到的估算，特别是在具有流水线和数据流等高级转换时，这些数字可能无法精确反映最终结果。如果您在报告中遇到问号，这可能是由于变量绑定循环造成，我们鼓励您为这些循环设置循环次数，以便能够在此报告中显示某些相关估算值。

配置信息摘要报告

SDAccel 环境运行时间自动收集主机应用上的剖析数据。应用完成执行后，配置信息摘要则以 HTM、CSV 和 Google Protocol Buffer 格式被保存到解决方案报告目录或工作目录中。这些报告可以在网页浏览器、电子数据表查看器或 SDAccel 环境中集成的“Profile Summary”视图中查看。配置信息报告在 SDAccel GUI 和 XOCC 命令行流程中生成。

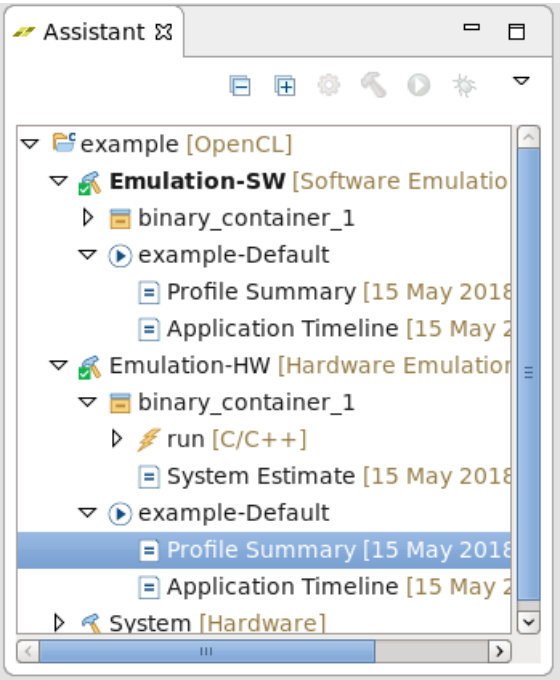
GUI 流程

当您从 SDAccel 环境编译和执行应用时，会自动生成配置信息摘要。

若要控制配置信息的生成，只需通过构建配置的上下文菜单来编辑运行配置，然后选择“Run → Run Configurations”。

配置运行后，“Assistant”视图允许从“Run Configuration”项下面方便地访问报告。运行配置执行完成后，现在可以通过“Assistant”视图中运行配置项的上下文菜单直接启动修改配置。

图 10: **SDAccel GUI 流程中的“Profile Summary”访问**



双击报告可打开它。

命令行

命令行用户在 SDAccel 环境外部执行独立的应用。要生成配置信息摘要数据，可以在不带任何额外选项的情况下编译您的设计。但是，链接比特流文件 (`xclbin`) 需要 `--profile_kernel` 选项。

通过 `--profile_kernel` 选项提供的实参可用于限制数据收集，在大型系统中可能需要进行数据收集。与配置信息摘要报告相关的 `profile_kernel` 选项的一般语法为：

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

可以指定以下三个字段来确定性能监控应用哪个接口：

- `kernel_name`
- `compute_unit_name`
- `interface_name`

但是，您也可以指定关键词 `all` 来将该监控应用于所有现存内核、计算单元和具有单选项的接口。最后一个选项 `<counters|all>` 可让您将信息收集限制在大型设计的 `counters`，而 `all`（默认）将包括实际追踪信息的收集。

注释: `profile_kernel` 选项是附加选项，可以在链接行上多次使用。

如果在 `sdaccel.ini` 文件中已指定 `profile = true` 选项，则当程序执行时将创建 `profile_summary.csv` 文件。

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

必须手动将 .csv 文件转换为 Google Protocol Buffer 格式 (.xprf) 才能在集成的“Profile Summary”视图中查看剖析结果。以下命令行示例将从 .csv 输入文件生成 .xprf 文件：

```
sdx_analyze profile profile_summary.csv
```

显示配置信息摘要

使用以下方法可显示从命令行创建的 SDAccel 环境“Profile Summary”视图。

1. 要在您选择的网页浏览器中显示该报告，请执行以下操作：
   a. 运行下列命令：

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

      该命令将创建一个 HTML 文件，呈现可被您选定的网页浏览器打开的数据。该文件包含的剖析结果与 GUI 流程 中呈现的结果相同。

   b. 导航到文件位置，然后双击生成的 HTML 文件。
2. 要在集成的 SDAccel “Profile Summary”视图中显示报告，请执行以下操作：
   a. 使用以下命令将 .csv 数据文件转换为 protobuf 格式。

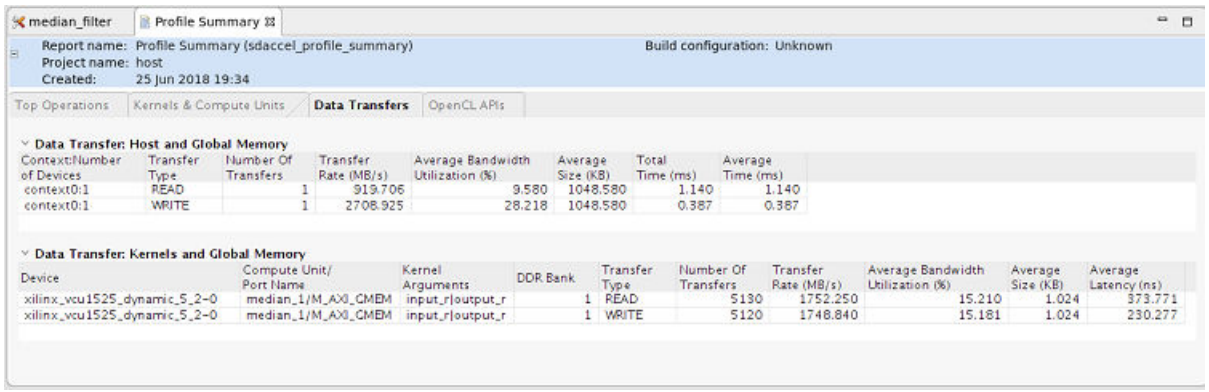
      ```
      sdx_analyze profile -i profile_summary.csv -f protobuf
      ```

   b. 运行 `sdx` 命令启动 SDAccel 工具 GUI：

      ```
      $sdx
      ```

   c. 在系统提示时选择默认的工作空间。
   d. 选择“File → Open File”。
   e. 浏览至并打开由步骤 a 中运行的 `sdx_analyze` 命令创建的 .xprf 文件。

      下图所示的“Profile Summary”视图显示了 OpenCL API 调用、内核执行、数据传输和配置文件规则检查 (PRC)。



数据解读

配置信息摘要包含 OpenCL 应用的大量有用统计数据。这些数据可为您提供您应用中功能瓶颈提供的总体概念。配置信息摘要由以下部分组成：

- “Top Operations”：
  - “Top Data Transfer: Kernels and Global Memory”：此表格显示 FPGA 和器件存储器之间顶层数据传输的配置信息数据。
  - “Device”：器件名
  - “Compute Unit”：计算单元名
  - “Number of Transfers”：在器件上监控的读写 AXI 传输事务的总和
  - “Average Bytes per Transfer”：（总读取字节数 + 总写入字节数）/（总读取 AXI 传输事务数 + 总写入 AXI 传输事务数）
  - “Transfer Efficiency (%)”：（每次传输的平均字节数）/分钟（4K，内存位宽/8 * 256））

  AXI4 规格将最大突发长度限制为 256 个字节，将最大突发量限制为 4K 字节。
  - “Total Data Transfer (MB)”：（总读取字节数 + 总写入字节数）/1.0e6
  - “Total Write (MB)”：（总写入字节数）/1.0e6
  - “Total Read (MB)”：（总读取入字节数）/1.0e6
  - “Transfer Rate (MB/s)”：（总数据传输）/（计算单元总时间）
  - “Top Kernel Execution”：
  - “Kernel Instance Address”：内核实例的主机地址（十六进制）
  - “Kernel”：内核名
  - “Context ID”：主机上的上下文 ID
  - “Command Queue ID”：主机上的命令队列 ID
  - “Device”：内核执行所在的器件的名称（格式：<器件>-<ID>）
  - “Start Time (ms)”：执行开始时间（毫秒）
  - “Duration (ms)”：执行持续时间（毫秒）
  - “Global Work Size”：内核的 NDRange
  - “Local Work Size”：内核的工作组大小
  - “Top Memory Writes: Host and Device Global Memory”：
  - “Buffer Address”：缓存的主机地址（十六进制）
  - “Context ID”：主机上的上下文 ID
  - “Command Queue ID”：主机上的命令队列 ID
  - “Start Time (ms)”：写入传输开始时间（毫秒）

- “Duration (ms)”：写入传输持续时间（毫秒）
- “Buffer Size (KB)”：写入传输大小（KB）
- “Writing Rate (MB/s)”：写入速率 = （缓存大小）/（持续时间）
- “Top Memory Reads: Host and Device Global Memory”：
- “Buffer Address”：缓存的主机地址（十六进制）
- “Context ID”：主机上的上下文 ID
- “Command Queue ID”：主机上的命令队列 ID
- “Start Time (ms)”：读取传输开始时间（毫秒）
- “Duration (ms)”：读取传输持续时间（毫秒）
- “Buffer Size (KB)”：读取传输大小（KB）
- “Reading Rate (MB/s)”：读取速率 = （缓存大小）/（持续时间）
- “Kernels & Compute Units”：
- “Kernel Execution (includes estimated device times)”：此表格显示了所有已计划和已执行的内核函数的配置信息数据摘要。
- “Kernel”：内核名
- “Number of Enqueues”：内核排队的次数
- “Total Time (ms)”：所有队列的运行时间的总和（从 OpenCL API 执行模型中的 START 至 END 进行测量）
- “Minimum Time (ms)”：所有队列的最小运行时间
- “Total Time (ms)”：所有队列的运行时间的总和（从 OpenCL API 执行模型中的 START 至 END 进行测量）
- “Average Time (ms)”：（总时间）/（队列数）
- “Maximum Time (ms)”：所有队列的最大运行时间
- “Compute Unit Utilization (includes estimated device times)”：此表格显示 FPGA. 上所有计算单元的摘要配置信息数据。
- “Device”：器件名称（格式：<器件>-<ID>）
- “Compute Unit”：计算单元名
- “Kernel”：与此计算单元关联的内核
- “Global Work Size”：内核的 NDRange（格式为 x:y:z）
- “Local Work Size”：本地工作组大小（格式为 x:y:z）
- “Number of Calls”：计算单元被调用的次数
- “Dataflow Execution”：表示是否启用顶层数据流执行
- “Maximum Overlapping Executions”：执行过程中的某一点有多少个执行实际正在并列运行
- “Dataflow Acceleration”：由于数据流加速带来的估计提升

- “Total Time (ms)”：所有调用的运行时间总和
- “Minimum Time (ms)”：所有调用的最小运行时间
- “Average Time (ms)”：（总时间）/（工作组数）
- “Maximum Time (ms)”：所有调用的最大运行时间
- “Clock Frequency (MHz)”：用于指定加速器的时钟频率 (MHz)
- “Data Transfers”：
- “Data Transfer: Host and Global Memory”：此表格显示主机和器件存储器之间通过 PCI Express® 链接进行的所有读写传输的配置信息数据。
- “Context:Number of Devices”：上下文 ID 和上下文中的器件数
- “Transfer Type”：READ 或 WRITE
- “Number of Transfers”：主机数据传输次数

注释: 可能包含 `printf` 传输

- “Transfer Rate (MB/s)”：（发送的总字节数）/（以微秒为单位的总时间），其中的总时间包括软件开销
- “Average Bandwidth Utilization (%)”：（传输速率）/（最大传输速率），其中最大传输速率 =（256/8 个字节）*（300 MHz）= 9.6 GBps
- “Average Size (KB)”：（发送的总 KB）/（传输次数）
- “Total Time (ms)”：传输次数总和
- “Average Time (ms)”：（总时间）/（传输次数）
- “Data Transfer: Kernels and Global Memory”：此表格显示 FPGA 和器件存储器之间进行的所有读写传输的配置信息数据。
- “Device”：器件名
- “Compute Unit/Port Name”：<计算单元名>/<端口名>
- “Kernel Arguments”：连接到此端口的实参的列表
- “DDR Bank”：此端口连接到的 DDR bank 编号
- “Transfer Type”：READ 或 WRITE
- “Number of Transfers”：在器件上监控的 AXI 传输事务数

注释: 可能包含 `printf` 传输)

- “Transfer Rate (MB/s)”：（发送的总字节数）/（计算单元总时间）
- “Compute Unit Total Time”：计算单元的总执行时间
- “Total Bytes Sent”：所有传输事务上的字节总和
- “Average Bandwidth Utilization (%)”：（传输速率）/（0.6 *最大传输速率），其中最大传输速率 =（512/8 个字节）*（300 MHz）= 19200 MBps
- “Average Size (KB)”：（发送的总 KB）/（AXI 传输事务次数）

- “Average Latency (ns)”：（所有传输事务的总时延）/（AXI 传输事务次数）
- “OpenCL API Calls”：此表格显示在主机应用中执行的所有 OpenCL 主机 API 函数调用的配置信息数据。
- “API Name”：API 函数的名称（例如，`clCreateProgramWithBinary`、`clEnqueueNDRangeKernel`）
- “Number of Calls”：此 API 的调用次数
- “Total Time (ms)”：所有调用的运行时间总和
- “Minimum Time (ms)”：所有调用的最小运行时间
- “Average Time (ms)”：（总时间）/（调用次数）
- “Maximum Time (ms)”：所有调用的最大运行时间

应用时间线

“Application Timeline”视图在共同的时间线上收集并显示主机和器件事件，以便帮助您了解和可视化各类系统的总体运行状况和性能。这些事件包括：

- 来自主机代码的 OpenCL API 调用。
- 器件追踪数据，包括 AXI 传输事务启动/停止、内核启动/停止等。

虽然对于应用的调试和剖析有用，但默认情况下并未收集时间线和器件追踪数据，因为运行时间需要定期从 FPGA 上传追踪数据，这样会给总体应用的执行增加额外的时间。但是，器件数据由 FPGA 内部的专用硬件收集，因此数据收集不会影响 FPGA 上的内核功能。以下各节说明了启用时间和器件数据收集所需的建立。

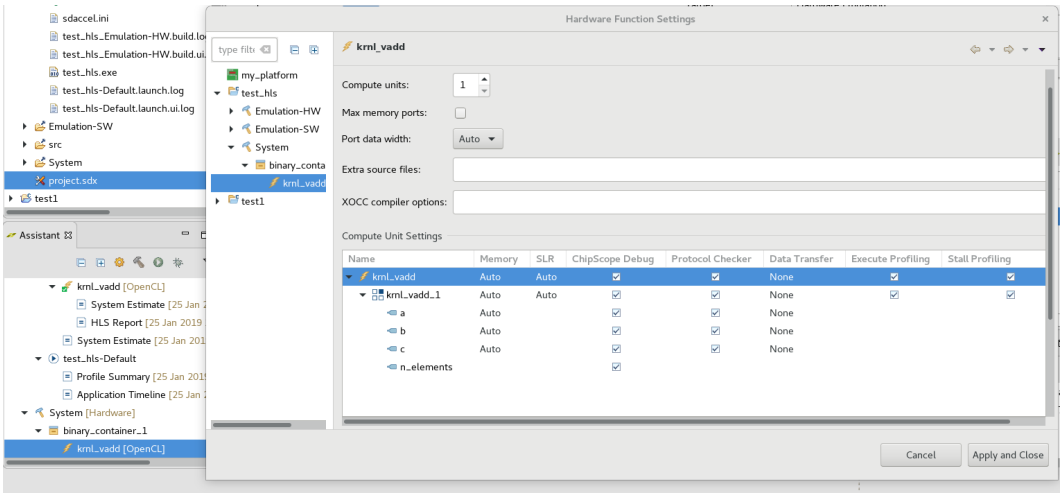
器件剖析的启用是侵入式的，可能会对总体性能产生负面影响。此功能应该只能用于系统性能调试。

注释: 器件剖析在“Emulation-HW”中使用时不会产生负面影响。

GUI 流程

时间线和器件追踪数据收集是从 SDAccel 集成环境创建的 SDAccel™ 项目的运行配置的一部分。请按照以下步骤来启用它：

1. 系统执行需要检测代码。这可以通过“Hardware Function Settings”对话框来完成。在“Assistant”视图中，右键单击“System [Hardware]”配置下的内核，然后选择“Settings Command”。



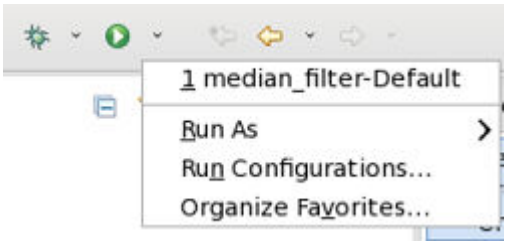
对于应用时间线功能，您可以启用“Data Transfer”、“Execute Profile”和“Stall Profiling”。这些选项检测任何内核的每个实例的所有端口。由于这些选项插入附加硬件，优化所有端口代价可能太大。为了实现这一目的，通过命令行提供了更多控件，如命令行一节中的详细说明。这些选项仅对系统运行有效。在硬件仿真过程中，此数据是默认生成的。

- “Data Transfer”：此选项可启用数据端口的监控。
- “Execute Profiling”：此选项在系统运行过程中提供最小端口数据收集。此选项记录计算单元的执行时间。执行剖析在进行数据和停滞剖析时被默认启用。
- “Stall Profiling”：此选项将停滞监控逻辑纳入到比特流中。

2. 规定运行过程中实际上要报告什么信息。

   注释: 仅报告系统执行过程中从硬件实际接触的信息。

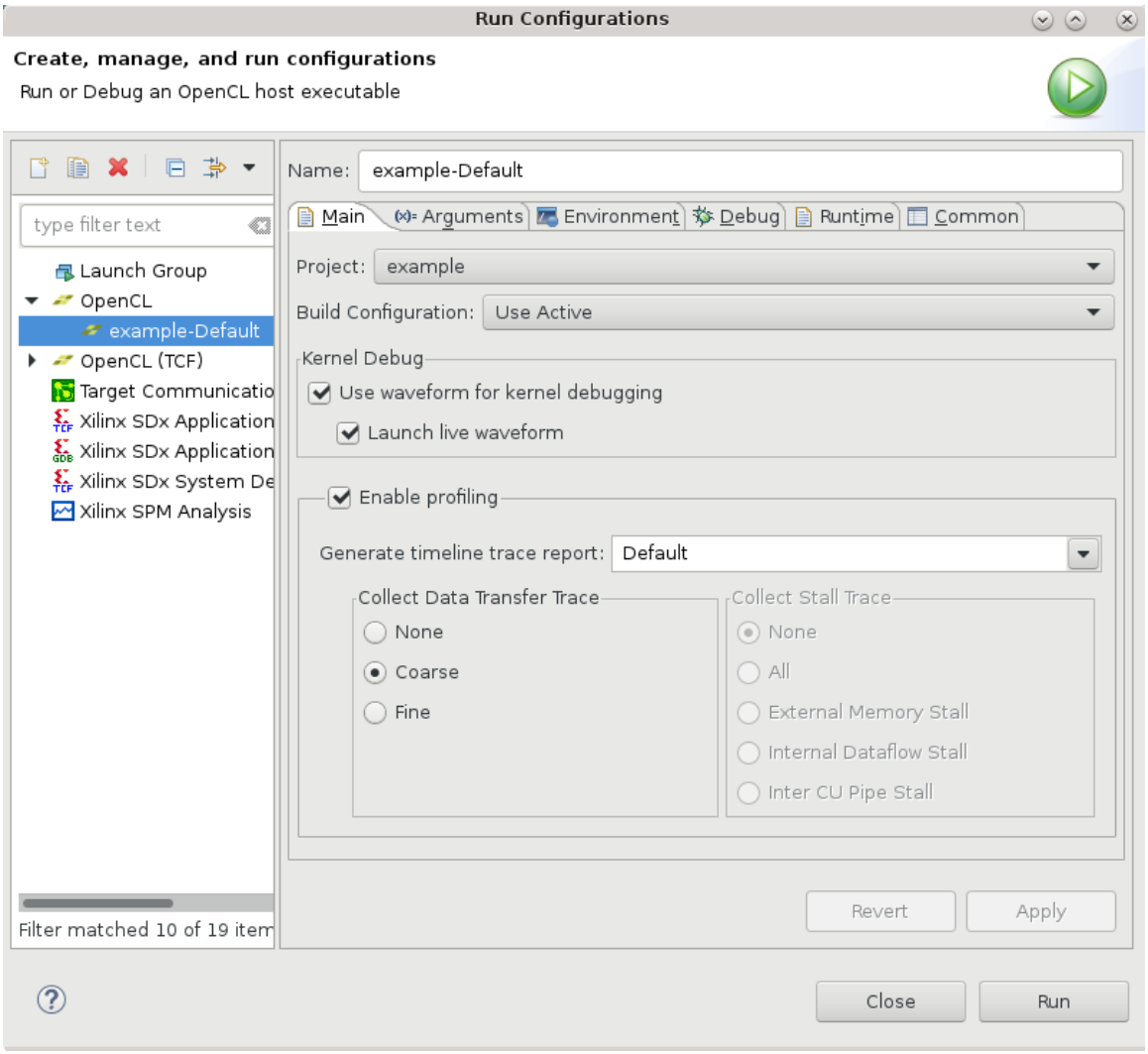
   要配置报告，单击“Debug”或“Run”按钮旁边的向下箭头，然后选择 “Run Configurations” 来打开“Run Configurations”窗口。



3. 在“Run Configurations”窗口中，单击“Profile”选项卡。

   确保已选中“Enable profiling”复选框。此复选框将启用基本的剖析支持。对于追踪数据，确保生成时间线追踪报告实际上收集您正在运行的构建配置中的信息。

   默认表示系统执行中不支持任何追踪数据捕获，但是在硬件仿真中被默认启用。



您还可以选择运行时间过程中要收集的信息量。分别为“Data Transfer Trace”和“Stall Trace”选择追踪数据收集的颗粒度。

“Data Transfer Trace”选项如下：

- “Coarse”：显示从第一个传输开始到最后一个传输结束（计算单元传输结束前）的计算单元传输活动。
- “Fine”：显示所有 AXI 级突发数据传输。
- “None”：关闭运行时间过程中器件级追踪的读取和报告。

“Stall Trace”选项如下：

- “None”：关闭任何停滞追踪信息收集。
- “All”：记录所有停滞追踪信息。
- “External Memory Stall”：存储器停滞到 DDR（例如，AXI4 从 DDR 读取）。
- “Internal Dataflow Stall”：内核之间流传输（例如，向数据流块之间的完整 FIFO 写入）。
- “Inter CU Pipe Stall”：内核内部管道（例如，向内核之间的完整 OpenCL™ 管道写入）。

如果同一个项目具有多个运行配置，您必须更改每个运行配置的配置信息设置。

4. 运行配置后，在“Assistant”视图中，双击“Application Timeline”打开“Application Timeline”视图。

命令行

完成以下步骤来启用命令行流程中的时间线和器件追踪数据收集：

1. 此步骤负责使用 SDx Accel 监视器（SAM）和 SDx 性能监视器（SPM）检测 FPGA 比特流。检测通过 `--profile_kernel` 执行，其中带有三个特别的检测选项（`data`、`stall` 和 `exec`）。

   注释: 除了用于系统编译和链接，请忽略 `--profile_kernel` 选项。在硬件仿真过程中，默认生成此数据。

   `--profile_kernel` 选项具有三个必要字段，用于确定监视器应用的具体内核接口。但是，如果资源使用不成问题，关键词 `all` 使您可以将监控应用于所有现有内核、计算单元和具有单一选项的接口。否则，您可以明确指定 `kernel_name`、`compute_unit_name` 和 `interface_name` 来限制检测。最后一个选项 `<counters|all>` 可使您将信息收集仅限制在大型设计的 `counters`，而 `all`（默认）包括实际追踪信息的收集。

   注释: `--profile_kernel` 选项是附加选项，可以在链接行上多次使用。

   - “`data`”：此选项可通过 SAM 和 SPM IP 来启用数据端口的监控。此选项仅需要在连接过程中设置。

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

   - “`stall`”：此选项需要在编译过程中应用：

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

     且此选项仅需要在链接过程中应用：

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

     此选项包括比特流中的停滞监控逻辑（使用 SAM IP）。但是，它确实需要停滞端口出现在内核接口上。为促成此事，需要该选项编译 C/C++/OpenCL API 内核模块。

   - “`exec`”：此选项提供系统运行过程中的最小端口数据收集。它只通过使用 SAM IP 记录内核的执行次数。此功能默认在使用数据或停滞数据收集的任何端口上启用。此选项仅需在链接过程中提供。

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

2. 内核经过检测后，必须在运行时间执行过程中启用数据收集。通过使用与主机可执行程序在同一目录中的 `sdaccel.ini` 文件来进行此操作。以下 `sdaccel.ini` 文件将在运行时间过程中启用最大信息收集：

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

   - `profile=<true|false>`：当此选项被指定为真时，即启用基本配置信息监控。没有其他任何额外选项时，此设置表示已启用主机运行时间记录配置信息摘要功能。但是，在不启用此选项的情况下，则根本不会执行任何监控。
   - `timeline_trace=<true|false>`：此选项将启用数据的时间线追踪信息收集。在不向 FPGA（数据）中添加配置信息 IP 的情况下，它将只显示主机信息。作为最低要求，要在时间线追踪中获得更多计算单元启动和结束的执行次数，计算单元需与 `--profile_kernel exec` 链接。
   - `data_transfer_trace=<coarse|fine|off>`：此选项可启用器件级 AXI 数据传输追踪：
     - `coarse`：显示从第一个传输开始到最后一个传输结束（在计算单元传输结束前）的计算单元传输活动。

- fine：显示所有 AXI 级突发数据传输。
- off：关闭运行时间过程中器件级追踪的读取和报告。

- stall_trace=<dataflow|memory|pipe|all|off>：规定时间线追踪内要捕获和报告的停滞类型。默认值为 off。
  - off：关闭任何停滞追踪信息收集。

    注释: 启用停滞追踪会经常填充追踪缓存，这样会导致时间线追踪不完整或可能损坏。可以通过设置 trace_stall=off 来避免这种情况。
  - all：记录所有停滞追踪信息。
  - dataflow：内核内流传输（例如，向数据流块之间的完整 FIFO 写入）。
  - memory：外部存储器停滞（例如，从 DDR 中进行 AXI4 读取）。
  - pipe：内核之间管道（例如，向内核之间的完整 OpenCL 管道写入）。

3. 要在“Application Timeline”中允许用户函数分析，必须作为仿真流程的一部分运行波形工具。这就必须通过 sdaccel.ini 文件来启动波形工具。

```
[Emulation]
launch_waveform=batch
```

- launch_waveform=<gui|batch>：此选项可在仿真过程中自动启动波形工具。如需了解更多详情，请参阅“Waveform”视图。
  - gui：启动实时波形视图的图形用户界面
  - batch：作为后台进程启动波形处理。

4. 在命令行模式中，生成 CSV 文件来捕获追踪数据。需使用 sdx_analyze 实用工具将这些 CSV 报告转换为“Application Timeline”格式后，才能在 SDAccel 环境 GUI 中打开和显示它们。

```
sdx_analyze trace timeline_trace.csv -k timeline_kernels.csv -f wdb
```

此操作将默认创建可从 GUI 中打开的 timeline_trace.wdb 文件。timeline_kernels.csv 文件包含具体的内核追踪数据，这些数据可能并非始终可用。在此情况下，应从 sdx_analyze 命令中忽略选项 -k timeline_kernels.csv。

5. 要查看时间线报告主机和器件波形，请执行以下操作：
   a. 运行以下命令，启动 SDx 环境：

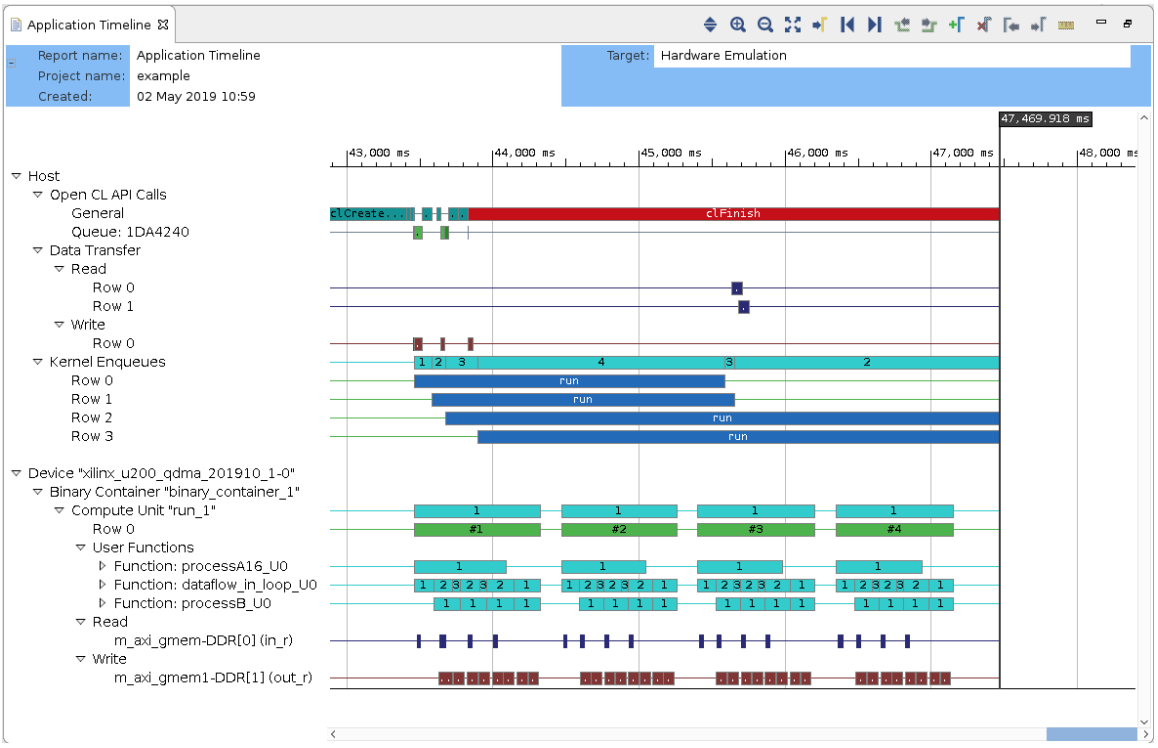
```
$sdx
```

   b. 在出现提示时选择工作空间。
   c. 选择“File → Open File”，浏览到在硬件仿真或系统运行过程中生成的 .wdb 文件，并打开它。

数据解读

下图显示的是可以在共同时间线上显示主机和器件事件的“Application Timeline”视图。此信息可帮助您了解应用执行的详情，并确定可进行提升的潜在领域。

图 11: **“Application Timeline”视图**



“Application Timeline”视图追踪包括两个主要部分：

- “Host”：显示源自主机一侧的所有活动的追踪。
- “Device”：显示 FPGA 上计算单元的活动。

在主机下，不同的活动可分类为 OpenCL™ API 调用、数据传输和内核。

完整的树具有以下结构：

- “Host”：
  - “OpenCL API Calls”：在此追踪所有 OpenCL API 调用。从主机透视图角度测量活动时间。
  - “General”：在此追踪所有常规 OpenCL API 调用，例如 `clCreateProgramWithBinary()`、`clCreateContext()` 和 `clCreateCommandQueue`。
  - “Queue”：在此追踪与特定命令队列相关联的 OpenCL API 调用。其中包括命令，例如 `clEnqueueMigrateMemObjects`、 `clEnqueueNDRangeKernel` 等。如果用户应用创建多个命令队列，则此部分将显示尽可能多的队列和活动。
  - “Data Transfer”：在此部分中，追踪 DMA 传输。从主机到器件的数据传输显示在“Write”下方，从器件到主机的传输显示在“Read”下方。另一部分“Copy”追踪内核之间的直接通信。
  - “Kernel Enqueues”：活动的内核执行显示在此处。此处的内核不能与器件上的内核/计算单元混淆。在此实例中，内核是指 NDRangeKernels 和由 `clEnqueueNDRangeKernels()` 和 `clEnqueueTask()`API 创建的任务，这些内核针对从主机角度测量而得的时间进行绘制。可以安排多个内核同时执行，它们的追踪将从安排运行的点开始直至内核执行结束。这就是存在多个输入的原因。行数取决于重叠的内核执行数。

注释: 内核的重叠不应错误地被认为是器件上实际真实的并行执行，因为进程可能并未做好实际立即执行的准备。

- “Device "name"”：
  - “Binary Container "name"”：
  - “Accelerator "name"”：这是 FPGA 上计算单元（又称为加速器）的名称。
  - “User Functions”：在 Vivado HLS 工具内核情况下，在此追踪作为数据流进程执行的函数。这些函数的追踪显示了这些当前正在并行执行的函数的活动实例数。当启用波形时，这些名称在硬件仿真中生成。

注释: 函数级活动仅在硬件仿真中才可能出现。

  - “Function: "name a"”
  - “Function: "name b"”
  - “Read”：计算单元通过 AXI-MM 端口从 DDR 中读取。此处显示由计算单元读取的数据的追踪。活动显示为传输事务且每个传输事务的工具提示显示 AXI 传输事务的更多详情。当使用 `--profile_kernel data` 时生成这些名称。
  - “m_axi_<bundle name>(port)”
  - “Write”：计算单元通过 AXI-MM 端口写入 DDR。此处显示由计算单元写入的数据的追踪。活动显示为传输事务且每个传输事务的工具提示显示 AXI 传输事务的更多详情。当使用 `--profile_kernel data` 时生成此名称。
  - “m_axi_<bundle name>(port)”

“Waveform”视图

SDx 开发环境在运行硬件仿真时可生成波形视图和启动实时波形查看器。它可在系统级、计算单元级和函数级上深入详细地显示仿真结果。详情包括内核和全局存储器之间的数据传输、内核之间管道通过的数据流以及内核内部管道通过的数据流。它们从系统级到单个函数调用上提供了对性能瓶颈的深入了解，以帮助开发者最优化他们的应用。

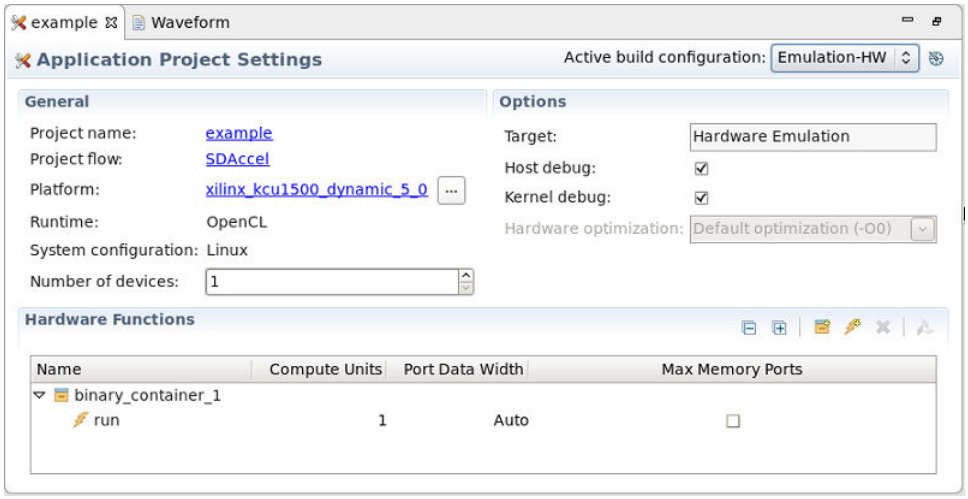
默认情况下，波形视图和实时波形查看器未启用。这是因为这些视图要求运行时间在硬件仿真过程中生成仿真波形，这会消耗更多时间和磁盘空间。以下各节说明了启用数据收集所需的建立。

注释: 波形视图允许您从 SDx 开发环境内部直接查看器件传输事务。相反，实时波形能力实际上可生成能够可视化硬件传输事务和用户选择的潜在内部信号的仿真波形视图。

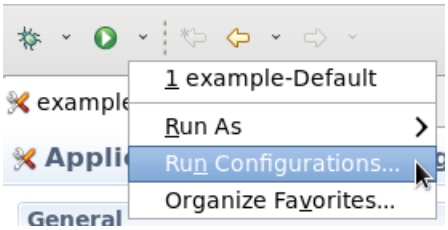
GUI 流程

按照以下步骤来启用波形数据收集并打开视图：

1. 打开“Application Project Settings”窗口，并选择“Kernel debug”复选框。



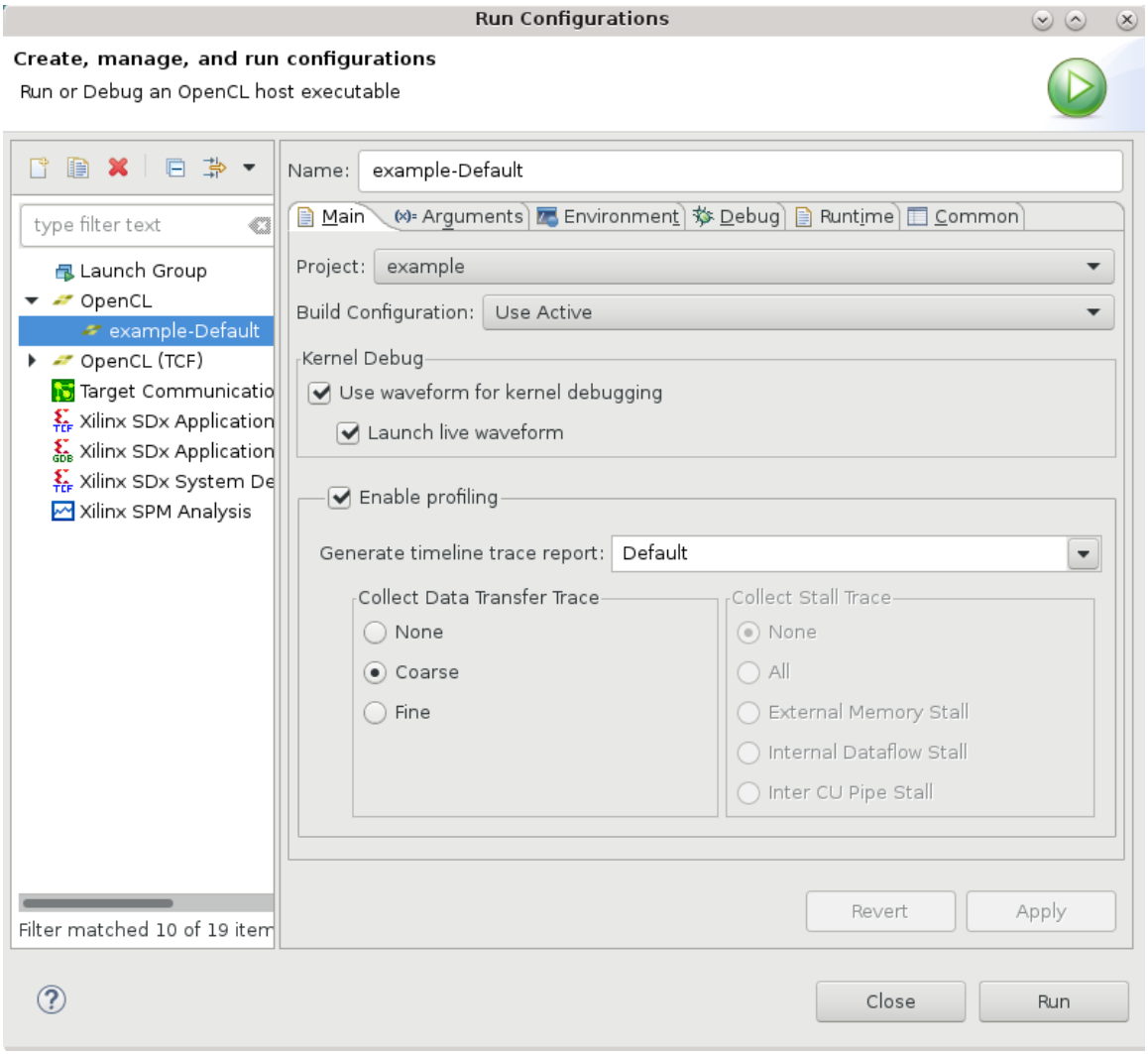
2. 单击“Run”按钮旁边的向下箭头，然后选择“Run Configurations”打开“Run Configurations”窗口。



3. 在“Run Configurations”窗口上，单击“Main”，然后选择“Use waveform for kernel debugging”复选框。

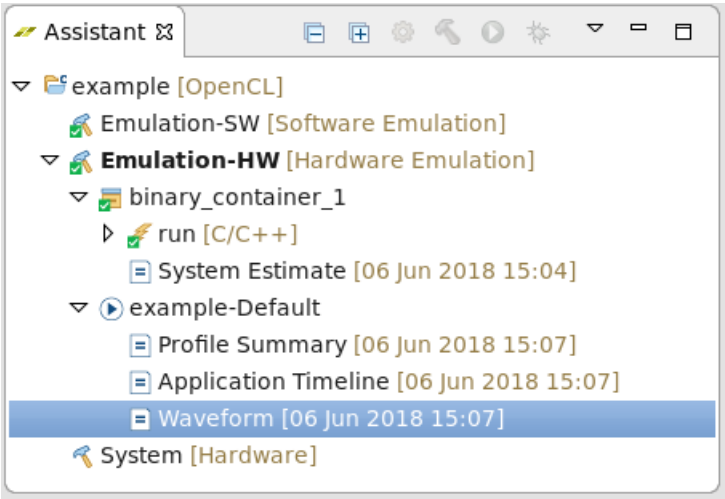
   可选：

   - 当硬件仿真正在运行时，要显示“Simulation”窗口来查看实时波形，请取消选中“Launch live waveform”。
   - 要启用基本剖析，请选择“Enable profiling”。



4. 如果同一个项目具有多个运行配置，请更改每个运行配置的配置信息设置。
5. 如果您尚未选择要自动启动的实时波形查看器，请从 SDx 开发环境打开“Waveform”视图。

   在 SDx 开发环境中，双击“Assistant”窗口中的“Waveform”可打开“Waveform”视图窗口。



命令行

在硬件仿真过程中，从命令行使用以下指令来启用波形数据收集，然后打开查看器：

1. 在内核编译过程中启动调试代码生成。

```
xocc -g -t hw_emu ...
```

2. 在与主机可执行相同的目录中创建 `sdaccel.ini` 文件，其内容如下：

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

这样可以启用最大的可观察性。详细的选项为：

- “profile=<true|false>”：将此选项设置为真，可启用配置信息监控功能。在不执行其他任何额外选项的情况下，这表示已启用主机运行时间记录配置信息摘要功能。但是，在不启用此选项的情况下，则不会执行任何监控。
- “timeline_trace=<true|false>”：此选项将启用数据的时间线追踪信息收集。

3. 要查看实时波形和其他模拟波形，请将以下内容添加到 `sdaccel.ini` 中的仿真部分：

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

- launch_waveform=<batch|gui>: `gui` 选项可启用实时波形查看器，而 `batch` 选项将记录波形活动情况以便用于后处理。

实时波形查看器在硬件仿真执行过程中生成，这样可以让您仔细查看波形的详情。

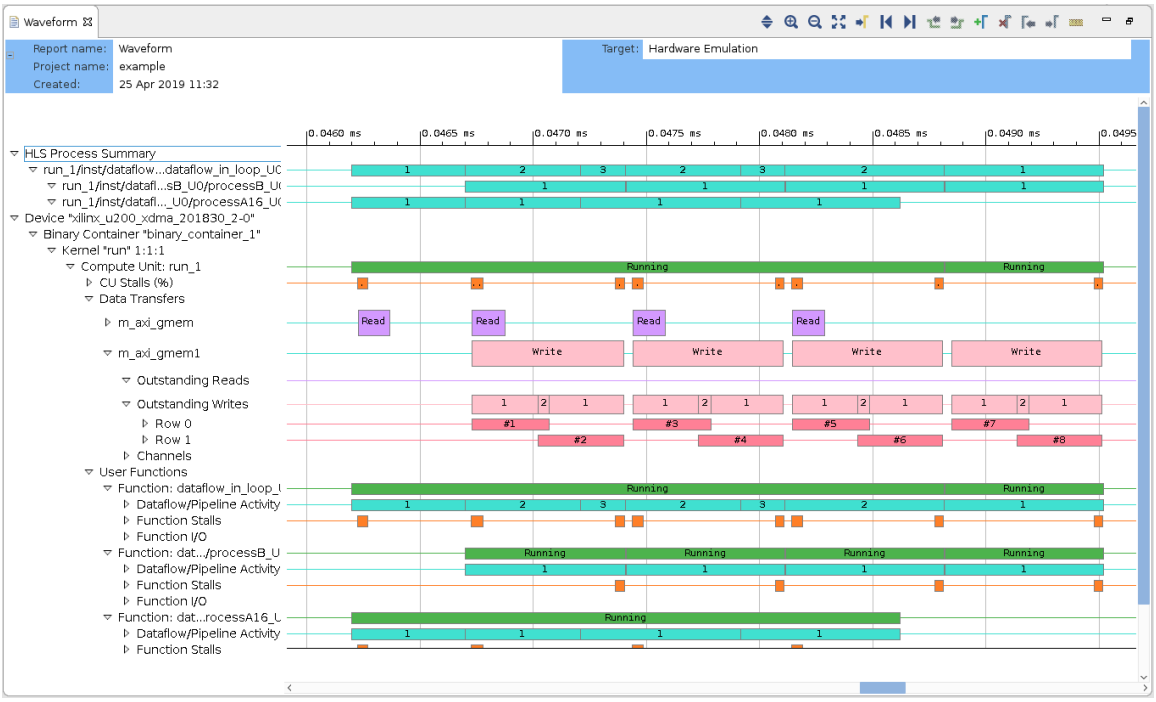
4. 执行硬件仿真。硬件数据传输数据已被收集到文件 `<hardware_platform>-<device_id>-<xclbin_name>.wdb` 中。
5. 如果未请求实时波形查看器，请按照以下步骤来打开“Waveform”视图：
   a. 运行以下命令：`$sdx`，启动 SDx IDE。
   b. 在系统提示时选择工作空间。
   c. 选择“File → Open File”，浏览到硬件仿真过程中生成的 .wdb 文件。

或者，可使用以下命令：`xsim --gui <file>.wdb`，用 xsim 打开 .wdb 文件。如需了解有关 xsim 的详情，请参阅《Vivado Design Suite 用户指南：逻辑仿真》 (UG900)。

“Waveform”视图的数据解读

以下图像显示的是“Waveform”视图：

图 12: **“Waveform”视图**



“Waveform”视图按层级组织，以方便导航。

注释: 此视图基于硬件仿真过程中生成的实际波形（内核追踪）。这样可使此视图降序排列直至产生抽象数据的单个信号。但是，由于它对数据进行后处理，因此无法添加其他信号，而且 DATAFLOW 传输事务等某些运行时间分析无法被可视化。

层级树和描述为：

- “HLS Process Summary”：此摘要部分包含在生成的 RTL 中所含的每个顺序进程的活动报告的层级显示。可视化 HLS 设计中的活动进程可以详细配置每个顶层模块内部哪些进程处于活动状态及其活动时间长度。因此，此视图可以针对单个进程性能以及各个独立进程的总体并行执行进行分析。根据阿姆达尔 (Amdahl) 定律，如果可以缩短进程执行时间，则主导总体执行的进程具有最大的性能提升潜力。
- “Device “name””：目标器件名。
  - “Binary Container “name””：二进制容器名称。
  - “Kernel “name” 1:1:1”：对于每个内核和该内核的每个计算单元，此部分细分了源自计算单元的各种活动。
  - “Compute Unit: “name””：计算单元名。

- “CU Stalls (%)”：当由于外部存储器访问、内部流（即数据流）或外部流（即 OpenCL 管道）等原因造成信号电路的一部分停滞时，HLS 工具将提供停滞信号来提醒您。详细内核追踪中显示的停滞总线将编译所有最低值停滞信号，并报告在任意时间点停滞的百分比。这样可以提供仿真中任意点上停滞内核的数量因子。

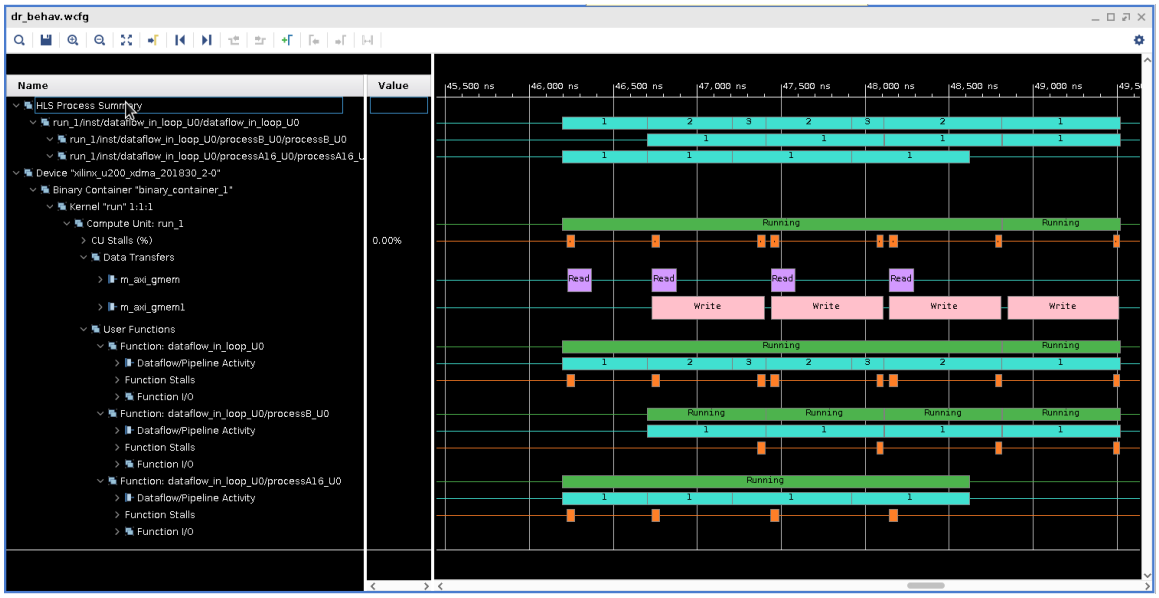
例如，如果有 100 个最低值停滞信号，且在给定的时钟周期内有 10 个活动信号，则 CU 停滞百分比为 10%，如果又有一个变为非活动状态，则停滞百分比为 9%。

- “Data Transfers”：可显示从每个计算单元主 AXI 端口到 DDR 发起的读取/写入数据传输访问数。
- “User Functions”：此信息可用于 HLS 工具内核，并可显示用户函数。
- “Function: "name"”：
- “Dataflow/Pipeline Activity”：如果函数作为数据流进程执行，此信息可显示函数的并行执行数。
- “Active Iterations”：此信息可显示数据流的当前活动迭代数。行数动态增加以适应任何并发执行的可视化。
- “StallNoContinue”：这是一个停滞信号，表示数据流进程是否遇到任何输出停滞（函数已执行，但是为能从相邻的数据流进程接收到继续执行信号）。
- “RTL Signals”：这些是底层 RTL 控制信号，用于解读上面的数据流进程传输事务视图。
- “Function Stalls”：显示进程遇到的不同类型停滞。
- “External Memory”：访问 DDR 存储器时出现停滞。
- “External Stream”：流传输触发的停滞。
- “Function I/O”：根据其相关联的块接口进行分组的实际接口信号。
- “Function: "name"”：函数名。
- “Function: "name"”：函数名。

实时波形数据

下图显示的是硬件仿真运行时的实时波形查看器。

图 13: **实时波形查看器**



实时波形查看器按层级组织，以方便导航。以下是层级树和描述。

注释: 由于实时波形查看器仅作为实际硬件仿真运行 (xsim) 的一部分来显示，您可以将其他信号和寄存器传输 (RTL) 的内部注释到同一个视图中。同时，可以将所有已分类和组合的组完全展开到实际汇入信号。如需了解更多有关使用 xsim 的信息，请参阅《Vivado Design Suite 用户指南：逻辑仿真》 (UG900)。

- “HLS Process Summary”：此摘要部分包含每个顺序进程的活动报告的层级显示，顺序进程包含在生成的 RTL 中。可视化 HLS 设计中的活动进程可以详细配置每个顶层模块内部哪些进程处于活动状态及其活动时间长度。因此，此视图可以针对单个进程性能以及各个独立进程的总体并发执行进行分析。根据阿姆达尔 (Amdahl) 定律，如果可以缩短进程执行时间，则主导总体执行的进程具有最大的性能提升潜力。
- “Device "name"”：目标器件名。
  - “Binary Container "name"”：二进制容器名称。
  - “Kernel "name" 1:1:1”：对于每个内核和该内核的每个计算单元，此部分细分了源自计算单元的各种活动。
  - “Compute Unit: "name"”：计算单元名。
  - “CU Stalls (%)”：当由于外部存储器访问、内部流（即数据流）或外部流（即 OpenCL™ 管道）等原因造成信号电路的一部分停滞时，Vivado HLS 工具将提供停滞信号来提醒您。详细内核追踪中显示的停滞总线将编译所有最低值停滞信号，并报告在任意时间点停滞的百分比。这样可以提供仿真中任意点上停滞内核的数量因子。

  例如：如果有 100 个最低值停滞信号，且在给定的时钟周期内有 10 个活动信号，则 CU 停滞百分比为 10%，如果又有一个变为非活动状态，则停滞百分比为 9%。

  - “Data Transfers”：可显示从每个计算单元主 AXI 端口到 DDR 发起的读取/写入数据传输访问数。
  - “User Functions”：此信息可用于 HLS 内核，并可显示用户函数。
  - “Function: "name"”：
  - “Dataflow/Pipeline Activity”：如果函数作为数据流进程执行，此信息可显示函数的并行执行数
  - “Active Iterations”：此信息可显示数据流的当前活动迭代数。行数动态增加以适应任何并行执行的可视化。

- “StallNoContinue”：这是一个停滞信号，表示数据流进程是否遇到任何输出停滞（函数已执行，但是为能从相邻的数据流进程接收到继续执行信号）。
- “RTL Signals”：这些是底层 RTL 控制信号，用于解读上面的数据流进程传输事务视图。
- “Function Stalls”：显示进程遇到的不同类型停滞。
- “External Memory”：访问 DDR 存储器时出现停滞。
- “External Stream”：流传输触发的停滞。
- “Function I/O”：根据其相关联的块接口进行分组的实际接口信号。
- “Function: "name"”：函数名。
- “Function: "name"”：函数名。

“Guidance”视图

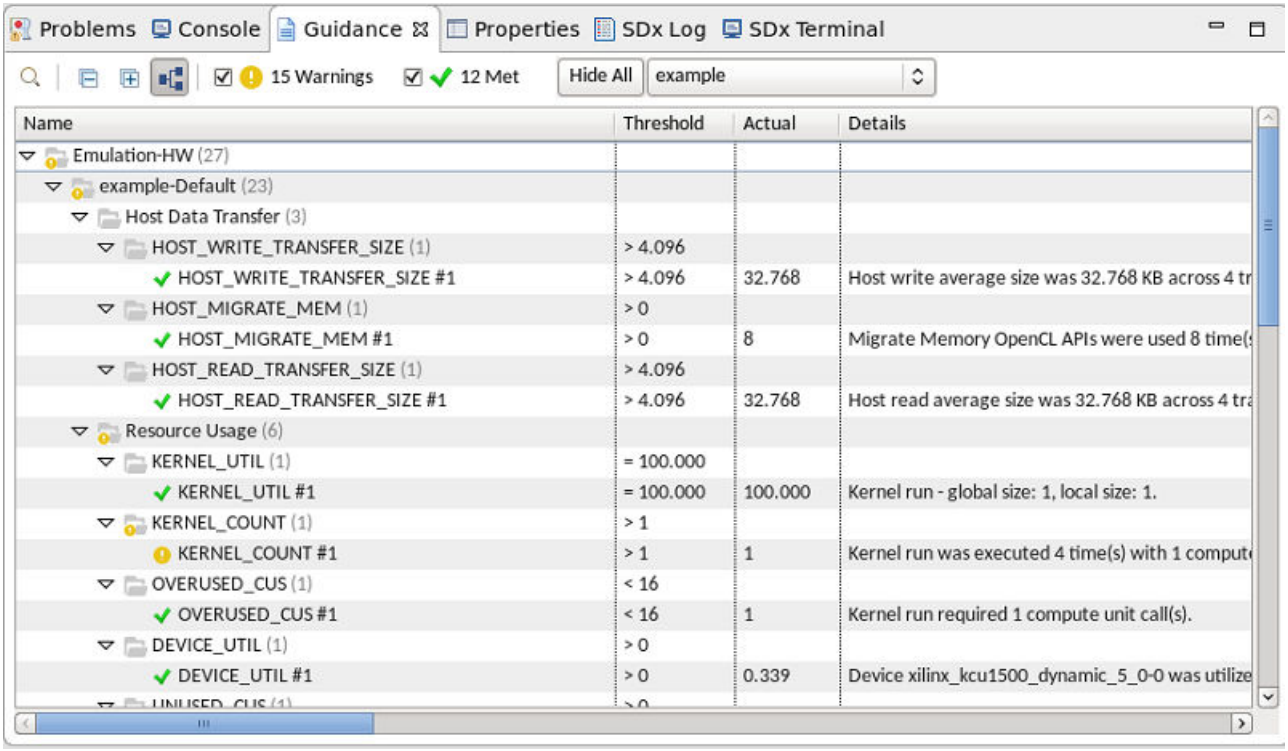
“Guidance”视图的设计目的是为您提供整个开发进程中的反馈。它在一个位置展示从构建实际设计到运行时间分析过程中出现的所有问题。

“Guidance”视图的目的是帮助您识别设计中的潜在问题，理解这一点至关重要。这些问题可能是源代码相关的问题，也可能是由于缺少工具最优化造成的问题。此外，规则是根据大量参考设计的经验而得的一般规则。然而，这些规则对于特定设计可能不适用。因此，理解特定指南规则并根据您的具体算法和要求采取适当的行动是您的责任。

GUI 流程

“Guidance”视图会自动填充并显示在底部中央的标签视图中。运行硬件仿真后，“Guidance”视图将如下所示：

图 14: **“Guidance”视图**



注释: 您可以通过 Vivado HLS 工具后编辑来生成“Guidance”视图，但是将无法获得“Profile Rule Checks”。

为了简化可视化指南信息，GUI 流程允许您搜索和筛选“Guidance”视图，以查找特定指南规则项。还可以折叠或展开树状视图或甚至抑制层级树表述，以及可视化指南规则的精简表述。最后，您可以选择“Guidance”视图中显示的内容。您可以启用或禁用警告的可视化和已满足的规则，并根据构建和仿真等消息来源限制特定内容。

默认情况下，“Guidance”视图显示下拉菜单中所选项目的全部指南信息。

要限制单个构建和运行步骤的内容，请执行以下操作：

1. 使用命令 “Window → Preferences”
2. 选择类“Xilinx Sdx → Guidance”。
3. 取消单击“Group guidance rule checks by project”。

命令行

通过合并了流程所有指南信息的 GUI 可对指南数据进行最佳分析。但是，该工具会自动生成包含指南信息的 HTML 文件。由于指南信息在工具的整个流程中生成，因此生成了多个指南文件。定位指南报告的最简单方法是搜索 `guidance.html` 文件。

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

此命令列出已生成的所有指南文件，可使用任何网页浏览器打开。

数据解读

“Guidance”视图将每个输入项放置在独立的行中。每行可能含有指南规则的名称、阈值、实际值以及该规则的简要而具体的描述。最后一个字段提供了指向参考材料的链接，这些参考材料的目的是为了帮助理解和解决任何违规。

在“GUI Guidance”视图中，指南规则按类别和名称列中唯一的 ID 进行分组，并注释有表示严重性的符号。这些在 HTML 报告中将单独列出。此外，由于 HTML 报告未显示工具提示，因此 HTML 报告中还包括了一个完整的“名称”列。

以下列表描述了 HTML 指南报告中包含的所有字段及其目的。

- “Id”：每个指南规则都分配了一个唯一的 ID。使用此 ID 可唯一识别指南报告中的特定消息。
- “Name”：“Name（名称）”列显示了一个助记名称，可唯一识别指南规则。这些名称的设计是为了帮助记住视图中的特定指南规则。
- “Severity”：“Severity（严重性）”列可以方便识别指南规则的重要性。
- “Full Name”：“Full Name（全名）”提供了与“名称”列中的助记名称相比更清晰的名称。
- “Categories”：大部分消息已在不同类别内进行了分组。这样 GUI 可以在“Guidance”视图中的通用树节点下的逻辑类别中显示消息组。
- “Threshold”：“Threshold （阈值）”列显示了预期的阈值，它决定规则是否得到满足。该阈值由许多符合良好设计和编码做法的应用决定。
- “Actual”：“Actual（实际值）”列显示具体设计上实际遇到的值。此值将与预期值做比较以确定是否满足规则。
- “Details”：“Details （详情）”列提供了描述当前规则相关情况的简要但具体的消息。
- “Resolution”：“Resolution （分辨率）”列提供了一个指向通用方法的指针，这些通用方法可以修改模型源代码或工具变换以满足当前规则。单击该链接可显示一个弹出窗口或一个带有提示并可应用于特定问题的代码段的文档。

使用实现工具

使用 Vivado HLS 进行内核最优化

所有使用 OpenCL 或 C/C++ 进行的内核最优化均可从 SDAccel 环境内部执行。主要的性能最优化，如本章中所讨论的（流水线化函数和循环、应用数据流来启用函数和循环之间更大的并发、展开循环等），可通过赛灵思 FPGA 设计工具 Vivado® HLS 工具来执行。

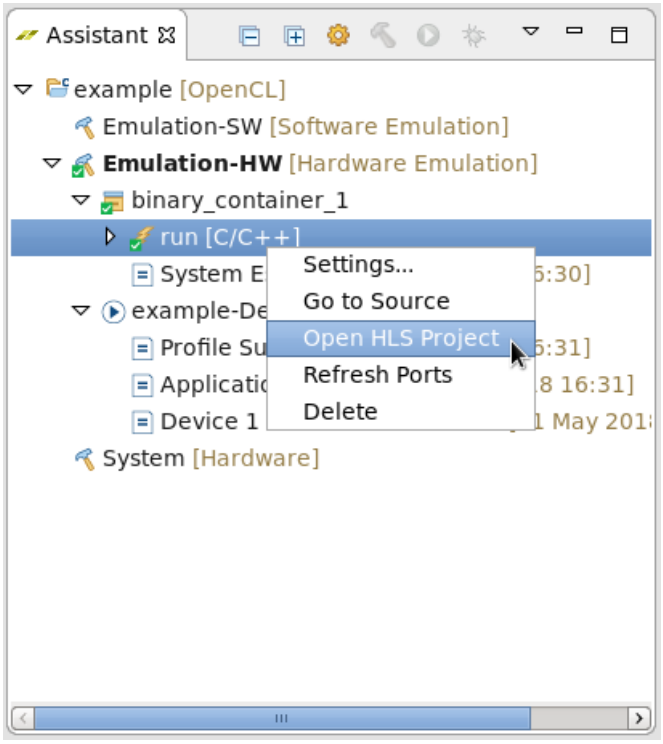
SDAccel 环境会自动调用 HLS 工具。但是，要使用 GUI 分析功能，您必须从 SDAccel 环境内部直接启动 HLS 工具。在独立模式中使用 HLS 工具可针对组最优化方法启用以下增强功能：

- 只专注于内核最优化，无需执行仿真。
- 创建多种解决方案、比较其结果以及利用解决方案空间来查找最佳设计的能力。
- 使用“Analysis Perspective”来分析设计性能的能力。

重要提示! 只有内核源代码被整合回到 SDAccel 环境中。利用最优化空间后，确保所有最优化作为 OpenCL 属性或 C/C++ 编译指示已被应用到内核源代码中。

要在独立模式中打开 HLS 工具，从“Assistant”窗口，双击硬件函数对象，然后选择 “Open HLS Project”，如下图所示。

图 15: **打开 HLS 工程**



使用 Vivado Design Suite 控制 FPGA 实现

SDAccel 开发环境提供了从 OpenCL/C/C++ 模型到 FPGA 加速实现的顺畅流程。在大多数情况下，此流程完全抽象化了 FPGA 中可编程区域已配置为执行内核功能的基础事实。这样使开发者完全避免了典型的硬件约束，如布线延迟和内核布局。但是，在某些情况下，必须考虑这些担心，特别是当执行大型设计时。为了实现此目的，开发环境允许您完全控制 Vivado Design Suite 后端工具。

SDAccel 环境调用 Vivado Design Suite 来自动运行 RTL 综合与实现。您还可以选择从 SDAccel 环境内部直接启动该设计套件。当在 SDAccel 环境中的独立模式下调用 Vivado 集成设计环境 (IDE) 时，您可以打开 Vivado 综合项目或 Vivado 实现项目，以编辑、管理和控制该项目。

当以系统配置为目标的构建完成后，Vivado 项目可以在 SDAccel 环境中打开。

要在独立模式下打开 Vivado IDE，请从赛灵思下拉菜单选择 “Vivado Integration” 和 “Open Vivado Project”。在 Vivado 综合项目和实现项目之间选择，然后单击 “OK”。

在独立模式下使用 Vivado IDE 可以利用各种综合和实现选项来进一步最优化内核的性能和区域。为了能够更好地使用大部分功能，建议您一定要熟悉该设计套件。

重要提示! 在独立项目中应用的最优化开关不会自动整合回到 SDAccel 环境中。利用最优化区域后，确保使用 `xocc` 的 `--xp` 选项将所有最优化参数传递给 SDAccel 环境。例如：

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

通过调用 `xocc -interactive` 来显示当前项目的 Vivado IDE，命令行流程中也支持此最优化流程。在 IDE 中，生成 DCP，它可以被保存并在与 xocc 链接过程中重复使用。具体选项为：

- “`--interactive`” 允许 Vivado IDE 从 `xocc` 环境内部启动，并加载正确的项目。
- “`--reuse_impl`” 允许将预执行和时序关闭的 Vivado 工具设计检查点 (.dcp) 文件引进来并在 SDx 环境流程中直接使用，以便生成 xclbin。

内核最优化

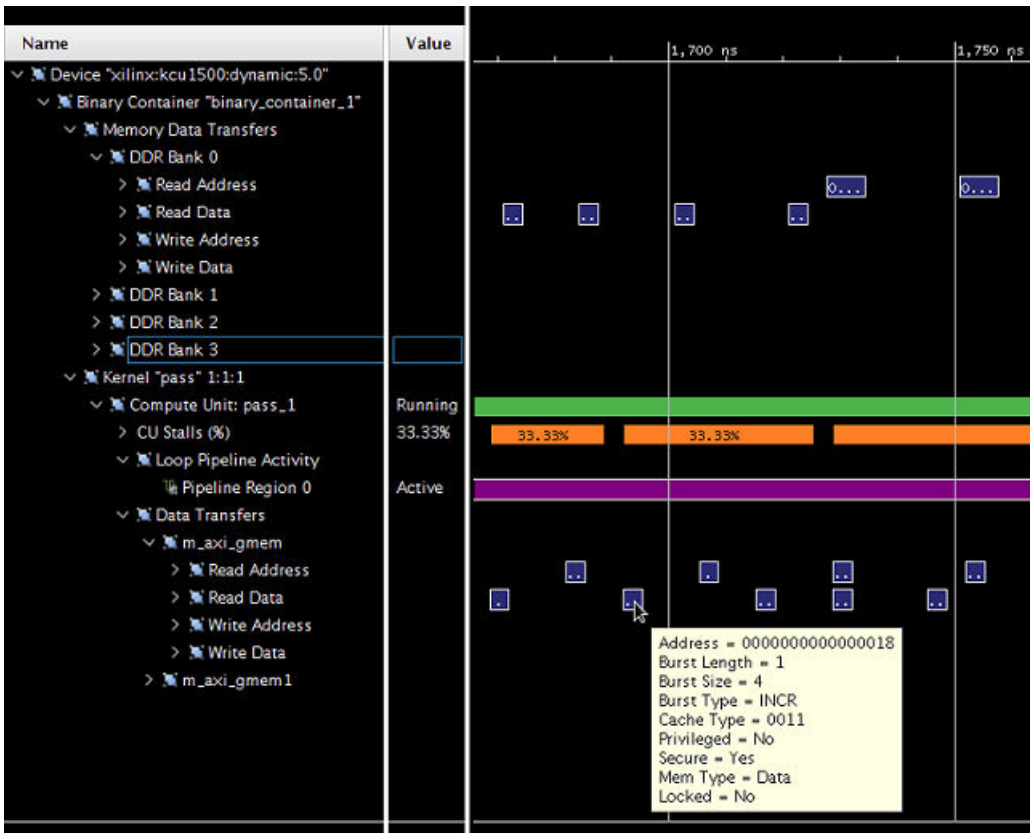
FPGA 的一个关键优势是其特别为您的算法创建定制设计的灵活性和能力。这可提供各种实现选择，以便在算法吞吐量和功耗之间进行利弊取舍。创建定制逻辑的缺点是设计需按照传统的 FPGA 设计流程进行。

以下指导原则可帮助管理设计的复杂性和实现所需的设计目标。

接口属性（详细的内核追踪）

详细的内核追踪可提供对 AXI 传输事务及其属性的方便访问。AXI 传输事务出现在全局存储器以及 AXI 互连的内核一侧 (Kernel "pass" 1:1:1) 。下图显示了一个新加速算法的典型内核追踪。

图 16: **已加速的算法内核追踪**



以下字段在性能方面最重要：

- “Burst Length”：描述在一个传输事务内发送多少次节拍
- “Burst Size”：描述作为一个节拍的一部分进行传输的字节数

如果突发长度为 1 且每个封装只有 4 个字节，则它将需要许多单独的 AXI 传输事务来传输任何合理的数据量。

注释: SDAccel™ 环境决不会创建小于 4 个字节的突发量，即时传输更小的数据也是如此。在这种情况下，如果未启用 AXI 突发即访问连续项，您会看到多个 AXI 读取同一个地址。

因此，较小的突发长度和突发量，特别是小于 512-位，是最优化接口性能的良好机会。以下个节显示了经过提升的实现：

- 使用突发数据传输
- 使用 AXI4 数据宽度

顶层数据流

通过重叠多个内核执行，顶层数据流最优化可为内核在并行处理架构中执行提供能力。在较长内核时延的情况下，此最优化很有用，并会提升应用的总体性能。赛灵思推荐通过将接口编译指示 `ap_ctrl_chain` 应用到返回端口（与 `s_axilite` 一起）而仅对在顶层函数中具有数据流的内核启用此最优化功能。

```
#pragma HLS INTERFACE ap_ctrl_chain port=return bundle=control
```

提供以下示例的目的是说明返回端口同时需要 `ap_ctrl_chain` 和 `s_axilite` 选项才能启用顶层数据流。

```
void N_stage_Adders(int *input, int *output, int incr, int size)
{
    ...
    #pragma HLS INTERFACE s_axilite port=return bundle=control
    #pragma HLS INTERFACE ap_ctrl_chain port=return bundle=control
    ...
    ...
    #pragma HLS dataflow
    read_input(input,streamArray[0],size);
    compute_loop: for (int i = 0 ; i < STAGES ; i++)
    {
        #pragma HLS UNROLL
        adder(streamArray[i],streamArray[i+1],incr,size);
    }
    write_result(output,streamArray[STAGES],size);
}
```

注释: 此最优化的剖析能力目前仅在硬件中提供。

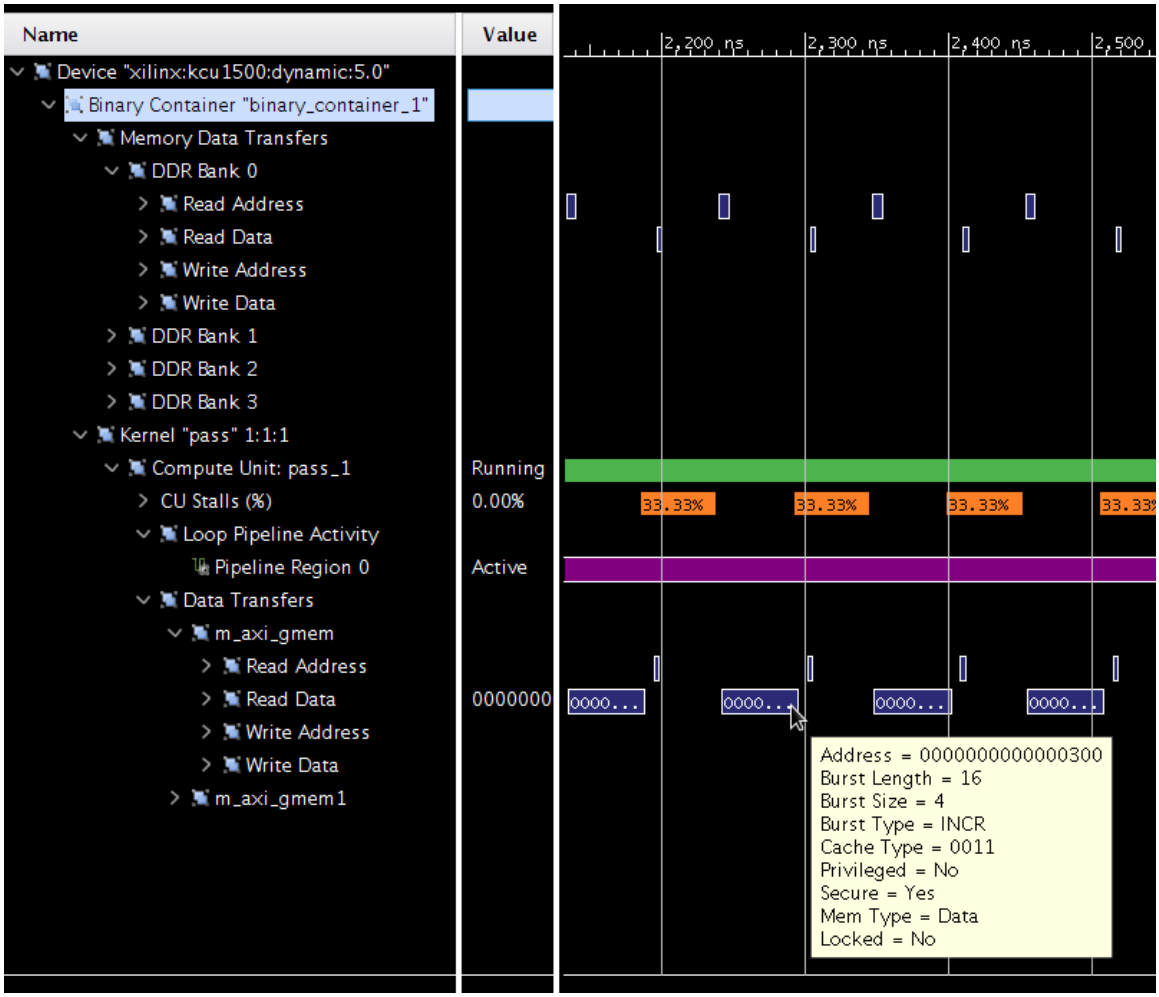
使用突发数据传输

以突发方式传输数据可隐藏存储器访问时延，并提升存储器控制器的带宽使用量和效率。

建议: 引用突发从连续地址位置传输数据的持续请求。如需了解更多详情，请参阅《SDAccel 环境编程指南》 (UG1277) 中的“自/至引用突发传输”。

如果出现突发数据传输，详细的内核追踪将把较高的突发速率反映为较大的突发长度数字：

图 17: **带详细内核追踪的突发数据传输**



在上图中，还可以观察到 AXI 互连后的存储器数据传输实际上以不同的方式执行（更短的传输事务时间）。如果您将鼠标停留在这些传输事务上，您将看到 AXI 互连已将 16x4 字节传输事务封装到单个 1x64 字节的突发传输事务中。这样可以有效使用更加有利的 AXI4 带宽。下一节将更加详细地讨论此最优化技术。

突发推断在很大程度上依赖于编码样式和访问模式。为了避免潜在的建模缺陷，请参阅《SDAccel 环境编程指南》(UG1277)。但是，您可以通过隔离数据传输和计算来方便检测和提升性能，如以下代码段所示：

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

总之，函数 `read` 负责从 AXI 输入读取到内部变量 (`tmpIn`)。计算由函数 `process` 针对内部变量 `tmpIn` 和 `tmpOut` 执行。函数 `write` 获取产生的输出并写入到 AXI 输出。

将读写函数与计算隔离可带来：

- 读取/写入函数中的简单控制结构（循环）使突发检测更简单。

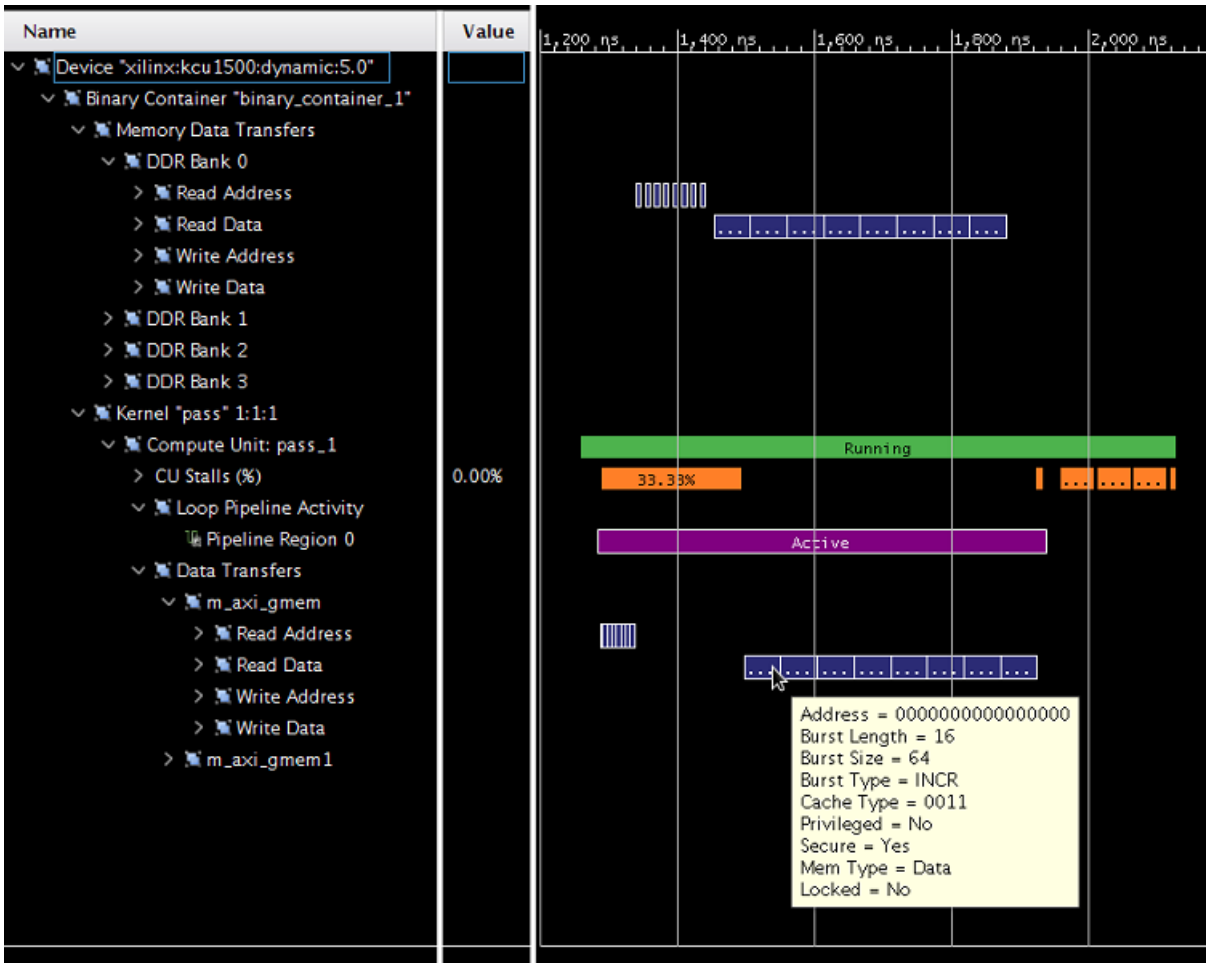
- 计算函数与 AXI 接口隔离可简化潜在的内核最优化。如需了解更多信息，请参阅内核最优化一章。
- 内部变量被映射到芯片存储器上，与 AXI 传输事务相比，这样做访问速度更快。SDAccel 环境中支持的加速平台可具有高达 10 MB 的片上存储器，可用作管道、本地存储器和专用存储器。有效使用这些资源可极大地提升应用的效率和性能。

使用 AXI4 数据宽度

在内核和存储器控制器之间的用户数据宽度可通过 SDAccel 环境编译器根据内核实参的数据类型予以配置。为了使数据吞吐量最大化，赛灵思推荐您选择映射到存储器控制器上完整数据宽度的数据类型。所有受支持加速卡中的存储器控制器均支持 512 位用户接口，该接口可以映射到 OpenCL™ 矢量数据类型，例如 `int16` 或 C/C++ 任意精度数据类型 `ap_int<512>`。

如下图所示，您可以观察到突发 AXI 传输事务（突发长度 16）和 512 位节拍规模（突发量 64 字节）。

图 18: **突发 AXI 传输事务**



本示例显示良好的接口配置，因为它可以最大化 AXI 数据宽度，并可显示实际突发传输事务。

由于存储器布局和数据封装的差异，用于声明接口的复杂结构或类可能会造成非常复杂的硬件接口。这可能会带来在复杂系统中很难调试的潜在问题。

建议: 对于可以封装到二的倍数边界的内核实参，请使用简单的结构。如需了解使用结构的推荐方法，请参阅赛灵思板载示例 GitHub 的 《kernel_to_gmem》 类别中的《定制数据类型示例》。

OpenCL API 属性

OpenCL API 提供各种属性，以支持使用更加自动化的方法来增加 AXI 数据宽度的使用。如上所述，API 还支持接口数据类型的改变，但是需要对算法进行与 C/C++ 相同的代码更改才能适应更大的输入矢量。

为了消除手动代码修改，将对以下 OpenCL 属性进行解释，以便执行数据路径拓宽和算法的矢量化。详细描述请参阅《SDx 编译指示参考指南》 (UG1253)。

- vec_type_hint
- reqd_work_group_size
- xcl_zero_global_work_offset

在以下示例中，查看下列案例的组合功能：

```
__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];
}
```

在此案例中，硬编码接口是一个 32 位宽的数据路径 `(int *c, int* a, int *b)`，如果直接执行，它会极大地限制存储器的吞吐量。但是，已根据三个属性的值应用了自动拓宽和变换。

- “__attribute__((vec_type_hint(int)))”：声明 `int` 是用于计算和存储器传输（32 位）的主要类型。此知识用于根据 AXI 接口（512 位）的目标带宽计算矢量化/拓宽系数。在本示例中，该系数将为 16 = 512 位/32 位。这表示，在理论上，如果可以应用矢量化，则可以处理 16 个值。
- “__attribute__((reqd_work_group_size(X, Y, Z)))”：定义工作项的总数（其中 `X`、`Y` 和 `Z` 为正常数） `X*Y*Z` 是工作项的最大数量，因此请定义尽可能大的矢量化系数，该系数将填满存储器带宽。在本示例中，工作项的总数是 `64*1*1=64`。

  要使用的实际矢量化系数将是由实际编码类型或 vec_type_hint 定义的矢量化系数和通过 `reqd_work_group_size` 定义的最大矢量化系数的最大公约数。

  最大矢量化系数的商除以实际矢量化系数将提供 OpenCL 描述的剩余循环计数。由于此循环进行流水线处理，使几个剩余循环迭代可能非常有利，这样能够充分利用流水线化实现的优点。如果矢量化 OpenCL 代码具有很长的时延，尤其如此。

有一个可选的参数，特别推荐要进行指定，用于 OpenCL 接口上的性能最优化。

- __attribute__((xcl_zero_global_work_offset)) 指示编译器，运行时间未使用任何全局偏移参数，且所有访问均已协调一致。这样可以向编译器提供关于工作组协调的有用信息，而该其反过来通常又会传输到存储器访问的协调中（更少的硬件）。

应注意，这些转换的应用将更改要综合的实际设计。部分展开的循环要求重塑存储数据的本地阵列。此操作通常会顺利进行，但在极少情况下交互可能会很差。

例如：

- 对于分区的阵列，当分区系数不能被展开/矢量化系数整除时。
  - 产生的访问将需要大量的多路复用器并将给调度器创造困难问题（可能会严重增加内存使用量和编译时间），赛灵思推荐您使用 2 的倍数的分区系数（因为矢量化系数始终是 2 的倍数）。
- 如果进行矢量化的循环具有无关的资源约束，则调度器表示 II 无法得到满足。
  - 这不一定与性能损失相关（通常它仍能够很好执行），因为 II 在未展开的循环（因而每个迭代具有成倍的吞吐量）上计算。
  - 调度器会通知您有关可能的资源约束，并告知您解决这些约束将进一步提升性能。
  - 请注意，一种常见的现象是本地阵列不会自动重塑（通常是因为在后面的代码段以非矢量化的方法对它进行访问）。

使用 OpenCL 管道减小内核至内核之间的通信时延

本节特别适用于 OpenCL 内核。对于 C++ 内核，已提供了内核至内核流传输。该内容详见 存储器数据传输类型 中的讨论。

OpenCL API 2.0 规格介绍了一种称为管道的新的存储器对象。管道以先进先出的方式存储数据。管道对象只能使用从管道读取和写入的内置函数来访问。管道对象无法从主机进行访问。管道可用于从 FPGA 内部将数据从一个内核流传输到另一个内核，而无需使用外部存储器，这样可以极大地提升总体系统时延。

在 SDAccel 开发环境中，管道必须在所有内核函数外部予以静态定义；目前不支持使用 `OpenCL 2.x clCreatePipe` API 的动态管道分配。管道深度必须使用管道声明中的 xcl_reqd_pipe_depth 属性来规定。

有效的深度值如下：

- 16
- 32
- 64
- 128
- 256
- 512
- 1024
- 2048
- 4096
- 8192
- 16384
- 32768

在不同内核中，一个给定的管道可以有且只能有一个生产者和消费者。

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

可在无阻塞模式中使用标准 OpenCL `read_pipe()` 和 `write_pipe()` 内置函数或在阻塞模式中使用赛灵思扩展 `read_pipe_block()` 和 `write_pipe_block()` 函数来访问管道。可以使用 OpenCL `get_pipe_num_packets()` 和 `get_pipe_max_packets()` 内置函数来查询管道的状态。如需了解有关这些内置函数的详情，请参阅 Khronos Group 的“OpenCL C 规格，2.0 版”。

以下函数签名是当前受支持的管道函数，其中 `gentype` 表示内置的 OpenCL C scalar 整数或浮点数据类型。

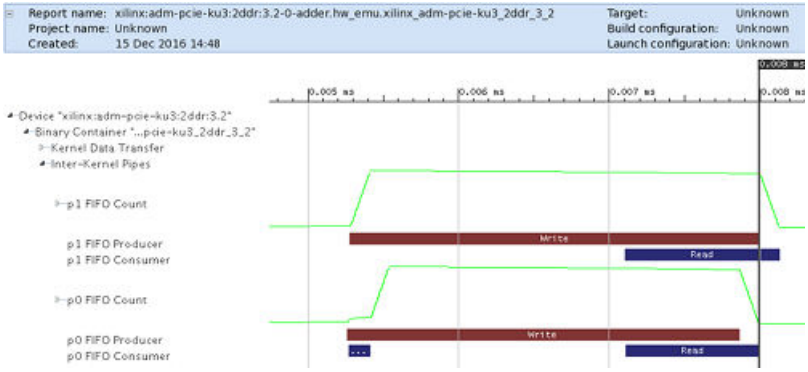
```
int read_pipe_block (pipe gentype p, gentype *ptr)
int write_pipe_block (pipe gentype p, const gentype *ptr)
```

以下 GitHub 上的 SDAccel 快速入门示例中的“阻塞管道示例”利用 `blocking read_pipe_block()` 和 `write_pipe_block()` 函数来使用管道将数据从一个处理阶段传递到另一个处理阶段：

```
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]);
    }
}
```

“Device Traceline”视图显示在硬件仿真运行后 OpenCL 管道上的详细活动和停滞情况。此信息可用于选择正确的先进先出规模，以便获得最佳的应用区域和性能。

图 19: **“Device Traceline”视图**



最优化计算并行度

默认情况下，C/C++ 不会为计算并行度建模，因为它始终按顺序执行任何算法。相反，OpenCL API 确实在工作组方面为计算并行度建模，但它在算法描述内部不使用任何额外的并行度。然而，类似于 FPGA 等可完全配置的计算引擎对利用计算并行度具有更多的自由度。

给数据并行度编码

要充分利用在 FPGA 上算法实现过程中的计算并行度，需要指出的是综合工具必须首先能够从源代码中识别计算并行度。循环和函数是反映计算并行度和源描述中计算单元的主要备选。但是，即使在这种情况下，验证实现充分利用计算并行度的优势很关键，因为在某些情况下，由于源代码的结构问题，SDAccel 工具可能无法应用所需的转换。

某些计算并行度可能从一开始就不会反映在源代码中，这种情况很常见。在此情况下，需要将它添加进去。典型的示例是可能被描述为在单个输入值上运行的内核，而 FPGA 实现可能在多个值上并行执行计算更为高效。这种并行建模在使用完整 AXI 数据宽度一节中描述。使用 OpenCL 矢量数据类型，如 `int16` 或 C/C++ 任意精度数据类型 `ap_int<512>` 可以创建 512 位接口。

注释: 这些矢量类型也可作为强大的方法用于在内核内部为数据并行度建模，在 `int16` 情况下，并行运行的数据路径可达 16 个。如需了解使用矢量的推荐方法，请参阅位于 GitHub 上 SDAccel 快速入门示例的《愿景》类别中的《中值滤波器示例》。

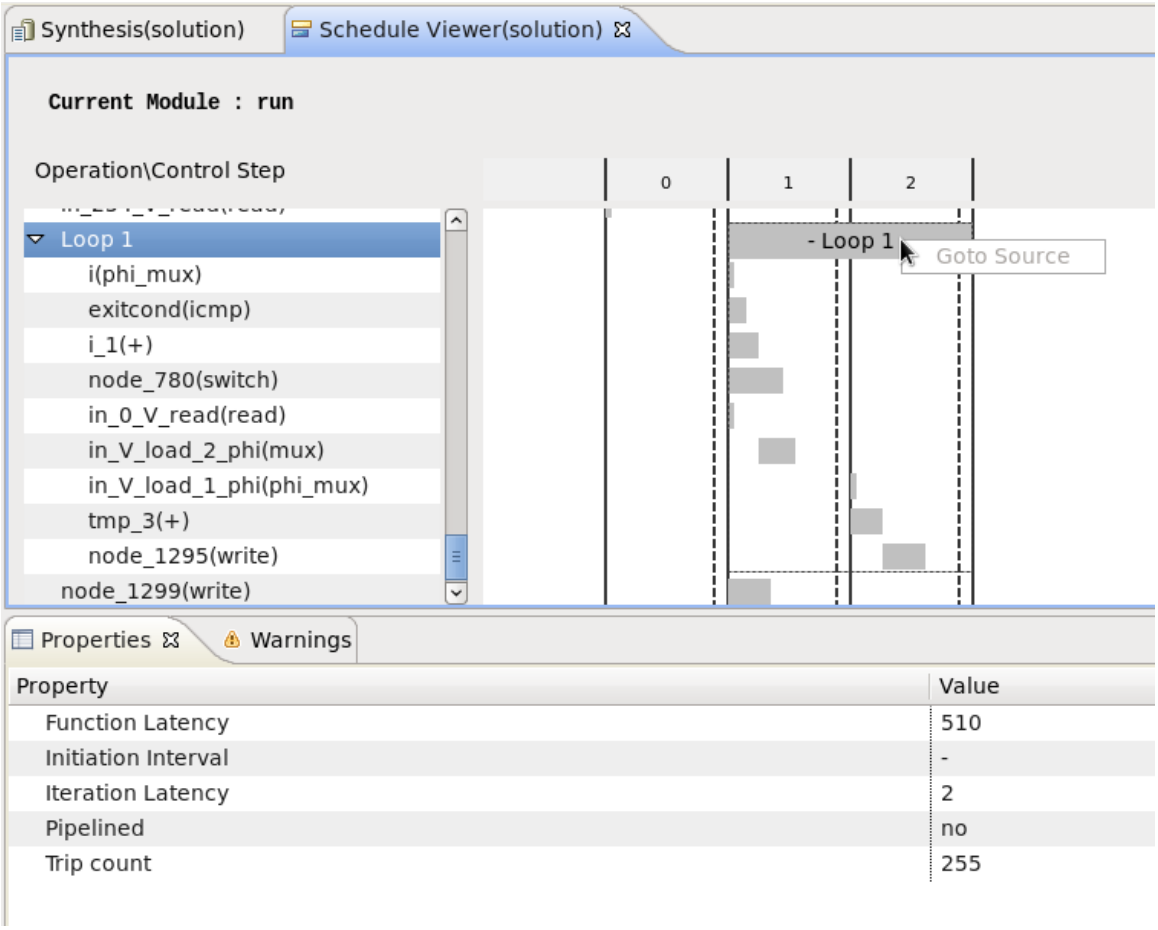
循环并行化

循环是表示重复算法代码的基本 C/C++/OpenCL™ API 方法。以下示例展示了循环结构的各个实现方面：

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

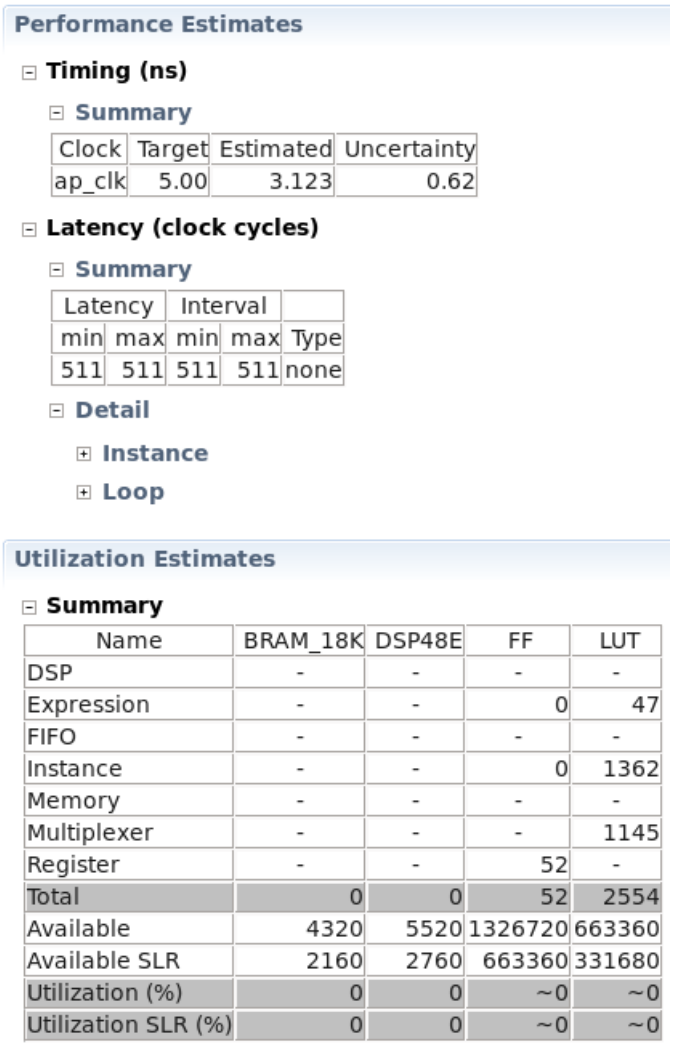
此代码在值阵列上进行迭代并添加连续值，最后一个值除外。如果此循环作为写入来执行，则每个循环迭代需要两个实现周期，这将导致总共 510 个实现周期。可以在 HLS 项目的调度器查看器中对此进行详细分析：

图 20: **调度器查看器中已执行的循环结构**



也可通过 Vivado 综合结果从总数和时延上对此进行分析：

图 21: **综合结果性能估算**



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

此处的关键数量是时延数量和总 LUT 使用情况。例如，取决于配置，您可以获得 511 个的时延和总数为 47 的 LUT 使用情况。如您所见，这些值根据不同实现选择可能会相差很大。当此实现需要非常小的区域时，会导致很大的时延。

展开循环

展开循环可启用要探索模型的完整并行度。要进行此操作，您只需标记要展开的循环，然后工具将以最多并行度来创建实现。要标记拟展开的循环，OpenCL API 循环可以使用 UNROLL 属性来标记：

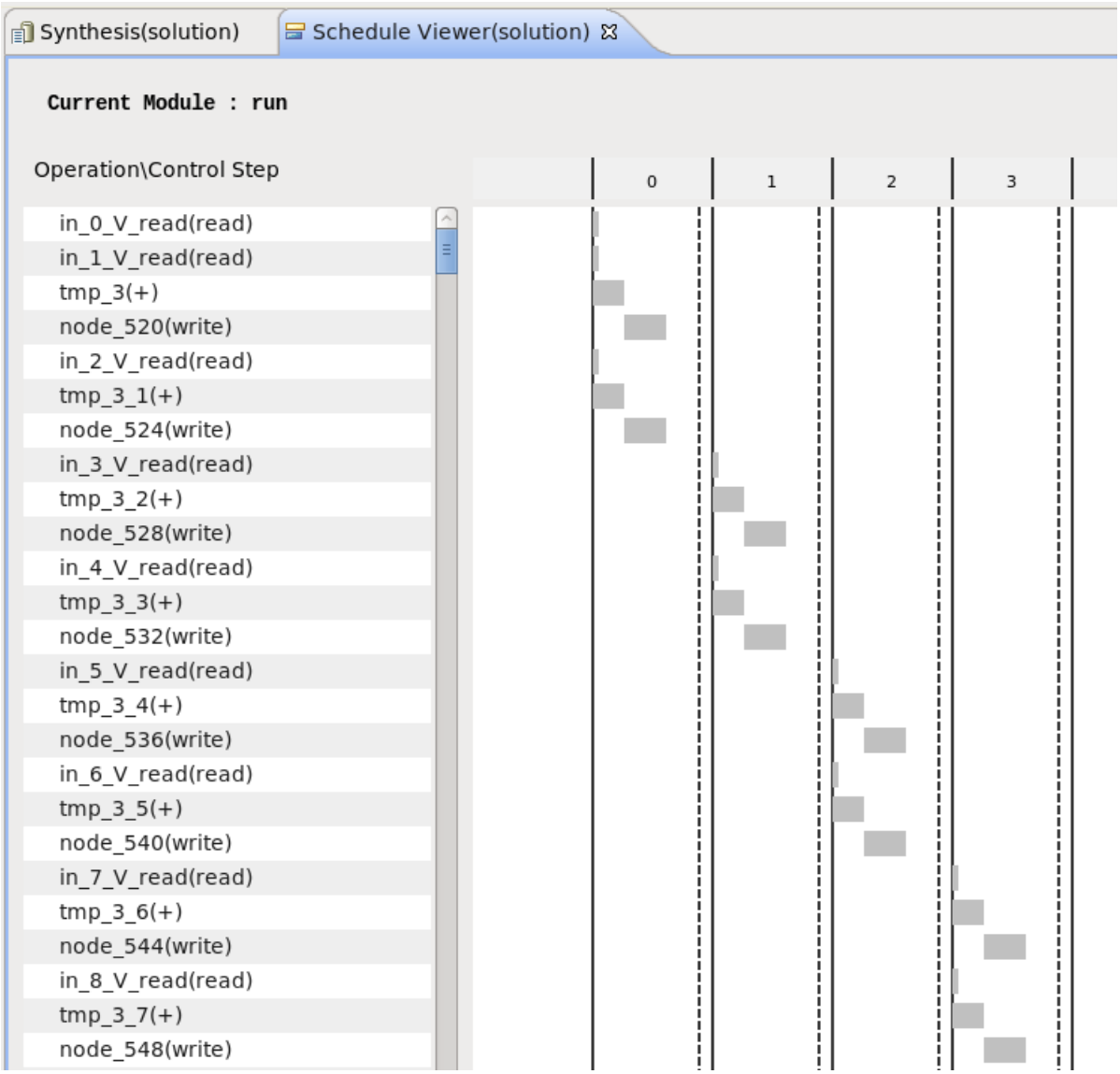
```
__attribute__((opencl_unroll_hint))
```

或 C/C++ 循环可以使用 UNROLL 编译指示：

```
#pragma HLS UNROLL
```

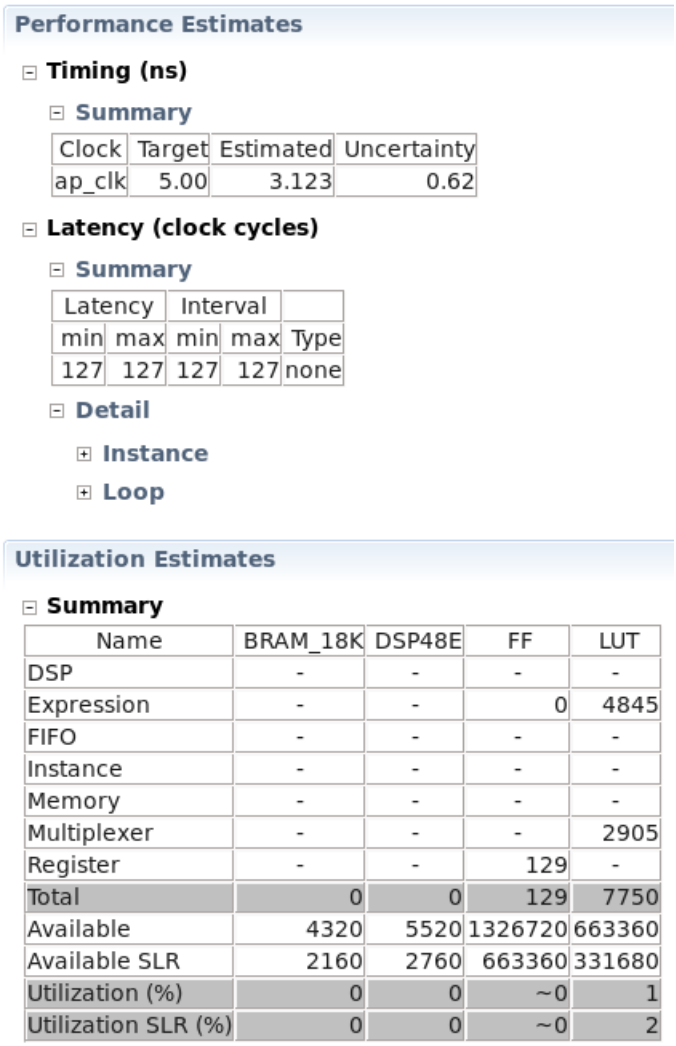
当应用于此具体示例时，HLS Project 中的 Schedule Viewer 将为：

图 22: **调度器查看器**



具有以下估算性能：

图 23: **性能估算**



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

您可以看到，总时延已大幅提升到 127 个周期，且如预期一样，计算硬件已提高到 4845 个 LUT，才能执行相同并行计算。

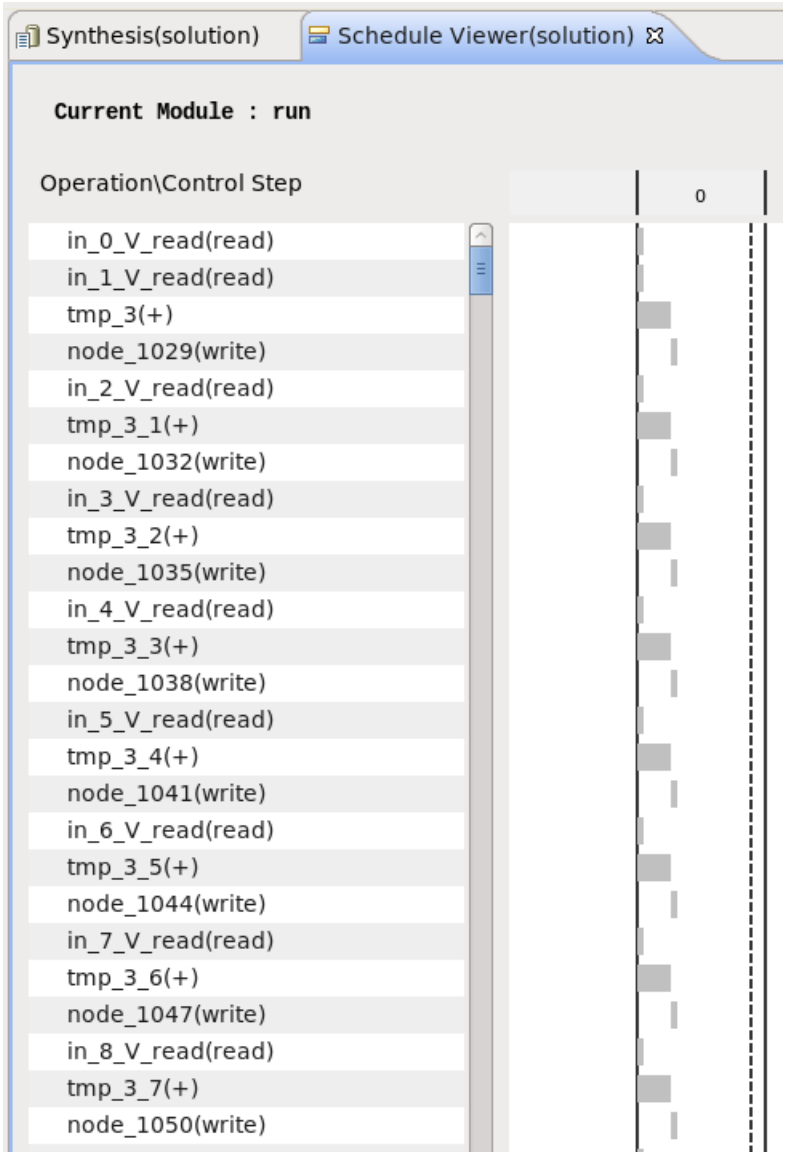
但是，如果您分析 for 循环，您可能会问，为什么此算法不能在单个周期中实现，因为每个加法都完全独立于前面的循环迭代。其原因在于要用于 `out` 变量的存储器接口。SDAccel 环境默认将双端口存储器用于阵列。但是，这表示每个周期最多可将两个值写入存储器。因此，要看到完全并行实现，您必须指定 `out` 变量保存在寄存器中，如以下示例所示：

```
#pragma HLS array_partition variable=out complete dim= 0
```

如需了解更多信息，请参阅 《SDx 编译指示参考指南》 (UG1253) 中的《pragma HLS array_partition》一节。

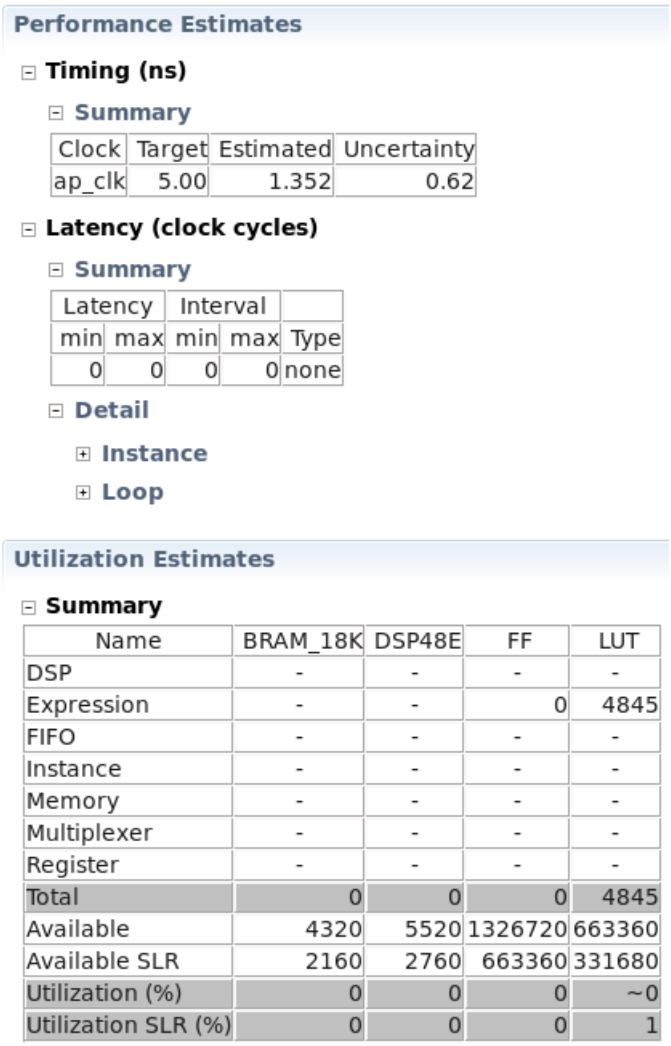
此变换的结果可在以下调度器查看器中查看：

图 24: **调度器查看器中的转换结果**



相关联的估算为：

图 25: **转换结果性能估算**



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

如您所看到的，此代码可作为只需周期的一小部分即可完成的组合函数来实现。

流水线循环

流水线循环可使您及时重叠循环的迭代。允许迭代并行运行通常是一种好的折衷方法，因为资源可以在迭代之间共享（资源消耗更少），同时与未展开的循环相比，需要更少的执行时间。

通过以下编译指示启用 C/C++ 中的流水线：

```
#pragma HLS PIPELINE
```

当 OpenCL API 使用以下属性时：

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

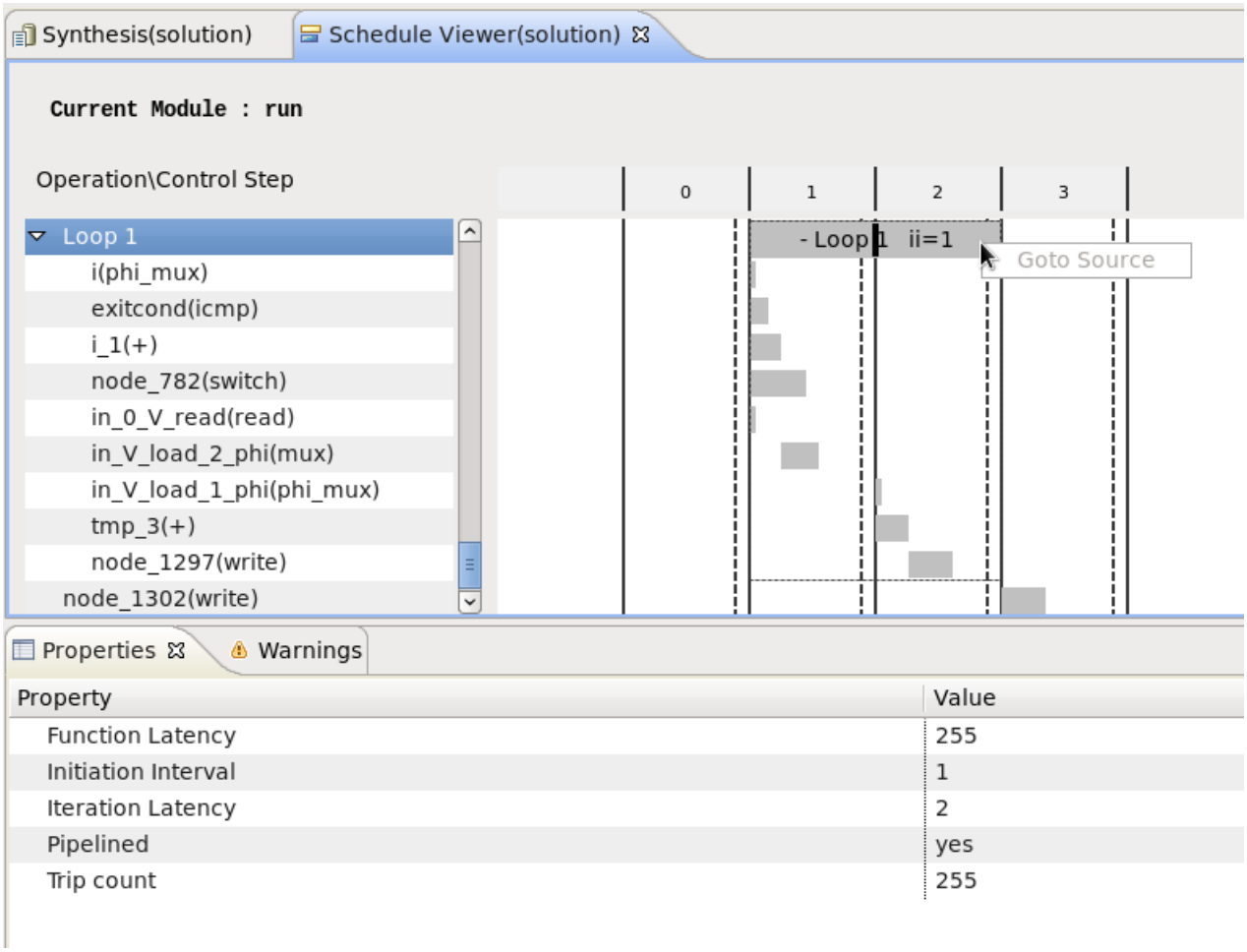
注释: OpenCL API 还有其他方法来指定循环流水线。此操作与未明确声明工作项循环有关，流水线化这些循环需要属性：

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

有关这些规格中任何一种规格的更多详情详见《SDx 编译指示参考指南》 (UG1253) 和《SDAccel 环境编程指南》 (UG1277)。

在此示例中，HLS 项目中的调度器查看器可产生以下信息：

图 26: **调度器查看器中的流水线循环**



其中总体估算为：

图 27: **性能估算**

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

因为循环的每个迭代仅消耗两个时延周期，因此只能有单个迭代重复。这样使得总时延将被切分为原来的一半，导致 257 个周期的总时延。但是，如果与展开相比，时延的减小通过利用更少资源来实现。

在多数情况下，循环流水线化本身可提升总体性能。但是，流水线化的有效性将取决于循环的结构。部分常用迭代为：

- 存储器端口或处理通道等具有有限可用性的资源可限制底迭代 (II) 的重叠。
- 与此类似，循环承载的相依性，例如由在一个影响下一个迭代的迭代中计算的变量条件创建的相依性，可能会增大流水线的初始间隔。

这些将在高层次综合过程中由工具进行报告，并可在调度器查看器中进行查看和检查。为了获得最佳性能，可能需要修改代码来消除这些限制因素，或者需要指示工具通过重构阵列的存储器实现或通打破所有相依性来消除某些相依性。

任务并行度

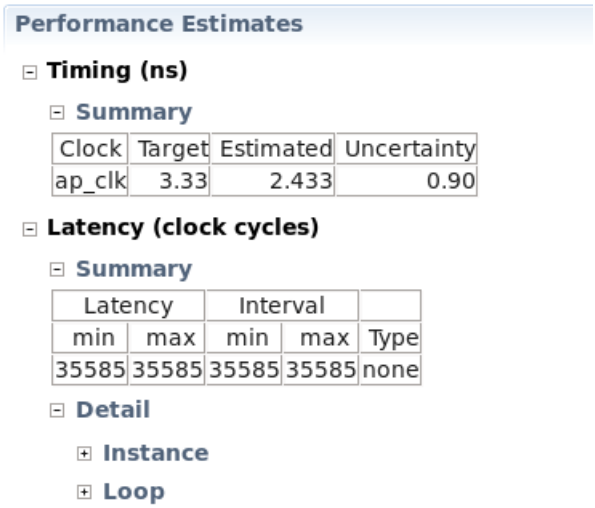
任务并行度可让您充分利用数据流并行度。与循环并行化相反，当部署任务并行度后，完全执行单元（任务）可并行运行，从而充分利用任务之间带来的额外缓存。

请看以下示例：

```
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]));
  }
}
```

当执行此代码时，将逐行顺序执行函数 processA 和 processB 128 次。如果循环中 processA 和 processB 的组合时延为 278，则总时延可估算为：

图 28: **性能估算**



额外的周期是因循环建立而产生，并可在调度器查看器中查看。

对于 C/C++ 代码，可将 DATAFLOW 编译指示添加到 for 循环中来执行任务并行度：

```
#pragma HLS DATAFLOW
```

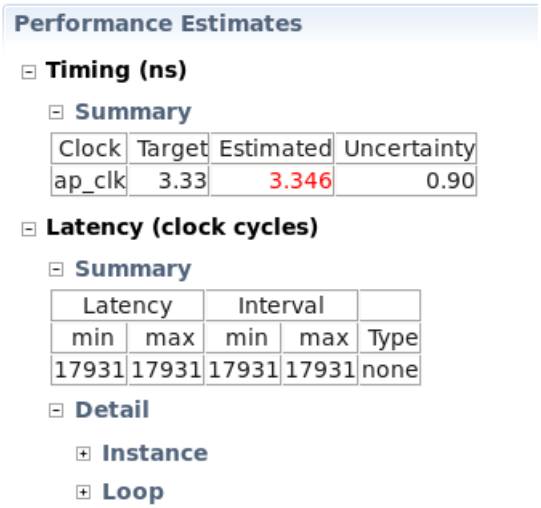
对于 OpenCL API 代码，将属性添加到 for 循环之前：

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

如需了解有关这些修饰符相关信息和限制的详情，请参阅《SDx 编译指示参考指南》 (UG1253) 和《SDAccel 环境编程指南》 (UG1277)。

如 HLS 报告中的估算所示，应用变换可以极大地提升总体性能，有效使用任务之间的双（乒乓）缓存方案：

图 29: **性能估算**

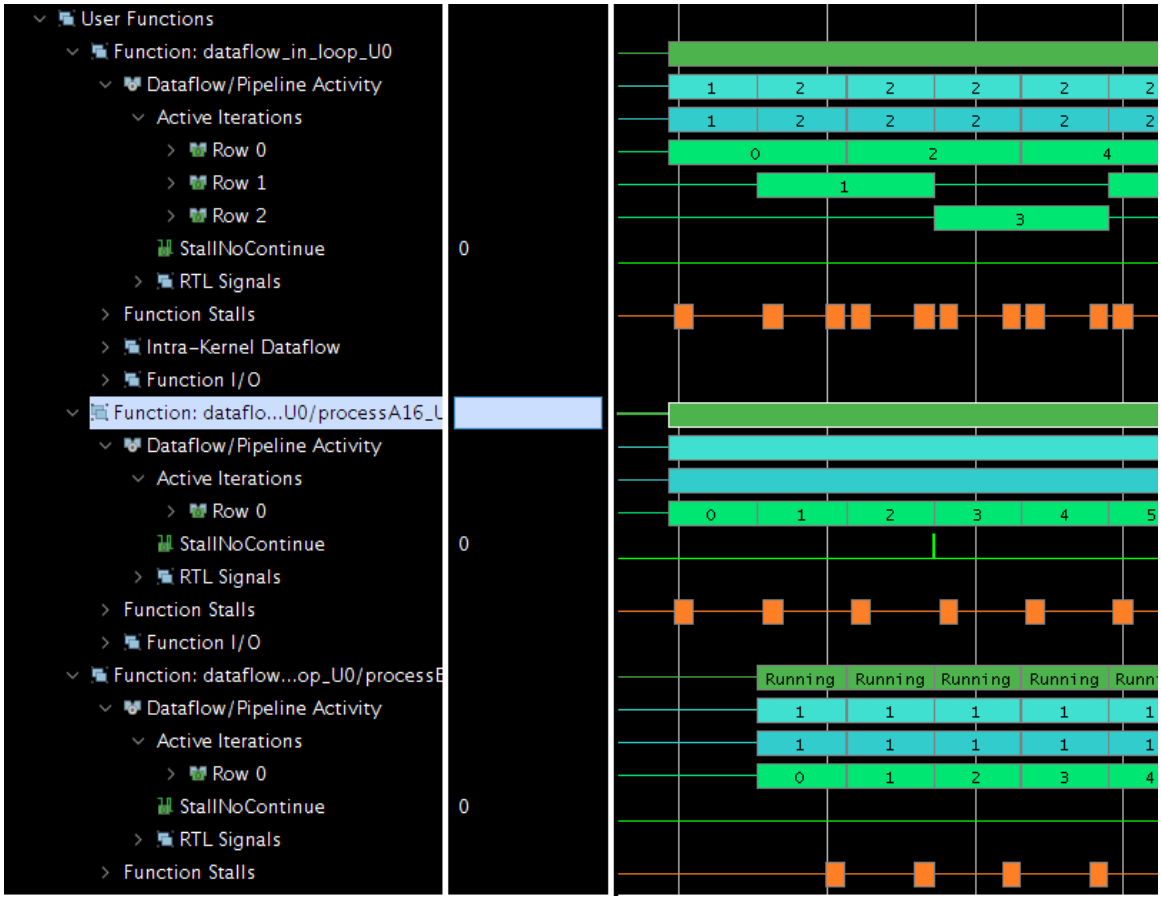


由于不同迭代的不同任务的并行执行，设计的总时延在此情况下几乎下降了一半。考虑到每个处理函数的 139 个循环和 128 个迭代的完全重叠，这将允许总时延为：

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

使用任务并行度对于实现是非常强大的提升性能的方法。但是，将 DATAFLOW 编译指示应用到具体和任意代码段的有效性可能会差异很大。有效应用 DATAFLOW 的编码指南提供在《SDx 编译指示参考指南》 (UG1253) 和《SDAccel 环境编程指南》 (UG1277) 中）。但是，为了了解 DATAFLOW 编译指示的最终实现，经常需要实际查看单个任务的执行模式。为了实现此目的，SDAccel 提供了详细的内核追踪，可以很好地展示并发执行。

图 30: **详细的内核追踪**



对于此详细的内核追踪，该工具显示了数据流循环的开始，如上图所示。它显示了 processA 如何在循环的开始启动，而 processB 等待直到 processA 完成，才启动其第一次迭代。但是，当 processB 完成循环的第一次迭代后，processA 开始运行第二次迭代，以此类推。

此消息更为抽象的表示出现在“Application Timeline (Host & Device)”和“Device Hardware Transaction”视图中（在硬件仿真过程中仅为器件）。

最优化计算单元

数据宽度

性能的一个方面是实现所需的数据宽度。工具在整个算法过程中传输端口宽度。在某些情况下，特别是当以算法描述作为开始时，C/C++/OpenCL™ API 代码即使在设计的端口也可能只使用像整数这样的大数据类型。但是，由于算法已映射到完全可配置的实现，10 或 12 位等更小的数据类型通常就已足够了。为了实现此目的，在最优化过程中检查“HLS Synthesis”报告中的基本操作的大小非常有益。总而言之，当 SDAccel 环境将算法映射到 FPGA 上时，需要大量的处理才能理解 C/C++/OpenCL API 的结构并提取操作相依性。因此，为了执行此映射，SDAccel 环境通常将源代码分区为多个操作单元，然后将这些操作单元映射到 FPGA 上。这些操作单元 (ops) 的数量和大小受到几个方面的影响，如工具中所示。

在下图中，报告了基本操作及其位宽。

图 31: **“Utilization Estimates”操作**

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

查找算法描述中通常使用的 16、32 和 64 位位宽，并验证来自 C/C++/OpenCL API 源的相关操作确实需要这样大的位宽。这样可以极大地提升算法的实现，应为更小的操作需要的计算时间更少。

定点运算

某些应用程序仅使用浮点运算，因为它们是为其他硬件架构进行最优化的。如使用赛灵思器件上 INT8 最优化进行深度学习中所解释，使用定点运算的各种应用（如深度学习）可以大量节省能耗和区域，而能保持相同的精度等级。

建议: 决定使用浮点运算之前，请先为您的应用探索使用定点运算。

宏运算

考虑更大的计算元素有时非常有利。该工具将在独立于其余的源代码的源代码上运行，有效地将算法映射到 FPGA 上，而无需考虑周围的运算。在应用时，SDAccel 工具将保持运算边界，有效创建特定代码的宏运算。这使用了以下原理：

- 至映射进程的运算位置。
- 降低启发式方法的复杂性。

在应用时这可能会产生巨大的不同结果。在 C/C++ 中，宏运算在以下命令的帮助下创建

```
#pragma HLS inline off
```

而在 OpenCL API 中，同类宏运算可通过不指定以下属性来生成，但要定义以下函数：

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

使用最优化的库

OpenCL 规格提供了许多内置数学函数。带有 `native_` 前缀的所有内置数学函数都映射到一个或多个本地器件指令，与对应的函数（不带 `native_` 前缀）比较通常具有更好的性能。这些函数的精度和输入范围（某些情况下）由实现定义。在 SDAccel™ 环境中，这些 `native_` 内置函数使用 Vivado® HLS 工具数学库中的对等函数，这些对等函数已在区域和性能上针对赛灵思 FPGA 进行过最优化。

建议: 如果精度满足应用要求，赛灵思推荐您使用 `native_` 内置函数或 HLS 工具数学库。

最优化存储器架构

实现的一个重要方面是存储器架构。由于访问带宽有限，可能会严重影响整体性能，如以下示例所示：

```
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
}
```

此代码添加与二维输入阵列内部维度相关联的四个值。如果在不进行任何其他修改的情况下执行，它会产生以下估算：

图 32: **性能估算**

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

4608（循环 2）的总体时延应该为 18 个周期的 256 次迭代（16 个周期消耗在内部循环中，加上总和复位，加上被写入的输出）。可以在 HLS 项目的调度器查看器中对此进行查看。当展开内部循环时，该估值会显著变得更好。

图 33: **性能估算**

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

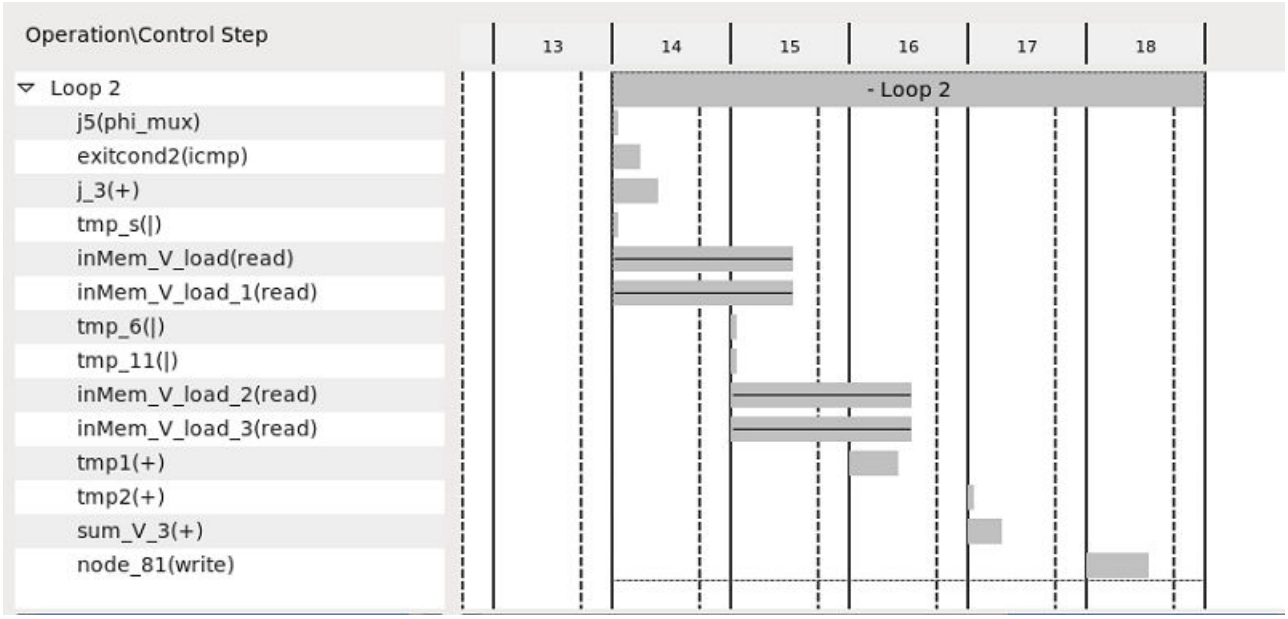
Instance

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

但是，此提升主要是由于此进程同时使用双端口存储器的两个端口所致。可以在 HLS 项目的调度器查看器中看到这一点：

图 34: **调度器查看器**

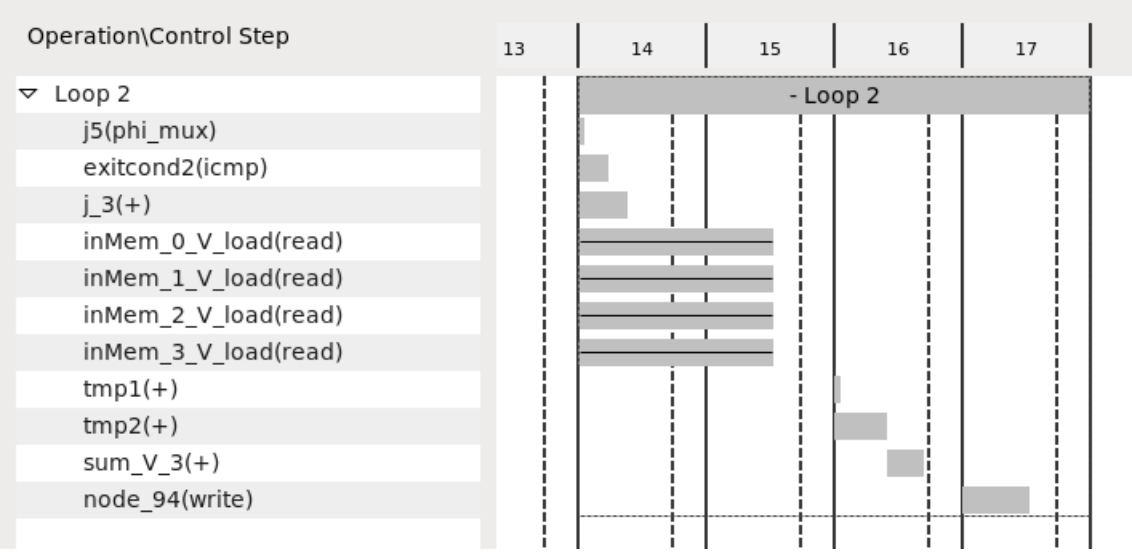


如您所见，每个周期执行两个读取操作，以便从存储器中访问所有值来计算总和。这通常是我们不愿看到的结果，因为这样会完全阻止对存储器的访问。为了进一步提升结果，存储器可以在第二个维度上划分为四个更小的存储器：

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

这样会导致四个阵列读取，所有读取使用一个端口在不同存储器上执行：

图 35: **已执行的四个阵列结果**



使用总周期数为 256 * 4 个 = 1024 个周期用于循环 2。

图 36: **性能估算**

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

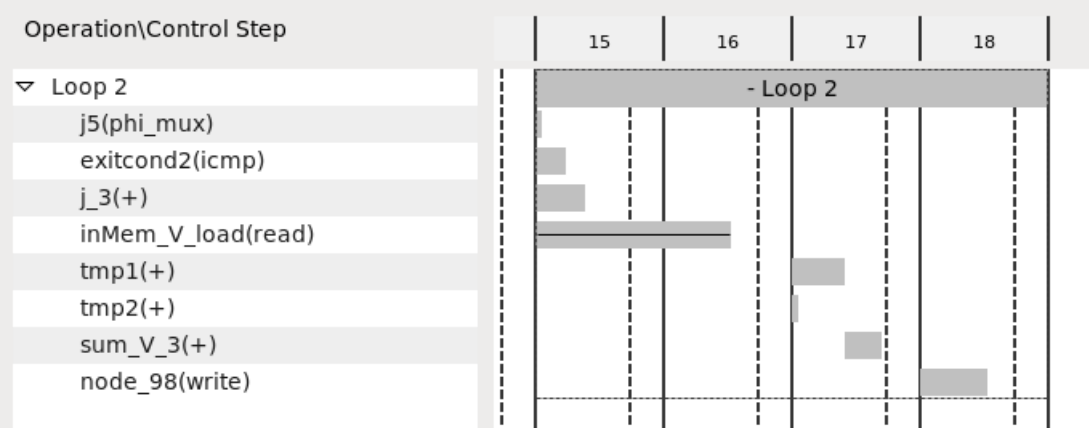
	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	1034	1034	12	1	1	1024	yes
- Loop 2	1024	1024	4	-	-	256	no
- Loop 3	257	257	3	1	1	256	yes

或者，存储器可以重构为带有四个并行字词的单个存储器。这可通过以下编译指示来执行：

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

这样会导致与阵列分区相同的时延，但是只有一个使用单个端口的存储器：

图 37: **时延结果**



虽然其中任何一种解决方案在总体时延和利用率上可创建类似的结果，但是重塑阵列可带来更清洁的接口和更少的布线拥塞，因而使其成为首选解决方案。

注释: 这样即可完成阵列最优化，在实际设计中，可通过利用循环并行化来进一步提升时延（请参阅循环并行化一节）。

```
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 4 章

主机最优化

本章重点讨论主机代码最优化。主机代码使用 OpenCL™ API 来调度各个计算单元的执行以及进出 FPGA 电路板的数据传输。因此，您需要考虑通过 OpenCL 队列进行的并行执行。本节详细讨论了常见错误以及如何识别和处理它们。

减少内核排队的开销

OpenCL API 执行模型支持数据并行和任务并行编程模型。内核通常由 OpenCL 运行时间进行多次排队，然后安排在器件上执行。您必须采用以下两种方式之一发送命令来启动内核：

- 对于数据并行情况，请使用 `clEnqueueNDRange` API。
- 对于任务并行情况，请使用 `clEnqueueTask` API。

分派过程在主机处理器上执行，实际命令和内核实参必须通过 PCIe® 链接发送到 FPGA。在当前赛灵思运行时间 (XRT) 中，分派命令和实参到 FPGA 的开销位于 30us 和 60us 之间，取决于内核上的实参数量。可以通过最小化内核需要执行的次数来降低此开销的影响。

对于数据并行情况，赛灵思推荐您仔细选择主机代码和内核的全局和局部工作规模，以便全局工作规模是局部工作规模的较小倍数。理想情况是，全局工作规模与以下代码片段中显示的局部工作规模相同：

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

对于任务并行情况，赛灵思推荐您将调用最小化为 `clEnqueueTask`。理想情况是，应该在单次调用 `clEnqueueTask` 中完成所有工作量。

数据传输

存储器数据传输类型

FPGA 存储器层级以及基于 PCIe 的主机到计算单元的数据传输，提供了大量的不同数据传输选项。为了获得最佳性能，必须评审这些选项。基本的利弊取舍在于以下三个方面之间：

- 要部署的存储器类型
- 流数据传输
- 计算单元之间没有主机存储器数据传输

注释: 如需了解更多有关使用下节所述的 `--sp` 选项、`--sc` 选项、`xlcbin --info` 命令和 `platforminform` 命令的信息，请参阅 《SDx 命令和工具参考指南》 (UG1279)。

存储器层级

FPGA 加速器卡包括多个不同的存储器层级，可用于主机和内核 (CU) 之间的通信。对于每个存储器层级，从最优化目的考虑，都有不同优点和缺点。

- DDR 存储器: 此存储器在 FPGA 外部。它是可选的最大存储器，但因而也是具有最长的访问时延。
  - 使用（链接标记）：`-sp [kernel|cu].[arg|port]:sptag`

  `sptag` 的值可以通过 `xclbin --info` 或 `platforminfo` 命令进行查找。
- PL 存储器: 此存储器在 FPGA 内部。它通常小于外部 DDR 存储器，但是如果受平台支持，它比 DDR 内存具有更低的时延。
  - 使用（链接标记）：`-sp [kernel|cu].[arg|port]:sptag`

  `sptag` 的值可以通过 `xclbin.info` 和平台信息进行查找。

流数据传输

当流部署在计算单元之间或计算单元与主机之间时，即建立了顺序数据路径。因此，传输的数据仅可按照发送的顺序接收。该工具区分两种类型的流数据传输，一种在计算单元之间，另一种在计算单元和主机之间。

- 内核至内核: 在内核至内核流传输通信情况下，端口通过 `-sc` 选项进行连接：

  使用（链接标记）： `-sc src.port:dst:port`

  根据内核接口上的 `ap_axiu` 数据类型，通过使用 hls::streams，此功能将在 SDAccel 环境中得到支持。这要求必须包含有 `ap_axi_sdata.h` 报头文件。
- 计算单元至主机: 在计算单元和主机之间流传输数据需要 QDMA 队列。因此，此流程需要 QDMA 平台。

  根据内核接口上的 `qdma_axis` 数据类型，通过使用 hls::streams，此功能将在 SDAccel 环境中得到支持。成员字段“last”用于表示指定工作负载完成的时间。这样可以流传输各种长度的通信。如需了解有关主机代码建模的详情，请参阅《SDAccel 环境编程指南》 (UG1277)。

计算单元之间没有主机存储器数据传输

在存储器映射的数据传输中，数据块写入到与端口相关的存储器或从与端口相关的存储器写入。这样读取器和写入器可以在用于通信的存储器缓存内部执行随机访问。通过使用需要相同存储器的进程之间的共享缓存，或在无需存储器交互的情况下在缓存之间即可实施高效复制的 `clEnqueueCopyBuffer` API 的帮助下，SDAccel 环境可在内核之间进行高效的基于缓存的通信。

点对点

SDAccel 开发环境可在单个主机上支持具有多个加速器卡的系统，以便它们共同工作来加速大型软件系统。具体而言，对于不同加速卡之间的通信，该开发环境可支持直接点对点通信。这可通过启用对允许直接通信的其中一个卡 DDR 存储器空间（无需主机存储器）的直接访问来实现。

启用对 DDR 存储器空间的直接访问

以下步骤描述了从本地源缓存到导出目标缓存的建立。但是，在实际修改主机代码之前，启用点对点通信非常重要。此操作通过针对接收器件使用 xbutil "xbutil p2p --enable" 来执行。器件的整个 DDR 地址空间将被映射到主机的 I/O 存储器空间。如需了解更多信息，请参阅《SDx 命令和工具参考指南》 (UG1279)。

此后，必须修改主机代码以为直接的点对点通信做好准备：

1. 使用 XCL_MEM_EXT_P2P_BUFFER 标签创建 buf_dst。

   buf_dst 应该没有任何关联的用户空间缓存（主机缓存）。

2. 使用 `xclGetMemObjectFd` 和 `xclGetMemObjectFromFd` API 导入和导出 buf_dst，上下文：buf_dst_exported。
3. 使用常规的 `clEnqueueCopyBuffer` API (src_command_queue, buf_src, buf_dst_exported, 0,0, buffer_size,,,,) 命令在器件之间复制缓存。

注释: 如步骤 1 和步骤 2 所述，导出的缓存必须为点对点缓存。如果导出的不是点对点缓存且已执行复制操作，则该缓存仍将被复制，但是通过主机存储器进行，而不是直接从卡到卡。卡到卡的传输在配置信息摘要报告中将单独报告。

有关点对点连接的更多信息，可以在 GitHub 上的 XRT 文档中找到。

使用内核计算的重叠数据传输

数据库分析等应用所具有的数据集比加速器件上的可用存储器要大得多。它们要求必须以块的形式传输和处理完整的数据。使用计算来重叠数据传输的技术对于实现这些应用的高性能至关重要。

以下代码片段用内核计算实例显示了赛灵思板载实例 GitHub 的主机类别中源自 OpenCL 重叠数据传输的矢量加法内核。

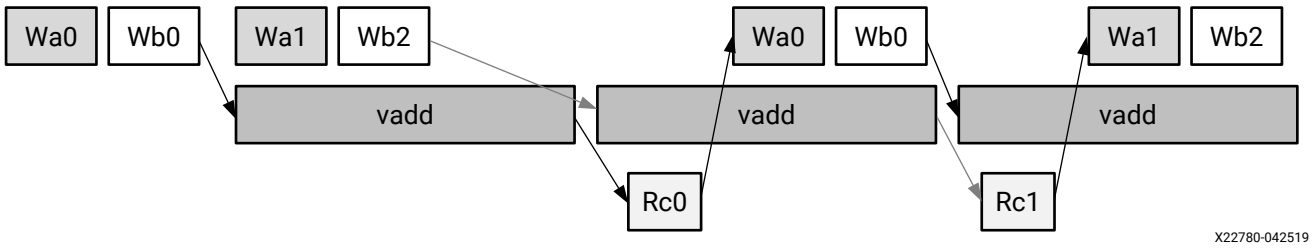
```
kernel __attribute__((reqd_work_group_size(1, 1, 1)))
void vadd(global int* c,
          global const int* a,
          global const int* b,
          const int offset,
          const int elements)
{
    int end = offset + elements;
    vadd_loop: for (int x=offset; x<end; ++x) {
        c[x] =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

对于本示例，主机应用内有四个任务要执行：

1. 写入缓存 a (Wa)
2. 写入缓存 b (Wb)
3. 执行 vadd 内核
4. 读取缓存 c (Rc)

OpenCL 数据传输和内核执行 API 的异步性使数据传输和内核执行可以重叠，如下图所示。在本示例中，所有缓存都使用双重缓存，以便在计算单元处理一组缓存的同时主机可以在另一组缓存上运行。OpenCL 事件对象为建立复杂的操作相依性及同步主机线程和器件操作提供了一种简便的方法。下图中的箭头显示了如何建立事件触发才能实现最优性能。

图 38: **事件触发建立**



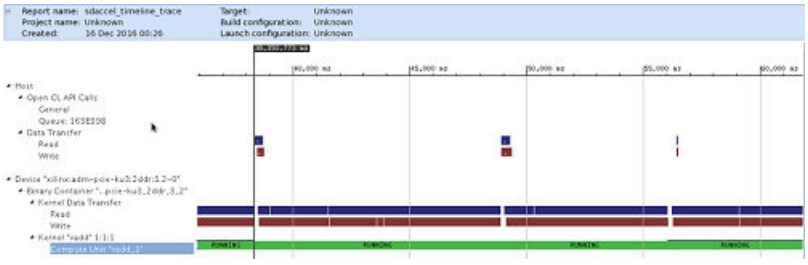
以下主机代码片段将四个任务排队到循环中。它还建立了不同任务之间的事件同步，以确保每个任务的数据相依性都得到满足。双重缓存可通过将不同存储器对象值传递给 `clEnqueueMigrateMemObjects` API 予以建立。事件同步可通过让每个 API 调用等待其他事件以及在 API 完成时触发其自己的事件予以实现。

```
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 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");


    OCL_CHECK(clSetKernelArg(kernel, 0, sizeof(cl_mem),
        &buffer_c[iteration_idx % 2]));
    OCL_CHECK(clSetKernelArg(kernel, 1, sizeof(cl_mem),
        &buffer_a[iteration_idx % 2]));
    OCL_CHECK(clSetKernelArg(kernel, 2, sizeof(cl_mem),
        &buffer_b[iteration_idx % 2]));
    OCL_CHECK(clSetKernelArg(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]));
}
```

“Application Timeline”视图清楚显示了数据传输时间已完全被隐藏，而 `vadd_1` 计算单元正在稳定的运行。

图 39: **“Application Timeline”视图中隐藏的数据传输时间**



缓存存储器分段

存储器缓存的分配和取消分配可能会导致 DDR 中的存储器分段。这可能会导致较差的计算单元性能，即使理论上它们可以并行执行。

当不同计算单元使用多个线程时经常会出现这个问题，此时在线程每次排队内核时会分配和释放具有不同大小的许多器件缓存。这种情况下，时间线追踪将在内核执行之间出现间隙，看起来就像进程处于休眠状态。

由运行时间分配的每个缓存在硬件中应该是连续的。对于较大的存储器，当分配和取消分配许多缓存时，可能需要等待很长时间才能释放该空间。这个问题可以通过分配器件缓存并在内核的不同队列之间重用缓存来解决。

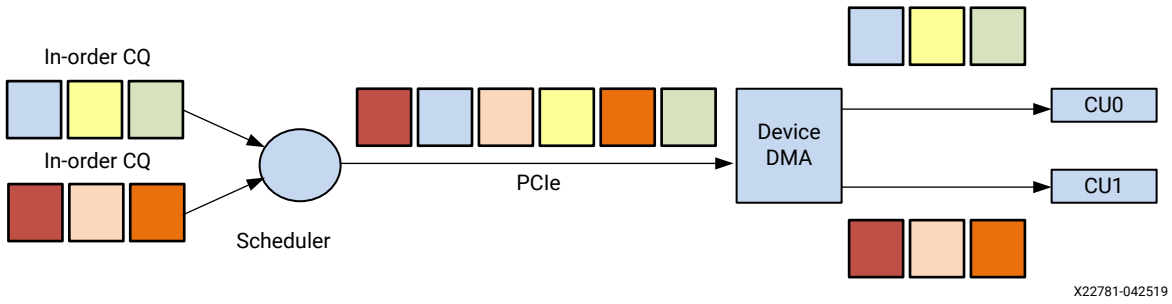
计算单元调度

调度内核运算对于整体系统性能非常关键。当执行多个计算单元（同一个内核或不同内核）时，这变得更加重要。本节将考察负责调度内核的不同命令队列。

多个排序命令队列

下图显示了一个带有两个排序命令队列 CQ0 和 CQ1 的示例。调度器从每个队列中按顺序发送命令，但是调度器可以按任何顺序从 CQ0 和 CQ1 中发出命令。如果需要，您必须管理 CQ0 和 CQ1 之间的同步。

图 40: **两个排序命令队列的示例**



来自位于 GitHub 上 SDAccel 快速入门示例的主机类别内《并行内核执行示例》的以下代码片段建立了多个排序命令队列并将命令排队到每个队列中：

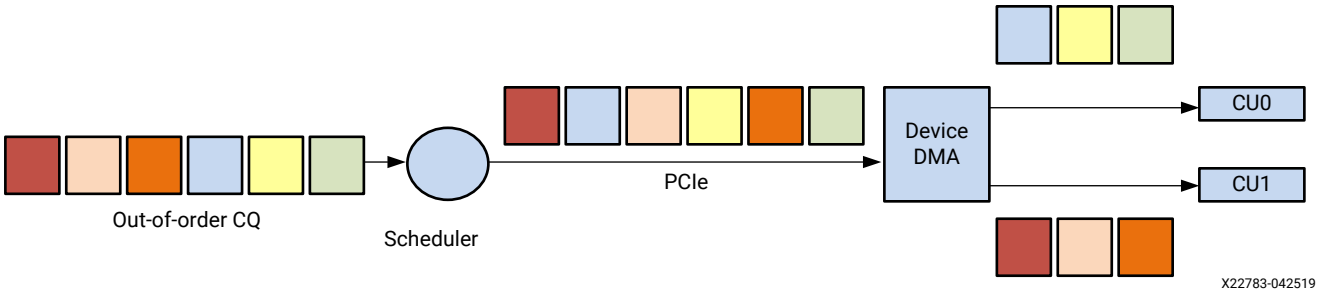
```
cl_command_queue ordered_queue1 = clCreateCommandQueue(
      world.context, world.device_id, CL_QUEUE_PROFILING_ENABLE, &err)

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]);
```

单个无序命令队列

下图显示了带有单个无序命令队列的示例。调度器可以任何顺序从队列中发出命令。如果需要，必须明确建立事件相依性和同步。

图 41: **带单个无序命令队列的示例**



来自 GitHub 上的 SDAccel 快速入门示例中的《并行内核执行示例》的以下代码片段建立了单个无序命令队列并将命令排队到每个队列中：

```
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])
```

下图显示了“Application Timeline”视图，其中计算单元 `mmult_1` 与计算单元 `mscale_1` 和 `madd_1` 并行运行，同时使用多个有序队列和单个无序队列方法。

图 42: **显示 mult_1 与 mscale_1 和 madd_1 并行运行的“Application Timeline”视图**

使用 clEnqueueMigrateMemObjects API 来传输数据

OpenCL 框架可提供大量 API，用于在主机和器件之间传输数据。通常而言，数据传输 API，如 `clEnqueueWriteBuffer` 和 `clEnqueueReadBuffer`，在排队后会将内存对象显式移植到存储器中。但是在数据传输时它们无法做出保证。这就使主机应用很难将内存对象的布局重叠到内核执行计算的器件上。

OpenCL 1.2 框架引入了一种新的 API：`clEnqueueMigrateMemObjects`。使用此 API，可以在依赖命令之前明确执行内存移植。这可使应用通过常规命令队列调度抢先更改内存对象的关联，为另一个即将执行的命令做好准备。这还允许应用在需要内存对象之前，将这些内存对象的布局与其他不相关的操作重叠，从而隐藏传输时延。在由 `clEnqueueMigrateMemObjects` API 关联的事件被标记为 CL_COMPLETE 后，在 mem_objects 中指定的内存对象将被成功移植到与 command_queue 相关的器件中。

`clEnqueueMigrateMemObjects` API 还可在内存对象创建后，用于指导内存对象的初始布局，从而可能避免将使用它的第一个已排队命令上的对象进行实例化的初始开销。

使用 `clEnqueueMigrateMemObjects` API 的另一个优点是可以在单个 API 调用中移植多个内存对象。这可减小一个以上内存对象调度和调用函数来传输数据的开销。

以下代码片段显示了 `clEnqueueMigrateMemObjects` API 的使用情况，该代码段来自 GitHub 上 SDAccel 快速入门示例主机类别中的 《XPR 器件的矢量乘法》示例。

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

XILINX

第 5 章

拓扑结构最优化

本章将重点讲述拓扑结构最优化，并探讨与多个计算单元的大体布局和实现相关的属性及其对性能的影响。

多个计算单元

根据 FPGA 上可用的资源，可以创建同一个内核（或不同内核）的多个计算单元以并行运行，这样可以提升系统的处理时间和吞吐量。

不同内核在 `xocc` 链接行上被作为单独的 .xo 文件提供。可以使用 `--nk` 选项添加多个内核计算单元：

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

注释: 每个单独的内核必须由主机代码分别驱动。

使用多个 DDR bank

在 SDAccel™ 环境中，受支持的加速卡提供一个、两个或四个 DDR bank 和最大 80 GB/s 原始 DDR 带宽。

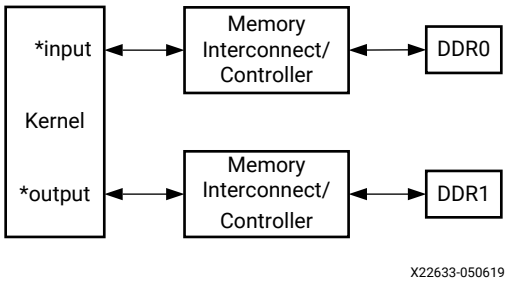
建议: 对于在 FPGA 和 DDR 之间传输大量数据的内核，赛灵思推荐您引导 SDAccel 编译器和运行库使用多个 DDR bank。

除了 DDR bank，主机应用可访问 PLRAM 来将数据直接传输到内核。此功能可通过在兼容的平台上使用 `xocc --sp` 选项来启用。

为了充分发挥 DDR bank 或 PLRAM 的优势，可使用 `--sp` 选项将加速器的单个实参映射到 xclbin 中所需的 DDR bank 或 PLRAM。此映射将被主机可执行程序自动拾取。

以下原理图显示了来自将输入指针连接到 DDR bank 0 和将输出指针连接到 DDR bank 1 的 GitHub 上 SDAccel 快速入门示例的“kernel_to_gmem” 类别的《全局存储器两个 bank 示例》。

图 43: **全局存储器两个 bank**



将内核端口连接到存储器 bank

创建多个 AXI 接口

OpenCL™ 内核、C/C++ 内核和 RTL 内核分配函数给 AXI 接口时具有不同的方法。

- 对于 OpenCL 内核，必须具有 `--max_memory_ports` 选项才能为内核实参上的每个全局指针生成一个 AXI4 接口。AXI4 接口名称基于实参列表上的全局指针的顺序。

  以下代码取自示例 `gmem_2banks_ocl`（该示例位于 GitHub 上 SDAccel 快速入门示例的 `kernel_to_gmem` 类别中）：

  ```
  __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) {
   ...
  }
  ```

  在此示例中，第一个全局指针 `input` 分配的 AXI4 名称为 `xi_gmem0`，第二个全局指针分配的 `output` 名称为 `axi_gmem1`。

- 对于 C/C++ 内核，通过在 HLS INTERFACE 编译指示中为不同全局指针指定不同的“捆绑”名称，可生成多个 AXI4 接口。如需了解更多信息，请参阅《SDAccel 环境编程指南》 (UG1277)。

  以下代码片段来自 `gmem_2banks_c` 示例，该示例将 `input` 指针分配给捆绑 `gmem0` 并将 `output` 指针分配给捆绑 `gmem1`。捆绑名称可以是任何有效的 C 字符串，而生成的 AXI4 接口名称将为 `m_axi_<bundle_name>`。例如，输入指针的 AXI4 接口名为 `axi_gmem0`，而输出指针的接口则为 `m_axi_gmem1`。

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

- 对于 RTL 内核，在 RTL Kernel Wizard 执行导入过程中，会生成端口名称。RTL Kernel Wizard 建议的默认名称为 `m00_axi` 和 `m01_axi`。如果未做更改，当通过 `--sp` 选项分配 DDR bank 时，必须使用这些名称。

将 AXI 接口分配给全局存储器

重要提示! 当使用一个以上 DDR 接口时，赛灵思要求您使用 `--sp` 选项为每个内核/计算单元指定 DDR 存储器 bank，并指定内核布局在哪个 SLR 中。如需了解有关 `--sp` 命令选项的详情，请参阅《SDx 命令和工具参考指南》 (UG1279) 中的 XOCC 命令。如需了解有关 SLR 布局的详情，请参阅《SDAccel 环境用户指南》 (UG1023)。

AXI4 接口使用 `--sp` 选项连接到 DDR bank。`--sp` 选项值的格式为 `<kernel_instance_name>.<interface_name>:<DDR_bank_name>`。

DDR 存储器或替代通信存储器（如 PLRAM 和 HBM）的完整列表可通过 `platforminfo` 命令查找。

注释: 并非所有平台都支持所有全局存储器选项。

以下是一个将输入指针 (`M_AXI_GMEM0`) 连接到 DDR bank 0 并将输出指针 (`M_AXI_GMEM1`) 连接到 DDR bank 1 的命令行示例：

```
xocc --max_memory_ports apply_watermark
--sp apply_watermark_1.m_axi_gmem0:DDR[0]
--sp apply_watermark_1.m_axi_gmem1:DDR[1]
```

您可以使用“Device Hardware Transaction”视图来观察实际的 DDR bank 通信，并分析 DDR 的使用情况。

图 44: **“Device Hardware Transaction”视图 DDR bank 上的传输事务**

将内核分配给 SLR 区域

分配端口给 DDR bank 需要在 FPGA 上将内核物理布线连接到已分配的 DDR。目前，大型 FPGA 使用带超级逻辑区域 (SLR) 的堆叠硅片器件。默认情况下，SDAccel 环境将把计算单元放置在与 shell 相同的 SLR 中。可能不会始终需要这样做，特别是当特定的 DDR bank 可能已在另一个 SLR 区域中使用时。因此，赛灵思推荐使用 `--slr` 选项来映射要向已使用的 DDR 存储器关闭的内核。例如，可以通过应用以下链接选项将 `apply_watermark_1` 内核映射到 SLR 1：

```
xocc -l --slr apply_watermark_1:SLR1
```

要更好地了解 DDR 和 SLR 数量等平台属性，请使用 `platforminfo`。如需了解更多信息，请参阅《SDx 命令和工具参考指南》 (UG1279)。

XILINX

附录 A

示例

为了帮助您快速入门使用 SDAccel™ 环境，GitHub 上的 SDAccel 快速入门示例托管有许多示例，这些示例可展示较好的设计实践、编码指南、常见应用的设计模式，以及最重要的是能够最大限度提高性能的最优化技术。板载示例分为几种主要类别。每种类别具有多种关键概念，在适用时通过 OpenCL™ C 和 C/C++ 的单个示例来展示。所有示例都包含用于运行软件仿真、硬件仿真和在硬件上运行的 Makefile 文件和详细解释该示例的 `README.md` 文件。

XILINX

附录 B

附加资源与法律提示

赛灵思资源

如需了解答复记录、技术文档、下载以及论坛等支持性资源，请参阅赛灵思技术支持。

Documentation Navigator 与设计中心

赛灵思 Documentation Navigator (DocNav) 提供了访问赛灵思文档、视频和支持资源的渠道，您可以在其中筛选搜索信息。DocNav 使用 SDSoC™ 和 SDAccel™ 开发环境安装。打开 DocNav 的方法：

- 在 Windows 中，单击“Start → All Programs → Xilinx Design Tools → DocNav”。
- 在 Linux 命令提示中输入“`docnav`”。

赛灵思设计中心提供了根据设计任务和其他话题整理的文档链接，您可以使用链接了解关键概念以及常见问题解答。访问设计中心：

- 在 DocNav 中，单击“Design Hub View”标签。
- 在赛灵思网站上，查看设计中心页面。

注释: 如需了解更多有关 DocNav 的信息，请参阅赛灵思网站上的 Documentation Navigotor。

参考资料

1. 《SDAccel 环境版本说明、安装和许可指南》 (UG1238)
2. 《SDAccel 环境剖析和最优化指南》 (UG1207)
3. 《SDAccel 环境入门教程》 (UG1021)
4. SDAccel™ 开发环境网页
5. Vivado® Design Suite 文档
6. 《Vivado Design Suite 用户指南：采用 IP 集成器设计 IP 子系统》 (UG994)
7. 《Vivado Design Suite 用户指南：创建和封装定制 IP》 (UG1118)
8. 《Vivado Design Suite 用户指南：部分重配置》 (UG909)

9. 《Vivado Design Suite 用户指南：高层次综合》 (UG902)
10. 《UltraFAST 设计方法指南（适用于 Vivado Design Suite）》 (UG949)
11. 《Vivado Design Suite 属性参考指南》 (UG912)
12. Khronos Group 网页：OpenCL 的文档
13. 赛灵思 Virtex® UltraScale+™ FPGA VCU1525 加速开发套件
14. 赛灵思 Kintex® UltraScale™ FPGA KCU1500 加速开发套件
15. 赛灵思 Alveo™ 网页

请阅读：重要法律提示