查看所有版本

AI 引擎工具和流程用户指南

UG1076 (v2022.2) 2022 年 10 月 19 日

AMD
XILINX

AMD
XILINX

目录

AMD XILINX

第 1 章

概述

Versal® 自适应计算加速平台 (ACAP) 将标量引擎 (Scalar Engine)、自适应引擎 (Adaptable Engine) 和智能引擎 (Intelligent Engine) 与领先的存储器和交互技术有机结合，从而为任何应用提供强大的异构加速功能。最重要的是，Versal ACAP 硬件和软件是专为数据科学家和软硬件开发者开展编程和优化工作而提供的。Versal ACAP 受到诸多工具、软件、资源库、IP、中间件和框架的广泛支持，适用于所有业界标准的设计流程。

基于 TSMC 7 nm FinFET 工艺技术构建的 Versal 产品服务组合提供了一个前所未有的平台，它将软件可编程性和特定领域的硬件加速与自适应能力有机结合在一起，以满足当今快速创新节奏的需求。该产品服务组合包含 6 大器件系列，其架构是专为跨不同市场的各种应用（从云、网络、无线通信到边缘计算和端点）提供可缩放性和 AI 推断功能而构建的。

Versal 架构将不同类型的引擎与丰富的连接和通信功能以及片上网络 (NoC) 相结合，从而支持实现覆盖整个器件的无缝式存储器映射访问。智能引擎包括适用于自适应推断和高级信号处理计算的 SIMD VLIW AI 引擎以及用于定点、浮点和复杂 MAC 运算的 DSP 引擎。自适应引擎将可编程逻辑块与存储器有机结合，它具备专为应对高计算密度需求而设计的架构。标量引擎包含 Arm® Cortex®-A72 和 Cortex-R5F 处理器，支持计算密集型任务。

AI 引擎

Versal AI Core 系列可借助 AI 引擎提供突破性的 AI 推断加速，此 AI 引擎的计算性能较当前服务器级 CPU 高 100 倍。此系列应用范围广泛，包括用于云端动态工作负载以及超高带宽网络，同时还可提供高级安全性功能。AI 和数据科学家以及软硬件开发者均可充分利用高计算密度的优势来加速提升任何应用的性能。鉴于此 AI 引擎所具备的高级信号处理计算能力，它十分适合用于高度优化的无线应用，例如，射频、5G、回程 (backhaul) 和其它高性能 DSP 应用。

AI 引擎是超长指令字 (VLIW) 处理器阵列，具有高度优化的单指令流多数据流 (SIMD) 矢量单元，专用于各种计算密集型应用，尤其是数字信号处理 (DSP)、5G 无线应用和人工智能 (AI) 技术（如机器学习 (ML)）等。

AI 引擎是硬化的块，可提供多级并行处理能力，包括指令级并行处理和数据级并行处理。指令级并行处理包括标量操作：最高 2 次移动、2 次矢量读取（加载）、1 次矢量写入（存储）和 1 条可执行的矢量指令，总计每个时钟周期达 7 路 VLIW 指令。数据级并行处理是通过矢量级操作来实现的，其中每个时钟周期可执行多组数据操作。每个 AI 引擎都包含矢量处理器和标量处理器、专用程序存储器、本地 32 KB 数据存储器、支持访问三个相邻方向内任一方向的本地存储器。它还可访问 DMA 引擎和 AXI4 互连开关，以通过流传输来与其它 AI 引擎进行通信或者与可编程逻辑 (PL) 或 DMA 进行通信。请参阅《Versal ACAP AI 引擎架构手册》(AM009) 以获取有关 AI 引擎阵列和接口的具体详细信息。

第 2 章：AI 引擎架构概述 提供了 AI 引擎架构、工具以及可供内核编程参考的文档的高层次综述。

AI 引擎内核

每个 AI 引擎内核都是 1 个使用专用 API 编写的 C/C++ 程序，专用于处理 VLIW 矢量处理器。AI 引擎内核代码是使用 AI 引擎编译器 (`aiecompiler`) 编译的，此编译器包含在 Vitis™ 核开发套件内。AI 引擎编译器可编译内核以生成在 AI 引擎处理器上运行的 ELF 文件。

AI 引擎 graph

AI 引擎程序包含以 C++ 编写的数据流 graph 规范。此规范可使用 AI 引擎编译器进行编译并执行。自适应数据流 (ADF) graph 应用由多个节点和边缘组成，其中节点表示计算内核函数，边缘则表示数据连接。应用中的内核经编译可在 AI 引擎上或器件的 PL 区域中运行。如需了解有关如何利用 AI 引擎来对 AI 引擎内核与 graph 进行开发、调试和最优化的更多信息，请参阅《AI 引擎内核与 Graph 编程指南》(UG1079)。其中还包含有关专用 graph 构造以及 AI 引擎 graph 的控制方法的信息。

控制 AI 引擎 graph

第 7 章：PS 主机应用编程 描述了创建主机应用来控制系统的 graph 和 PL 内核的进程。在硬件上部署设计时，您可安装驱动以简化通过 PS 上运行的主机应用来初始化和控制 graph 的执行的操作，或者在器件启动时加载并运行 AI 引擎 graph。

AI 引擎编译器在编译 AI 引擎设计 graph 与内核代码的过程中，会生成应用专用的 AI 引擎控制代码。AI 引擎控制代码可以：

- 控制 AI 引擎内核的初始加载。
- 运行 graph 以进行多次迭代、更新与 graph 关联的运行时参数 (RTP)、退出并复位 AI 引擎。

  注释：每个 graph 均可包含多个内核、输入端口和输出端口。所谓 graph 的连接，指的是在内核之间、内核与输入端口之间或者内核与输出端口之间的连接（等同于数据流 graph 中的信号线），并且可作为连接来加以配置。如果 graph 所耗用的数据样本数量与 graph 中内核所期望的数据窗口或数据串流相等，并且所生成的数据样本数量与 graph 中所有内核的输出时所期望的数据窗口或数据串流相同，即表示 graph 完成一轮迭代运行。

Vitis 核开发套件可提供 `xilinx_vck190_base_202220_1` 平台和 `xilinx_vck190_base_dfx_202220_1` 平台，用于以 VCK190 评估板为目标来构建、仿真、调试和部署您的 AI 引擎设计。它支持开发包含 AI 引擎与 PL 内核的设计，设计中所含的主机应用的目标为 PS 中的 Arm 处理器上运行的 Linux 操作系统。在此平台上开发的设计可使用硬件仿真流程来加以验证。这些设计也可在 VCK190 评估板上运行。

程序编译和仿真

第 3 章：对 AI 引擎 Graph 应用进行编译 详细描述了 AI 引擎编译器提供的不同类型的编译、可传入的选项和输入文件以及期望的输出。您可以独立编译 graph 与内核，也可以将其编译包含在更大的系统设计内，并通过设计中的设置，在运行时捕获和剖析事件追踪数据。

第 4 章：对 AI 引擎 graph 应用进行仿真 详细描述了 AI 引擎仿真器以及用于功能仿真的 x86 仿真器。AI 引擎仿真器用于将 graph 应用作为独立实体来进行仿真，或者将其作为更大的系统设计的硬件仿真的一部分来进行仿真。

第 5 章：仿真期间对 AI 引擎 graph 应用进行性能分析 描述了如何在运行硬件仿真构建或硬件构建时，通过执行事件追踪来提取性能数据。此数据可用于对 AI 引擎内核与 graph 进行进一步的最优化。

第 11 章：映射器/布线器方法论 描述了在编译器和/或布线器阶段处理 AI 引擎编译器内的故障时要使用的编译器和布线器方法。

将 AI 引擎 graph 作为 Versal ACAP 系统设计的一部分进行集成和部署

先前步骤中开发的 AI 引擎内核与 graph 可作为更大的 Versal ACAP 系统设计的一部分来使用，此系统设计可由 AI 引擎内核、HLS PL 内核、RTL 内核与主机应用组成。Vitis 编译器负责构建此更大的系统。

如 第 8 章：使用 Vitis 工具流程来集成应用 中所述，您可使用命令行方法来构建系统，或者使用基于 GUI 的方法，如 第 9 章：使用 Vitis IDE 中所述。任一方法都允许您执行仿真来验证设计、在交互式调试环境中调试设计，以及构建设计以供部署在硬件上。

第 8 章：使用 Vitis 工具流程来集成应用 还介绍了赛灵思的 Dynamic Function eXchange (DFX) 平台部署流程，如 以 DFX 平台为目标 中所述。DFX 流程允许您在运行时将 `xclbin` 加载或者重新加载到 DFX 区域内，从而复位 AI 引擎设计和 PL 内核。

在硬件中利用 AI 引擎对设计进行剖析和调试

第 6 章：在硬件上对 AI 引擎 graph 应用进行性能分析 描述了在硬件中运行设计时，如何执行事件追踪来剖析和提取性能数据。

第 10 章：调试 AI 引擎应用 为您演示了如何从命令行或者从 Vitis IDE 运行和使用调试环境。系统性能的评估以及应用调试是达成应用目标的关键步骤。

第 12 章：AI 引擎硬件剖析和调试方法论 描述了在硬件中利用 AI 引擎 graph 来运行 Versal 设计时可使用的五阶段式剖析和调试方法论。

按设计进程浏览内容

赛灵思文档按一组标准设计进程进行组织，以便帮助您查找当前开发任务相关的内容。所有 Versal® ACAP 设计进程的对应设计中心和设计流程助手资料均可在 Xilinx.com 网站上找到。本文档涵盖了以下设计进程：

- 系统和解决方案规划：确认系统级别的组件、性能、I/O 和数据传输要求。包括解决方案到 PS、PL 和 AI 引擎的应用映射。本文档中适用于此设计进程的主题包括：
  - 第 7 章：PS 主机应用编程
  - 第 8 章：使用 Vitis 工具流程来集成应用
- 嵌入式软件开发：从硬件平台创建软件平台，并使用嵌入式 CPU 开发应用代码。还涵盖 XRT 和 Graph API。本文档中适用于此设计进程的主题包括：
  - 第 7 章：PS 主机应用编程
- AI 引擎开发：创建 AI 引擎 Graph 及内核、库用法、仿真调试与剖析以及算法开发。还包含 PL 与 AI 引擎内核的集成。
- 系统集成与确认：集成和确认系统功能性能，包括时序收敛、资源使用情况和功耗收敛。本文档中适用于此设计进程的主题包括：
  - 第 8 章：使用 Vitis 工具流程来集成应用
  - 第 9 章：使用 Vitis IDE
  - 第 10 章：调试 AI 引擎应用

AMD XILINX

第 2 章

AI 引擎架构概述

AI 引擎阵列编程要求充分理解要实现的算法、AI 引擎的功能以及各功能单元之间的整体数据流。AI 引擎阵列支持三种级别的并行度：

- SIMD：通过矢量寄存器，允许并行计算多个元素。
- 指令级别：通过 VLIW 架构，允许在单个时钟周期内执行多项指令。
- 多核：通过 AI 引擎阵列，即可在其中并行执行数百个 AI 引擎。

虽然对于 AI 引擎而言，大部分标准 C 语言代码均可编译，但代码可能需要大幅重构才能在 AI 引擎阵列上实现最优性能。AI 引擎的优势在于，它能够在每个时钟周期内执行矢量 MAC 运算、为下一项运算加载 2 个 256 位矢量、存储来自上一项运算的单个 256 位矢量并递增一个指针或执行另一次标量运算。AI 引擎编译器不会执行任何自动矢量化或基于编译指示的矢量化。代码必须重写后才能使用 SIMD 内部数据类型（例如，`v8int32`）和矢量内部函数（例如，`mac(...)`），而这些都必须在单个流水打拍循环内执行，才能达成最优性能。32 位标量 RISC 处理器具有一个 ALU、部分非线性函数和数据类型转换。每个 AI 引擎所能访问的存储器量是有限的，这表示大型数据集需要分区。

AI 引擎内核即 AI 引擎上运行的函数，这些函数构成数据流 graph 规范的基本构建块。数据流 graph 是 Kahn 进程网络，具有确定性行为，不依赖于各种计算延迟或通信延迟。AI 引擎内核声明为空的 C/C++ 函数，通过提取窗口实参或串流实参来建立 graph 连接。内核还可包含静态数据和运行时参数实参，包括异步实参或触发实参。每个内核都应在其自己的源文件内定义。

为了达成总体系统性能，必须广泛阅读了解有关架构、分区、AI 引擎数据流 graph 生成和数据流连接最优化等方面的知识，积累相关经验。如需了解更多详细信息，请参阅《Versal ACAP AI 引擎架构手册》(AM009)。

赛灵思提供了 DSP 和通信库，其中包含经最优化的代码，可用于 AI 引擎，应尽可能多加利用。其中提供的源代码也提供了强大的资源，供用户学习有关 AI 引擎内核编码的信息。

AI 引擎组件

AI 引擎阵列由二维 AI 引擎拼块 (tile) 阵列构成，其中每个 AI 引擎拼块均包含一个 AI 引擎、存储器模块和拼块互连模块。

- AI 引擎：每个 AI 引擎都是一个超长指令字 (VLIW) 处理器，其中包含一个标量单元、一个矢量单元、两个加载单元和一个存储单元。
- AI 引擎拼块：每个 AI 引擎拼块都包含一个 AI 引擎、一个本地存储器模块，搭配多条通信路径以促进拼块间的数据交换。
- AI 引擎阵列：AI 引擎阵列是指 AI 引擎拼块的完整二维阵列。
- AI 引擎程序：AI 引擎程序包含以 C/C++ 编写的数据流 graph 规范。该程序是使用 AI 引擎工具链来编译并执行的。
- AI 引擎内核：内核是使用 AI 引擎矢量数据类型和内部函数以 C/C++ 编写的。这些函数均为 AI 引擎上运行的计算函数。内核构成数据流 graph 规范的基本构建块。

- ADF graph：ADF graph 是具有单个 AI 引擎内核或多个 AI 引擎内核（以数据串流连接）的网络。它凭借如下特定构造来与可编程逻辑、全局存储器和处理器系统进行交互，此类构造有：PLIO（graph 编程中的端口属性，用于建立往来可编程逻辑的串流连接）、GMIO（graph 编程中的端口属性，用于建立往来全局存储器的外部存储器映射连接）和 RTP。

工具

Vitis 集成设计环境

Vitis™ 集成设计环境 (IDE) 可用于为赛灵思器件执行系统编程，包括含多个 AI 引擎内核的 Versal® 器件。在此工具中，有下列功能特性可用。

- 最优化 C/C++ 编译器，用于编译内核与 graph 代码，执行所有必要的连接、布局和检查以确保器件上的各项功能正常工作。
- 周期近似的仿真器、加速的功能仿真器和剖析工具。
- 调试环境，可在仿真环境和硬件环境下使用。

Vitis 命令行工具

命令行工具可用于构建、仿真以及生成输出文件和报告。捕获由 IDE 生成的命令行输出后，有助于后续集成到客户构建环境内。Vitis 分析器 IDE 可用于查看报告并分析由命令行工具生成的输出文件和报告。

Vitis Model Composer

Vitis™ Model Composer 可提供基于 MATLAB® 和 Simulink® 的高层次图形输入环境，用于包含 AI 引擎、HLS 和 RTL 组件的设计仿真与代码生成。如需了解更多信息，请参阅《Vitis Model Composer 用户指南》(UG1483)。

- 将 AI 引擎内核、graph、HLS 内核与基于 RTL 的块导入同一个 Simulink® 设计，用于快速协同仿真。
- 从 Simulink 库浏览器中，将经过最优化的 AI 引擎函数（例如，有限脉冲响应 (FIR) 和 FFT 滤波器）拖放到设计中。
- 使用 MATLAB 或 Simulink 中生成的激励来验证设计、直观显示结果并将结果与黄金参考结果进行比较。生成 graph 代码并测试矢量。
- 汇编导入的代码和块库代码以供馈送到下游工具中。

文档

以下链接对于 AI 引擎内核开发和编程很实用。

- 《Versal ACAP AI 引擎内部函数文档》(UG1078) 提供了当前版本中支持的所有内部调用 API 和数据类型的列表。对于 AI 引擎支持的所有内部调用 API 和数据类型，它都可作为行之有效的参考指南来使用。
- 《AI 引擎内核与 Graph 编程指南》(UG1079)
- 《Versal ACAP AI 引擎架构手册》(AM009)
- 《Chess 编译器用户手册》提供了所有“编译指示”和函数的列表，以帮助您对 AI 引擎内核代码进行最优化。此文档可在 AI 引擎专区中找到。

- 《Versal ACAP AI 引擎寄存器参考资料》(AM015)
- 《AI 引擎 API 用户指南》(UG1529)
- 《Vitis Model Composer 用户指南》(UG1483)

AMD XILINX

第 3 章

对 AI 引擎 Graph 应用进行编译

本章描述了传递给 AI 引擎编译器 (`aiecompiler`) 的所有命令行选项。它提取数据流 graph 代码和各内核代码来生成可在各 AI 引擎目标平台（例如，仿真器和 AI 引擎器件）上运行的镜像。AI 引擎编译器会对 graph 进行静态编译，并在 AI 引擎内对内核进行映射和布局。

提示：除非另行指定，否则所有输入文件路径都与当前目录有关，所有输出文件路径都与 `Work` 目录有关。

AI 引擎 graph 和内核均可单独编译，或者可作为独立应用来进行编译，此独立应用在 AI 引擎处理器阵列内通过仿真或硬件来运行。graph 与内核还可包含在较大的系统设计中使用，此类设计将 AI 引擎 graph 与 Versal® 器件上运行的 ELF 应用以及器件的可编程逻辑内运行的可编程逻辑 (PL) 内核加以集成。AI 引擎编译器用于编译 graph 与内核，包括作为独立配置和包含在更大的系统内的 graph 与内核。

如 第 9 章：使用 Vitis IDE 所示，Vitis™ IDE 可用于创建和管理工程构建设置并运行 AI 引擎编译器。或者，您可从命令行构建工程（如 第 8 章：使用 Vitis 工具流程来集成应用 中所述）或者在脚本或 Makefile 中构建工程。上述任一方法都允许您执行仿真来验证 graph 应用或集成系统设计、在交互式调试环境下调试设计并构建设计以供在硬件上运行和部署。无论您采用任何方法来使用这些工具，都应首先设置环境。

设置 Vitis 工具环境

AI 引擎工具是作为 Vitis 统一软件平台的一部分来交付和安装的。因此，准备运行 AI 引擎工具时，以 AI 引擎编译器和 AI 引擎仿真器为例，必须设置 Vitis 工具。Vitis 统一软件平台包含 2 个必要组件，这 2 个组件必须完成安装和配置并搭配有效的 Vitis 工具许可证才能正常工作。

- Vitis 工具和 AI 引擎工具
- 目标 Vitis 平台，例如，用于 AI 引擎应用的 `xilinx_vck190_base_202220_1` 平台

如需了解更多信息，请参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393)。

安装 Vitis 软件平台的组件时，请运行以下脚本以将环境设置为在特定命令 shell 下运行。

```
#setup XILINX_VITIS and XILINX_HLS variables
source <Vitis_install_path>/Vitis/<version>/settings64.csh
```

提示：`settings64.sh` 和 `setup.sh` 脚本同样在相同目录中提供。

最后，使用以下环境变量定义可用目标平台的位置，以搭配 Vitis IDE 和 AI 引擎工具一起使用：

```
setenv PLATFORM_REPO_PATHS <path to platforms>
```

提示：`PLATFORM_REPO_PATHS` 环境变量指向包含平台文件 (XPFM) 的目录。这样您只需使用平台的文件夹名称即可指定平台。

您可使用以下任一命令来确认工具的安装和设置。

```
which vitis
which aiecompiler
```

您可使用以下命令来确认平台安装。

```
platforminfo --list
```

输入

AI 引擎编译器会提取多种形式的输入，并生成可执行应用以供在 AI 引擎器件上运行。用于运行 AI 引擎编译器的命令行如下：

```
aiecompiler [options] <Input File>
```

其中：

- `<Input File>` 用于指定数据流 graph 代码，此代码用于为 AI 引擎 graph 定义 `main()` 应用。输入流 graph 是使用数据流 graph 语言来指定的。如需了解数据流 graph 的描述，请参阅《AI 引擎内核与 Graph 编程指南》(UG1079) 中的创建数据流 graph（包含内核）。

AI 引擎编译器命令示例：

```
aiecompiler --verbose --pl-freq=100 --workdir=./myWork --
platform=xilinx_vck190_base_202220_1.xpfm\
--include="./" --include="./src" --include="./src/kernels" --include="./data" --include="$
{XILINX_HLS}/include"  \
./src/graph.cpp
```

命令行还包含其它输入选项，例如：

- `--constraints=<jsonfile>` 用于指定约束，例如，位置或布局边界框。

输出

默认情况下，AI 引擎编译器会将所有输出都写入名为 `Work` 的目录和 `libadf.a` 文件，其中，`Work` 是启动工具的当前目录的子目录，`libadf.a` 文件则用作为 Vitis 编译器的输入，此文件是在启动 AI 引擎编译器的相同目录中创建的。输出的类型和输出目录的内容取决于指定的 `--target`，如 AI 引擎编译器选项 中所述。如需了解有关 Vitis 编译器的更多信息，请参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的 Vitis 编译器命令。

提示：您可使用 `--workdir` 选项指定不同的输出目录。

下表中描述了 `./Work` 目录的结构和内容。

表 1：**工作目录结构**

目录/文件	描述
./Work/	
<name>.aiecompile_summary	这是一个生成文件，可在 Vitis 分析器中将其打开以查看编译汇总信息。
config/scsim_config.json	该 JSON 脚本用于为 SystemC 仿真器指定选项。它包含 AI 引擎阵列拼块几何结构、输入/输出文件规范及其与串流开关的连接。
arch/	
logical_arch_aie.larch	该 JSON 文件旨在描述 AI 引擎应用的硬件要求。
aieshim_constraints.json	该 JSON 文件如果存在，即表示 AI 引擎阵列与通过 AI 引擎应用所提供的可编程逻辑之间的用户定义的物理接口约束。
aieshim_solution.aiesol	该 JSON 文件旨在描述从逻辑通道到物理通道的映射，这些通道跨越 AI 引擎阵列与可编程逻辑之间的接口。
cfgraph.xml	该 XML 文件旨在描述 AI 引擎应用的硬件要求。此文件供 Vitis 工具流程使用。
aie/	
Makefile	此 Makefile 文件用于为所有 AI 引擎编译代码。
array and programmable<n>_<m>/	这些目录是各个 AI 引擎编译目录。
Release/	Synopsys 版本目录，适用于包含 ELF 文件的 AI 引擎。
<n>_<m>.lst	内核微码，位于 <n>_<m>。
<n>_<m>.map	用于显示位于 <n>_<m> 的内核的存储器映射。它还包含存储器大小、宽度和偏移。
scripts/	Synopsys 编译器工程和连接器脚本。
src/	处理器（包含内核与 main）的源文件。
ps/c_rts/	此目录包含基于 C 语言的运行时协议，用于对 PS 交互进行建模。
aie_control.cpp	这是 AI 引擎控制代码，它是通过为程序中存在的特定 graph 对象实现 init、run、end graph API 后所生成的。此文件与 main 应用相链接，为仿真器和裸机创建 PS 线程。
aie_control_xrt.cpp	这是 AI 引擎控制代码，它是通过为程序中存在的特定 graph 对象实现 init、run、end graph API 后所生成的。此文件与 main 应用相链接，为 Linux 应用创建 PS 线程。
systemC/	此目录包含 PS main 的 SystemC 模型。
Makefile	此 Makefile 文件用于编译所有 PS SystemC 模型。
generated-source/	其中包含 SystemC 封装文件，用于 PS main。
generated-objects/	包含已编译的共享库，用于 PS main。
ps/cdo/	此目录包含生成器代码，用于以配置数据对象格式 (CDO) 进行 graph 配置和初始化。可在 SystemC-RTL 仿真和实际硬件执行期间使用。
Makefile	此“Makefile”用于编译 graph CDO
generateAIEConfig	此 bash 脚本用于构建 graph CDO
generated-sources/	包含用于生成 CDO 的 C++ 程序。
generated-objects/	包含用于生成 CDO 的已编译程序。
pthread/	
PthreadSim.c	这表示从输入数据流 graph 到 C 语言程序的源码到源码转换，使用 pthreads 来实现。
sim.out	GCC 编译的二进制文件，适用于 PthreadSim.c。

表 1：**工作目录结构** (续)

目录/文件	描述
reports/	
<graph>_mapping_analysis_report.txt	此映射报告用于描述如何将内核分配给 AI 引擎以及如何将窗口缓冲器分配给 AI 引擎存储器组。
<graph>.png	此 bitmap 文件用于显示基于 AI 引擎的内核 graph 连接和分区。
<graph>.xpe	此 XML 文件用于描述基于所使用的硬件资源估算的 graph 功耗剖析。此文件可搭配 Xilinx® Power Estimator (XPE) 工具来使用。
sync_buffer_address.json	显示内核同步缓冲器地址，并包含局部和全局地址。
lock_allocation_report.json	描述端口以及与内核关联的锁定和缓冲器。
dma_lock_report.json	显示 AI 引擎的输入/输出的 DMA 锁定及其连接到的内核（含缓冲器信息）。
temp/	此目录包含由 AI 引擎编译器生成的部分临时文件，可用于调试。此外，此处默认还会创建 CF graph .o 文件。

AI 引擎编译器选项

表 2：**AI 引擎选项**

选项名称	描述
--constraints=<string>	约束（位置、边界框等）可使用 JSON 文件指定。该选项支持您指定一个或多个约束文件。
--heapsize=<int>	每个 AI 引擎使用的堆大小（以字节为单位） 对于栈、堆和同步缓冲器（32 字节，包括 graph 运行迭代数信息），可分配最多 32768 字节的数据存储器。默认堆大小设置为 1024 字节。将堆大小更改为其它值之前，请确保栈、堆和同步缓冲器大小总和不超过 32768 字节。 该选项用于分配文件作用域内的任意剩余数据，这些数据在用户 graph 中均未显式连接。
--stacksize=<int>	每个 AI 引擎使用的栈大小（以字节为单位） 对于栈、堆和同步缓冲器（32 字节），可分配最多 32768 字节的数据存储器。默认栈大小设置为 1024 字节。将栈大小更改为其它值之前，请确保栈、堆和同步缓冲器大小总和不超过 32768 字节。 该选项用作为标准编译器调用约定，包括栈分配的局部变量和寄存器溢出。
--pl-freq=<value>	为所有 PLIO 指定接口频率（以 MHz 为单位）。默认频率为 AI 引擎频率的四分之一，支持的最大频率为 AI 引擎频率的一半。在 graph 中提供了每个接口特定的 PL 频率。
--pl-register-threshold=<value>	为寄存的 AI 引擎到 PL 交汇指定频率（以 MHz 为单位）阈值。默认频率为 AI 引擎频率的八分之一，视特定器件速度等级而定。 **注释：**如果值高于 AI 引擎阵列频率的四分之一，则将被忽略，并改用四分之一值。

表 3：**CDO 选项**

选项名称	描述
--broadcast-enable-core	使用广播启用与 graph 关联的所有 AI 引擎。该选项会在阵列中保留一条广播通道，用于核的使能。默认值为 true。

表 4：**编译器调试选项**

选项名称	描述
`--adf-api-log-level=<value>`	ADF API log 日志级别。可用的值如下： 0：错误 1：级别 0 + 警告 2：级别 1 + 参考消息 3：级别 2 + 调试消息 默认值为 2。
`--kernel-linting`	执行 graph 与内核之间的一致性检查。默认为 false。
`--known-tripcount`	将未知循环次数转换为已知循环次数。
`--quiet`	禁止 AI 引擎编译器输出。
`--verbose`	AI 引擎编译器详细输出，在编译各阶段发射编译器消息。这些调试和追踪 log 日志可提供有关编译进程的实用消息。

表 5：**设计规则检查选项**

选项名称	描述
`--drc.disable=<string>`	禁用对指定 ID 执行设计规则检查。不执行已禁用的检查。
`--drc.enable=<string>`	启用对指定 ID 执行设计规则检查。
`--drc.severity=<string>`	更改设计规则检查的严重性：`format <ID>:<severity>[:context]`。
`--drc.waive=<string>`	豁免对指定 ID 执行设计规则检查。豁免检查仍需执行，但会将其标记为豁免。

表 6：**执行目标选项**

选项名称	描述
`--target=<hw|x86sim>`	AI 引擎编译器支持多种构建目标。默认值为 `hw`。 • `hw` 目标会生成 `libadf.a` 以供在目标平台上的硬件器件中和在硬件仿真中使用。 • `x86sim` 目标会编译代码以供在“x86 simulator”（x86 仿真器）中使用，如 x86 功能仿真器 中所述。

表 7：**文件选项**

选项名称	描述
`--include=<string>`	该选项可用于在 include 路径内包含额外目录，以供编译器前端处理。 指定一个或多个 include 目录。
`--output=<string>`	为输入数据流 graph 文件指定由前端生成的 `output.json` 文件。此输出文件会传递至后端，用于 AI 引擎器件的映射和代码生成。对于其它类型的输入，忽略此选项。
`--output-archive=<string>`	指定输出存档名称，其中将包含已编译的 AI 引擎工件。默认值为 `libadf.a`。
`--platform=<string>`	指向 Vitis 平台文件的路径，执行硬件设计及其 RTL 协同仿真时，此平台文件用于定义可用的硬件和软件组件。可采用平台规范 (XPFM) 格式或硬件规格 (XSA) 格式。
`--workdir=<string>`	默认编译器将所有输出都写入当前目录中名为 `Work` 的子目录。该选项可用于指定其它输出目录。

表 8：**常规选项**

选项名称	描述
`--help`	列出可用的 AI 引擎编译器选项，按此处所列出的分组进行排序。
`--help-list`	显示 AI 引擎编译器选项的字母列表。
`--version`	显示 AI 引擎编译器的版本。

表 9：**其它选项**

选项名称	描述
`--disable-multirate`	禁用 ADF graph 中的多重速率。默认为 false。
`--no-init`	该选项用于禁用 AI 引擎数据存储器中的窗口缓冲器的初始化。该选项能够加速将二进制镜像加载到 SystemC-RTL 协同仿真框架中。默认为 false。 **提示：**这不影响静态初始化的查找表。
`--nodot-graph`	默认情况下，AI 引擎编译器会生成 `.dot` 和 `.png` 文件，以在 AI 引擎上直观显示用户指定的 graph 及其分区。该选项可用于消除 DOT graph 输出。默认为 false。

表 10：**模块专用选项**

选项名称	描述
`--Xchess=<string>`	该选项可用于将内核专用选项传递给 CHESS 编译器，此编译器用于为每个 AI 引擎编译代码。 选项字符串指定为 `<kernel-function>:<optionid>=<value>`。在映射指定内核函数的 AI 引擎上编译生成的源文件期间包含该选项字符串。
`--Xelfgen=<string>`	该选项可用于将附加命令行选项传递给编译器的 ELF 生成阶段，此阶段当前作为 `make` 命令来运行，用于构建所有 AI 引擎 ELF 文件。 例如，为了将并行编译数量限制为 4，请写入 `-Xelfgen="-j4"`。 **注释：**如果编译期间，log 日志中出现 `bad_alloc` 错误，或者如果 Vitis IDE 崩溃，可能是由于工作站上内存不足而导致的。除了增大机器上可用内存外，还有一种可行的变通方法是在代码生成阶段限制编译器使用的并行度。具体方法是在 GUI 中指定编译器 CodeGen 选项 `-j1` 或 `-j2`，或者在命令行上指定 `-Xelfgen=-j1` 或 `-Xelfgen=-j2`。
`--Xmapper=<string>`	该选项可用于将其它命令行选项传递到编译器的映射器阶段。例如： `--Xmapper=DisableFloorplanning` 如果设计在映射或布线阶段无法收敛，或者如果您要尝试通过减少存储体冲突来提升性能，则可尝试这些选项。 请参阅 映射器和布线器选项 以获取选项列表及其描述。
`--Xpreproc=<string>`	该选项可用于将常规选项传递到“PREPROCESSOR”阶段，用于执行所有源码编译 (AIE/PS/PL/x86sim)。例如： `--Xpreproc=-D<var>=<value>`
`--Xpslinker=<string>`	该选项用于将常规选项传递到“PS LINKER”阶段。例如： `--Xpslinker=-L<libpath> -l<libname>`

表 10：**模块专用选项** (续)

选项名称	描述
`--Xrouter=<string>`	该选项用于将常规选项传递到“ROUTER”阶段。例如： `-Xrouter=dmaFIFOsInFreeBankOnly`
`--fast-floats`	该选项支持快速实现线性浮点标量运算，例如，add、sub、mul 和 compare。
`--fast-nonlinearfloats`	该选项支持快速实现非线性浮点标量运算，例如，sine/cosine、sqrt 和 inv。
`--fastmath`	该选项支持快速实现 float2fix、fplt 和 fpge。

注释：在 AI 引擎 graph 的后续编译中，仅对已修改的 AI 引擎进行重新编译。任何未修改的内核都不会进行重新编译。

表 11：**事件追踪选项**

选项名称	描述
`--event-trace=<value>` 其中，`<value>` 为以下值之一： • `functions` • `functions_partial_stalls` • `functions_all_stalls` • `runtime`	事件追踪配置值。其中指定的 `<value>` 表示： • 不含停滞的函数转换视图。 • 含串流/锁定/级联停滞的函数转换视图。 • 含所有停滞（串流/锁定/级联/存储器）的函数转换视图。 • 运行时事件追踪配置。
`--event-trace-mem-tile` 其中 `<value>` 是 L1	存储器拼块的事件追踪配置级别 • L1：运行事件的 DMA 端口。
`--event-trace-port=<value>` • `plio` • `gmio`	设置 AI 引擎事件追踪端口。默认值为 `gmio`；赛灵思建议使用 `gmio` 作为 event-trace-port（事件追踪端口）配置。如需了解更多信息，请参阅 事件追踪构建流程。 • 将 AI 引擎事件追踪端口设置为 `plio` • 将 AI 引擎事件追踪端口设置为 `gmio`
`--num-trace-streams=<int>`	表示追踪串流的数量。默认值为 16。
`--trace-plio-width=<int>`	表示追踪串流的 PLIO 宽度。默认值为 64。允许的值为 32 和 64。

表 12：**最优化选项**

选项名称	描述
`--xlopt=<int>`	基于 `opt` 级别启用内核最优化组合。允许的值为 0 到 2；默认值为 1。 • `xlopt=1` ○ 自动编译堆大小：使用内核分析来自动计算每个 AI 引擎的堆要求，由此保证易用性。因此，您无需指定堆大小。 ○ 指导信息：通过提供指导信息来高亮未对齐的变量、可能由映射器分配的全局阵列、restrict 的不当使用以及可能出现的先读取后写入冲突。 ○ 编译指示插入：在内核代码中自动推断和插入编译指示。 • `xlopt=2` ○ 自动内联：如果可自动内联函数，则执行此操作，即使这些函数未声明为 `__inline` 或 `inline` 也是如此。 ○ 对展开的循环进行循环剥离：通过剥离，使循环迭代次数成为展开因子的倍数。基于循环的迭代次数和利益启发，将循环拆分为多个循环，并在拆分的循环上添加平铺编译指示。 **注释：**编译器最优化 (`xlopt > 0`) 可能降低调试可视性。
`--Xxloptstr=<string>`	用于在 `xlopt` 级别 1 和 2 中启用/禁用最优化的选项字符串。 • `-annotate-pragma=false`：关闭循环编译指示的自动插入 • `-xlinline-threshold=T`：将自动内联阈值设置为 T（默认 T = 5000） • `-annotate-pragma`：自动插入循环展开、流水打拍和平铺编译指示（默认值 = true）

注释：两个保留码字：`aie` 和 `adf` 在 graph 编程中均为无效的名称空间标识符。

映射器和布线器选项

表 13：**映射器选项**

选项	描述
`DisableFloorplanning`	该选项会在映射器中禁用 auto-floor-planning（自动布局规划）阶段。该选项对于密集约束的设计很有用，在此类设计中，位置约束用于指引映射阶段。
`BufferOptLevel[1-9]`	这些选项可用于通过减少存储体冲突来改善吞吐量。如果“BufferOptLevel”较高，映射器会尝试减少映射到相同存储体的缓冲器数量，从而降低影响整体性能的存储体冲突的概率。提高“BufferOptLevels”会导致总体映射区域增大，在极端罕见情况下甚至可能无法找到解决方案。“BufferOptLevels”默认值为 `BufferOptLevel0`。
`disableSeparateTraceSolve`	默认追踪行为会强制 AI 引擎映射器在使用追踪调试功能特性时，终止原始设计位置中的所有 PLIO/GMIO。但如果原始解决方案没有保留任何空间用于追踪 GMIO，那么除非移动设计 PLIO，否则将无解决方案可用。在此类情况下即可使用该选项。

注释：您可以在下一次编译时重新循环先前设计布局。这样即可显著缩短映射器运行时间。运行编译器时，它会在 `Work/temp` 目录中生成布局约束文件 `graph_aie_mapped.aiecst`。赛灵思建议您保存 `Work/temp/graph_aie_mapped.aiecst`，以便在后续编译中使用，因为每次重新编译都会重新生成 `Work` 文件夹。在命令行上可以为下一次迭代指定此约束文件。

```
aiecompiler --constraints Work/temp/graph_aie_mapped.aiecst src/graph.cpp
```

提示：映射器并不知晓每核 16K 程序存储器的限制。有一种变通方法是更改运行时使用规范，将内核映射到不同的核。

表 14：**布线器选项**

选项	描述
`dmaFIFOsInFreeBankOnly`	该选项可确保 DMA FIFO 仅插入不含任何其它已映射的缓冲器的存储体。如果观测发现，由于同时访问 DMA FIFO 缓冲器和同一存储体中布局的某些其它设计缓冲器，导致存储器停滞，即可使用该选项。
`disableSSFifoSharing`	禁用布线器在任一信号线的两个或两个以上终端之间共享串流开关 FIFO 的能力。仅当器件中没有足够串流开关 FIFO 可用于为每个终端提供其自己的专用 FIFO 时，才应使用该选项。
`disablePathBalancing`	禁用布线器向信号线添加额外 FIFO 以平衡再收敛的路径之间的时延的能力。

在 Vitis 分析器中查看编译结果

完成 AI 引擎 graph 编译后，AI 引擎编译器会编写名为 `<graph-file-name>.aiecompile_summary` 的编译结果汇总报告，可在 Vitis 分析器内查看。此汇总包含各报告的集合，以及反映已编译的构建中实现的 AI 引擎应用的状态。此汇总报告会写入 AI 引擎编译器的工作目录，该目录由 `--workdir` 选项指定，默认为 `./Work`。

要打开 AI 引擎编译器汇总，请使用以下命令：

```
vitis_analyzer ./Work/graph.aiecompile_summary
```

这样会打开 Vitis 分析器并显示报告的“Summary”（汇总）页面。“Report Navigator”（导航器报告）视图会列出“Summary”中可用的不同报告。如需获取 Vitis 分析器的完整信息，请参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的使用 Vitis 分析器。

列出的报告包括：

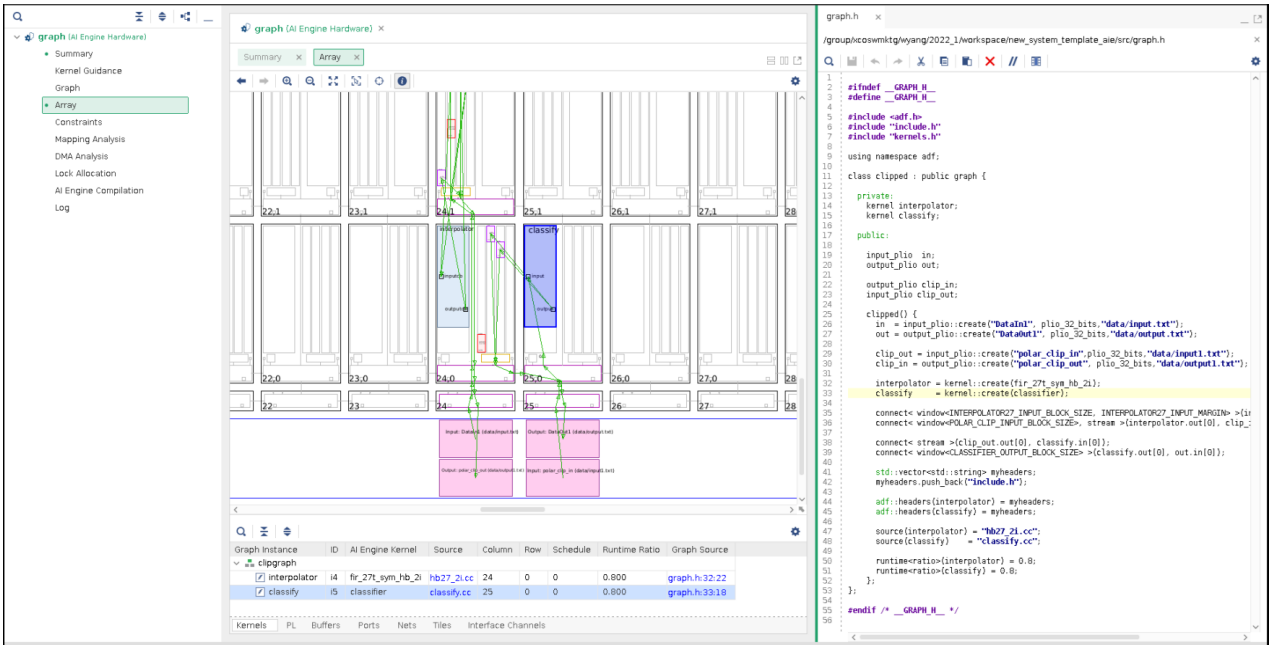
- “Summary”（汇总）：这是顶层报告，用于报告构建的详细信息，例如，创建构建的日期、工具版本、graph 链接以及使用的命令行。
- “Kernel Guidance”（内核指南）：显示各种消息，提供有关内核最优化的指导信息。
- “Graph”：提供 AI 引擎 graph 流程图，显示流经各种内核的数据流。您可按需放大和平移 graph 显示。在“Reports”（报告）视图底部提供了一张表，其中汇总了 graph 以及内核、缓冲器、端口和信号线相关的信息。单击 graph 图中的对象即可在表中高亮所选对象。（请参阅 graph 和阵列详细信息）。
- “Array”（阵列）：提供 Versal 器件上的 AI 引擎处理器阵列的图形表示法。graph 内核与连接布局在阵列上下文内。您可以放大并选择阵列图中的元素。选中阵列中的对象也会在“Reports”视图底部的表格中高亮所选对象。

  注释：“Graph”报告和“Array”报告共享相同的表格。选中任一视图中的项时，会同时选中另一视图中的该项。例如，选中“Graph”视图中的信号线就会在“Array”视图中选中同一条信号线。
- “Constraints”（约束）：显示 graph 内使用的所有约束以及来自 .json 约束文件的所有约束。
- “Mapping Analysis”（映射分析）：显示文本报告 `graph_mapping_analysis_report.txt`。报告 graph 到器件资源的块映射、端口映射和存储体映射。
- “DMA Analysis”（DMA 分析）：显示文本报告 `DMA_report.txt`，提供来自 graph 的 DMA 访问汇总信息。
- “Lock Allocation”（锁定分配）：显示文本报告 `Lock_report.txt`，监听端口实例上的 DMA 锁定。

- “AI Engine Compilation”（AI 引擎编译）：显示单一内核编译 log 日志文件。

下图显示了 Vitis 分析器中打开的 `graph.aiecompile_summary` 报告，其中显示了“Array”图、该图和表格视图中选中的 AI 引擎内核，并在“Source Code”（源代码）视图中显示了内核的源代码。

图 1：**Vitis 分析器 graph 汇总信息**



graph 和阵列详细信息

在 Vitis 分析器的 graph 和阵列视图中，有多个表格着重讲解 graph 与内核的详细信息。以下章节提供了每个表的详细描述以及每个不同表中的列中提供的信息。

内核

“Kernels”（内核）表显示了有关 ADF graph 所使用的内核的详细信息。例如，下表显示了两个内核：“interpolator”（插入程序）和“classify”（分类）。以下代码示例显示了 `fir_27t_sym_hb_2i` 和 `classifier` 内核函数例化为 graph 中的内核的过程。

```
interpolator = kernel::create(fir_27t_sym_hb_2i);
classify = kernel::create(classifier);
```

图 2：**内核表**

表 15：**列描述**

列	描述
“Graph Instance” （graph 实例）	显示设计 graph 的分层视图以及子 graph 与内核。
“ID”	从 `aiecompiler` 提供给内核的唯一 ID。
“AI Engine Kernel” （AI 引擎内核）	内核函数名称。此名称无需与 graph 类中的内核例化的名称相匹配。例如，`fir_27t_sym_hb_2i` 是函数名称，且例化为 “interpolator” ，如前述代码所示。
“Source” （源文件）	内核源文件。单击此文件名即可打开内核的源文件。
“Column” （列）	AI 引擎中的列，对该列中的内核进行映射。
“Row” （行）	AI 引擎中的行，对该行中的内核进行映射。
“Schedule” （调度）	映射到同一个拼块（相同的列和行）的内核的执行顺序。0 表示未设置任何调度。
“Runtime Ratio” （运行时比率）	graph 中使用 `runtime<ratio>(<kernel>) = n` 约束设置的运行时比率。
“Graph Source” （graph 源文件）	含行号的源文件 (`graph.h`)，其中的内核将执行例化。单击此链接即可打开该行号处的源文件。

可编程逻辑 (PL)

下图所示 “PL” 表提供了有关 PLIO 到 ADF graph 的连接的详细信息。例如，在此图中有 4 个与 graph 关联的 PLIO 对象。示例中还提供了与每个 PLIO 连接关联的 PLIO 连接名称、PLIO 数据连接宽度、以及仿真测试激励文件。

```
input_plio  in  = input_plio::create("DataIn1", plio_32_bits,"data/
input.txt");
output_plio out = output_plio::create("DataOut1", plio_32_bits,"data/
output.txt");
input_plio  clip_out = input_plio::create("polar_clip_in",
plio_32_bits,"data/input1.txt");
output_plio clip_in = output_plio::create("polar_clip_out",
plio_32_bits,"data/output1.txt");
```

图 3：**PL 表**

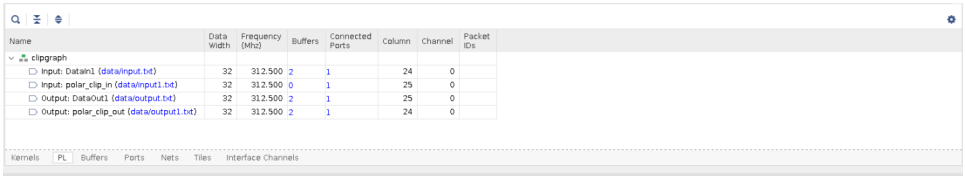


表 16：**列描述**

列	描述
“Name” （名称）	PLIO 连接的名称以及它属于输入还是输出。
“Data Width” （数据宽度）	构造函数中定义的 PLIO 连接的数据宽度。宽度可采用 32 位、64 位或 128 位。
“Frequency (MHz)” （频率 (MHz)）	（可选）PLIO 构造函数中为 PLIO 连接定义的频率（以 MHz 为单位）。
“Buffers” （缓冲器）	PLIO 连接中所使用的缓冲器数量。如果 PLIO 端口连接到使用两个缓冲器的 AI 引擎内核的窗口端口，则表示乒乓缓冲器。从 PLIO 端口到 AI 引擎内核的串流端口的连接不耗用任何缓冲器。
“Connected Ports” （已连接的端口）	PLIO 连接到的端口数。此 PLIO 数据可多播至 AI 引擎中的多个目标。欲知详情，请参阅《AI 引擎内核与 Graph 编程指南》(UG1079) 中的多播支持。

表 16：**列描述** (续)

列	描述
“Column” （列）	供 PLIO 使用的接口列，由 `aiecompiler` 分配。值的范围为 0-49。
“Channel” （通道）	PLIO 所使用的接口列中的通道。
“Packet ID” （包 ID）	包切换功能特性允许您在多个目标之间往返发送数据包。这些数据包可在 PL 与 AI 引擎之间往返发送。该列显示了使用包切换时，所使用的包的 ID。欲知详情，请参阅《AI 引擎内核与 Graph 编程指南》(UG1079) 中的显式包切换。

缓冲器

“Buffers” （缓冲器）表包含有关映射到 ADF graph 的缓冲器的信息。通常，缓冲器在窗口连接中使用。

注释：如果使用 `buf#` 和 `buf#d`，则表示它是乒乓缓冲器。

图 4：**缓冲器表**

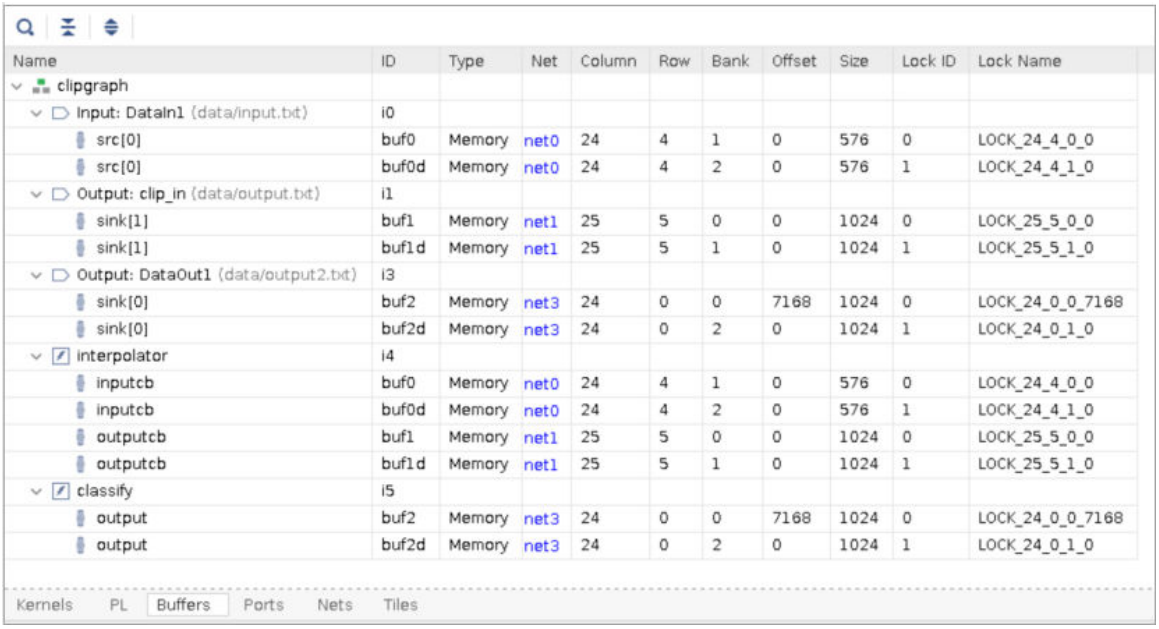


表 17：**列描述**

列	描述
“Name” （名称）	分配有缓冲器的列的名称。
“ID”	由 AI 引擎编译器提供给缓冲器的唯一 ID。
“Type” （类型）	当前使用的缓冲器的类型。类型可设为 “Memory” （存储器）或 “Stream” （串流）。窗口连接使用乒乓缓冲器，串流连接可使用 DMA 缓冲器。
“Net” （信号线）	与缓冲器关联的信号线。
“Column” （列）	拼块的列位置，编译器会对此拼块中的缓冲器进行映射。
“Row” （行）	拼块的行位置，编译器会对此拼块中的缓冲器进行映射。

表 17：**列描述** (续)

列	描述
“Bank”	拼块所在的 bank，对此拼块中的缓冲器进行映射。bank 为：0、1、2 或 3。 **注释：**硬件视图是位宽为 128 位的 8 个 bank。软件视图是位宽为 256 位的 4 个 bank。
“Offset” （偏移）	缓冲器相对于 bank 的地址偏移。
“Size” （大小）	缓冲器大小（以字节为单位）。
“Lock ID” （锁定 ID）	布局在 bank 中的每个缓冲器的唯一 ID。
“Lock Name” （锁定名称）	与缓冲器关联的锁定的唯一名称。可用于调试缓冲器上的锁定停滞。

端口

“Ports” （端口）表包含设计的所有端口，包括 GMIO 端口、PLIO 端口以及内核上的输入、输入输出和输出端口。

图 5：**端口表**

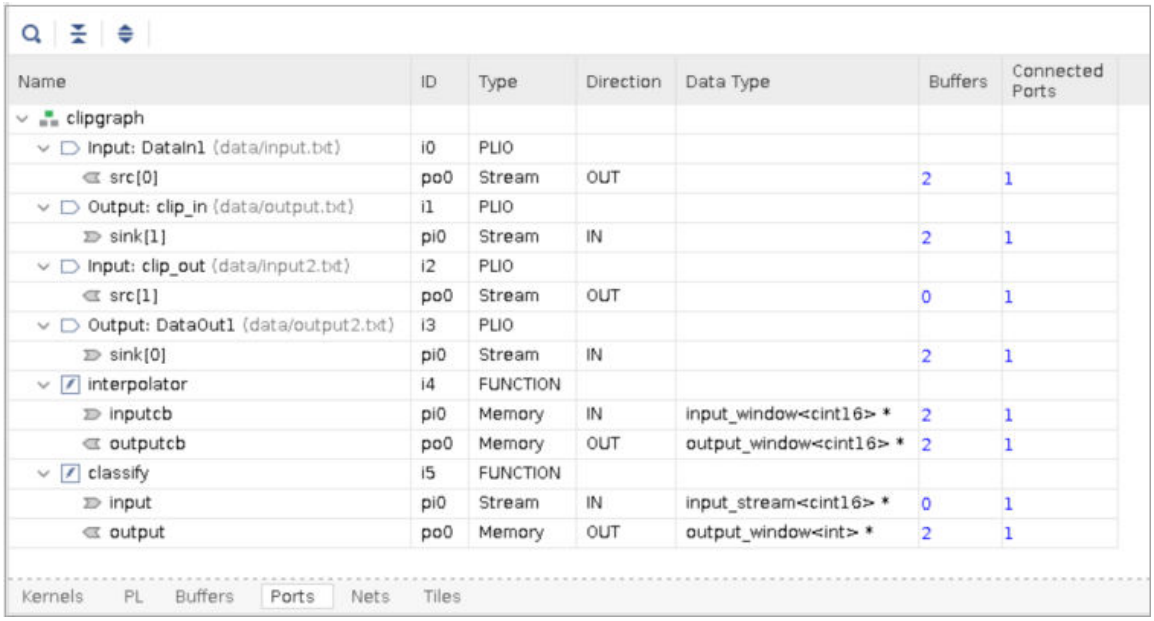


表 18：**列描述**

列	详细信息
“Name” （名称）	表示内核的输入、输入输出和输出端口、GMIO 端口或 PLIO 端口的端口名称。
“ID”	AI 引擎编译器指定的端口的唯一 ID。
“Type” （类型）	端口类型。“PLIO”端口可包含 Stream（串流）和 Packet Switching （包切换），“GMIO”端口包含 Global Memory（全局存储器），“Function” （函数）可包含 Memory（存储器）或 Stream（串流）。
“Direction” （方向）	端口方向。可设为：IN（输入）、OUT（输出）或 INOUT（输入输出）。
“Data Type” （数据类型）	内核的端口的类型定义。例如，`input_window<int16>*` 或 `input_stream<int16>*`。
“Buffers” （缓冲器）	表示针对连接而进行例化的缓冲器的数量。对于串流连接，不使用缓冲器。对于窗口连接，使用的是乒乓缓冲器。

表 18：**列描述** (续)

列	详细信息
“Connected Ports” （已连接的端口）	表示特定端口连接到的端口数。端口可多播至多个端口。欲知详情，请参阅《AI 引擎内核与 Graph 编程指南》(UG1079) 中的多播支持。

信号线

“Nets” （信号线）表中显示了 AI 引擎内核之间或者 AI 引擎内核与 PLIO/GMIO 端口之间的信号线连接详细信息。例如，在以下所示 `graph.cpp` 文件片段中提供了连接约束的示例，此连接约束用于连接到 graph 中的 AI 引擎内核之间的串流和窗口连接，或者用于连接到 PLIO/GMIO 端口。

```
connect< window >(in.out[0], interpolator.in[0]);
connect< window, stream >(interpolator.out[0], clip_in.[0]);
connect< stream >(clip_out.out[0], classify.in[0]);
connect< window >(classify.out[0], out.in[0]);
```

图 6：**信号线表**

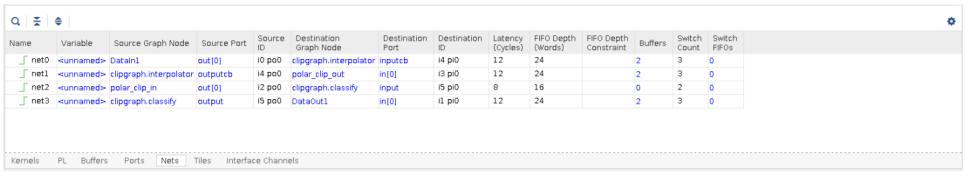


表 19：**列描述**

列	描述
“Name” （名称）	内部生成的信号线的名称。
“Variable” （变量）	信号线连接的名称（可选择在连接约束中指定）。“<unnamed>.net#” 表示在 graph 的连接约束中，`connect<>` 不含唯一命名。
“Source Graph Node” （源 graph 节点）	graph 连接的源节点，可能是 AI 引擎内核、PLIO 或 GMIO 节点。
“Source Port” （源端口）	graph 连接的源端口，可能是 AI 引擎内核、PLIO 或 GMIO 端口。
“Source ID” （源 ID）	`aiecompiler` 指定的源端口的唯一 ID。
“Destination Graph Node” （目标 graph 节点）	graph 连接的目标节点，可能是 AI 引擎内核、PLIO 或 GMIO 节点。
“Destination Port” （目标端口）	graph 连接的目标端口，可能是 AI 引擎内核、PLIO 或 GMIO 端口。
“Destination ID” （目标 ID）	AI 引擎编译器指定的目标端口的唯一 ID。
“Latency (Cycles)” （时延（周期数））	将数据从源节点传输到目标节点所需的最小周期计数。
“FIFO Depth (Words)” （FIFO 深度（字数））	通过信号线中的布线资源分配的 FIFO 存储器。这包括配置为 DMA FIFO、串流开关端口和串流开关 FIFO 的缓冲器。FIFO 深度的单位为 32 位码字。
“FIFO Depth Constraint” （FIFO 深度约束）	反映设计中提供的 FIFO 深度约束。
“Buffers” （缓冲器）	信号线连接所使用的缓冲器数量。
“Switch Count” （开关计数）	信号线连接遍历的开关数量。

表 19：**列描述** (续)

列	描述
“Switch FIFOs” （开关 FIFO）	信号线连接所使用的串流开关 FIFO 数量。

拼块

“Tiles” （拼块）表显示了 ADF graph 中具有已映射的内核与缓冲器的所有拼块。例如，在此设计中使用了 3 个拼块，其中 2 个拼块包含内核（Tile [24,0] 和 Tile [25,0]），2 个拼块具有已映射的缓冲器（Tile[24,0] 和 Tile[24,1]）。

图 7：**拼块表**

Tile	Column	Row	Kernels	Buffers
Tile [24,0]	24	0	fir_27t_sym_hb_2i	3
Tile [24,1]	24	1		5
Tile [25,0]	25	0	classifier	

表 20：**列描述**

列	描述
“Tile” （拼块）	拼块 ID。
“Column” （列）	拼块的列位置。
“Row” （行）	拼块的行位置。
“Kernels” （内核）	映射到此拼块的内核数量。
“Buffers” （缓冲器）	映射到此拼块的缓冲器数量。这包括信号线上的缓冲器及内核内部的缓冲器。

接口通道

接口通道表包含设计中使用的通道的列表，这些通道是 AI 引擎与 PL 之间的接口或 AI 引擎与全局存储器之间的接口。通常，这些通道定义为 graph 对象中的 `input_plio`、`output_plio`、`input_gmio` 或 `output_plio` 对象。

图 8：**接口通道**

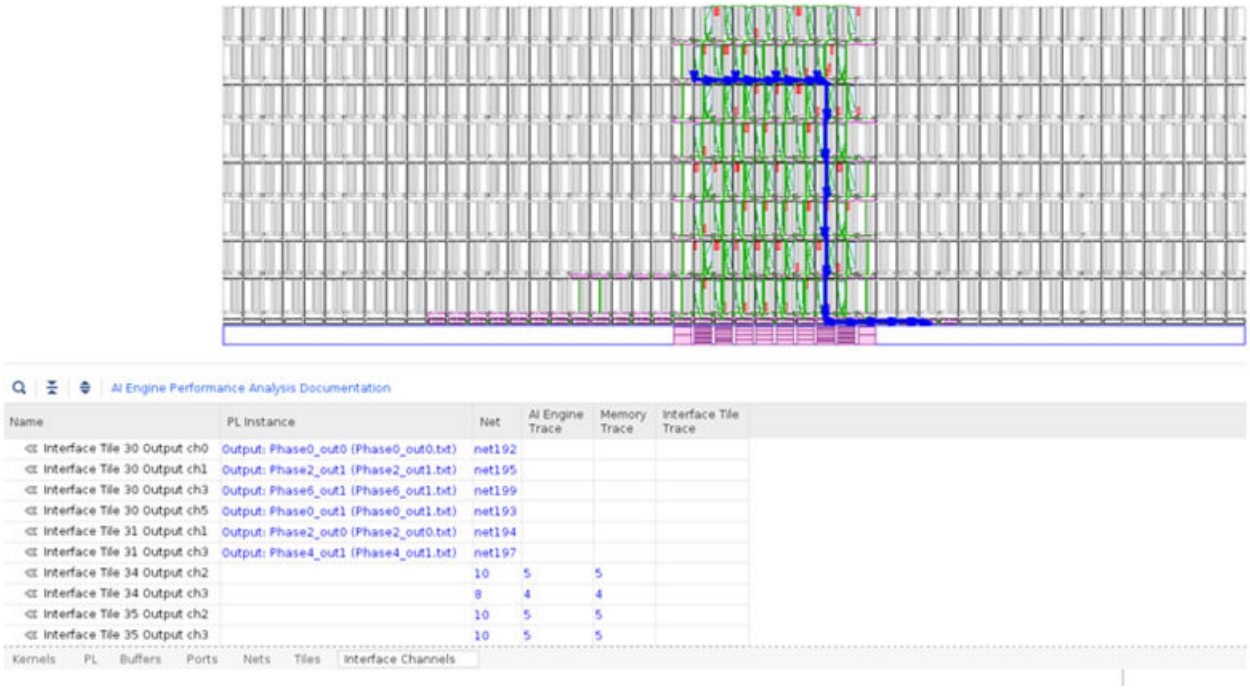


表 21：**表格列的描述**

列	描述
“Name”（名称）	接口通道的名称，包括拼块编号、输入或输出以及通道编号。
“PL Instance”（PL 实例）	设计的 PLIO 对象信息，包含输入或输出以及关联的输入或输出文件。
“Net”（信号线）	与接口通道关联的信号线。如果该值为数值，那么串流将广播到“N”条子信号线。如果该值为“netXXX”，那么这表示此信号线是单一路径。
“AI Engine Trace”（AI 引擎追踪）	合并到接口拼块中特定通道的 AI 引擎追踪包串流的数量。
“Memory Trace”（存储器追踪）	合并到接口拼块中特定通道的存储器追踪包串流的数量。
“Interface Tile Trace”（接口拼块追踪）	合并到接口拼块中特定通道的接口拼块追踪包串流的数量。

编译汇总

完成 AI 引擎 graph 编译后，即可在 Vitis 分析器中打开编译汇总。编译汇总提供了有关工具版本、构建开始时间和结束时间、构建持续时间、AI 引擎频率和 AI 引擎设计所使用的资源的信息。此外，该资源包含 AI 引擎内核详细信息，并提供了指向 graph 的链接。

图 9：**状态**

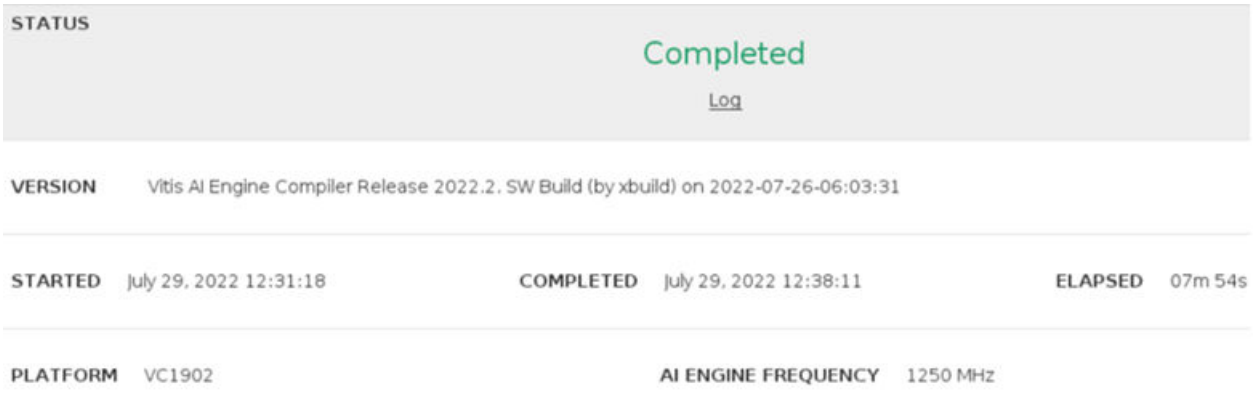
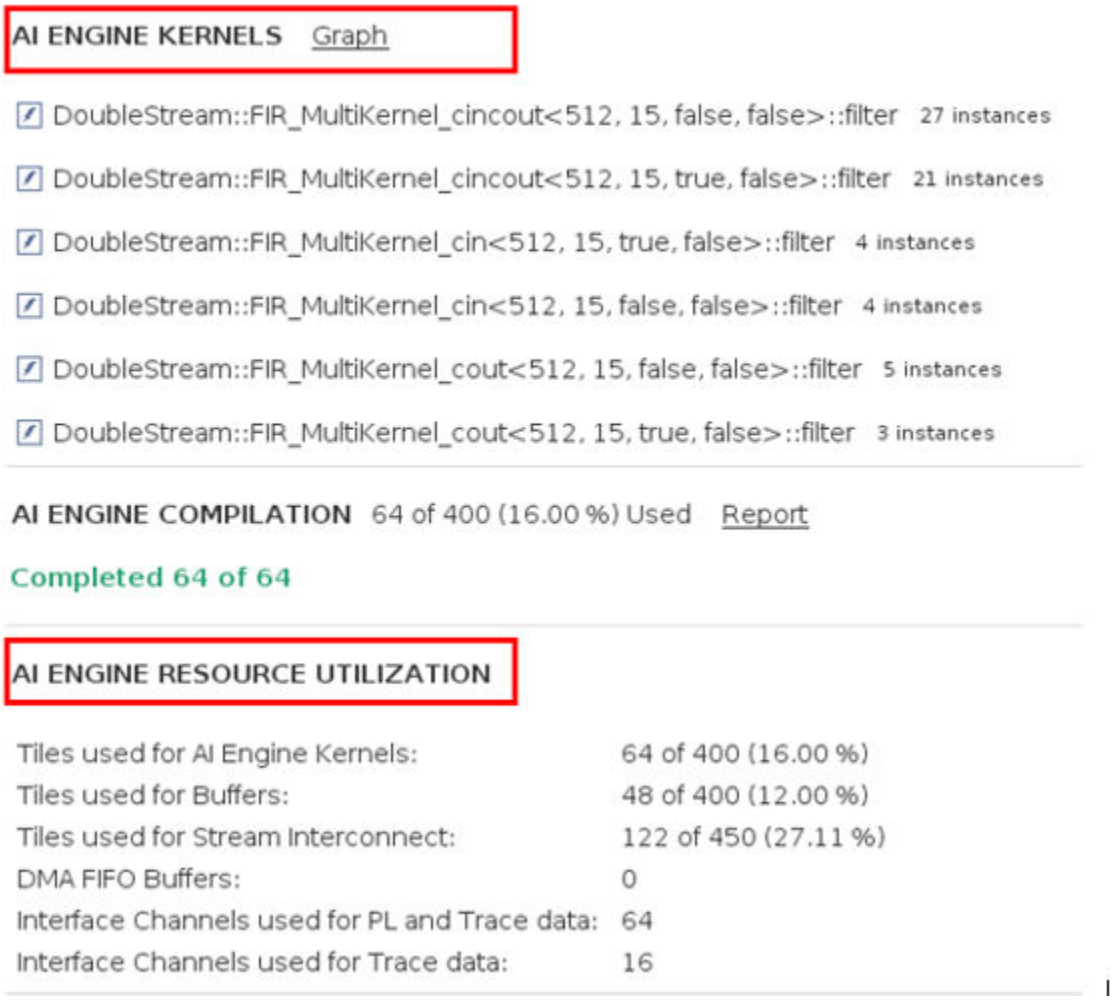


图 10：**AI 引擎内核和 AI 引擎资源使用情况**



`AI ENGINE KERNELS` 部分高亮显示了 AI 引擎内核，这些内核对相同的内核函数进行例化，但所使用的参数以及内核对应的实例数量不尽相同。

`AI ENGINE COMPILATION` 部分提供了设计使用的 AI 引擎总数。

`AI ENGINE RESOURCE UTILIZATION` 部分提供了设计的资源使用情况概述。下表中提供了详细信息。

表 22：**AI 引擎资源使用情况描述**

使用情况参数	描述
Tiles used for AI Engine Kernels	设计中 AI 引擎内核使用的拼块数量。格式为：`X of Y (Z%)`。
Tiles used for Buffers	设计所使用的缓冲器数量。格式为：`X of Y (Z%)`。
Tiles used for Stream Interconnect	用于通过串流与其它 AI 引擎、PL 或 DMA 进行通信的拼块数量。
DMA FIFO Buffers	设计所使用的 DMA FIFO 缓冲器数量。
Interface channels used for PL and Trace data	用于 PL 和追踪串流数据的接口通道数量。
Interface Channels used for Trace data	用于追踪串流数据的接口通道数量。

AI 引擎编译器指南

AI 引擎编译器完成 AI 引擎设计编译后，它会分析设计，并基于 AI 引擎规则或软件最佳实践来提供有关如何改善设计的指南。部分指南可能会由 AI 引擎编译器自动纠正。指南文件会罗列发现的所有结果，包括严重性、按拼块编号分类、详细信息，是否由 AI 引擎编译器完成纠正以及建议的解决方案。

指南文件 `guidance.html` 位于 `Work/reports` 目录中。Web 浏览器可用于复查此指南文件。建议您按设计指南报告来更新自己的设计，然后再在仿真器中或硬件上运行设计。

以下提供了 AI 引擎编译器生成的指南示例，这些指南均包含在 `Work/reports/guidance.html` 中。

- 变量先引用，然后再初始化。

Id	Name	Severity	Impact	Full Name	Categories	Details	Resolution
1	AIE-MEM-04	advisory		Memory Management	Memory.25_0	Global variable 'state' reads its default/external initialization in kernel 'producer', and is not explicitly written before this read	Ensure that for all global variables, an initialization/write (default = zero) precedes the read. For more information, refer to UG1076.

- 全局变量从内核实现本地进行初始化。

Id	Name	Severity	Impact	Full Name	Categories	Details	Resolution
1	AIE-MEM-04	advisory		Memory Management	Memory.24_0	Global variable 'state' reads its default/external initialization in kernel 'producer', and is not explicitly written before this read	Ensure that for all global variables, an initialization/write (default = zero) precedes the read. For more information, refer to UG1076.

- 数据阵列并非 128 位（16 字节）对齐。

Id	Name	Severity	Impact	Full Name	Categories	Details	Resolution
1	AIE-MEM-01	advisory		Memory Management	Memory.0_0	Alignment of global array lut1 is not 128 bits. Automatically correcting it.	Align global arrays to 16 byte boundary. Refer to the chess user manual for details on specifying the alignment constraints.

- 使用 `__restrict` 限定符会导致未定义的行为。仅当在硬件上运行时，才会呈现出此未定义的行为。

Id	Name	Severity	Impact	Full Name	Categories	Details	Resolution
...	...	...	...	...	...	...	...
3	AIE-MEM-05	advisory		Memory Management	Memory.24_0	Store to restrict child pointer *_Z23equalizer_24tap_complexP12input_windowI6cint16EP13output_windowIS0_E::inputcb.addr must be used in a different block-level scope than the parent pointer	Ensure proper usage of restrict keyword. Refer to the formal definition of restrict in Section 6.7.3.1 of C99 Standard.
4	AIE-MEM-05	advisory		Memory Management	Memory.24_0	Store to restrict child pointer *_Z23equalizer_24tap_complexP12input_windowI6cint16EP13output_windowIS0_E::outputcb.addr must be used in a different block-level scope than the parent pointer	Ensure proper usage of restrict keyword. Refer to the formal definition of restrict in Section 6.7.3.1 of C99 Standard.

注释： `--xlopt` ⩾1 选项是编译设计所必需的，它允许 AI 引擎编译器生成相应的指南文件。

AMD XILINX

第 4 章

对 AI 引擎 graph 应用进行仿真

本章描述了可用于在不同的抽象层、精度层和速度层对 AI 引擎应用进行仿真的各种执行目标。AI 引擎 graph 可在 4 种不同环境内进行仿真。

x86 仿真器是一种快速功能仿真器，如 x86 功能仿真器 中所述。它应用于对 AI 引擎 graph 进行功能仿真，适用于内核与 graph 的功能开发与验证。但它无法提供时序、资源或性能信息。

AI 引擎仿真器 (`aiesimulator`) 会对 AI 引擎阵列的时序和资源进行建模，同时为 NoC 和 DDR 存储器使用传输事务级 SystemC 模型。这样即可加速完成 AI 引擎应用的性能分析、准确估算 AI 引擎资源使用情况并提供周期近似的时序信息。

在 x86 功能仿真器与 AI 引擎仿真器中，与 AI 引擎对接的 PL 接口可通过未定时的外部流量来实践。同样，使用用户 graph 的 main() 函数作为 C 语言测试激励文件即可配置和控制 AI 引擎。此测试激励文件数据输入和输出并未定时，用户 graph 的 main() 函数充当虚拟仿真平台。

如需对整个系统进行快速功能仿真（包括 AI 引擎 graph、PL 逻辑以及基于 XRT 的主机应用）以控制 AI 引擎和 PL，则应使用 Vitis™ 软件仿真流程。软件仿真流程使用基于 HLS 的内核或基于 RTL 的内核的 C 语言模型，以便与 AI 引擎 graph 对接，并使用同样可在硬件上运行的主机代码对其进行控制。此流程包含 AI 引擎和 PS 的 x86 功能模型。PS 的 x86 功能你行是软件仿真流程中的默认模型。Arm QEMU 模型同样可作为 PS 的替代模型来使用。

最后，当您准备好对整个系统（包括 AI 引擎 graph 和 PL 逻辑以及用于控制 AI 引擎和 PL 的基于 XRT 的主机应用）进行仿真后，您应针对特定开发板和平台使用 Vitis 硬件仿真流程。此流程包括 AI 引擎的 SystemC 模型，以及用于 NoC、DDR 存储器、PL 内核 (RTL) 和 PS（在 QEMU 上运行）的传输事务级 SystemC 模型。您也可为自己的平台或设计包含 RTL 逻辑和测试激励文件 PL 逻辑。

下表列出了 4 种仿真流程及其是否支持功能级或性能级调试，以及源代码调试的支持级别。此外还提供了有关在您的 AI 引擎设计开发的相应阶段内如何使用这些仿真流程的建议。

表 23：**仿真流程**

仿真工具流程	功能调试	性能分析与调试	源码级调试	设计开发阶段使用
x86 仿真器	支持	不支持	支持	AI 引擎内核与 graph 调试
AI 引擎仿真器	支持	支持（此仿真会单步执行 AI 引擎汇编代码，对于性能分析和最优化很有用）	允许单步执行 AI 引擎编译器所生成的汇编代码，这有助于代码最优化，但可能由于编译器最优化而限制源码级可视性	AI 引擎 graph 性能调试
Vitis 软件仿真	支持	不支持	支持	系统级仿真和功能调试
Vitis 硬件仿真	支持	支持	可能；但由于编译器最优化，源码级可视性受到限制	系统级仿真和性能调试

仿真模型

下表列出了仿真流程，并对 AI 引擎设计开发的相应阶段内如何使用这些仿真流程以及用于各种 Versal® 架构域的仿真模型类型提供了建议。仿真模型的类型和所使用的仿真工具流程可用于判定模型结果的准确性。

表 24：**仿真模型**

仿真工具流程	AI 引擎内核	PL 内核	PL 平台	NoC/DDR 模型	PS 模型
x86 仿真器	x86 线程	C/SystemC/外部流量生成器	不适用	不适用	不适用
AI 引擎仿真器	SystemC	SystemC/外部流量生成器	不适用	SystemC	不适用
Vitis 软件仿真	x86 线程	SystemC/RTL/外部流量生成器	不适用	不适用	x86（默认）QEMU（可选）
Vitis 硬件仿真	SystemC	SystemC/RTL/外部流量生成器	RTL/SystemC	SystemC	QEMU

仿真功能特性

在 `aiesimulator` 或硬件仿真中运行设计时，可获取剖析数据。分析此数据有助于您测量内核效率、与每个 AI 引擎关联的停滞和活动时间，并确定性能可能未处于最优状态的 AI 引擎内核。您还可收集有关设计时延、吞吐量和带宽的数据。如需获取有关运行和分析剖析数据的详细信息，请参阅 第 5 章：仿真期间对 AI 引擎 graph 应用进行性能分析。

事件追踪功能特性允许您捕获并分析程序执行的系统级视图。它有助于识别程序执行期间的问题，包括执行正确与否以及性能问题。AI 引擎架构可为仿真和硬件仿真期间的事件生成、收集和串流（作为追踪数据）提供直接支持。如需获取有关运行和分析事件追踪数据的详细信息，请参阅 第 5 章：仿真期间对 AI 引擎 graph 应用进行性能分析。

`x86simulator` 和 `aiesimulator` 通过 `graph.cpp` 的 `main()` 函数来对设计进行仿真。QEMU 仿真支持可用于软件仿真和硬件仿真中的主机应用。请注意，在软件仿真中，QEMU 支持为可选。在软件仿真中，您可以 Linux-XRT 为目标来创建主机应用，并进行主机应用仿真。在硬件仿真中，您可以裸机或 Linux-XRT 为目标来创建主机应用，并进行主机应用仿真。通过 graph 的 main() 函数（充当虚拟测试激励文件平台）即可将测试激励文件数据提供给 x86 仿真器和 AI 引擎仿真器。有多个级别的测试激励文件支持可用于仿真流程。测试激励文件数据可基于文件，或者也可以通过外部流量生成器来传输。如需了解有关此功能特性的更多详细信息，请参阅 生成流量用于软硬件仿真。

在 AI 引擎内核之间流动的数据可供使用，也可作为数据快照文件或者通过 Vitis 分析器 GUI 来查看。如需了解有关 x86 仿真器快照功能特性的更多详细信息，请参阅 数据快照。对于运行 AI 引擎仿真器所得结果的追踪数据，如需了解有关数据可视化的更多信息，请参阅 追踪视图数据可视化。

下表列出了 4 种仿真流程所支持的仿真功能特性的类型。

表 25：**仿真功能特性**

仿真工具流程	走线	剖析	主机应用	测试激励文件支持	AI 引擎数据流可视性
x86 仿真器	不支持	不支持	通过 graph.cpp 中的 main() 函数	基于文件/外部流量生成器	支持（通过快照功能特性）
AI 引擎仿真器	支持	支持	通过 graph.cpp 中的 main() 函数	基于文件/外部流量生成器	支持（通过 Vitis 分析器中的“Trace”视图）
Vitis 软件仿真	不支持	不支持	通过以 Linux-XRT 为目标的主机应用中的 main() 函数	基于文件/外部流量生成器	支持（通过快照功能特性）
Vitis 硬件仿真	支持	支持	通过以裸机或 Linux-XRT 为目标的主机应用中的 main() 函数	基于文件/外部流量生成器	支持（通过 Vitis 分析器中的“Trace”视图）

x86 功能仿真器

在 AI 引擎 graph 和内核上启动开发时，验证设计行为至关重要。这称为功能仿真，对于识别设计中的错误很有用。例如，graph 中的窗口大小与内核中的迭代数息息相关。通过故障排除来查看大型 graph 中是否每个内核都正在吸收和生成正确数量的样本，这本质上是一个耗时且迭代性的操作。x86 仿真器因其给开发者所提供的高速迭代和高层次的数据可视性，而成为对此类行为进行测试、调试和验证的理想选择。但 x86 仿真无法提供时序、资源或性能信息。x86 仿真器专在工具开发机器上运行。这意味着其性能和存储器使用均由开发机器来定义。

虽然 AI 引擎仿真器会对 graph 与内核的存储器进行完整建模，但验证设计行为仍然至关重要。这称为功能仿真，对于识别设计中的错误很有用。例如，graph 中的窗口大小与内核中的迭代数息息相关。通过故障排除来查看大型 graph 中是否每个内核都正在吸收和生成正确数量的样本，这本质上是一个耗时且迭代性的操作。x86 仿真器因其给开发者所提供的高速迭代和高层次的数据可视性，而成为对此类行为进行测试、调试和验证的理想选择。但 x86 仿真无法提供时序、资源或性能信息。x86 仿真器专在工具开发机器上运行。这意味着其性能和存储器的使用由开发机器来定义。对于 AI 引擎阵列，这也意味着 AI 引擎仿真器受到 AI 引擎存储器空间的限制。x86 仿真器则不受此限制，可提供近乎无限量的调试 `printf()s`、大型阵列和变量。通过搭配单步执行内核的能力，即可快速发现并解决非常复杂的问题。

其中还提供了多个宏用于改善内核调试。这些宏旨在搭配 x86 仿真器一起使用，可保留在代码中以便于维护。x86 仿真器有利有弊。部分类型的 graph 构造不受支持，在 AI 引擎仿真器与 x86 仿真器之间难以保证行为的周期精确性。x86 仿真器并非旨在替代 AI 引擎仿真器。AI 引擎仿真器仍必须在所有设计上运行以验证行为并获取性能信息。对于工程开发的不同阶段，总有某一种工具更适合您的需求。

注释：要了解有关 x86 仿真限制的更多信息，请参阅 限制。

要运行 x86 仿真器，请将 AI 引擎编译器目标更改为 `x86sim`。

```
aiecompiler --target=x86sim graph.cpp
```

为 x86 仿真完成应用编译后，即可按如下方式调用 x86 仿真器。

```
x86simulator
```

以下代码中显示了完整的 x86 仿真器命令帮助信息。

```
$
x86simulator [-h] [--help] [--h] [--pkg-dir=PKGDIR]
optional arguments:
-h,--help --h show this help message and exit
--pkg-dir=PKG_DIR Set the package directory. ex: Work
--i, -i ,--input-dir=PATH Set the input directory
--o, -o ,--output-dir=PATH Set the output directory
--timeout=secs Terminate simuation after specified number of seconds
--dump Enable snapshots of data traffic on kernel ports
--gdb Invoke from gdb
--valgrind Run simulator under valgrind to detect access violations
--valgrind-gdb Run simulator under valgrind and debug via gdb server
--valgrind-args=ARGS Override default options for valgrind. Used in
conjunction with --valgrind.
--disable-stop-on-deadlock    disable deadlock detection
--trace Enable trace of kernel stall events
--trace-print Print kernel stall events during simulation
--enable-handshake-ext-tb Enable handshake protocol with external testbench
```

编译后的 x86 本机仿真二进制文件是由 AI 引擎编译器生成的，置于 `Work` 目录（请参阅 第 3 章：对 AI 引擎 Graph 应用进行编译）下，并由 x86 仿真器自动启动。

在以下 graph 代码片段中指定了输入和输出文件。

```
adf::input_plio in1=adf::input_plio::create("In", adf::plio_32_bits,
"In1.txt");
adf::output_plio out1=adf::output_plio::create("Out", adf::plio_32_bits,
"Out1.txt");
```

x86 仿真器在运行时会在当前工作目录中查找 `data/In1.txt`，即 ADF graph 所使用的输入之一。为了将 x86 仿真器的输出文件与 AI 引擎 graph 和内核的输出文件加以区分，重要的是验证设计仿真器中的 `Out1.txt` 是否位于 `current_working_dir/x86simulator_output/data/` 内。

（可选）由仿真器生成的输出文件可以与黄金输出进行比较（忽略空格差异）。

```
diff -w <data>/golden.txt <data>/output.txt
```

仿真器 log 日志和输出均可在 `vitis_analyzer` 中直观查看：

图 11：**Vitis 分析器 log 日志视图**

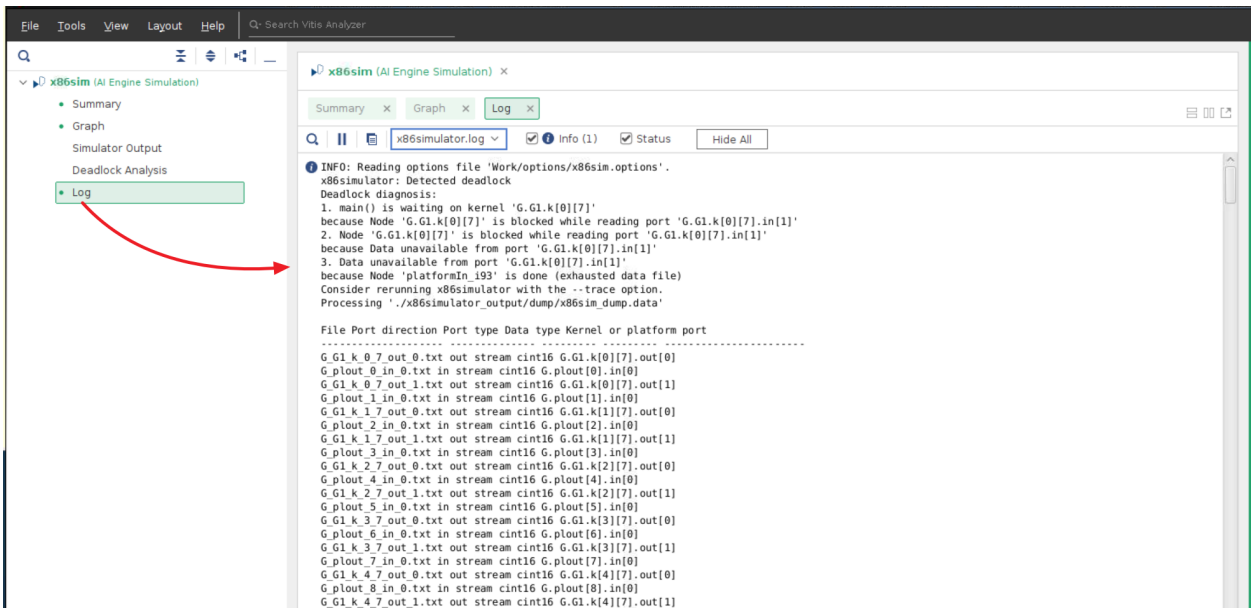
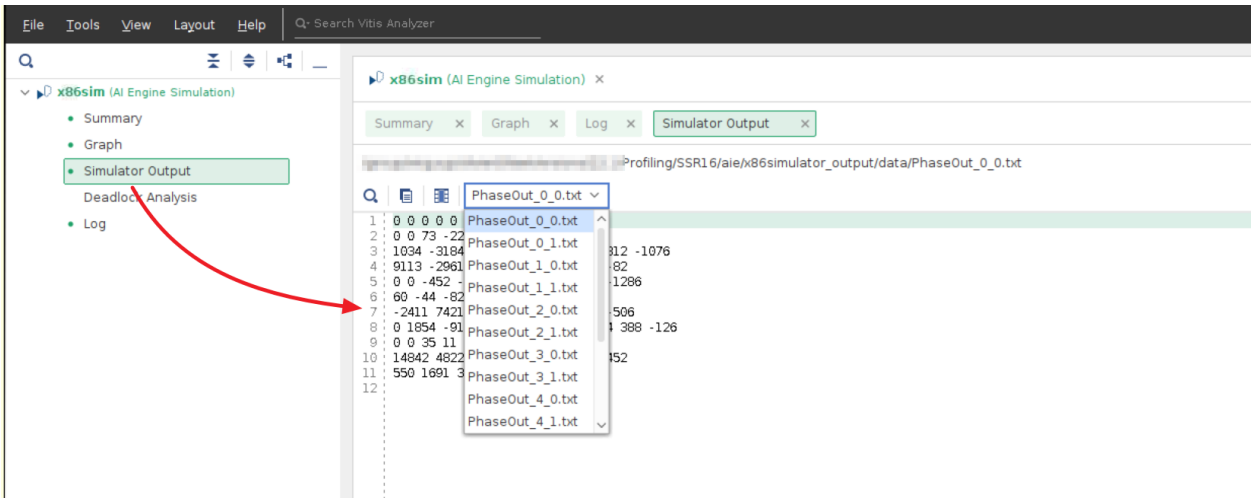


图 12：**Vitis 分析器仿真器输出视图**



“Simulator Output”（仿真器输出）视图允许您选择应显示的输出。

设计编译

GNU 调试器支持与基于 IDE 调试器类似的 C/C++ 调试。它允许设置断点、单步步进、单步跳过函数和断点上的多次命中计数。对于 AI 引擎内核开发，x86 仿真器支持使用 GDB 堆内核代码进行单步调试。

AI 引擎编译器的 `target` 实参必须设置为 `x86sim` 才能使用 GDB。

```
aiecompiler --target=x86sim graph.cpp
```

此外，使用预处理器指令 `-O0` 进行编译会最大程度减少可改善调试可视性的最优化操作。如果需要额外调试可视性，可以降低编译器最优化级别。可按如下方式将最优化参数传递给预处理器。

```
aiecompiler --target=x86sim --Xpreproc=-O0 graph.cpp
```

仿真器选项

本节中描述了完整的 x86 仿真器 (`x86simulator`) 选项集合。

表 26：**x86 仿真器选项**

选项	描述
`-h` 和 `--help --h`	显示帮助消息。
`--pkg-dir=PKG_DIR`	设置包的目录，例如：`./Work`
`--timeout=secs`	在指定秒数过后，终止仿真。
`--dump`	在内核端口上启用数据流量快照。
`--gdb`	从 GDB 调用。
`--valgrind`	在 Valgrind 下运行仿真器以检测访问违例。
`--valgrind-gdb`	在 Valgrind 下运行仿真器并通过 GDB 服务器进行调试。
`--valgrind-args=ARGS`	覆盖 Valgrind 的默认选项。搭配 `--valgrind` 一起使用。

表 26：**x86 仿真器选项** (续)

选项	描述
`--disable-stop-on-deadlock`	默认情况下，仿真器会对仿真进行连续扫描，如果检测到死锁，则停止仿真。该选项会禁用此扫描进程。
`--trace`	启用内核停滞事件的追踪。
`--trace-print`	在仿真期间打印内核停滞事件。
`--enable-handshake-ext-tb`	通过外部测试激励文件启用握手协议。

数据快照

`x86simulator --dump` 数据快照功能特性允许用户无需使用调试器即可对内核端口处的数据流量进行转储和检验。此功能特性无需对内核代码执行任何检测。它还有助于诊断含 `x86simulator` 的相同设计的两个版本之间的仿真结果不匹配问题。对于内核端口（包括串流、窗口、包串流、级联串流和 RTP 端口），支持生成数据快照。所有平台级别的端口（如，PLIO、GMIO、RTP 端口以及对特定 RTP 端口进行的 PS 访问）也同样支持该功能特性。

对于输入窗口端口，每次内核获取窗口并且快照包含裕度时，都会生成数据快照。对于输出窗口端口，每次内核发布窗口并且其中不含裕度时，都会生成快照。在上述任一情况下，快照均包含内核迭代计数和样本数据。每个内核端口各生成一个文本文件。此外，还会记录内核的迭代计数。

例如，以下提供的 graph 数据快照包含由 2 个内核构成的内核链，内核所含窗口端口中，第一个窗口含非零裕度。

```
$x86simulator --pkg-dir=./Work --i=.. --dump
INFO: Reading options file './Work/options/x86sim.options'.
Processing './x86simulator_output/dump/x86sim_dump.data'
File Port direction Port type Data type Kernel or platform port
------------------------ -------------- --------- ---------
------------------------
platform_src_0.txt out window cint16 platform.src[0]
mygraph_first_in_0.txt in window cint16 mygraph.first.in[0]
mygraph_second_out_0.txt out window cint16 mygraph.second.out[0]
platform_sink_0.txt in window cint16 platform.sink[0]
mygraph_first_out_0.txt out window cint16 mygraph.first.out[0]
mygraph_second_in_0.txt in window cint16 mygraph.second.in[0]
Wrote './x86simulator_output/dump/platform_src_0.txt'
Wrote './x86simulator_output/dump/mygraph_first_in_0.txt'
Wrote './x86simulator_output/dump/mygraph_second_out_0.txt'
Wrote './x86simulator_output/dump/platform_sink_0.txt'
Wrote './x86simulator_output/dump/mygraph_first_out_0.txt'
Wrote './x86simulator_output/dump/mygraph_second_in_0.txt'
Simulation completed successfully returning zero
$> more ./x86simulator_output/dump/mygraph_first_in_0.txt
# port: mygraph.first.in[0]
# port_dir: in
# port_type: window
# data_type: cint16
# Iteration 1; snapshot 1
0 0
0 0
0 0
0 0
0 0
0 0
0 0
0 0
0 1
```

```
2 3
4 5
6 7
8 9
10 11
12 13
14 15
16 17
18 19
20 21
22 23
24 25
26 27
28 29
30 31
32 33
34 35
36 37
38 39
40 41
42 43
44 45
46 47
48 49
50 51
52 53
54 55
56 57
58 59
60 61
62 63
# Iteration 2; snapshot 2
48 49
50 51
52 53
54 55
56 57
58 59
60 61
62 63
1 0
3 2
5 4
7 6
9 8
11 10
13 12
15 14
17 16
19 18
21 20
23 22
25 24
27 26
29 28
31 30
33 32
35 34
37 36
39 38
41 40
43 42
45 44
```

```
47 46
49 48
51 50
53 52
55 54
57 56
59 58
61 60
63 62
```

对于串流端口，每个含 4 字节数据的数据区块各生成一个数据快照。每个快照均包含 TLAST 的值以及内核迭代计数。对于级联端口，粒度为 8 字节。

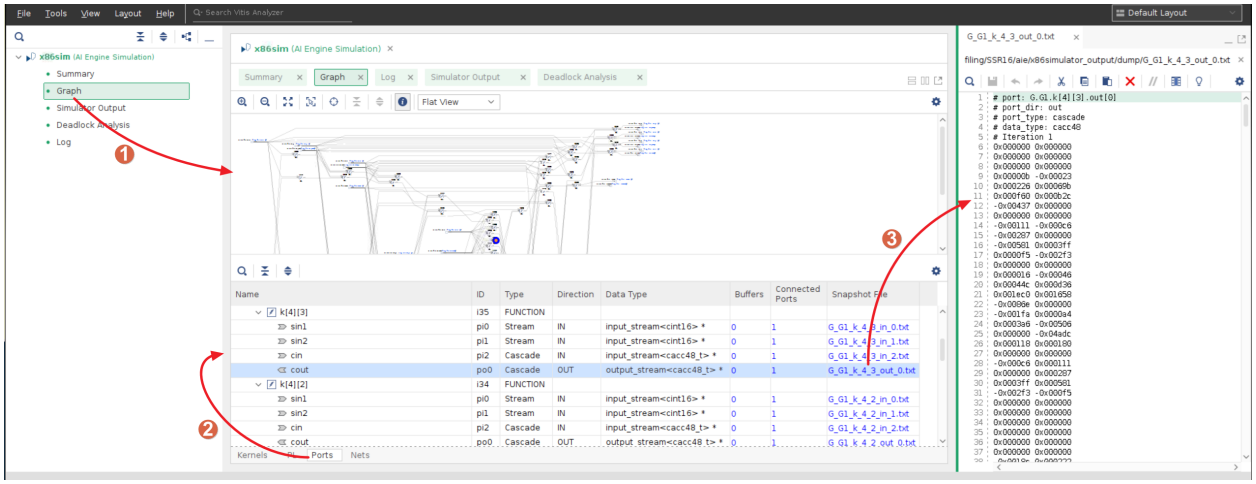
例如，以下提供的 graph 数据快照包含 3 个内核，这些内核通过串流端口相连接。

```
$ x86simulator --pkg-dir=./Work --i=.. --dump
INFO: Reading options file './Work/options/x86sim.options'.
Processing './x86simulator_output/dump/x86sim_dump.data'


File Port direction Port type Data type Kernel or platform port
------------------------- -------------- --------- ---------
-----------------------
platform_src_0.txt out stream int32 platform.src[0]
fifo_graph_dist_in_0.txt in stream int32 fifo_graph.dist.in[0]
fifo_graph_aggr_out_0.txt out stream int32 fifo_graph.aggr.out[0]
platform_sink_0.txt in stream int32 platform.sink[0]
fifo_graph_comp0_in_0.txt in stream int32 fifo_graph.comp0.in[0]
fifo_graph_comp1_in_0.txt in stream int32 fifo_graph.comp1.in[0]
fifo_graph_dist_out_0.txt out stream int32 fifo_graph.dist.out[0]
fifo_graph_comp0_out_0.txt out stream int32 fifo_graph.comp0.out[0]
fifo_graph_aggr_in_0.txt in stream int32 fifo_graph.aggr.in[0]
fifo_graph_comp1_out_0.txt out stream int32 fifo_graph.comp1.out[0]
fifo_graph_aggr_in_1.txt in stream int32 fifo_graph.aggr.in[1]


Wrote './x86simulator_output/dump/platform_src_0.txt'
Wrote './x86simulator_output/dump/fifo_graph_dist_in_0.txt'
Wrote './x86simulator_output/dump/fifo_graph_aggr_out_0.txt'
Wrote './x86simulator_output/dump/platform_sink_0.txt'
Wrote './x86simulator_output/dump/fifo_graph_comp0_in_0.txt'
Wrote './x86simulator_output/dump/fifo_graph_comp1_in_0.txt'
Wrote './x86simulator_output/dump/fifo_graph_dist_out_0.txt'
Wrote './x86simulator_output/dump/fifo_graph_comp0_out_0.txt'
Wrote './x86simulator_output/dump/fifo_graph_aggr_in_0.txt'
Wrote './x86simulator_output/dump/fifo_graph_comp1_out_0.txt'
Wrote './x86simulator_output/dump/fifo_graph_aggr_in_1.txt'
Simulation completed successfully returning zero


$ more ./x86simulator_output/dump/fifo_graph_dist_in_0.txt
# port: fifo_graph.dist.in[0]
# port_dir: in
# port_type: stream
# data_type: int32
# Iteration 1
0
1
2
3
4
5
```

```
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
...
```

Vitis 分析器可用于直观显示这些快照：

图 13：**Vitis 分析器窗口**



1. 单击左侧菜单中的“Graph”即可显示“Graph”视图。
2. 选择 graph 下的“Ports”（端口）选项卡，最后选择要显示的快照文件即可。
3. 所选快照的详细信息就会显示在右侧。

死锁检测

AI 引擎用户可能会遇到仿真器挂起。常见原因是对于所请求的 graph 迭代次数，输入数据不足；串流数据的生成和耗用之间存在不匹配；与串流、级联串流或异步窗口之间存在周期依赖性，或者阻塞协议调用（获取异步窗口、读/写串流）顺序错误。

x86 仿真器将自动检测死锁，除非指定了 `--disable-stop-on-deadlock` 选项。如果检测到死锁，则停止仿真，并打印死锁诊断报告。

此外，在 `x86simulator_output/simulator_state_post_analysis.dot` 中会生成 graph。此 `.dot` 文件会对 graph 中有关模块框图的描述进行解码，在此模块框图中会以红色高亮显示死锁中所涉及的代理。要将此文件转换为 `.png` 文件，必须使用 `dot` 程序，如下所示。

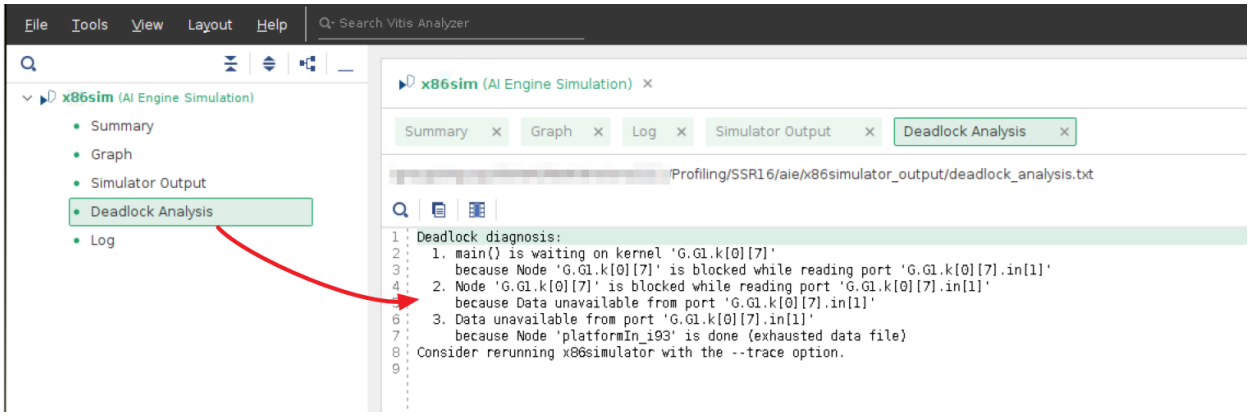
```
dot -Tpng x86simulator_output/simulator_state_post_analysis.dot >
simulator_state_post_analysis.png
```

重要提示！x86 仿真中不含死锁并不意味着 SystemC 仿真中也没有死锁。X86 仿真并不会对时序和资源约束进行建模，因而可能导致死锁的因素更少。另一方面，如果 x86 仿真发生死锁，SystemC 仿真也会发生死锁，因此先修复 x86 仿真中的死锁再继续 SystemC 仿真是很有益的。

注释：软件仿真或者含外部测试激励文件的用例不支持 `--stop-on-deadlock` 选项。

在 `vitis_analyzer` 中可直观显示死锁的文本描述：

图 14：**Vitis 分析器的“Deadlock Analysis”视图**



追踪报告

追踪功能用于调试仿真挂起，且无需检测内核代码，也无需使用 `gdb`。对于外部测试激励文件，追踪功能不可用。存在两个使用模型。

第一个使用模型适用于独立 x86 仿真，无外部测试激励文件。当 x86 仿真器检测到死锁时，它会打印一条消息，指明设计发生死锁，仿真将终止，并建议在启用追踪选项的情况下重新运行仿真。在此第二次仿真中，仿真器会给仿真期间发生的事件生成文本报告。学习此报告有助于您识别死锁的根本原因。根本原因可能只是仿真平台的数据文件中的输入数据不足或者可以增加参与的输入数据。

第二个使用模型适用于已启用 `--trace` 和 `--timeout=secs` 选项的仿真。超时到期后，仿真器会终止并生成追踪报告。正如第一个使用模型，分析追踪报告有助于识别死锁的根本原因。

其中包含如下选项：

- `--trace` 用于在仿真结束时获取完整追踪报告。
- `--trace-print` 用于在仿真运行期间，在控制台上显示输出。

追踪报告内容

追踪报告可显示在 x86 仿真器中进行设计仿真期间，所发生的一连串事件。生成的文件名为 `x86simulator_output/trace/x86sim_event_trace.data.txt`。记录的事件类型包括：

- 内核迭代开始
- 内核迭代结束
- 串流停滞开始，即，从由于缺乏数据而发生阻塞的内核串流端口开始读取
- 串流停滞结束，即，初始存在阻塞、最后返回的串流端口中的读取点
- 锁定停滞开始，即，开始尝试获取窗口端口，此端口中锁定初始不可用
- 锁定停滞结束，即，对初始存在阻塞、最后返回的窗口端口尝试执行获取的时间点

--trace-print 与 --trace 的输出对比

`--trace-print` 的输出的精细程度不及 `--trace` 生成的文件。如果要快速查看仿真中发生的情况，或者如果计划通过 CTRL-C 代替 `--timeout=secs` 来终止仿真，请使用 `--trace-print`。如果您的仿真不终止（死锁或分段错误），并且您不想指定 `--timeout=secs`，那么也适用该选项。

`--trace-print` 的输出列包含以下信息：

- 时间戳：与 `x86sim_event_trace.data.txt` 相同。
- 内核的内部名称（`x86sim_event_trace.data.txt` 使用用户名）。
- 事件类型。
- 数字值，其意义取决于事件类型：它会对您正在等待锁定或串流停滞的端口进行编码。它会对迭代事件开始的迭代次数进行编码。

存储器访问违例和 Valgrind

当内核在对象界外执行读取或写入或者读取未初始化的存储器时，可能发生存储器访问违例。这种违例可能以多种方式显现，例如，仿真器崩溃或挂起。它还可能导致仿真器结果不可重复。`x86simulator --valgrind` 选项将在内核源代码中查找存储器访问违例。

注释：要使该功能特性正常工作，需安装 Valgrind。赛灵思建议使用 Valgrind v3.16.1。

该选项允许在 x86 仿真期间使用 Valgrind 检测内核源代码中的存储器访问违例。可检测到的访问违例类型如下。

- 出界写入
- 出界读取
- 读取未初始化的存储器

该选项有两种使用方法：

- `x86simulator --valgrind`：开启访问违例检测的前提下运行仿真。仿真结束时，Valgrind 会打印有关访问违例的报告。如无违例，那么报告会以“ERROR SUMMARY: 0 errors from 0 contexts”结尾。否则，报告会列出找到的每一项访问违例。这其中包括栈追踪，即高亮内核源代码中发生访问违例的位置对应的行号。
- `x86simulator --valgrind-gdb`：开启访问违例检测的前提下运行仿真，并以 GDB 进行调试。仿真会显示在 GDB 中并在 main() 处中止。此时您可设置额外断点。继续后，如果检测到访问违例，仿真将会停止。此时，您可检验局部变量和栈，以便诊断问题。

无论在上述任何情况下，均可使用 `flag --valgrind-args='list of arguments for valgrind'` 向 Valgrind 命令添加部分实参。例如：

```
--valgrind-args='-v --leak-check=no --track-origins=yes'
```

这样将不会跟踪存储器泄漏，并且发现访问违例时将显示整个栈。

```
x86simulator --pkg-dir=Work --valgrind-gdb --valgrind-args='-v --leak-check=no --track-origins=yes'
==1021329== Memcheck, a memory error detector
==1021329== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1021329== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==1021329== Command: Work/pthread/sim.out
==1021329==
==1021329== (action at startup) vgdb me ...
==1021329==
==1021329== TO DEBUG THIS PROCESS USING GDB: start GDB like this
==1021329==   /path/to/gdb Work/pthread/sim.out
==1021329== and then give GDB the following command
==1021329==   target remote | /tools/baton/valgrind/3.16.1/lib/valgrind/../../bin/vgdb --pid=1021329
==1021329== --pid is optional if only one valgrind process is running
==1021329==
```

注释：运行 x86simulator 并搭配 Valgrind 选项将增加仿真运行时间。

使用 GDB

对相应目标成功完成编译后，即可启动仿真并将 GDB 实例自动附加到仿真中。要启动交互式 GDB 会话，请以 `--gdb` 开关运行以下命令。

```
x86simulator --gdb
```

默认情况下，使用 `--gdb` 命令行开关运行 x86 仿真器时，它会在 `graph.cpp` 中进入 `main()` 之前立即中断。这样即可在任意 AI 引擎内核开始之前暂停执行，因为 graph 尚未运行。要退出 GDB，请输入 `quit` 或 `help` 以获取更多命令。

有多种方式可用于设置断点。一种方法是使用以下语法：

```
break <kernel_function_name>
```

通过输入 `continue`（简略为 `c`），调试器即可运行直至达到 `<kernel function name>` 中的断点为止。达到断点时，可以检验局部栈变量和函数实参。下表显示了允许用于检验这些变量的部分常用 GDB 指令。

表 27：**常用 GDB 指令**

命令	描述
`info stack`	报告位于当前断点的函数调用栈。
`info locals`	显示局部变量的当前状态，这些变量位于调用栈内显示的函数调用作用域内。
`print <local_variable_name>`	打印单个变量的当前值。
`finish`	退出当前函数调用，但使所有仿真保持暂停状态。
`continue`	使调试器运行直至完成。

GDB 是非常强大的调试器，具有诸多功能特性。GDB 的完整文档不在本文档讨论范围内。如需获取 GDB 文档，请参阅 https://sourceware.org/gdb/current/onlinedocs/gdb.pdf 和 https://sourceware.org/gdb/current/onlinedocs/refcard.pdf。

宏

赛灵思提供了多个预定义的编译器宏，以帮助使用 x86 仿真器。在顶层 graph 测试激励文件（通常名为 `graph.cpp`）中，最好将以下预处理器宏与有条件的 `#if` 搭配使用以帮助包含或者排除相应的代码。

表 28：**宏**

宏	描述
__X86SIM__	此预定义宏用于指定仅适用于 `x86sim` 流程的代码。
__AIESIM__	此预定义宏用于指定仅适用于 `aiesimulator` 流程的代码。
X86SIM_KERNEL_NAME	此宏搭配 `printf()` 一起使用，以标记含有内核实例名称的检测文本。

注释：对于以下划线 _ 包围的宏，在前后各有两个下划线字符。

宏代码示例如下所示。

```
myAIEgraph g;
#if defined(__AIESIM__) || defined(__X86SIM__)
int main()
{
   g.init();
   g.run(4);
   g.end();
   return 0;
}
#endif
```

提示：__AIESIM__ 宏仅适用于 AI 引擎仿真器，__X86SIM__ 适用于 x86 仿真器。

前述示例显示 `__X86SIM__` 宏包围了 `main()`，后者供 `graph.cpp` 文件使用。此 `main()` 必须从仿真流程中排除，这些宏则可提供此类灵活性。此外，建议仅限在 x86 仿真期间才考虑使用 `__X86SIM__` 宏来选择性启用调试检测。

Printf() 宏

x86 仿真器可在独立线程上并行执行多个内核。这意味着在标准输出中查看 `printf()` 调试消息时，通常这些消息可能呈交织和不确定状态。`X86SIM_KERNEL_NAME` 宏对于确定哪个内核正在打印哪一行是很有用的。以下示例显示了如何将其与 `printf()` 相结合。

注释：要使用 X86SIM_KERNEL_NAME，您必须包含 `adf/x86sim/x86simDebug.h`，如以下代码所示。

```
#include <adf/x86sim/x86simDebug.h>

void simple(input_window_float * in, output_window_float * out) {
   for (unsigned i=0; i<NUM_SAMPLES;i++) {
       float val = window_readincr(in);
```

```
        window_writeincr(out,val+0.15);
    }
    static int count = 0;
    printf("%s: %s %d\n",__FILE__,X86SIM_KERNEL_NAME,++count);
}
```

在内核中使用 printf()

在 AI 引擎内核代码中，矢量数据类型是最常用的类型。要在内核中调试矢量运算，使用 `printf` 是很有帮助的。

将 printf() 与矢量数据类型搭配使用

要对本机矢量类型执行 `printf()`，可使用如下技巧。

```
v4cint16 input_vector;
...
int16_t* print_ptr =(int16_t*)&input_vector;
    for (int qq=0; qq<4;qq++) //4 here so we print two int16s, real + imag per loop.
    printf("vector re: %d, im: %d\r\n",print_ptr[2*qq],print_ptr[2*qq+1]);
}
```

对于 AI 引擎仿真器，`--profile` 选项是观测 `printf()` 输出所必需的选项。对于 x86 仿真器，启用 `printf` 调用无需任何其它选项。这就是 x86 仿真器的优势之一。

重要提示！赛灵思建议在内核与主机代码中避免使用 `std::cout`。鉴于 x86 仿真器的多线程性质，如果使用 `std::cout`，其输出可能显示为交织。建议改为使用 `printf()`。

限制

x86 仿真器使用的是 AI 引擎架构的简化模型。在此 x86 仿真器中，AI 引擎阵列的各内核的布局布线是一个简化的模型。在 x86 仿真器中，布线延迟和 FIFO 长度均未建模。

x86 仿真由共享 x86 处理器的多个线程组成。因此，存储器访问冲突和串流访问冲突均未建模。

以 x86 仿真器作为目标时，AI 引擎编译器会将所有矢量处理器专用的指令转换为一系列 x86 指令。因此，x86 仿真器将不会估算任何设计吞吐量。此外，在 x86 仿真中，软件流水打拍也未建模。

存储器模型

如果内核需将状态从一次调用（迭代）保留至下一次调用，则可使用全局变量或静态变量来存储此状态。含静态存储类的变量（例如，全局变量和静态变量）可能导致 x86 仿真与 AI 引擎仿真之间出现不一致。根本原因在于，对于 x86 仿真，所有内核的源文件都编译到单个可执行文件内，而对于 AI 引擎仿真，以 AI 引擎为目标的每个内核都独立编译。因此，如果含静态存储类的变量被两个内核引用，并且这些内核映射到同一个 AI 引擎，那么 x86 仿真与 AI 引擎仿真共享此变量。但如果这些内核映射到不同 AI 引擎，那么对于 x86 仿真，仍共享该变量，但对于 AI 引擎仿真，每个 AI 引擎都有其自己的副本，无共享。如果该变量供内核读取和写入，那么这就会导致 x86 仿真与 AI 引擎仿真之间出现不匹配。

要跨内核迭代延续建模状态，首选方法是使用 C++ 内核类（请参阅《AI 引擎内核与 Graph 编程指南》(UG1079) 中的 C++ 内核类支持）。这样可以避免含静态存储类的变量的缺陷。或者，可将全局变量或静态变量的存储类更改为 thread_local，但这仅适用于 x86 仿真。在此情况下，x86 仿真中的内核的每个实例都有其自己的变量副本。这样如果使用的变量映射到不同的 AI 引擎，即可与 AI 引擎仿真的行为相匹配。在以下示例中，内核通过全局变量 delayLine 和静态变量 pos 跨内核迭代延续状态。这导致如有多个内核实例使用此源文件，x86 仿真与 AI 引擎仿真之间就会出现不匹配。通过将这些变量的存储类更改为 thread_local 即可避免此问题。

原始内核源代码：

```
// fir.cpp
#include <adf.h>
cint16 delayLine[16] = {};
void fir(input_window<cint16> *in1,
         output_window<cint16> *out1)
{
  static int pos = 0;
..
}
```

重做的内核源代码：

```
// fir.cpp
#include <adf.h>
#ifndef __X86SIM__
cint16 delayLine[16] = {};
#else
thread_local cint16 delayLine[16] = {};
#endif
void fir(input_window<cint16> *in1,
         output_window<cint16> *out1)
{
#ifndef __X86SIM__
  static int pos = 0;
#else
  static thread_local int pos = 0;
#endif
..
}
```

另一种可能是使用 `X86SIM_THREAD_LOCAL` 宏，使全局读/写线程安全。该宏在 `adf.h` 中定义，如下所示：

```
#ifdef _X86SIM_
#define X86SIM_THREAD_LOCAL thread_local
#else
#define X86SIM_THREAD_LOCAL
#endif
```

这使其定义为仅适用于 X86 仿真。

前述代码简化如下：

```
// fir.cpp
#include <adf.h>

X86SIM_THREAD_LOCAL cint16 delayLine[16] = {};

void fir(input_window<cint16> *in1,
         output_window<cint16> *out1)
{
static X86SIM_THREAD_LOCAL int pos = 0;
..
}
```

graph API 调用

在 AI 引擎仿真器与 x86 仿真器中，对于定时执行（`graph.wait(N)`、`graph.resume()` 和 `graph.end(N)`），graph 构造的行为不尽相同。对于 AI 引擎仿真器，N 用于指定处理器周期超时，而对于 x86 仿真器，它指定挂钟超时限制（以毫秒为单位）。因此，如果您的测试激励文件使用定时执行，那么 AI 引擎仿真器和 x86 仿真器可能产生不同结果，尤其是生成的输出数据量可能不同。

预处理器注意事项

以 `x86sim` 为目标运行 AI 引擎编译器时，编译器会忽略 `--Xchess` 选项。这表示，`x86sim` 流程不支持内核专用的编译选项。

请参阅以下示例以便更好地理解此注意事项。对 C 语言代码执行编译时修改的常用方法是使用预处理器指令，如 `#ifndef`。要控制这些预处理器指令，最好通过命令行编译器选项来传递 `#defines`。以下代码块示例基于预处理器指令执行两项不同的操作。

```
void example_kernel()
{
  #ifdef SIM
    printf("Sumulation Mode\n");
  #else
    printf("Default Mode\n");
  #endif
}
```

要在编译时使用 AI 引擎编译器以硬件为目标 (`hw`) 来定义 SIM 宏，您可执行以下操作。

```
aiecompiler -target=hw -Xchess="example_kernel:-DSIM"
```

由于当编译目标设为 `x86sim` 时，会忽略 `-Xchess` 实参，因此对于 x86 仿真器不定义 SIM，并且内核输出为“Default Mode”（默认模式）。

如果需要为 x86 仿真器指定预处理器选项，可使用 `aiecompiler -target=x86sim --Xpreproc` 替代 `-Xchess`。值得注意的是，以此方式传递的任何选项会应用于所有源代码和所有目标流程。

表 29：**AI 引擎编译器命令行选项**

选项	描述
`--Xchess=<string>`	该选项可用于将内核专用选项传递给 CHESS 编译器，此编译器用于为每个 AI 引擎编译代码。选项字符串指定为 `<kernel-function>:<optionid>=<value>`。在映射指定内核函数的 AI 引擎上编译生成的源文件期间包含该选项字符串。
`--Xpreproc=<string>`	该选项可用于将常规选项传递到 PREPROCESSOR 阶段，用于执行所有源代码的编译 (`AIE`/`PS`/`PL`/`x86sim`)。例如：`--Xpreproc=-D<var>=<value>`

包切换和 RTP

就包切换和运行时参数 (RTP) 而言，x86 仿真器与 AI 引擎仿真器之间具有一些重要的差异。如果您尚未熟悉这些构造，请参阅《AI 引擎内核与 Graph 编程指南》(UG1079) 中的运行时参数规范和显式包切换。

包切换和 RTP 所展现的行为即可揭示 AI 引擎仿真器与 x86 仿真器之间的差异。两者间虽有差异，但都是正确的，认清这一点至关重要。正如《AI 引擎内核与 Graph 编程指南》(UG1079) 中的显式包切换所述，包切换串流在所有设计流程（x86 仿真器、AI 引擎仿真器和硬件仿真）中都有不确定性。

包切换

包串流连接具有名为“packet ID”（包 ID）的字段。如果“packet ID”字段源来自 ADF graph 内部，那么 x86 仿真器会为这些包 ID 使用基于零 (0) 的规范索引方案。拆分节点的输出上的第一个分支的包 ID 等于 0，后接 1、2、3 以此类推。如果包 ID 的源来自 ADF graph 外部，那么 AI 引擎仿真器与 x86 仿真器之间将出现不一致。要解决这些不一致和获取有关当源位于 ADF graph 外部时提供定制包 ID 的更多信息，请参阅《AI 引擎内核与 Graph 编程指南》(UG1079) 中的包切换和 AI 引擎仿真器。

包合并的本质意味着 x86 仿真器与 AI 引擎仿真器会生成非确定性结果。如果包拆分分支上的 AI 引擎在任何其它分支之前完成数据处理，那么其数据会首先出现在包合并的输出上。具体哪个核首先出现取决于内核代码与传入数据。因此，任何下游处理块都必须准备好处理此行为。

同步和异步 RTP

同步和异步运行时参数在 x86 仿真器中均受到充分支持。但在 x86 仿真器与 AI 引擎仿真器之间，RTP 更新发生的精确时序和周期精度有所不同。尤其是异步 RTP，因其性质为异步，故而不影响特定已知周期上的内核。对于 x86 仿真器和 AI 引擎仿真器中的异步 RTP 都是如此。

注释：x86 仿真器就其性质而言，属于功能级别的仿真器，而 AI 引擎仿真器则用于周期建模（近似）。显而易见，两者之间存在差异。赛灵思建议您将自己的设计分区为多个部分，以便充分利用 x86 仿真器来获益。

最佳实践

x86 仿真器搭配下列建议使用最有效。

- 使变量保留在内核函数作用域内。
- 使 x86 仿真器测试激励文件保持简单 - 使用 `init()`、`run(X)` 和 `end()`。
- 使变量保留在 ADF graph 类作用域内（请参阅《AI 引擎内核与 Graph 编程指南》(UG1079) 中的全局 graph 作用域内的表和 C++ 内核类支持）。

软件仿真

Vitis™ 软件仿真可对整个系统提供快速功能仿真，包括 AI 引擎 graph、PL 逻辑以及基于 XRT 的主机应用，以控制 AI 引擎和 PL。软件仿真流程使用基于 HLS 的内核或基于 RTL 的内核的 C 语言模型，以便与 AI 引擎 graph 对接，并使用同样可在硬件上运行的主机代码对其进行控制。此流程包含 AI 引擎和 PS 的 x86 功能模型。（可选）您还可配置软件仿真，以使用 PS 的 Arm® QEMU 仿真器模型（未定时）。如需了解有关此流程的更多详细信息，请参阅 运行软件仿真。

复用 x86 仿真器选项

`x86simulator` 会在使用命令行实参时自动生成选项文件，并将其存储在 `Work/options` 目录中。生成的选项文件可根据下表中指定的其它选项来手动调整。

注释：在软件仿真流程中，仅支持一部分 x86 仿真器选项。

根据软件仿真的配置，当 PS 配置为使用 QEMU 模型或 X86 模型时，您可复用 x86 仿真选项。

当 PS 配置为使用 x86 模型时，请使用以下环境变量来指定 x86 选项文件的位置。

```
setenv X86SIM_OPTIONSPATH <x86 options file location>
```

当 PS 配置为使用 QEMU 模型时，请使用以下命令行选项并使用 `launch_sw_emu.sh` 进行软件仿真，如 运行硬件仿真 中所述。

```
./launch_sw_emu.sh -x86-sim-options ${FULL_PATH}/Work/options/x86sim.option
```

其中 `${FULL_PATH}` 表示指向此文件的完整路径。

表 30：**软件仿真流程中支持的 x86 仿真器选项**

x86 仿真选项	值	描述
`dump`	<yes>/<no>	转储输出每个 AI 引擎内核端口的每次迭代的输入和输出数据快照。
`trace`	<yes>/<no>	提供在 AI 引擎 graph 执行期间发生的事件视图（含时间戳）。这包括带有相应时间戳的停滞、graph 启动/停止以及迭代启动/停止。
`trace-print`	<yes>/<no>	在终端内打印输出追踪，并导出追踪文件结果

AI 引擎 SystemC 仿真器

Versal® ACAP AI 引擎 SystemC 仿真器 (`aiesimulator`) 包含全局存储器（DDR 存储器）和片上网络 (NoC) 的建模，以及 AI 引擎阵列的建模。使用 SystemC 仿真目标完成应用编译后，即可调用 AI 引擎 SystemC 仿真器，如下所示。

```
aiesimulator –-pkg-dir=./Work
```

重要提示！使用 AI 引擎仿真器需按 设置 Vitis 工具环境 所述进行设置。

各项配置和二进制文件都是由 AI 引擎编译器生成的，置于 `Work` 目录下（请参阅 第 3 章：对 AI 引擎 Graph 应用进行编译），并使用 `--pkg-dir` 选项来为仿真器指定这些配置和文件。graph 由 `main` 应用中所述的控制线程来进行初始化、运行和终止。AI 引擎编译器负责对该控制线程进行编译，并将 PS IP 封装文件直接加载到仿真器中。

默认情况下，`graph.run()` 选项会指定永续运行的 graph。AI 引擎编译器会生成代码，以通过永续 `While` 循环来执行数据流 graph，因此仿真也会永续运行。要创建终止程序用于调试，请在 graph 代码中指定 `graph.run(<number_of_iterations>)` 以将迭代的执行次数限制为指定次数。指定的迭代数可以是任意正整数。

注释： `graph::run(-1)` 可指定永续运行的 graph。

AI 引擎仿真器命令首先按 `Work/config/scsim_config.json` 文件中生成的编译器中指定的方式来配置仿真器。这包括加载 PL IP 块及其连接、配置 I/O 数据文件驱动程序以及配置 NoC 和全局存储器（DDR 存储器）连接。随后，它会执行指定的 PS 应用，最终退出仿真器。

AI 引擎仿真器具有可选 `--profile` 选项，该选项会在内核代码中启用 `printfs` 以将其显示在控制台上，并生成剖析信息。并且，`--dump-vcd <filename>` 选项还会在仿真持续期间生成值更改转储 (VCD)。随后，即可使用 `--simulation-cycle-timeout <number-of-cycles>` 在指定时钟周期数过后退出仿真。

重要提示！如果不提供时钟周期数或者 `graph.run()` 的运行轮数，那么仿真会永久运行。您需要按两次“Ctrl+c”才能退出仿真器。

提示：在相同设计上的不同仿真运行期间，您可能会观察到不同的周期计数。这是因为仿真器会等待数秒钟以便暂挂的传输事务（例如，DMA）全部完成。在此等待时间内，仿真器进程仍会计时，但可由操作系统进行上下文切换，因此每一轮运行的总周期数可能不同。要确保每一轮运行的总周期数相同，应使用 AI 引擎仿真器的 `--simulation-cycle-timeout` 选项在精确周期内停止仿真器。这样剖析报告上显示每一轮运行的总周期数都相同。

注释：请勿在内核代码中包含 `<iostream>` 以启用 `printfs`。在内核代码中使用 `#include <iostream>` 会导致 x86 仿真器和 SystemC 仿真器都出现编译错误。

仿真器选项

本节中描述了完整的 AI 引擎仿真器 (`aiesimulator`) 选项集合。在大部分情况下，只需指向 `pkg-dir` 即足矣。

表 31：**AI 引擎仿真器选项**

选项	描述
`-h` 和 `--help`	显示如下帮助消息并退出。
`--display-run-interval=<time in ns>`	每次该选项所指定的时间耗尽（以 ns 为单位）后，定期显示时间戳。例如，`--display-run-interval=10`。
`--dump-vcd=<file>`	将 VCD 波形信息转储到 `<file>` 中。由于该工具会将 `.vcd` 追加到指定文件名后，因此无需包含文件后缀。 **注释：**要从 aiesimulator 生成的 .vcd 文件生成 XPE 文件，请使用 `vcdanalyze` 工具，如下所示： `vcdanalyze --vcd <vcdfile> --xpe [OPTIONS]` 可用选项包括： • -s=<STARTTIME>：指定开始时间（可选）。 • -e=<ENDTIME>：指定结束时间（可选）。 • --out=<RPT>：以“.rpt”格式生成工作负载估算报告。 • -v：生成详细输出（可选）。 • --xpe：生成功耗估算报告。 • --xpe-dir=<XPEOutputDir>：指定用于生成 XPE 报告的目录。默认值为 `./aiesim_xpe`。 如需了解有关使用 XPE 文件的信息，请参阅《适用于 Versal ACAP 的 Xilinx Power Estimator 用户指南》(UG1275)。
`--enable-handshake-ext-tb`	在 aiesimulator 与外部测试激励文件之间启用逐个样本传输事务
`--enable-memory-check`	启用运行时程序和数据存储器边界访问检查。任何违例访问都将以 [ERROR] 消息方式来报告。默认禁用该选项。
`--hang-detect-time=<time in ns>`	如果在该选项所指定的时间段（以 ns 为单位）过后，所有活动的核都处于停滞状态，则仿真退出。例如，`--hang-detect-time=10`。
`-i`	`--input-dir=<dir>` 选项的别名。
`-o`	`--output-dir=<dir>` 选项的别名。
`--pkg-dir=<dir>`	指定封装目录，例如，`./Work`。

表 31：**AI 引擎仿真器选项** (续)

选项	描述
`--profile`	为所有已用的核生成剖析数据。允许在 `stdout` 上生成 `printf` 追踪消息，并在仿真期间收集剖析统计数据。这可能稍许减缓仿真器速度。 （可选）可使用 `--profile=(col,row)(col,row)...` 标记指定对特定核进行剖析。
`--simulation-cycle-timeout=<cycles>`	应用完成加载后，运行指定的周期数。 **提示：**指定 `--simulation-cycle-timeout` 选项即可在指定次数的超时后终止仿真会话。但在调试进程期间指定仿真超时时，请务必指定较大数值的周期数，因为达到超时周期后，调试将终止。
`--online`	调用 `vcdanalyze` 以动态解析 VCD 数据，这样即可（可选）生成通用追踪格式 (CTF) 或者波形数据库 (WDB) 输出。 **提示：**`--online` 选项与 `--dump-vcd` 选项不得搭配一起使用。如果同时指定这两个选项，只有 `--online` 选项会生效。
`--output-time-stamp`	使用 `--output-time-stamp=no` 可获取无时间值的输出文件，使用 `--output-time-stamp` 可获取输出文件中的统一时间值（以 ns 为单位）。

挂起检测

AI 引擎用户可能会遇到仿真器挂起。常见原因是对于所请求的 graph 迭代次数，输入数据不足；串流数据的生成和耗用之间存在不匹配；与串流、级联串流或异步窗口之间存在周期依赖性，或者阻塞协议调用（获取异步窗口、读/写串流）顺序错误。

`aiesimulator` 的 `--hang-detect-time=<time_in_ns>` 选项将允许该工具检查内核上发生的挂起状况，并在指定延迟后触发仿真退出。它可以检测电路切换或包切换通信的锁定停滞和串流停滞。以下提供了挂起检测输出的部分示例：

图 15：**串流停滞检测**

```
Set iterations for the core(s) of graph g
Enabling core(s) of graph g
Waiting for core(s) of graph g to finish execution ...
1747600 ps Array::all cores with kernels are in hang condition, stopping simulation
|---------------- Core Stall Status ----------------|
 (21,1) -> Stream stall ->  SS0
 (21,2) -> Stream stall ->  SS0
 (21,3) -> Stream stall ->  SS0
 (22,1) -> Stream stall ->  SS0
 (22,2) -> Stream stall ->  SS0
 (22,3) -> Stream stall ->  SS0
 (24,1) -> Stream stall ->  SS0
 (24,2) -> Stream stall ->  SS0
 (25,1) -> Stream stall ->  SS0
 (25,2) -> Stream stall ->  SS0
 (26,1) -> Stream stall ->  SS0
 (26,2) -> Stream stall ->  SS0
 (26,3) -> Stream stall ->  SS0
 (27,1) -> Stream stall ->  SS0
 (27,2) -> Stream stall ->  SS0
 (28,1) -> Stream stall ->  SS0
 (28,2) -> Stream stall ->  SS0
|------------------------------------------------|
Exiting!
```

图 16：**锁定停滞检测**

```
Enabling core(s) of graph rx0
Waiting for core(s) of graph rx0 to finish execution ...
852600 ps Array::all cores with kernels are in hang condition, stopping simulation
|---------------- Core Stall Status ----------------|
 (24,1) -> Lock stall ->  Lock_East
 (24,2) -> Lock stall ->  Lock_East
 (25,1) -> Lock stall ->  Lock_North
|------------------------------------------------|
Exiting!
core(s) are done executing
```

仿真输入和输出数据串流

仿真输入文件

输入/输出串流的默认位宽为 32 位。位宽用于指定仿真输入文件上每行的样本数。输入文件的每一行上样本的解读取决于期望的数据类型和 PLIO 数据宽度。下表根据数据类型及其对应的 PLIO 接口规范，显示了输入数据文件中的样本解读方式。

表 32：**仿真输入数据与数据类型和 PLIO 宽度的依赖关系**

数据类型	PLIO 32 位	PLIO 64 位	PLIO 128 位
	`adf::input_plio in = adf::input_plio::create("DataIn1", adf::plio_32_bits, "input.txt");`	`adf::input_plio in = adf::input_plio::create("DataIn1", adf::plio_64_bits, "input.txt");`	`adf::input_plio in = adf::input_plio::create("DataIn1", adf::plio_128_bits, "input.txt");`
int8	每行 4 个值。例如： 6 8 3 2	每行 8 个值。例如： 6 8 3 2 6 8 3 2	每行 16 个值。例如： 6 8 3 2 6 8 3 2 6 8 3 2 6 8 3 2
int16	每行 2 个值。例如： 24 18	每行 4 个值。例如： 24 18 24 18	每行 8 个值。例如： 24 18 24 18 24 18 24 18
int32	每行 1 个值 2386	每行 2 个值。例如： 2386 2386	每行 4 个值。例如： 2386 2386 2386 2386
int64	不适用	45678	每行 2 个值。例如： 45678 95578
cint16	每行 1 个 cint 值：实数，虚数。例如： 1980 485	每行 2 个 cint 值。例如： 1980 45 180 85	每行 4 个 cint 值。例如： 1980 485 180 85 980 48 190 45
cint32	不适用	每行 1 个 cint 值：实数，虚数。例如： 1980 485	每行 2 个 cint 值。例如： 1980 45 180 85
float	每行 1 个浮点值。例如： 893.5689	每行 2 个浮点值。例如： 893.5689 3459.3452	每行 4 个浮点值。例如： 893.5689 39.32 459.352 349.345
cfloat	不适用	每行 1 个浮点 cfloat 值：实数，虚数。例如： 893.5689 24156.456	每行 2 个浮点 cfloat 值：实数，虚数。例如： 893.5689 24156.456 93.689 256.46

仿真输出文件

在每个输出 PLIO 端口上，仿真器可使用相同类型的声明作为输入 PLIO 数据文件来自动创建包含串流内容的文件：

```
adf::output_plio out1 =
adf::output_plio::create("DataOut1",adf::plio_32_bits,"output1.txt");
adf::output_plio out2 =
adf::output_plio::create("DataOut2",adf::plio_64_bits,"output2.txt");
adf::output_plio out3 =
adf::output_plio::create("DataOut3",adf::plio_128_bits,"output3.txt");
```

对于输出创建的文件，适用的格式与输入相同。根据数据类型和 PLIO 带宽，将在每一行上显示一定量的数据。仿真器会为每个输出行添加时间戳，以便我们估算仿真期间的数据吞吐量。此时间戳并没有统一的时间单位，可以采用：

- 皮秒 (ps)
- 纳秒 (ns)
- 微秒 (us)
- 毫秒 (ms)
- 秒 (s)

如果串流的来源是在每一帧末尾生成的 TLAST 标记，那么此 TLAST 也会写入输出文件。以下提供了此类输出文件示例：

```
...
T 15984 ns
4552 4555
T 15988 ns
4558 4561
T 15992 ns
4564 4567
T 15996 ns
4570 4573
T 16 us
4576 4579
T 16004 ns
4582 4585
T 16008 ns
4588 4591
T 16012 ns
4594 4597
T 16016 ns
4600 4603
T 16020 ns
4606 4609
T 16024 ns
TLAST
4612 4615
T 17940 ns
4618 4621
T 17944 ns
4624 4627
T 17948 ns
4630 4633
T 17952 ns
...
```

综上，PLIO 端口的每项输出都具有如下格式：

- 时间戳
- 可选 TLAST
- 样本值

您可基于输出来为此 PLIO 端口估算设计吞吐量。时间戳仅与有效输出相关。当 PLIO 端口处于静默状态时，输出文件上没有任何指示信息。

吞吐量计算方式为输出样本数除以时间戳差值：

$$\text{吞吐量} = \text{样本数} / (\text{最后一个时间戳} - \text{第一个时间戳})$$

如果不考虑最后一个输出样本之后发生的所有时钟周期，那么此简单公式将过高估算吞吐量。

对于基于帧的输出，可能会过高估算此结果。在此情况下，输出格式如下：

- 帧 0 输出（tstart_0 到 tend_0）
- 帧间静默

- 帧 1 输出（tstart_1 到 tend_1）
- 帧间静默
- ...
- 帧 N-1 输出（tstart_N-1 到 tend_N-1）
- 帧间静默
- 帧 N 输出（tstart_N 到 tend_N）

在此情况下，必须考虑每个帧输出的帧间耗用时间。如果仅使用前 N 个帧（从 0 到 N-1），那么这是可以做到的。在此情况下，可以将吞吐量公式的时间戳替换为：

- FirstTimestamp = tstart_0
- LastTimestamp = tstart_N（最后一帧的第一个输出时间戳）

以下提供了计算 PLIO 吞吐量的 Python 脚本示例，其中对仿真输出文件进行了分析并对吞吐量进行了计算。

```
import numpy as np
from math import *
import sys
import argparse

def GetTime_ns(Stamp):
  Time_ns = float(Stamp[1])
  if(Stamp[2] == 'ps'):
    Time_ns = Time_ns/1000.0
  elif(Stamp[2] == 'us'):
    Time_ns = Time_ns*1000.0
  elif(Stamp[2] == 'ms'):
    Time_ns = Time_ns*1000000.0
  elif(Stamp[2] == 's'):
    Time_ns = Time_ns*1000000000.0
  return(Time_ns)


def ReadFile(filename):
  # Detect the number of data per PLIO output
  fdr = open(filename,'r')
  ts = fdr.readline()
  d = fdr.readline()
  dw = d.split()
  fdr.close()


  coltime = 0
  coldata = 1
  numdata = len(dw)
  coltlast = numdata + 1


  # Initializes the output array
  # Format: timestamp (in ns) val1 val2 ... valN TLAST (0 or 1)
  a = np.zeros((0,numdata+2))
  fdr = open(filename,'r')
  line = ' '
  lnum = 0;

  while line !="" :
```

```
    line = fdr.readline()
    if line=='':
      continue
    res = line.split()

    if(res[0] != 'T'): # It should be a timestamp
      continue

    l = np.zeros((1,numdata+2))
    # Extract the time stamp
    l[0][0] = GetTime_ns(res)


    line = fdr.readline()
    res = line.split()
    # extract the TLAST
    if(res[0]=='TLAST'):
      tlast = 1
      line = fdr.readline()
      res = line.split()
    else:
      tlast = 0

    l[0,coltlast] = tlast
    # Extract all values
    for i in range(numdata):
      l[0,i+1] = float(res[i])

      # Appends to the whole array
      a = np.append( a , l,axis=0)

  fdr.close()
  return(a)

def Throughput(Filename,IsComplex):
  V = ReadFile(Filename)
  print("\n==============================")
  print(Filename)
  print("\n")

  NRows = V.shape[0]
  NCols = V.shape[1]
  NFullFrames = int(np.sum(V[:,NCols-1]))
  print("Number of Full Frames: " + str(NFullFrames))


  # Basic Throughput computation
  if IsComplex:
    Ratio = 0.5
  else:
    Ratio = 1
  RawThroughputMsps = float(NRows*(NCols-2))/(V[NRows-1,0]-
V[0,0])*Ratio*1000.0
  print("Raw Throughput: %.2f" % RawThroughputMsps)

  # If the output is frame based, compute a more precise throughput
  tlast = np.where(V[:,NCols-1] == 1.0)
  if(len(tlast[0])<=1):
    TotalThroughput = RawThroughput
  else:
    tlast = tlast[0]
    EndRow = tlast[len(tlast)-2]+1
    # EndRow is the number of Rows I take into account for the number of
```

```
datasource
    # The timestamp I am interested in is the timestamp of the next
transaction
    TotalThroughputMsps = float(EndRow*(NCols-2))/(V[EndRow,0]-
V[0,0])*Ratio*1000.0
    print(" Throughput: %.2f" % TotalThroughputMsps)

  print("\n")

# Entry point of this file
if __name__ == "__main__":
  parser = argparse.ArgumentParser(prog=sys.argv[0], description='Compute
the throughput corresponding to some output of AIE Simulations')
  parser.add_argument('--iscomplex', action='store_true', help='Indicates
Complex data in the file')
  parser.add_argument('filename',nargs='+')
  Args = sys.argv
  Args.pop(0)
  args = parser.parse_args(Args)

for f in args.filename:
  Throughput(f,args.iscomplex)
```

全局存储器仿真

当应用使用 GMIO 规范访问全局存储器（请参阅《AI 引擎内核与 Graph 编程指南》(UG1079) 中的配置 input_gmio/output_gmio）时，仿真需对 DDR 存储器建模，布线网络则会将 DDR 存储器连接至 PL 和 AI 引擎。AI 引擎到 DDR 存储器的连接由 AI 引擎阵列接口中嵌入的 DMA 数据移动器进行调解，并通过 PS 程序中的 GMIO API 进行控制。从 PL 上的 AXI4-Stream 端口到 DDR 存储器的连接由软核 GMIO 数据移动器块进行调解，此块是由 AI 引擎编译器自动生成的，用于仿真。数据移动器会通过 NoC 将源自 PL 块的串流接口转换为 AXI4 存储器映射接口传输事务，其中包含特定起始地址、块大小和突发量，如《AI 引擎内核与 Graph 编程指南》(UG1079) 中的配置 input_gmio/output_gmio 所示。

对全局存储器进行仿真时，可使用附加的 `--gm-init-file` 选项提供存储器数据文件，该选项使用预定义数据来初始化 DDR 存储器。此文件是从给定地址起始的 DDR 存储器的文本字节转储。此文件格式如下：

```
<startaddr>:
<byte>
<byte>
...
```

例如，可按如下方式通过全局存储器初始化来调用 AI 引擎仿真器：

```
aiesimulator –-pkg-dir=./Work –-gm-init-file=dump.txt
```

仿真器还会在仿真输出目录（默认目录：`aiesimulator_output`）中，为所使用的 DDR 存储器生成输出字节转储。输出文件的名称基于从基址 `0x0` 开始的 DDR 存储体（例如，`DDRMC_SITE_X1Y0.mem`）的内部位置。您可使用此转储来验证全局存储器传输事务。

硬件仿真

要对整个系统（包括 AI 引擎 graph 和 PL 逻辑以及用于控制 AI 引擎和 PL 的基于 XRT 的主机应用）进行仿真，您必须针对特定开发板和平台使用 Vitis 硬件仿真流程。此流程包括 AI 引擎的 SystemC 模型，以及用于 NoC、DDR 存储器、PL 内核 (RTL) 和 PS（在 QEMU 上运行）的传输事务级 SystemC 模型。您也可为自己的平台或设计包含 RTL 逻辑和测试激励文件 PL 逻辑。如需了解有关此流程的详细信息，请参阅 运行硬件仿真。

复用 AI 引擎仿真器选项

AI 引擎仿真器会生成一个选项文件，其中列出了用于对 AI 引擎 graph 应用进行仿真的选项。运行 AI 引擎仿真器时，会自动生成该选项文件。后续在系统级别硬件仿真中，您可以复用在初始 graph 级别仿真运行期间所使用的 AI 引擎仿真器选项。您也可以手动编辑选项文件以按需指定其它选项。下表列出了可在 `aiesim_options.txt` 文件内指定的选项。此文件位于 `aiesimulator_output` 目录中，如果 `--dump-vcd` 选项（仅对 XSIM 仿真器有效）或者 `--profile` 选项与 `aiesimulator` 命令搭配使用，就会创建此文件。在命令行选项中可指定此文件，以使用 `launch_hw_emu.sh` 脚本启动硬件仿真器，如 在硬件中运行系统 中所述。命令行示例如下所示。

```
./launch_hw_emu.sh \
-add-env VITIS_LAUNCH_WAVEFORM_BATCH=1 \
-aie-sim-options ${FULL_PATH}/aiesimulator_output/aiesim_options.txt
```

其中 `${FULL_PATH}` 必须是指向此文件或目录的完整路径。

表 33：**适用于硬件仿真的 AI 引擎选项**

命令	实参	描述
`AIE_DUMP_VCD`	<filename>	指定 `AIE_DUMP_VCD` 时，仿真会生成 VCD 数据，并将其写入指定的 `<filename>.vcd`。
`AIE_DEBUG_AXIMM`	True | False	启用 `AIE_DEBUG_AXIMM` (`True`) 时，仿真会生成存储器映射 AXI4 传输事务数据，并将其写入 `aiesim_debug_axi_mm_dump.txt` 文件。
`AIE_PROFILE`	All | (1,2)(3,4)...	该选项会剖析所有已使用的 AI 引擎或列出的选定 AI 引擎。硬件仿真会在 `sim/behav_waveform/xsim` 目录中生成剖析数据文件，在 Vitis 分析器中打开 `default.aierun_summary` 文件即可查看这些文件。该选项还会将 ADF 内核 `printf` 数据记录到 `sim/behav_waveform/xsim/simulate.log` 文件中。

重要提示！您必须使用下列任一方法将 AI 引擎编译器 `workdir` 环境变量设置为由 AI 引擎编译器生成的 `Work` 目录：

- 在 `launch_hw_emu.sh` 的命令行中使用 `-add-env`。例如，在 Makefile 中，`./launch_hw_emu.sh -aie-sim-options ./sim_options.txt -add-env AIE_COMPILER_WORKDIR=/yourdesigndirectory/Work`。
- 启动赛灵思仿真器之前，请在 shell 窗口中输入 `setenv AIE_COMPILER_WORKDIR /yourdesigndirectory/Work`。

注释：包含文件路径的任何命令都必须使用绝对路径。

手动创建仿真选项文件时，需遵循 `COMMAND=ARGUMENT` 格式，每条命令单独一行。最佳实践如以下示例所示。

```
AIE_PROFILE=All
AIE_VCD=foo
```

以下命令用于在硬件仿真期间启动赛灵思仿真器波形 GUI。

```
./launch_hw_emu.sh -g
```

此外，您可以添加更多高级选项来记录波形数据，而无需随 Vivado 逻辑仿真器 GUI 启动仿真。命令行示例如下所示。

```
./launch_hw_emu.sh \
-user-pre-sim-script pre-sim.tcl
```

`pre-sim.tcl` 包含用于添加波形或记录设计波形的 Tcl 命令。如需获取示例，请参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393)，如需查看 Tcl 命令，请参阅《Vivado Design Suite 用户指南：逻辑仿真》(UG900)。

提示：如需了解有关如何在硬件仿真中调试应用的详细信息，请参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的硬件仿真中的调试技巧。

启用第三方仿真器

对设计执行硬件仿真时，支持使用第三方仿真器，例如，Questa Advanced Simulator (Mentor Grahpics)、Xcelium (Cadence)、VCS (Synopsys) 和 Riviera Simulator (Aldec)。您可以通过更新 Vitis 配置文件（`config.ini` 或 `system.cfg`）来启用这些仿真器。

表 34：**Vitis 链接设置**

仿真器	v++ --link 配置

Questa Advanced Simulator	`[advanced]`
	`param=hw_emu.simulator=QUESTA`
	`[vivado]`
	`prop=project.__CURRENT__.simulator.questa_install_dir=<SIMULATOR DIRECTORY>/questa/2022.2/bin`
	`prop=project.__CURRENT__.compxlib.questa_compiled_library_dir=<SIMULATOR LIBRARY DIRECTORY>/questa/2022.2/lin64/lib/`
	`prop=fileset.sim_1.questa.compile.sccom.cores={16}`
	`prop=fileset.sim_1.questa.elaborate.vopt.more_options={-stats=all}`
	`prop=fileset.sim_1.questa.simulate.vsim.more_options={-stats=all}`

Xcelium	`[advanced]`
	`param=hw_emu.simulator=XCELIUM`
	`[vivado]`
	`prop=project.__CURRENT__.simulator.xcelium_install_dir=<SIMULATOR DIRECTORY>/xcelium/bin/`
	`prop=project.__CURRENT__.compxlib.xcelium_compiled_library_dir=<SIMULATOR LIBRARY DIRECTORY>/xcelium/21.09.009/lin64/lib/`
	`prop=fileset.sim_1.xcelium.elaborate.xmelab.more_options={-timescale 1ns/1ps -STATUS]`

VCS	`[advanced]`
	`param=hw_emu.simulator=VCS`
	`[vivado]`
	`prop=project.__CURRENT__.simulator.vcs_install_dir=<SIMULATOR DIRECTORY>/vcs/S-2021.09-SP2/bin/`
	`prop=project.__CURRENT__.compxlib.vcs_compiled_library_dir=<SIMULATOR LIBRARY DIRECTORY>/vcs/S-2021.09-SP2/lin64/lib/`
	`prop=project.__CURRENT__.simulator.vcs_gcc_install_dir=<SIMULATOR DIRECTORY>/synopsys/vg_gnu/2021.09/linux64/gcc-9.2.0_64/bin`
	`param=project.alignLibraryPathEnvForVCS=true`
	`prop=fileset.sim_1.vcs.compile.vlogan.more_options={-v2005}`

Riviera	`[advanced]`
	`param=hw_emu.simulator=RIVIERA`
	`[vivado]`
	`prop=project.__CURRENT__.simulator.riviera_install_dir=<SIMULATOR DIRECTORY>/riviera/2022.04-lin64/bin/`
	`prop=project.__CURRENT__.compxlib.riviera_compiled_library_dir=<SIMULATOR LIBRARY DIRECTORY>/riviera/2022.04/lin64/lib/`
	`prop=project.__CURRENT__.simulator.xcelium_gcc_install_dir=<SIMULATOR DIRECTORY>/riviera/2022.04-lin64/gcc_Linux64/bin/`
	`prop=fileset.sim_1.riviera.simulate.asim.more_options={+access +r}`

完成修改后，请照常构建设计、运行 `launch_hw_emu.sh` 脚本，这将使用新的仿真器。如需了解有关仿真的更多信息，请参阅 在硬件中运行系统。

AMD XILINX

第 5 章

仿真期间对 AI 引擎 graph 应用进行性能分析

程序执行的系统级视图有助于识别程序执行期间的问题，包括执行正确与否以及性能问题。使用传统交互式调试器进行调试难以奏效，例如，可能出现包括锁定缺失、锁定不匹配、缓冲器溢出以及 DMA 缓冲器编程错误等问题。因此需要采用系统化的方法来收集程序执行的系统级追踪。AI 引擎架构可为仿真、硬件仿真或硬件执行期间的事件生成、收集和串流（作为追踪数据）提供直接支持。

在硬件仿真中分析 AI 引擎状态

生成和分析 AI 引擎状态

在硬件仿真期间，可定期生成状态输出。在 Vitis™ 分析器中可分析状态和警告以便调试。

1. 在 `xrt.ini` 中添加以下选项即可输出 AI 引擎状态。这是一次性设置，它会导致定期输出状态数据，包括死锁检测。
   - `aie_status=true`：开启输出生成功能特性。
   - `aie_status_interval_us`：指定 AI 引擎状态输出的时间间隔。

     示例：

     ```
     [Debug]
     aie_status=true
     aie_status_interval_us=1000
     ```

   硬件仿真会生成以下文件。
   - `xrt.run_summary`：由硬件仿真生成的汇总文件，可在 Vitis 分析器内打开。
   - `aie_status_edge.json`：AI 引擎和 AI 引擎存储器的状态。
   - `aieshim_status_edge.json`：AI 引擎接口拼块的状态。
   - `summary.csv`：汇总 CSV 文件。
2. 汇总文件和 JSON 文件必须使用 `scp` 命令从 QEMU Linux 系统复制到您的本地系统。状态文件必须位于本地目录中，才能在 Vitis 分析器内执行分析。如需了解有关如何将这些文件复制到您的本地系统的更多详细信息，请参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的在嵌入式处理器平台上运行仿真
3. 在 Vitis 分析器中分析汇总文件和 JSON 文件。欲知详情，请参阅 在 Vitis 分析器中分析 AI 引擎状态。

基于 AI 引擎仿真的性能分析

在仿真中，要查看带时间戳的事件、不同的事件类型以及与每一起事件相关联的数据，可使用值更改转储 (VCD) 文件。VCD 文件提供了已仿真的硬件信号的详细转储信息。此外，剖析汇总还可提供有关整体应用性能的详细信息（含注解）。

基于 AI 引擎仿真的剖析

剖析数据生成

在仿真框架内，AI 引擎仿真器可以为完整应用生成剖析报告。此报告是使用 `-profile` 标志生成的。

```
aiesimulator -pkg-dir=Work -profile
```

文本文件和 .xml 文件都是在 `aiesimulator_output` 目录中生成的。对于位于 C 列和 R 行中的拼块，会生成两种类型的文件。`*_funct` 用于报告每个函数的调用次数和周期数。`*_instr` 则是深入汇编代码的报告。使用 Vitis 分析器即可直观显示此报告。

```
vitis_analyzer aiesimulator_output/default.aierun_summary
```

“Profile”（剖析）选项卡会打开“Profile”报告，其中显示了含信息的各节菜单。

- “Summary”（汇总）：报告总周期计数、总指令计数和存储器中的程序大小。
- “Function Reports”（函数报告）：在表格和图示中显示逐个函数的多项关键指示符。
  - 调用次数
  - 函数总时间（周期数和 %）
  - 函数总时间 + 后代时间（周期数和 %）
  - 最小/平均/最大函数时间（周期数）
  - 最小/平均/最大函数时间 + 后代时间（周期数）
  - 程序计数器低位/高位
- “Profile Details”（剖析详细信息）：显示逐个函数的汇编代码（含实用的精度）。各列如下表所示。

表 35：**剖析详细列描述**

列名	内容
PC	程序计数器
Instruction（指令）	每行最多 16 字节
Assembly（汇编）	汇编代码助记符，含完整的 7 路指令字
Exe-count（执行计数）	处理器执行该行的次数
Cycles（周期数）	所需的周期数
User Count（用户计数）	
Wait States（等待状态）	对于部分指令，可能会出现存储器冲突，导致多次进入等待状态
Relative cycle use within function（函数内的相对周期使用情况）	显示为“*”行，其中相对长度会直观显示该指令在函数内的相对周期使用情况

表 35：**剖析详细列描述** (续)

列名	内容
Relative cycle use within simulation（仿真内的相对周期使用情况）	显示为“*”行，其中相对长度会直观显示该指令在仿真（包括 `main()` 和所有函数）内的相对周期使用情况
Relative wait-state use within function（函数内的相对等待状态使用情况）	显示为“W”行，其中相对长度会直观显示在函数内执行该指令期间，等待状态所使用的相对周期
Relative wait state use within simulation（仿真内的相对等待状态使用情况）	显示为“*”行，其中相对长度会直观显示在仿真（包括 `main()` 和所有函数）内执行该指令期间，等待状态所使用的相对周期

使用剖析数据进行性能调试

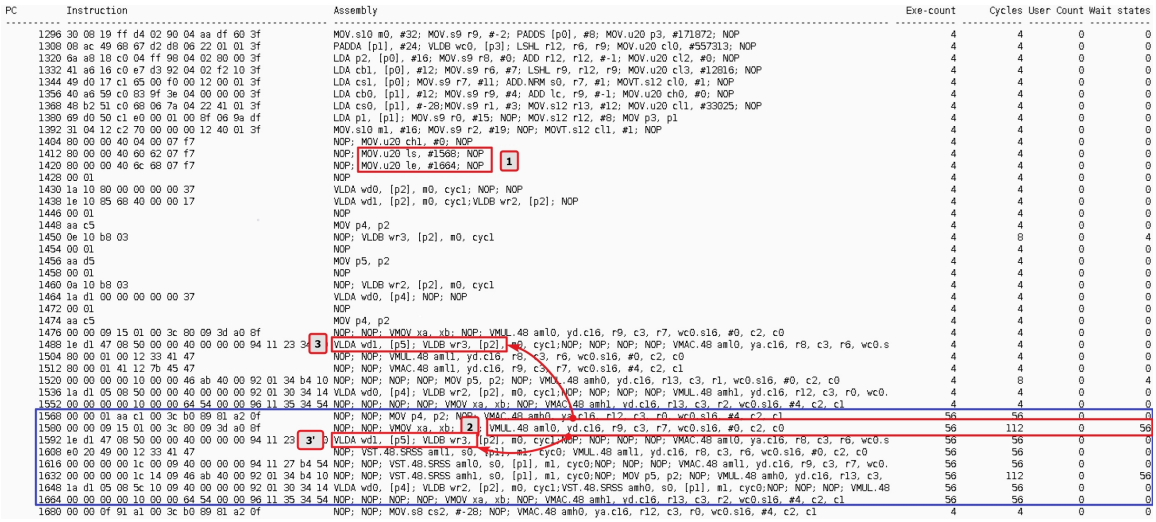
要改善设计的性能，应首先从最优化占用周期数最多的函数开始。完成后即可对占用周期数和比例略少的函数进行最优化。为达成此目的，函数总时间 graph 将帮助您选择这些函数。

对于最优化本身，您可使用编译指示（chess_prepare_for_pipelining 和 chess_loop_range）来执行寻常的循环流水打拍和展开。“Profile Details”（剖析详细信息）选项卡提供了有关等待状态的洞察。即使内层循环完美完成最优化，仍可能由于等待状态而浪费几个周期。当资源访问中存在冲突时，这些等待状态就会出现。

- 在本地 AI 引擎内或者两个连续 AI 引擎内的同一个存储体中执行两次读取或者执行一次读取和一次写入。
- 本地 AI 引擎会尝试访问存储体（读取或写入），而存储体 DMA 同时也在访问该存储体以执行某些数据传输。

以下是剖析详细信息的示例。

图 17：**Profile Details**



在此截屏中，内层循环首先使用 ls 和 le（即循环开始和循环结束 PC）进行本地化。此内层循环显示在蓝色矩形中。这看似正确，因为这些行上的 Exe-Count 远比其它行高。

在第二步中，由于等待状态已本地化，一条指令会持续耗用两个时钟周期，而不是一个时钟周期：

- Exe-count = 56
- Cycles = 112

VMUL 指令会提取寄存器 yd 中的数据和 wc0 中的系数。加载耗时 7 个时钟周期，随后数据才能有效加载到该寄存器中。第三步是对第一次迭代 (3) 的 7 个时钟周期之间的代码进行分析，或者对前一次迭代 (3') 上的循环内的代码进行分析。此处可以看到，每一种情况下都有 2 次加载，且发生在同一个 bank 上（所有读取都来自源代码中的相同窗口）。

基于 AI 引擎仿真的值更改转储

在仿真框架中，AI 引擎仿真器可以值更改转储 (VCD) 文件形式来生成硬件信号的详细转储。凭借一组已定义的抽象事件即可描述多核 AI 引擎程序执行这些事件的情况。`aiesimulator --dump-vcd` 命令可用于启用 VCD 文件输出。

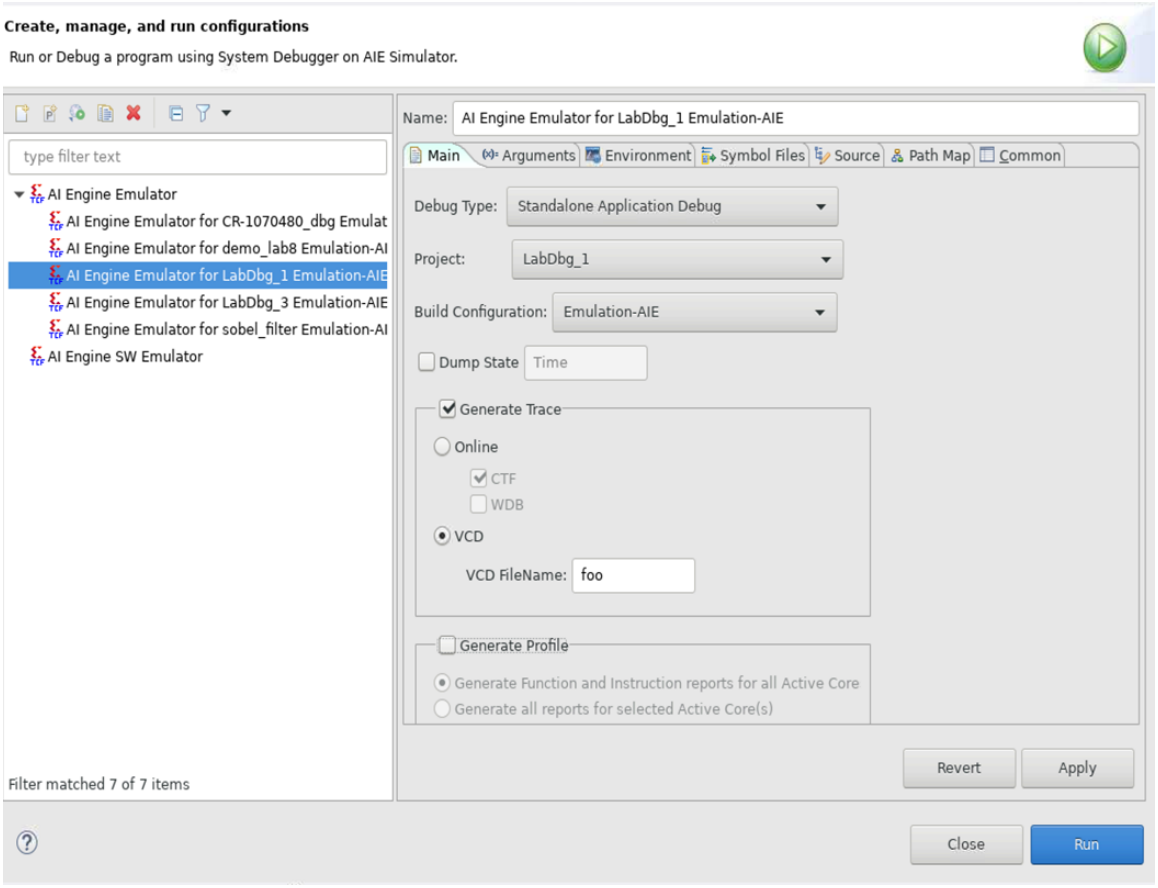
仿真后，可将此 VCD 文件处理至事件中，并在 Vitis™ 分析器内的时间线上查看。这些事件包含与每一起事件相关联的时间戳、不同的事件类型和数据等信息。此信息会与编译器生成的调试信息相关联。其中包括映射到函数名称和指令偏移的程序计数器值，以及存储器访问的源码级符号数据偏移。

AI 引擎抽象事件与 VCD 格式无关，将直接从硬件中提取。事件追踪可作为纯文本、逗号分隔值 (CSV)、通用追踪格式 (CTF) 或波形数据库 (WDB) 来生成，生成的事件追踪数据可在 Vitis 分析器内查看。

VCD 文件生成

要从 Vitis IDE 生成 VCD 文件，请从“Explorer”（资源管理器）视图中右键单击 AI 引擎 graph 工程，然后选择“Run As”→“Run Configurations”（运行方式 > 运行配置），如 创建 AI 引擎 graph 工程和顶层系统工程 中所述。这样即可打开当前工程的“Run Configurations”对话框。

图 18：**通过 Vitis IDE 启用 VCD 文件生成**



选中“AI Engine Emulator”（AI 引擎仿真器）选项，双击打开新配置。选中“Generate Trace”（生成追踪）复选框，以启用追踪捕获，然后选中“VCD Trace”（VCD 追踪）按钮。默认情况下，这样即可在当前目录中名为 `foo.vcd` 的文件内生成 VCD 转储。您可按需重命名此文件。

也可在命令行上调用 AI 引擎仿真器并指定 `--dump-vcd <filename>` 选项来生成 VCD 文件。生成 VCD 文件的目录即运行仿真的目录。假定使用 AI 引擎编译器来编译程序，则可在 shell 中通过 VCD 选项调用仿真器。

```
$ aiesimulator –-pkg-dir=./Work --dump-vcd=foo
```

此命令会生成 VCD 文件 (`foo.vcd`)，并将其写入当前目录。

从 VCD 执行 AI 引擎追踪

提供 `vcdanalyze` 实用工具的目的是为了从 VCD 文件生成 AI 引擎事件追踪。此进程会自动集成到 Vitis 工具流程内。在 Vitis IDE 中，当仿真运行完成 AI 引擎事件捕获后，您可右键单击“Explorer”（资源管理器）视图中的工程，然后选择“Analyze AIE Events”（分析 AI 引擎事件）。这样即可在当前工程的 `Traces/AIE_AXI_Trace` 下生成追踪数据，并且会在当前工程中自动载入各种视图。

目录 Traces/AIE_AXI_Trace/ctf/events.txt 下的原始事件追踪应如下所示：

```
time=1741000,event=FROM_PL,name=tl.me.shim.tile_0_0.pl_interface.pl_to_shim0.data0,col=0,str
eamid=0,data0=65536,data1=0,tlast=0
time=1742000,event=FROM_PL,name=tl.me.shim.tile_0_0.pl_interface.pl_to_shim0.data0,col=0,str
eamid=0,data0=196610,data1=0,tlast=0
time=1743000,event=FROM_PL,name=tl.me.shim.tile_0_0.pl_interface.pl_to_shim0.data0,col=0,str
eamid=0,data0=327684,data1=0,tlast=0
time=1743000,event=CORE_RESET,col=1,row=0
time=1744000,event=FROM_PL,name=tl.me.shim.tile_0_0.pl_interface.pl_to_shim0.data0,col=0,str
eamid=0,data0=458758,data1=0,tlast=0
time=1745000,event=FROM_PL,name=tl.me.shim.tile_0_0.pl_interface.pl_to_shim0.data0,col=0,str
eamid=0,data0=589832,data1=0,tlast=0
time=1746000,event=FROM_PL,name=tl.me.shim.tile_0_0.pl_interface.pl_to_shim0.data0,col=0,str
eamid=0,data0=720906,data1=0,tlast=0
time=1747000,event=FROM_PL,name=tl.me.shim.tile_0_0.pl_interface.pl_to_shim0.data0,col=0,str
eamid=0,data0=851980,data1=0,tlast=0
time=1748000,event=FROM_PL,name=tl.me.shim.tile_0_0.pl_interface.pl_to_shim0.data0,col=0,str
eamid=0,data0=983054,data1=0,tlast=0
time=1749000,event=FROM_PL,name=tl.me.shim.tile_0_0.pl_interface.pl_to_shim0.data0,col=0,str
eamid=0,data0=1,data1=0,tlast=0
time=1750000,event=FROM_PL,name=tl.me.shim.tile_0_0.pl_interface.pl_to_shim0.data0,col=0,str
eamid=0,data0=131075,data1=0,tlast=0
time=1751000,event=FROM_PL,name=tl.me.shim.tile_0_0.pl_interface.pl_to_shim0.data0,col=0,str
eamid=0,data0=262149,data1=0,tlast=0
time=2186000,event=DM_WRITE_REQ,col=0,row=0,port=tl.me.array.tile_0_1.mm.dm.port_AXI_write_b
6
time=2190000,event=DM_WRITE_REQ,col=0,row=0,port=tl.me.array.tile_0_1.mm.dm.port_AXI_write_b
7
time=2194000,event=DM_WRITE_REQ,col=0,row=0,port=tl.me.array.tile_0_1.mm.dm.port_AXI_write_b
6
time=2198000,event=DM_WRITE_REQ,col=0,row=0,port=tl.me.array.tile_0_1.mm.dm.port_AXI_write_b
7
time=2202000,event=DM_WRITE_REQ,col=0,row=0,port=tl.me.array.tile_0_1.mm.dm.port_AXI_write_b
2
time=2206000,event=DM_WRITE_REQ,col=0,row=0,port=tl.me.array.tile_0_1.mm.dm.port_AXI_write_b
3
```

以下命令会以文本格式在 ./trdata/events.txt 文件中为 foo.vcd 生成 AI 引擎追踪数据。

```
vcdanalyze -vcd foo.vcd
```

提示： vcdanalyze -h 可用于获取命令帮助。

以下命令可根据来自 foo.vcd 文件的 AI 引擎追踪数据生成 CSV 文件。

```
vcdanalyze -vcd=foo.vcd -csv
```

以下命令可根据来自 foo.vcd 文件的 AI 引擎追踪数据生成波形数据文件。

```
vcdanalyze -vcd foo.vcd -wdb
```

在 Vitis 分析器中查看运行汇总

运行系统后，无论是在仿真、硬件仿真还是硬件中，正确完成应用配置时都会生成 run_summary 报告。

在 AI 引擎 graph 仿真期间，AI 引擎仿真器或硬件仿真会捕获性能和活动指标，并将报告写入输出目录 `./aiesimulator_output` 和 `./sim/behav_waveform/xsim`。生成的汇总名为 `default.aierun_summary`。

`run_summary` 可在 Vitis 分析器中查看。此汇总包含报告集合，这些报告捕获了 AI 引擎应用运行时捕获的性能剖析汇总信息。例如，要打开 AI 引擎仿真器运行汇总，请使用以下命令：

```
vitis_analyzer ./aiesimulator_output/default.aierun_summary
```

这样会打开 Vitis 分析器并显示报告的“Summary”（汇总）页面。该工具的“Report Navigator”（导航器报告）视图会列出汇总中可用的不同报告。如需获取 Vitis 分析器的完整信息，请参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的使用 Vitis 分析器。

注释： `default.aierun_summary` 同样包含一部分与 `<GRAPH_TB_FILE_NAME>.aiecompile_summary` 相同的报告。这些报告是“Graph”和“Array”。要查看这些报告，请转至 在 Vitis 分析器中查看编译结果。

汇总报告

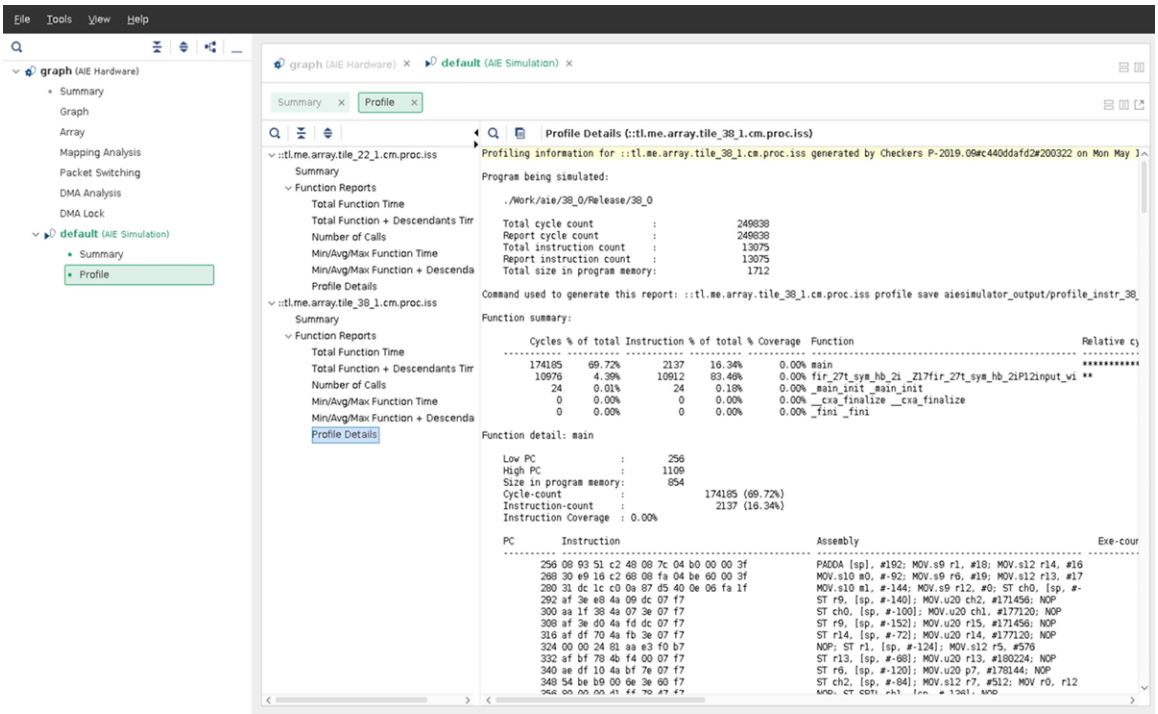
这是顶层报告，用于报告运行的详细信息，例如，用于启动仿真器的日期、工具版本和命令行。

剖析汇总

指定 `aiesimulator --profile` 选项后，仿真器会收集有关 AI 引擎 graph 和内核的剖析数据，并呈现 AI 引擎 graph 的高层次视图、映射到处理器的内核以及指标数据的表格和图形化表示。

“Profile Summary”报告可提供有关整体应用性能的详细信息（含注解）。应用执行期间生成的所有数据将按类别进行分组。“Profile Summary”（剖析汇总）允许您检验处理器/DMA 存储器停滞、死锁、推断、关键路径和最大争用。这对于系统级别的性能调优和调试很有用。系统性能根据时延（用于执行系统的周期数）和吞吐量（数据/耗费的时间）来呈现。系统性能欠佳会强制您检验并控制（通过约束）映射和缓冲器封装、串流和包切换分配、与相邻处理器的交互以及外部接口。以下显示了原始“Profile Summary”（剖析汇总）报告的示例。

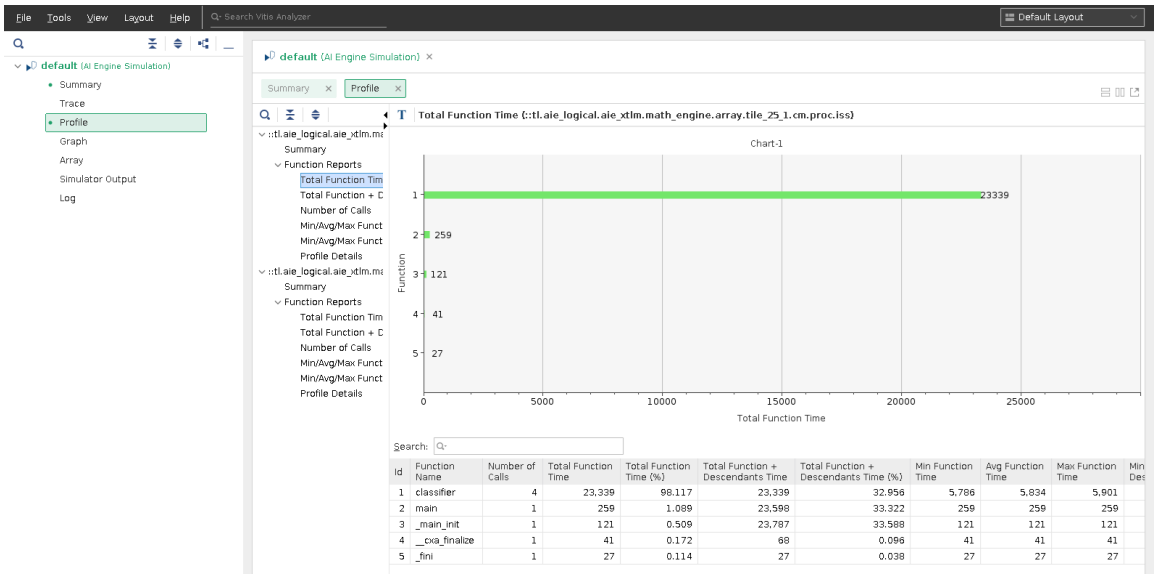
图 19：**剖析汇总**



注释：源于剖析报告的 tile（拼块）编号的行号值是实际拼块编号加 1。例如，tile_38_1 源自先前截图中的 tile(38, 0)。

特定的表可用于查看特定于内核的剖析信息。此信息以图表形式来显示，其中包含显示拼块上当前运行的操作的表格。图表示例如下所示。

图 20：**图表示例**



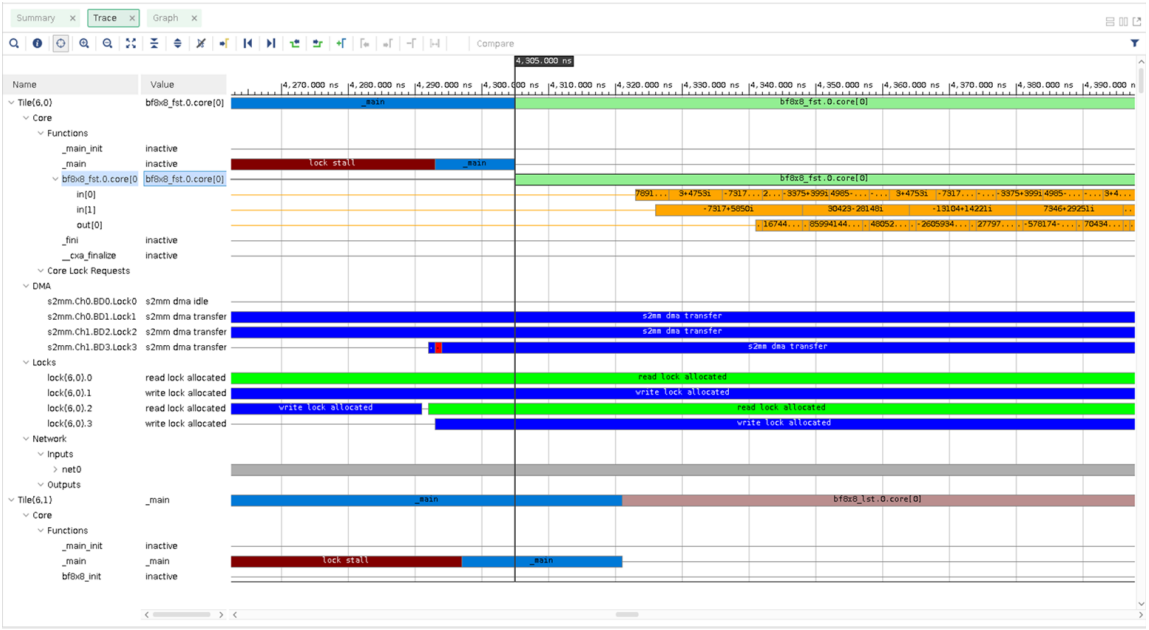
Id	Function Name	Number of Calls	Total Function Time	Total Function Time (%)	Total Function + Descendants Time	Total Function + Descendants Time (%)	Min Function Time	Avg Function Time	Max Function Time
1	classifier	4	23,339	98.117	23,339	32.956	5,786	5,834	5,901
2	main	1	259	1.089	23,598	33.322	259	259	259
3	_main_init	1	121	0.509	23,787	33.588	121	121	121
4	__cxa_finalize	1	41	0.172	68	0.096	41	41	41
5	_fini	1	27	0.114	27	0.038	27	27	27

在此视图中，您可看到显示“Total Function Time”（总计函数时间）的图表，其中显示了用于运行 graph 的函数的周期总数。Y 轴显示的是函数的“id”，可供在下表中引用。此信息对于判定函数中最为耗时的操作很有用，故而有助于最优化或调试。

追踪报告

锁定缺失、锁定不匹配、缓冲器溢出以及 DMA 缓冲器编程错误等问题，使用传统交互式调试技巧进行调试难以奏效。事件追踪可以提供系统化的方法来收集编程事件的系统级别追踪，为硬件事件（作为追踪）的生成、收集和串流提供直接支持。下图显示了 Vitis 分析器中打开的“Trace”（追踪）报告。

图 21：**追踪报告**



注释：此示例演示了内核函数和由编译器添加的内核函数：

- `_main`：核 `main` 函数。此类函数与顶层文件中使用的函数不同。
- `_main_init`：每次执行 graph 时运行一次的内核 `init` 函数。
- `_cxa_finalize`：调用全局 C++ 对象的析构函数。
- `_fini`：本部分可保存用于终止进程的可执行指令。如果程序正常退出，那么系统会运行这部分中的代码。

注释：如果 VCD 文件过大，Vitis 分析器分析 VCD 并打开“Trace”视图耗时过长，那么您可在运行 AI 引擎仿真器时执行 VCD 在线分析。随后，Vitis 分析器会打开现有 WDB 和 CTF 文件，而不是分析 VCD 文件。AI 引擎仿真器命令如下。

```
aiesimulator --pkg-dir=./Work --online -wdb -ctf
```

追踪报告的功能特性如下。

- 报告每个拼块。在每个拼块内，报告包含核、DMA、锁定和 I/O（前提是 graph 中存在 PL 块）。
- 映射到核的每个内核都有一条独立的时间线。当内核在执行（蓝色）时或者因存储器冲突或等待串流数据而停滞（红色）时，会显示此报告。
- 您可使用核、DMA 和锁定部分中的锁定 ID 来识别各核及 DMA 通过获取和释放锁定来彼此交互的方式。
- 锁定部分显示了拼块中的锁定活动，包括对应读取和写入锁定请求的分配和释放活动。相邻拼块可以分配特定锁定。因此，本部分并不一定与图中左侧窗格所示核的核锁定请求相匹配。
- 如果锁定未释放，则会出现一长条红色条形，延申至仿真时间末。
- 单击左箭头或右箭头可分别转至状态开始和结束位置。
- 数据视图可显示流经串流开关网络的数据，在每个中继段中都带有从入口点和主出口点。这对于查找布线延迟以及因包切换而产生的网络拥塞影响最有用，在此类情况下，如果两个包共享相同串流通道，那么其中一个包可能延迟至另一个包之后。

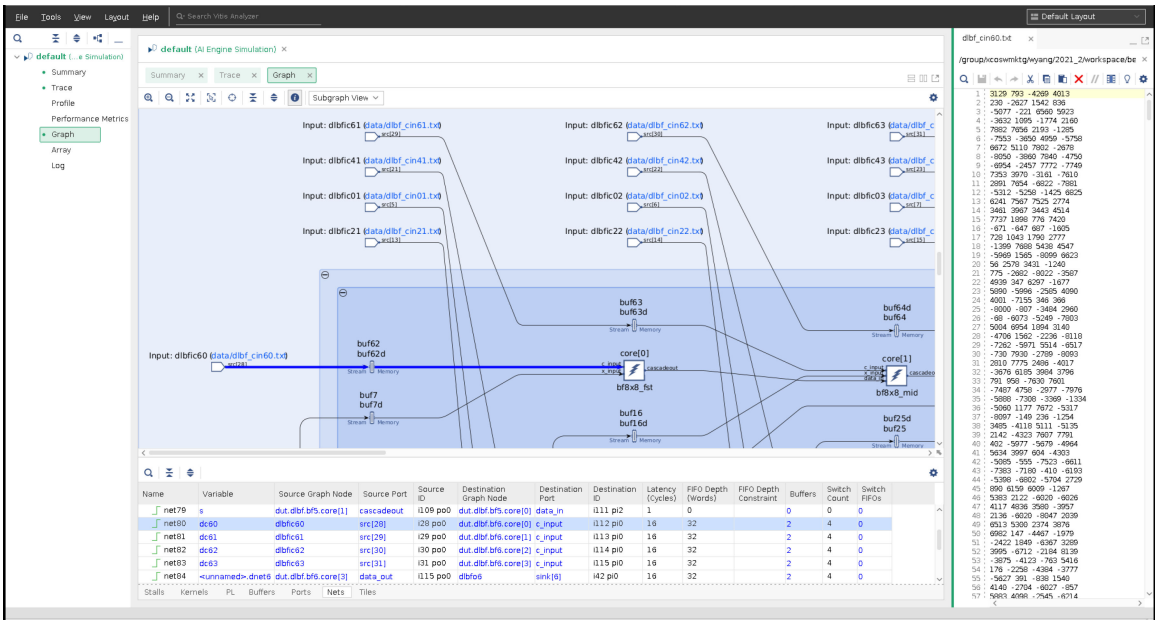
追踪视图数据可视化

追踪视图数据可视化允许您并排显示 I/O 端口处的各种事件，以便您审查设计并检验相关事件时序。这样即可提供有关设计在多核环境下的工作状况的深入见解。

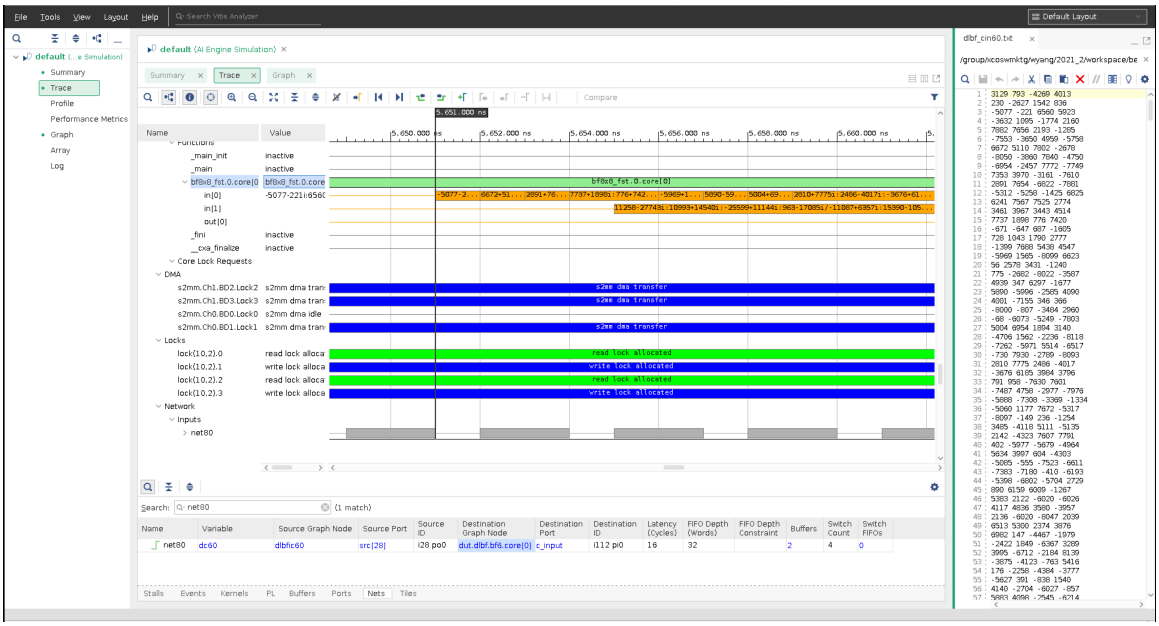
窗口数据分析

追踪视图中的窗口数据分析允许您执行从 graph 视图中的特定输入/输出窗口信号线连接到事件追踪视图中的相应位置的交叉探测。在内核开始执行之前，您可对正在缓冲的窗口输入数据进行追踪。要在窗口接口上执行数据分析，请使用以下步骤。

1. 下图显示了“Graph”视图中已选中的信号线连接。在该窗口底部的 graph 视图和信号线表中，选定的信号线 (`net80`) 均已高亮显示。此外，“Graph”视图还会将仿真输入和输出文件与相应的信号线加以关联。与 `net80` 相关联的仿真输入文件会显示在右侧。



2. 在 graph 视图中，选中连接到 `net80` 的文件源，如下图所示。输入数据显示在窗口右侧。
3. 切换至“Trace”（追踪）视图即可查看事件和详细的事件时序。



4. 在屏幕底部的事件表中，从列选择器中选中“Net”（信号线）。
5. 在相邻的筛选框内输入 `net81`。
6. 这样就会在事件表中高亮显示与 `net80` 相匹配的信号线事件。如有多项事件，可使用“Previous”/“Next”（上一项/下一项）工具栏按钮显示匹配的事件，以便您浏览各项事件。
7. 高亮的数据与来自输入文件的数据（显示在右侧窗格内）相匹配。

   注释：仅当所有数据都在窗口接口上可用时，AI 引擎内核才会启动。

您可使用这些步骤来检验所需窗口连接的 I/O 数据，以确保往来内核的 I/O 数据正确无误。

受支持的窗口数据类型

表 36：**受支持的窗口数据类型**

数据类型	显示格式示例
int8[1]	±123
int16	±12345
int32	±1234567890
int64	±1234567890123456789
uint8	123
uint16	12345
uint32	1234567890
uint64	12345678901234567890
cint16	±12345+±12345i
cint32	±1234567890+±1234567890i
float	±1.234567
cfloat	±1.234567+±1.234567i
v4cint16	±1+2i:3+4i:5+6i:7+8i

表 36：**受支持的窗口数据类型** (续)

数据类型	显示格式示例
v4int32	±1:2:3:4
v4cint32	±1+2i:3+4i
v4int64[4]	±1:2/3:4
v4float	±1.000000:2.000000:3.000000:4.000000
v4cfloat	±1.000000+2.000000i:3.000000+4.000000i
v8int16	±1:2:3:4:5:6:7:8
v8cint16	±1+2i:3+4i:5+6i:7+8i
v8int32	±1:2:3:4:5:6:7:8
v8float	±1.000000:2.000000:3.000000:4.000000:5.000000:6.000000:7.000000:8.000000
v16int8	±1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16
v16uint8	1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16
v16int16[2]	±1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16
v16uint16[2]	1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16
v16int32	±1:2:3:4:5:6:7:8
v16float[2]	±1.000000:2.000000:3.000000:4.000000:5.000000:6.000000:7.000000:8.000000:9:000000:10.000000:11.000000:12.000000:13.000000:14.000000:15.000000:16.000000
v16cfloat[2]	±1.000000+2.000000i:3.000000+4.000000i:5.000000+6.000000i:7.000000+8.000000i:9:000000:10.000000i:11.000000:12.000000i:13.000000:14.000000i:15.000000:16.000000i
v32int8[3]	±1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:20:21:22:23:24:25:26:27:28:29:30:31:32
v32uint8[3]	1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:20:21:22:23:24:25:26:27:28:29:30:31:32
v32int16	±1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16
v32cint16	±1+2i:3+4i:5+6i:7+8i:9+10i:11+12i:13+14i:15+16i
v32int32	±1:2:3:4:5:6:7:8
v32float	±1.000000:2.000000:3.000000:4.000000:5.000000:6.000000:7.000000:8.000000
v64int8	±1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:20:21:22:23:24:25:26:27:28:29:30:31:32 ±33:34:35:36:37:38:39:40:41:42:43:44:45:46:47:48:49:50:51:52:53:54:55:56:57:58:59:60:61:62:63:64
v64uint8	1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:20:21:22:23:24:25:26:27:28:29:30:31:32 33:34:35:36:37:38:39:40:41:42:43:44:45:46:47:48:49:50:51:52:53:54:55:56:57:58:59:60:61:62:63:64
v64int16	±1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16 ±17:18:19:20:21:22:23:24:25:26:27:28:29:30:31:32

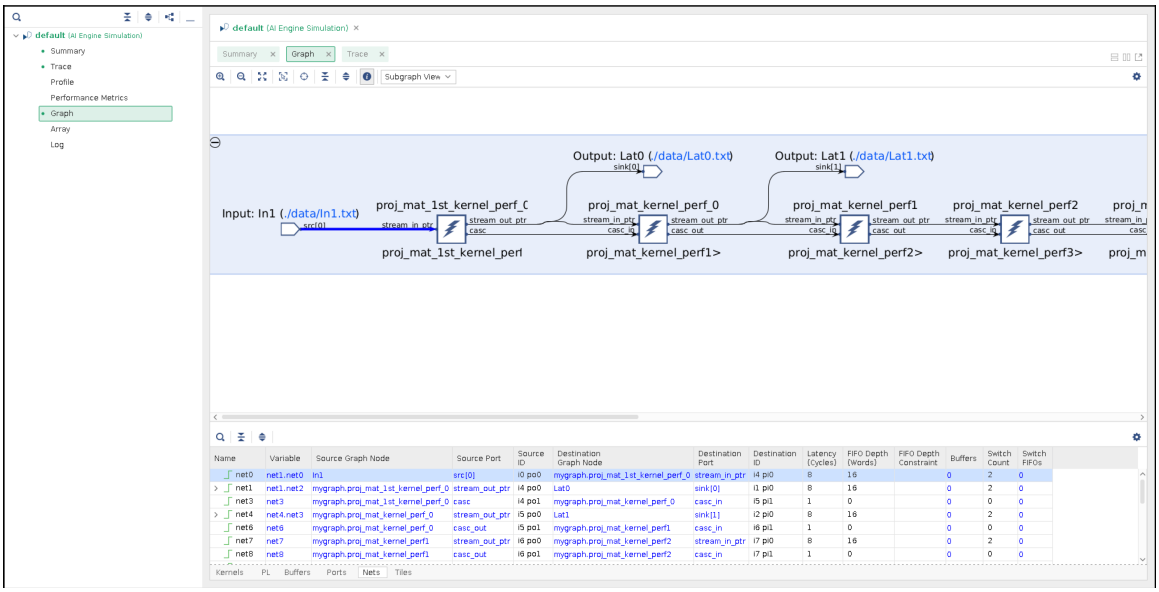
注释：

1. 前置 0 和“+”不予显示。
2. 当前时间线视图和事件表中的数据未对齐或无序排列。
3. 对于 256 位数据，每个 128 位数据均以“ / ”代替“ : ”来加以分隔。例如，1:2:3:4 / 5:6:7:8。
4. 2 个样本位于时间戳 x+1 处，后 2 个样本位于时间戳 x 处。

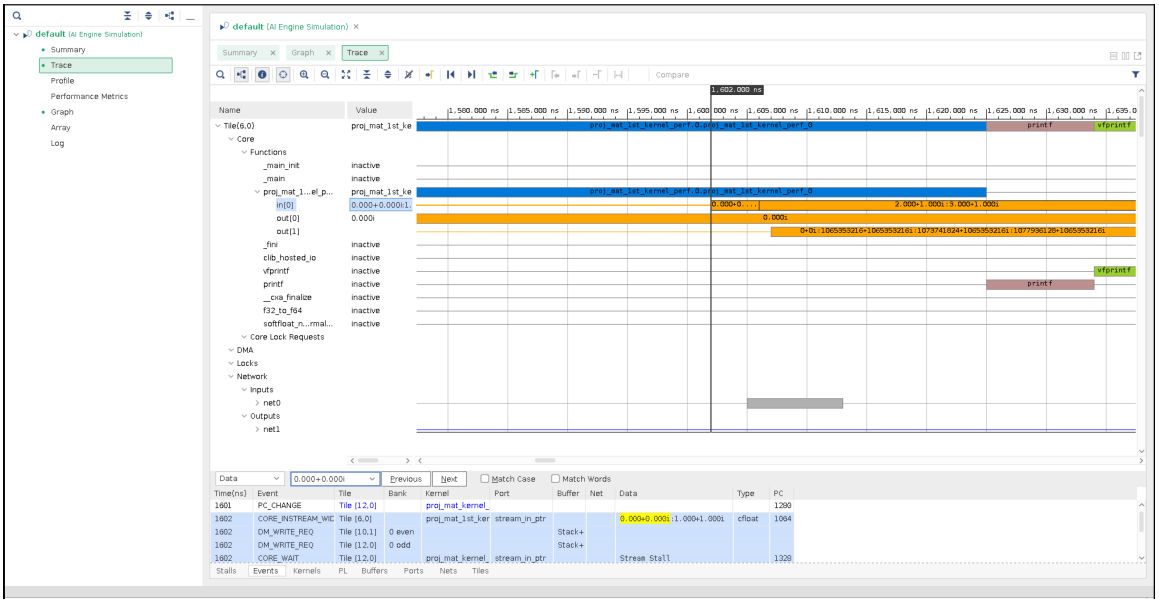
串流数据分析

追踪视图中的串流数据分析允许您执行从 graph 视图中的特定输入/输出串流信号线连接到事件追踪视图中的相应位置的交叉探测。在内核执行期间，您可对当前接收的串流输入数据进行追踪。要在串流接口上执行数据分析，请使用以下步骤。

1. 下图显示了“Graph”视图中已选中的内核端口。在该窗口底部的“Graph”视图和信号线表中，会高亮显示选定的 ID 为 `net0` 的信号线。



2. 切换至“Trace”（追踪）即可查看事件和详细的事件时序。



3. 拖动窗口上半部分中的垂直线（时间标记）即可将时间标记移至串流数据开始处。
4. 在窗口上半部分中，您可以检验与所选输入端口相关联的数据。
5. 在屏幕底部的事件表中，从下拉菜单选择“Data”（数据）。
6. 将 `0.000+0.000i` 输入相邻（筛选）框中，然后单击“Next”（下一步）。事件表中会高亮显示期望的时间和数据。

   注释：输入值取决于数据类型。此示例中的 `0.000+0.000i` 对应于复数浮点类型。

您可使用这些步骤来检验所需串流连接的 I/O 数据，以确保往来内核的 I/O 数据正确无误。

受支持的串流数据类型

表 37：**受支持的串流数据类型**

数据类型	显示格式
int8[1]	±123
int16	±12345
int32	±1234567890
int64[3]	±1234567890123456789
uint8	123
uint16	12345
uint32	1234567890
uint64[3]	12345678901234567890
cint16	±12345+±12345i
cint32[2]	±1234567890+±1234567890
float	±1.234567
cfloat[2]	±1.234567+±1.234567
acc48	±1:2:3:4:5:6:7:8
cacc48	±1+1i_2+2i_3+3i_4+4i
acc80[4]	不适用
cacc80[4]	不适用
accfloat	±1.000000:2.000000:3.000000:4.000000:5.000000:6.000000:7.000000:8.000000
caccfloat	1.000000+2.000000i:3.000000+4.000000i:5.000000+6.000000i:7.000000+8.000000i
v2cint32	±1+2i:3+4i
v4cint16	±1+2i:3+4i:5+6i:7+8i
v4int32	±1:2:3:4
v4float	±1.000000:2.000000:3.000000:4.000000
v8int16	±1:2:3:4:5:6:7:8
v16int8	±1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16
v16uint8	1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16
v8acc48	±1:2:3:4:5:6:7:8
v4cacc48	±1+2i_3+4i_5+6i_7+8i

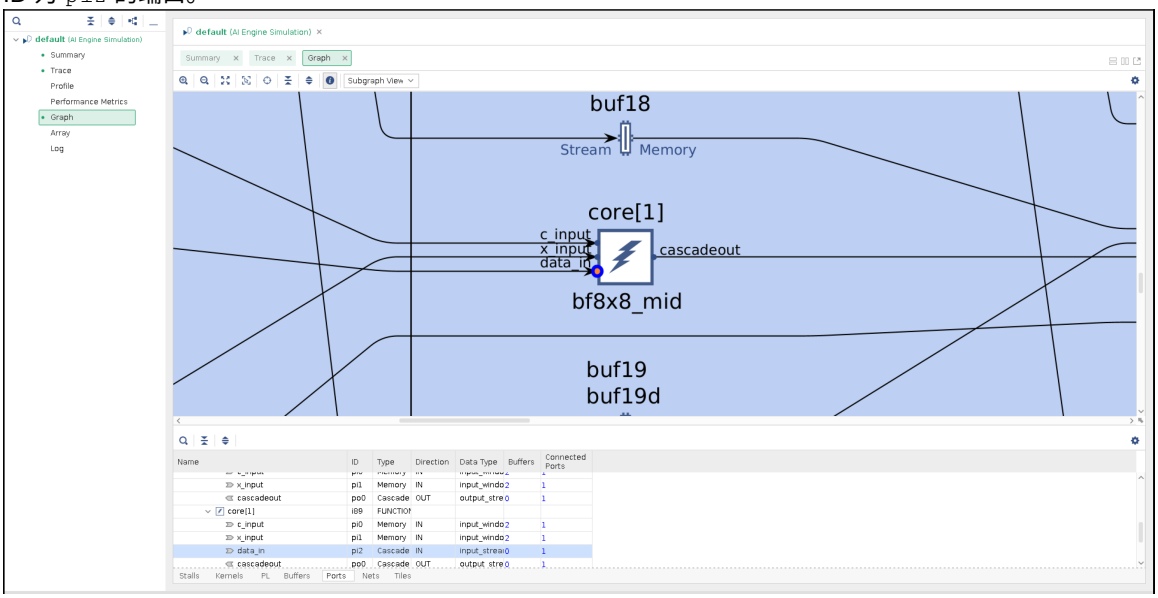
注释：

1. 前置 0 和“+”不予显示。
2. 实数部分后接虚数部分，分别位于 x 和 x+1 时间戳处。
3. 最低有效位 32 位值和最高有效位 32 位值显示在 x 和 x+1 时间戳处。
4. 当前正在开发中。

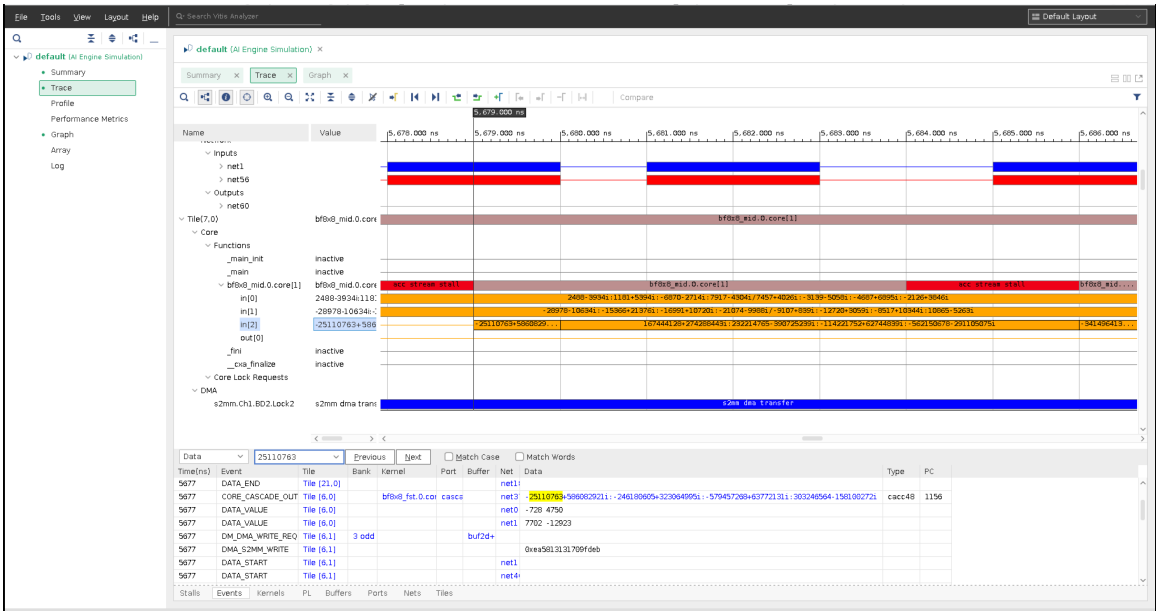
级联数据分析

级联数据的追踪与串流数据的追踪相似。追踪视图中的级联数据分析允许您执行从“Graph”视图中的特定输入/输出级联连接到事件追踪视图中的相应位置的交叉探测。在内核执行期间，您可对当前接收的级联输入数据进行追踪。要在级联接口上执行数据分析，请使用以下步骤。

1. 下图显示了“Graph”视图中已选中的内核端口。在该窗口底部的“Graph”视图和端口表中，会高亮显示选定的 ID 为 `pi2` 的端口。



2. 切换至“Trace”（追踪）即可查看事件和详细的事件时序。



3. 查看与选定输入端口关联的数据。
4. 在屏幕底部的事件表中，从列选择器下拉菜单选择“Data”（数据）。
5. 将 `25110763` 输入相邻（筛选）框中，然后单击“Next”（下一步）。事件表中会显示期望的时间和数据。

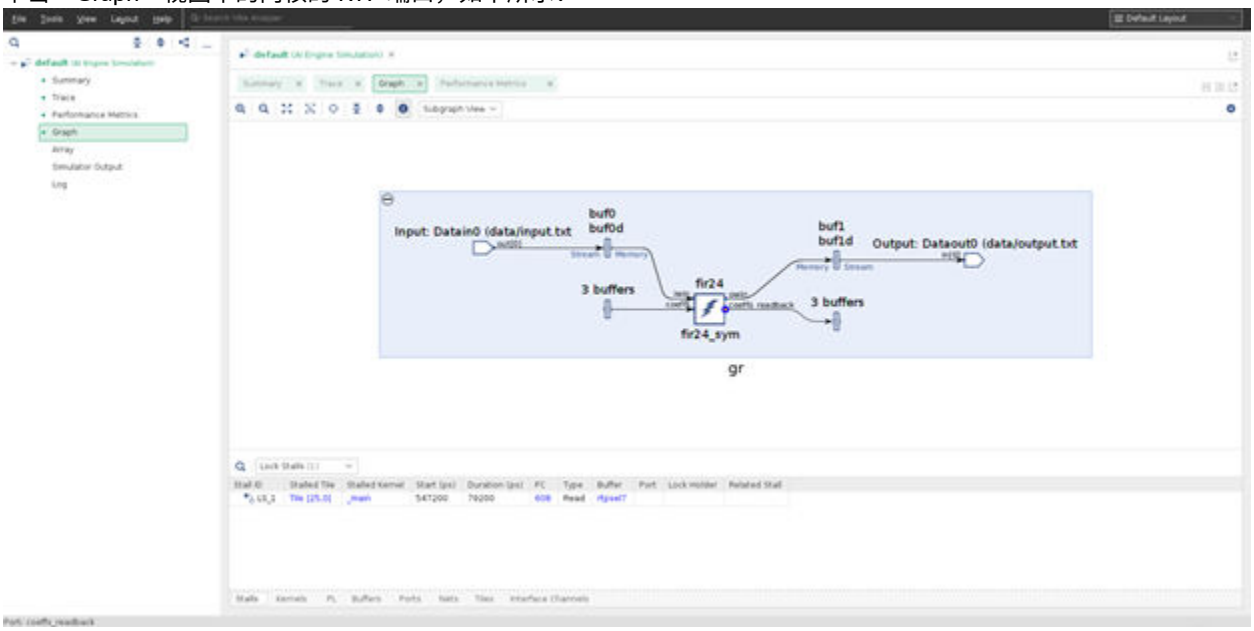
   注释：输入值取决于数据类型。此示例中的 `-25110763+5860829211` 对应于复数的整数数据类型。

您可使用这些步骤来检验所需级联连接的 I/O 数据，以确保往来内核的 I/O 数据正确无误。

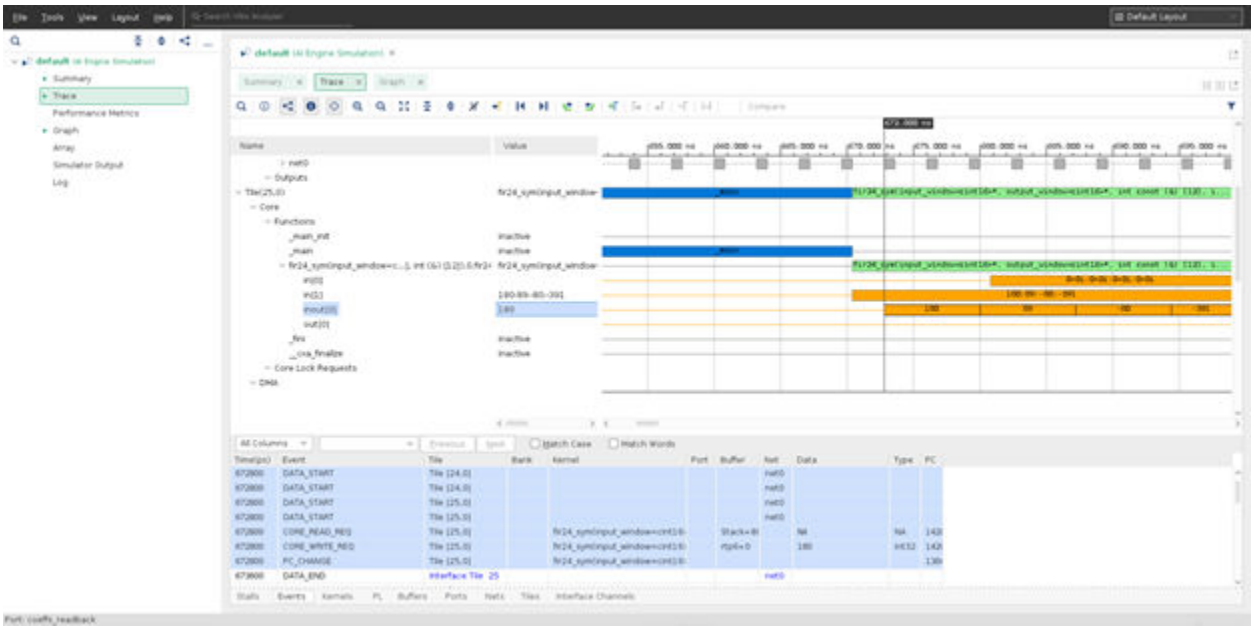
标量 RTP 数据分析

追踪视图中的标量 RTP 数据分析允许您执行从“Graph”视图中的特定标量输入/输出 RTP 端口到事件追踪视图中的相应位置的交叉探测。在内核执行期间，您可对当前读取或写入的 RTP 数据进行追踪。要在 RTP 端口上执行数据分析，请使用以下步骤。

1. 单击“Graph”视图中的内核的 RTP 端口，如下所示：



2. 切换到“Trace”（追踪）视图，选中的端口将高亮显示。“Events”（事件）视图会显示内核执行的 RTP 数据读取和写入操作。将光标置于“Trace”视图中的 RTP 数据开始处即可在“Events”视图中高亮显示该周期，如下所示：



注释：在“Trace”视图中只能对标量输入、输入输出同步以及异步 RTP 数据进行分析。矢量 RTP 数据无法直观显示。

数据显示限制

追踪视图数据可视化功能特性的限制如下。

- 在某些情况下，数据会在时间线上重复两次，因为超长指令字 (VLIW) CPU 会在同一个周期内从两个不同加载端口提取数据，所以存储器会在两个连续周期内仲裁两批加载数据，因此会在时间线上出现重复的相同数据。
- 64 位非矢量窗口数据类型显示为 2 个 32 位高位值和低位值。

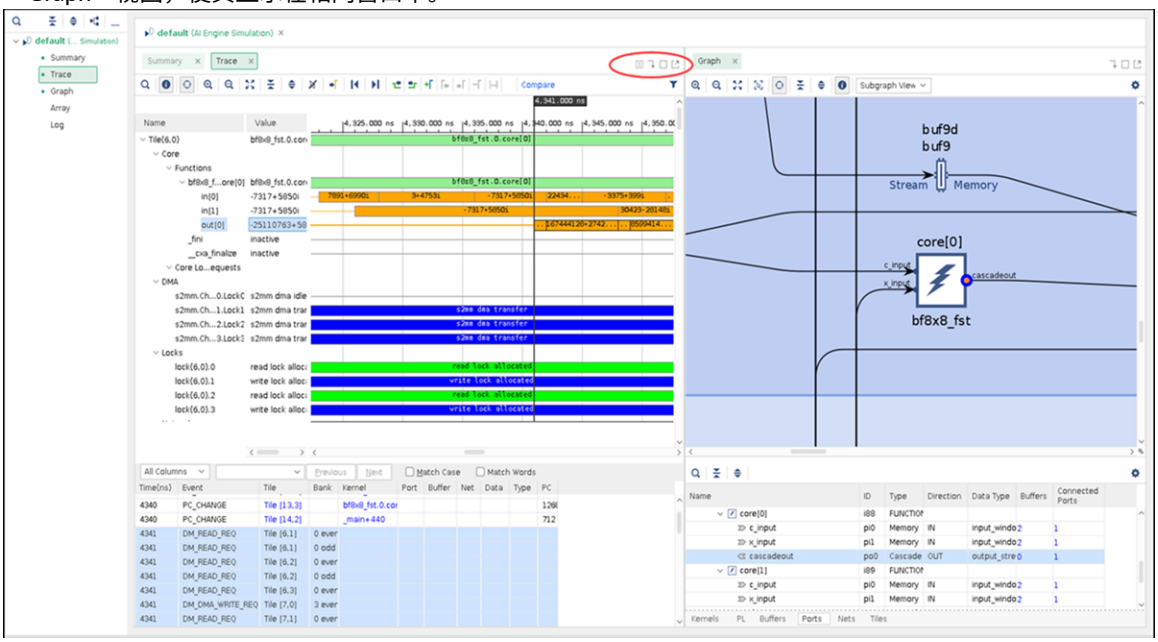
  例如，无符号的 64 位整数 `0x0000000100000002` 会分别显示为 2L 和 1H。L 和 H 表示低位和高位 32 位。

  另一个示例是复数 32 位值，其中实数 32 位首先显示，后接虚数 32 位。
- `input_pktstream/output_pktstream` 数据类型以整数格式显示。
- 不支持显示模板类中使用的 int64 数据类型。
- 如果内核函数由 AI 引擎编译器内联，则不显示在追踪内。
- 内联的内核函数 I/O 端口不显示在追踪内。

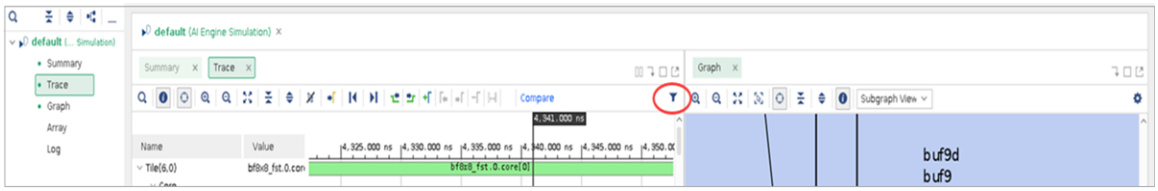
交叉探测

设计调试是一个复杂的进程，通常需要在各视图/报告之间进行切换。Vitis 分析器支持此功能，并且允许详细检查各项事件及其关联的时序。交叉探测允许您在不同视图中探测数据，使您在同一个窗口内即可查看不同透视图中的所有 I/O 数据，包括特定拼块或端口上的数据、事件发生的时间。“Trace”（追踪）和“Graph”视图可显示在同一个窗口中，在“Trace”视图中移动时间标记或者在“Graph”视图中选择对象的操作同时适用于这两个视图。

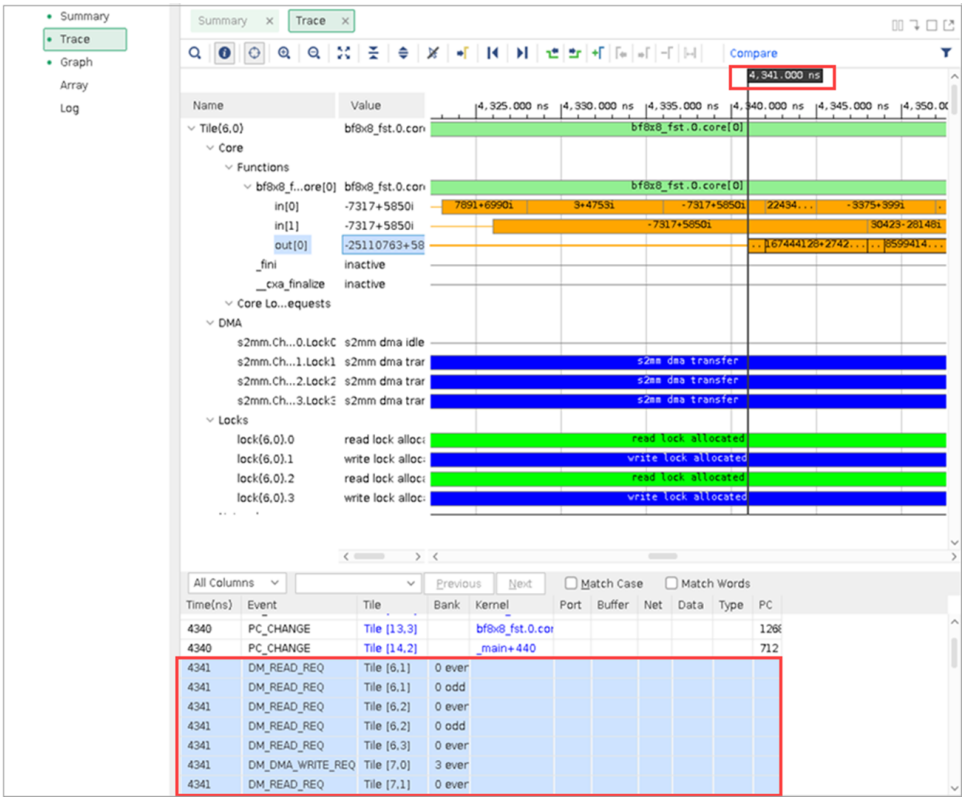
1. “Window Layout”（窗口布局）工具栏（下图中已圈出）可用于管理视图/报告。在下图中，已选中“Trace”和“Graph”视图，使其显示在相同窗口中。



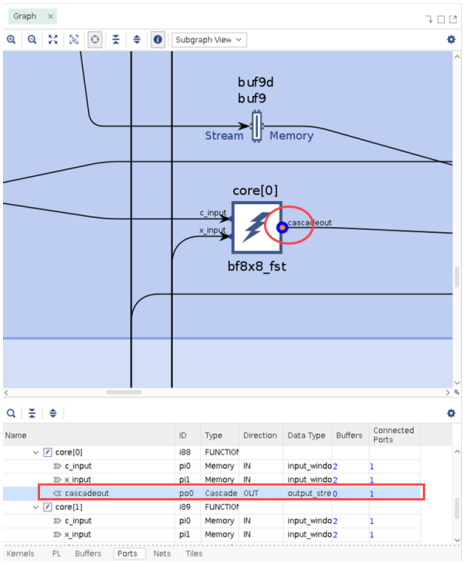
2. 筛选按钮功能（下图中已圈出）用于选择要在“Trace”视图中包含或排除的拼块、函数、输入/输出端口、DMA、锁定以便聚焦感兴趣区域。



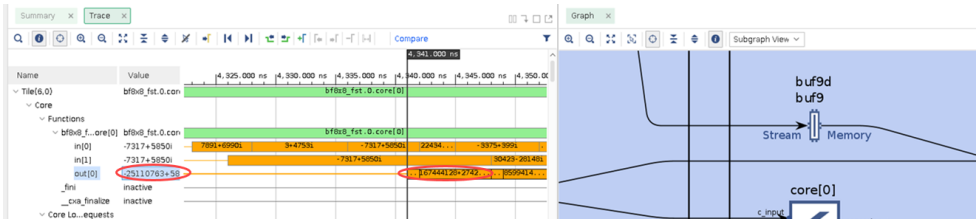
3. 拖动时间标记即可将时间向后或向前移动，以便评估给定时间的事件。在“Trace”视图的事件表（屏幕底部）中会高亮显示在选定的时间后发生的事件。



4. 选择“Graph”视图中的对象映射 graph、拼块、I/O 端口和信号线连接，以查看对象 ID、类型、方向、数据类型、缓冲器和已连接的端口。



5. 在“Trace”视图中提供了 I/O 端口的视图（下图中已圈出）。此设计示例使用复杂的 16 位值类型 (`cint16`)。



这些步骤在“Graph”视图（右侧）中显示了一个输出端口对象，此对象位于设计执行后的 4,341.000 ns 处，在“Trace”视图（左侧）中输出数据可用处也显示了此输出端口对象。将时间标记移至“Trace”视图中靠后的时间，就会在事件表（“Trace”视图的下半部分）中高亮显示相应的事件，以指示该时间点发生的事件。此信息对于多处理器环境内的相关事件很有用。其它示例可以在 Vitis 分析器工具中利用此交叉探测功能特性。

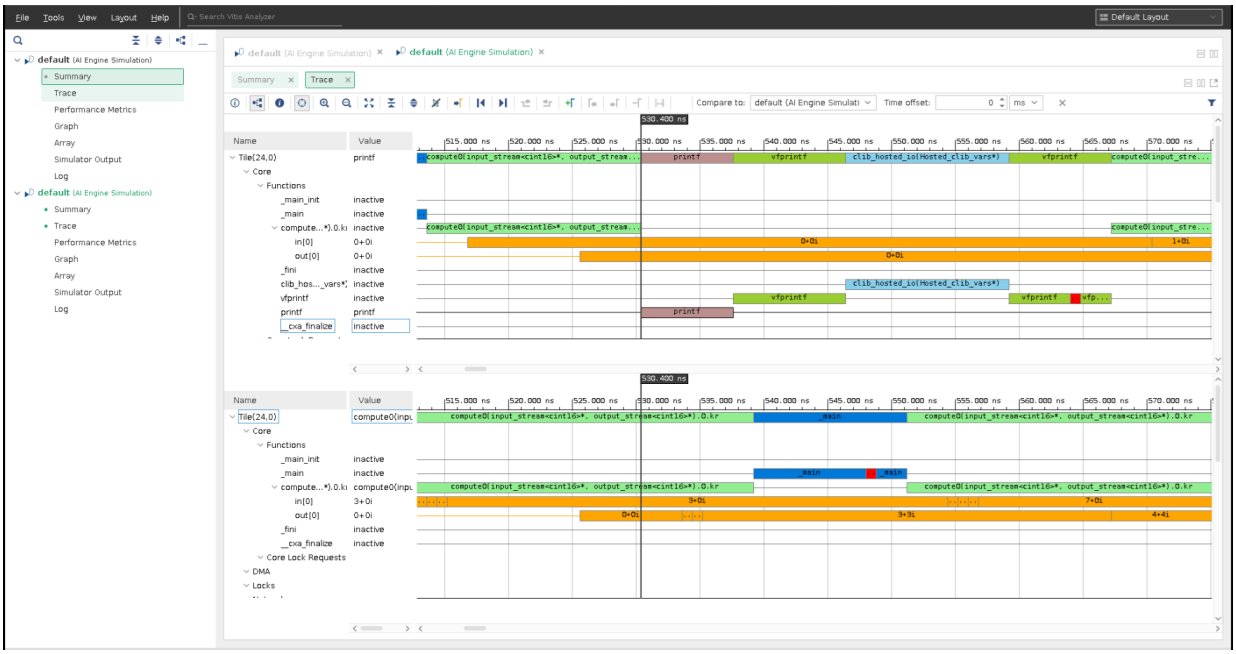
追踪比较

Vitis 分析器中的追踪比较允许对两个不同的设计执行进行比较。两次设计运行之间的比较有助于查看设计运行中引入的一种或多种变化所产生的影响。这样无论调整任何变量，您都能检验性能差异。

1. 打开两个要比较的汇总文件。
2. 选择任一设计的“Trace”（追踪）视图，右键单击并选择“Compare to” → “<filename>_summary”（对应另一个设计），或者单击“Compare”（比较）链接。可从任意追踪开始比较。

注释：在“Trace Compare”（追踪比较）模式下，Vitis 分析器中的功能特性对两个“Trace”视图都适用。

此示例适用相同的设计，但在内核中添加了 `printf()`。通过观察此“Trace Compare”示例，可以看到上方设计在 `printf()` 调用上耗费了时间，导致减缓了设计整体执行速度。



在 Vitis 分析器中执行 AI 引擎停滞分析

AI 引擎的执行可能因多种来源的停滞逻辑而发生停滞，包括下列来源。

- 外部 AXI4 存储器映射主接口：任意外部 AXI4 主接口（例如，PS）均可向特定 AI 引擎发出停滞信号。
- 锁定模块：AI 引擎有权访问锁定以执行硬件同步。获取锁定时，如果该核无法获得锁定，那么 AI 引擎将停滞，直至锁定变为解锁为止。
- AXI4-Stream 接口为空或已满：AI 引擎在读取空的输入 FIFO 或写入已满的输出 FIFO 时可能停滞。
- 数据存储器冲突：在两个不同 AI 引擎之间或者当某个 AI 引擎尝试访问单个存储体（例如，尝试在同一个周期内对同一个存储体执行加载和存储）时，可能发生存储体停滞。
- 来自事件单元的事件操作：来自事件单元的事件操作可能导致 AI 引擎停滞。

当 AI 引擎停滞时，AI 引擎的所有存储器接口（包括编程存储器接口）都会停滞。修复停滞原因后即可解决停滞。

Vitis 分析器可使用来自 AI 引擎仿真的 VCD 追踪来执行停滞分析，这样即可按指标显示停滞状态概况。这样也有助于您检测发生停滞的位置以及可能的原因。

为了使 Vitis 分析器能执行停滞分析，建议运行 AI 引擎仿真器搭配 `--online -wdb -ctf` 选项在后台生成事件追踪信息。

```
aiesimulator --pkg-dir=./Work --online -wdb -ctf
```

注释：如果分析需要使用 VCD 文件，则可改为使用 AI 引擎仿真器的 `--dump-vcd` 选项。但请注意，Vitis 分析器从 VCD 文件生成事件追踪需要时间。大型设计尤其如此，耗时较长，难以忽略。因此，建议使用 `vcdanalyze` 搭配 AI 引擎仿真器来准备事件追踪，以便 Vitis 分析器执行 AI 引擎停滞分析。

```
aiesimulator --pkg-dir=./Work --dump-vcd foo
vcdanalyze --vcd=foo.vcd --wdb --ctf
```

注释：请勿使用 vcdanalyze 的 `--outdir` 选项将输出数据置于非默认目录内。

要了解启动 Vitis 分析器以查看 AI 引擎仿真结果的步骤，请参阅 在 Vitis 分析器中查看运行汇总。

```
vitis_analyzer ./aiesimulator_output/default.aierun_summary
```

要在硬件仿真流程中通过 Vitis 分析器执行 AI 引擎停滞分析，还需要增加下列额外设置。

1. 编写仿真器选项文件（例如，`sim_options.txt`），其中包含下列内容：

   ```
   AIE_DUMP_VCD=foo
   ```

2. 以下列选项启动硬件仿真：

   ```
   ./launch_hw_emu.sh -aie-sim-options ./sim_options.txt -add-env
   AIE_COMPILER_WORKDIR=<ABSOLUTE_PRJ_PATH>/Work
   ```

3. （可选）运行 `vcdanalyze`：

   ```
   cd ./sim/behav_waveform/xsim/; vcdanalyze --pkg-dir=../../../Work --
   vcd=foo.vcd --wdb -ctf
   ```

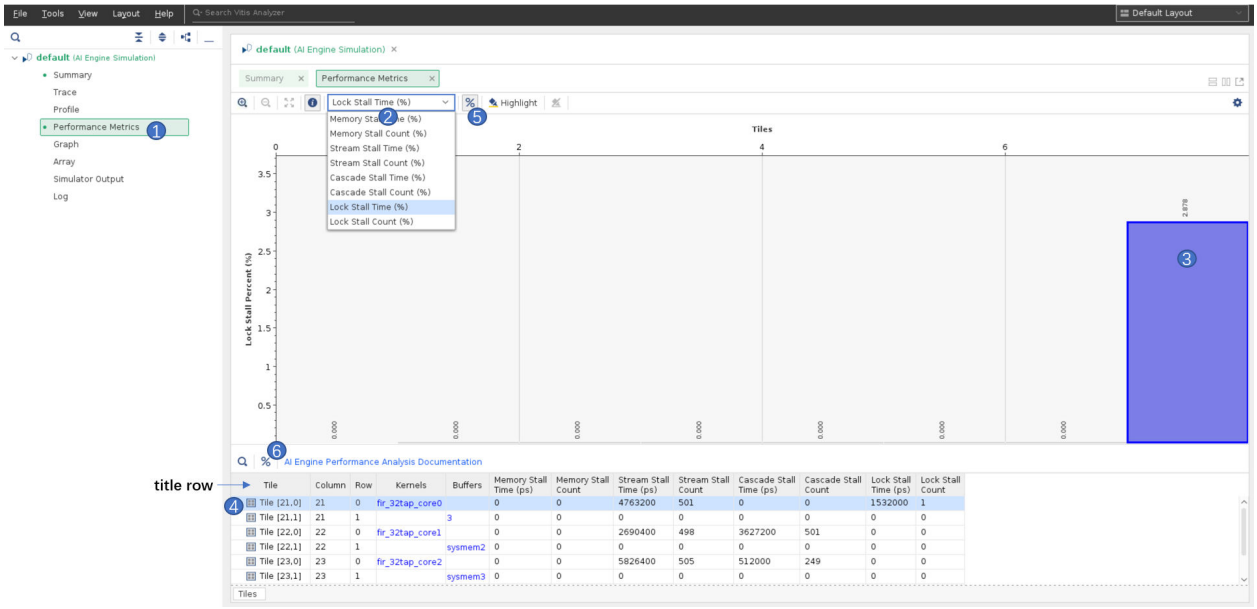
4. 启动 Vitis 分析器：

   ```
   vitis_analyzer ./sim/behav_waveform/xsim/default.aierun_summary
   ```

性能指标

“Performance Metrics”（性能指标）视图可显示每种类型的停滞占总仿真时间或总停滞计数的百分比。

图 22： **“Performance Metrics”视图**

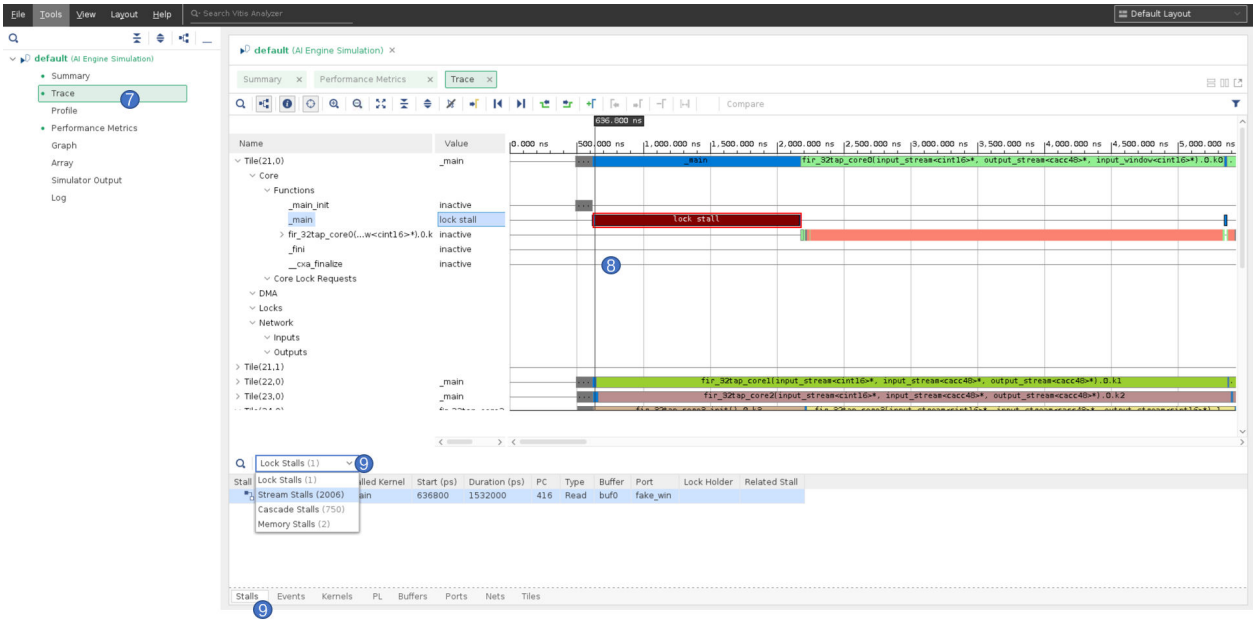


1. 单击“Performance Metrics”视图。
2. 从下拉列表中选择停滞类型。指标区分不同类型的停滞。它将显示仿真结果中现有的停滞。
   - Lock Stall Time (%)（锁定停滞时间 (%)）：锁定停滞时间百分比，在此期间获取 AI 引擎拼块内的缓冲器。
   - Lock Stall Count (%)（锁定停滞计数 (%)）：特定拼块中锁定停滞计数占所有 AI 引擎拼块中的锁定停滞总数的百分比。
   - Memory Stall Time (%)（存储器停滞时间 (%)）：存储器停滞时间百分比，在此期间 AI 引擎拼块内的存储器访问存在冲突。
   - Memory Stall Count (%)（存储器停滞计数 (%)）：特定拼块中存储器停滞计数占所有 AI 引擎拼块中的存储器停滞总数的百分比。
   - Stream Stall Time (%)（串流停滞时间 (%)）：由于串流已满或为空而导致串流停滞时间的百分比。
   - Stream Stall Count (%)（串流停滞计数 (%)）：特定拼块中串流停滞计数占所有 AI 引擎拼块中的串流停滞总数的百分比。
   - Cascade Stall Time (%)（级联停滞时间 (%)）：由于级联串流已满或为空而导致级联串流停滞时间的百分比。
   - Cascade Stall Count (%)（级联停滞计数 (%)）：特定拼块中级联串流停滞计数占所有 AI 引擎拼块中的级联串流停滞总数的百分比。
3. 每个 AI 引擎拼块在“Performance Metrics”视图中均显示为一个条形。停滞百分比越高，条形越偏右。对于最高的条形，应多加关注。单一其中某一个条形即可选中要关注的拼块。
4. 底部的“Tiles”（拼块）视图列出了所有 AI 引擎拼块以及有关列、行、内核、缓冲器、所有停滞时间、百分比和计数的信息。您可单击标题列，按特定列排序。

   提示：如果视图中的数值或信息为蓝色，则可与其它视图进行交叉探测。

5. 在下拉列表旁有个 % 按钮。单击此按钮即可切换停滞时间的显示形式：百分比或纳秒 (ns)。
6. 单击“Tiles”视图上方的“Show Percentage”（显示百分比）按钮即可在“Tiles”视图中切换停滞的显示方法：按时间和计数显示停滞或者按百分比显示停滞。在“Performance Metrics”视图中浏览信息时，通常与其它视图进行交叉探测是很有用的。例如：

图 23： **“Trace”视图**



7. 选择停滞时间最长的拼块。然后转至“Trace”（追踪）视图以查看停滞的位置和频率。
8. 缩放“Trace”视图以获取时间线中停滞的更清晰的视图。
9. 单击“Stalls”（停滞）视图并从下拉列表中选择要检验的停滞的类型。选中“Stalls”视图中的停滞即可在“Trace”视图中高亮显示此停滞。

“Performance Metrics”视图、“Trace”视图、“Graph”视图和“Array”视图可彼此交叉探测。“Graph”视图有助于理解 graph 中发生停滞的位置，“Array”视图则有助于查看硬件中对象的位置。在后续章节中，对每一种停滞类型的分析进行了解读并提供了更多详细信息。

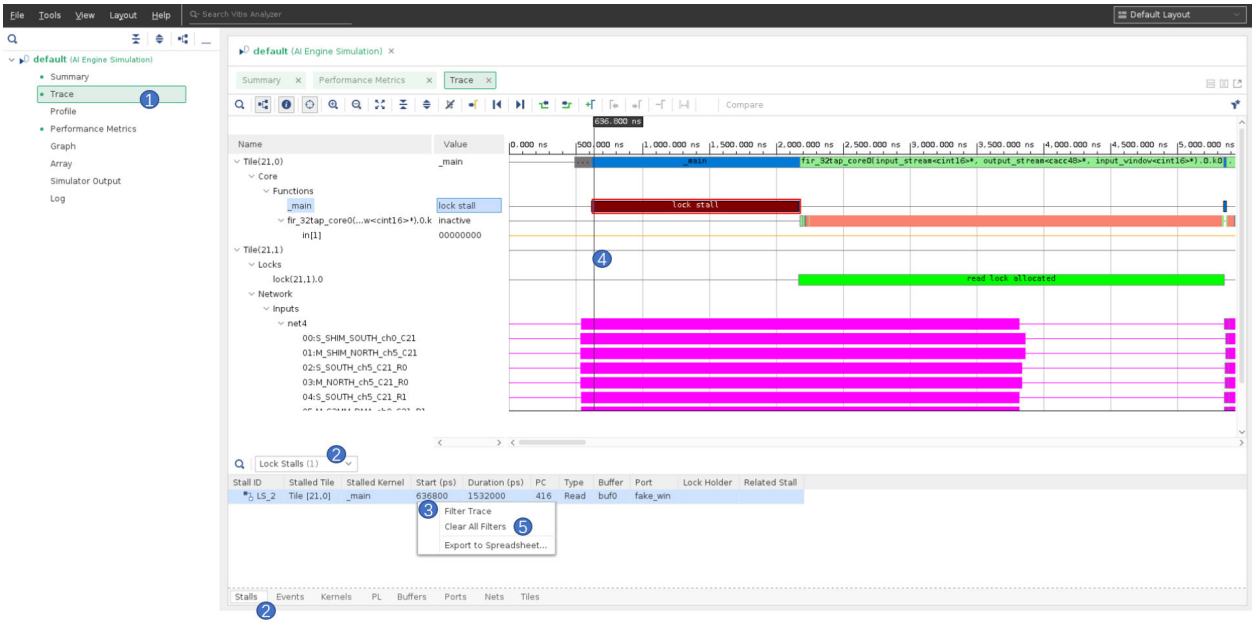
锁定停滞分析

“Performance Metrics”（性能指标）分析选项卡可帮助识别是否需要对锁定停滞进行分析。以下步骤演示了如何在 Vitis 分析器中开始分析锁定停滞。

1. 选择“Trace”（追踪）视图。
2. 在“Stalls”（停滞）视图中，选中下拉列表中的“Lock Stalls”（锁定停滞）。

提示：这样“Stalls”视图即可随“Trace”视图、“Graph”视图和“Array”（阵列）视图一起使用。

图 24：**“Trace”视图中的锁定停滞**



每次停滞都包含下列关联信息。

- “Stall ID”（停滞 ID）：锁定停滞名为 LS_<NUM>。此编号在所有类型的停滞中都是唯一的。停滞发生时间越早，编号越小。
- “Stalled Tile”（停滞的拼块）：已停滞的内核所在的 AI 引擎 tile。
- “Stalled Kernel”（停滞的内核）：已停滞的内核。此内核名为 `<Kernel_function_name>.<Schedule_ID>.<Graph_instance_name>`。有时，它显示为 `_main`，随后，需通过交叉探测来查找实际的内核函数。
- “Start (ns)”（开始 (ns)）：停滞的开始时间。
- “Duration (ns)”（持续时间 (ns)）：停滞的持续时间。
- “PC”：发生停滞时的程序计数器。
- “Type”（类型）：停滞的内核会尝试对缓冲器执行读取或写入。
- “Buffer”（缓冲器）：停滞的内核尝试读取或写入的缓冲器。
- “Port”（端口）：停滞的内核尝试读取或写入缓冲器时所使用的端口。
- “Lock Holder”（锁定保存器）：保存缓冲器的锁定的源代码。
- “Related Stall”（相关停滞）：可能导致停滞的其它停滞。

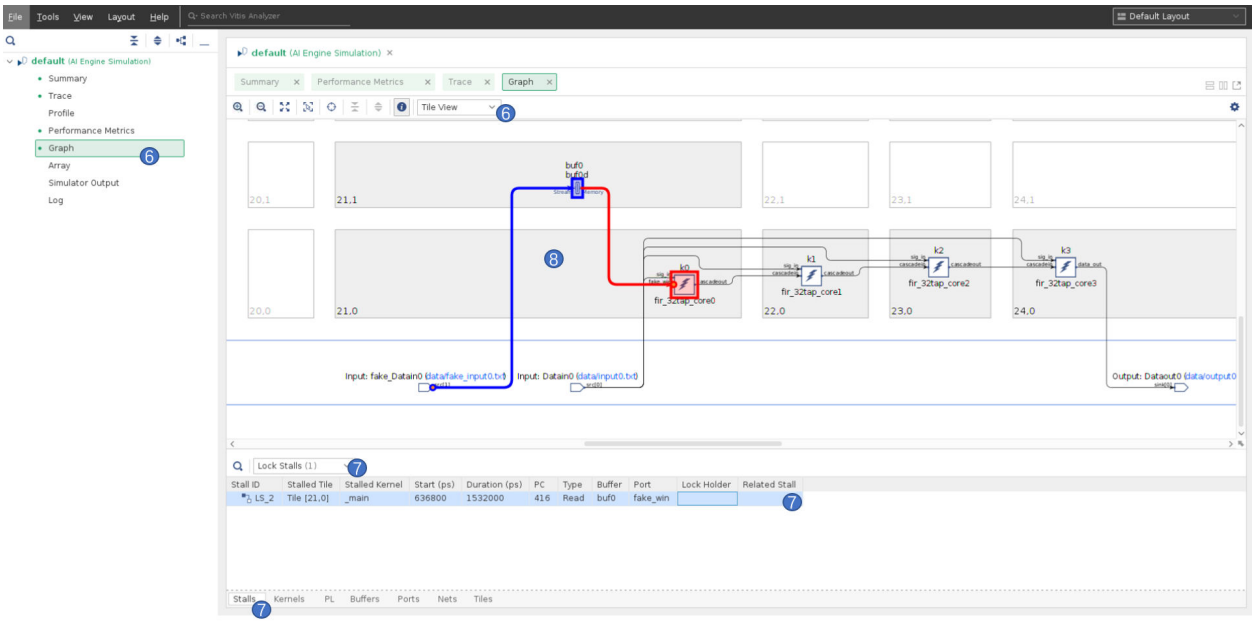
提示：蓝色项可与其它视图进行交叉探测。

3. 选中一行停滞。它将转至“Trace”视图中该停滞的开始位置。（可选）右键单击停滞并选择“Filter Trace”（筛选追踪）即可筛选与该停滞相关的所有信号。在“Trace”视图中，会显示与停滞相关的信号。不相关的信号则隐藏。如果采用大型设计，那么这样更便于浏览追踪。

提示：“Filter Trace”可能无法显示与相关停滞存在关联的信号。

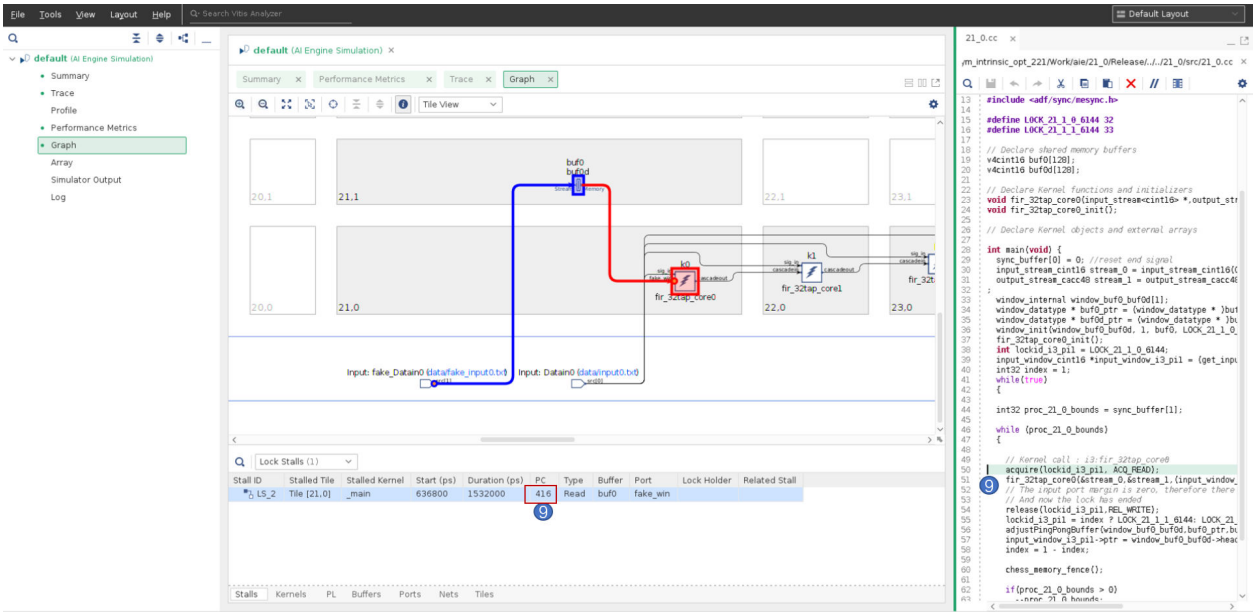
4. “Trace”视图可用于查看时间线中的锁定停滞。对于特定锁定停滞，只需为其分配写入锁定和读取锁定即可查看。根据停滞的位置以及停滞前后的事件，可分析停滞原因。例如，如果已分配写入锁定并且锁定类型为“Read”（读取），那么这表示生产者尚未释放缓冲器。使用者正在等待该缓冲器变为可读。生产者可在“Lock Holder”中找到。
5. 要清除先前筛选的追踪，请右键单击并选中“Clear All Filters”（清除所有筛选）。
6. 在“Graph”视图中显示停滞路径的概览是很有帮助的。选中“Graph”视图，然后从该视图的下拉列表中选中“Tile View”（拼块视图）。“Graph”视图的“Tile”视图会显示 AI 引擎拼块内的 graph。

图 25：**“Graph”视图中的锁定停滞**



7. 如果“Stalls”视图未显示，请从下拉列表中选中“Lock Stalls”（锁定停滞），并选择要分析的停滞。它将高亮显示“Graph”视图的“Tile”视图中的相关路径。
8. 红色路径显示的是停滞发生的位置。蓝色路径显示的是停滞发生的来源。
9. 单击 PC 值。这样即可打开源代码并转至其中发生停滞的行。

图 26：**对源代码进行 PC 交叉探测人**



下表列出了可能导致锁定停滞的场景以及可能的解决方案。

表 38：**锁定停滞场景和解决方案**

来源	目标	目标对象	停滞类型	可能的解决方案
AI 引擎内核	同步窗口锁定	AI 引擎内核	锁定停滞	· 如果使用单个缓冲器，请使用乒乓缓冲器（默认）或者将内核布局到相同 AI 引擎拼块内。 · 如果在执行中内核处于不平衡状态，请平衡内核间的吞吐量。
AI 引擎内核	异步窗口锁定（`window_acquire` 和 `window_release` API）	AI 引擎内核	锁定停滞	· 如果使用单个缓冲器，请使用乒乓缓冲器（默认）。 · 及时获取和释放缓冲器。按需使用本地缓冲器。
PL 接口	窗口锁定	AI 引擎内核	锁定停滞	· 确保 PL 接口吞吐量与 AI 引擎吞吐量相匹配。 · 检查 PL 接口频率与宽度是否设置正确。请参阅《AI 引擎内核与 Graph 编程指南》(UG1079) 中的 AI 引擎/可编程逻辑集成。
AI 引擎	窗口锁定	PL 接口	锁定停滞	· 确保 PL 接口吞吐量与 AI 引擎吞吐量相匹配。 · 检查 PL 接口频率与宽度是否设置正确。请参阅《AI 引擎内核与 Graph 编程指南》(UG1079) 中的 AI 引擎/可编程逻辑集成。

注释：DMA 锁定停滞不包含在 Vitis 分析器锁定停滞分析内。

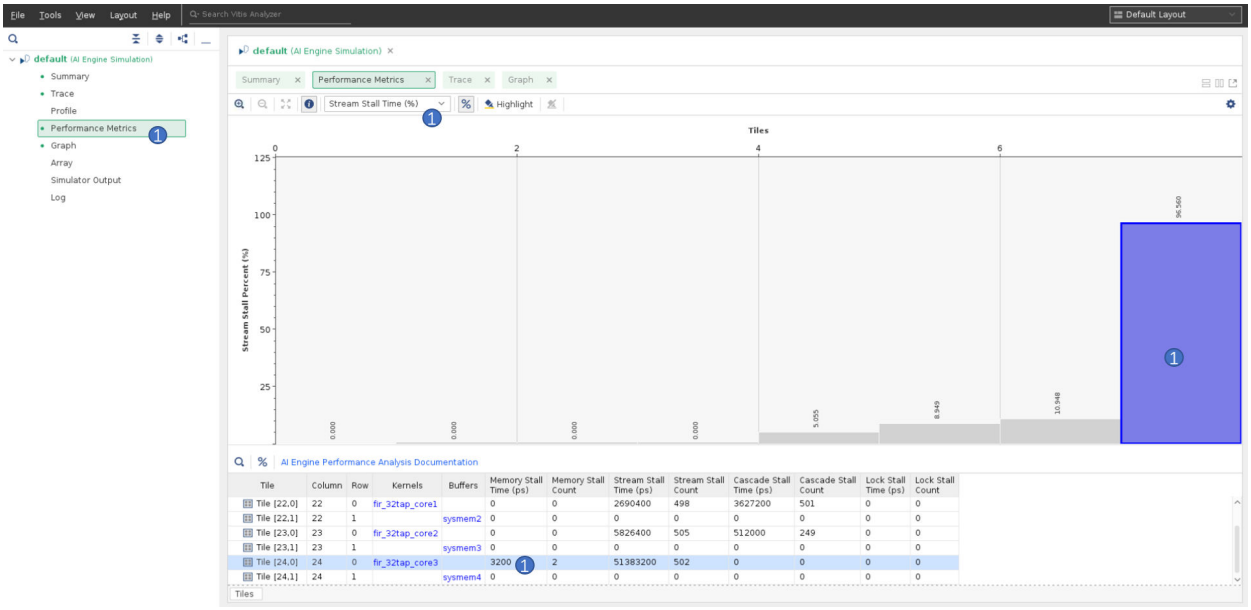
串流停滞分析

在“Performance Metrics”（性能指标）视图中，您可识别是否需对串流停滞以及导致停滞的拼块进行分析。

以下步骤演示了如何在 Vitis 分析器的“Performance Metrics”选项卡中开始分析串流停滞。

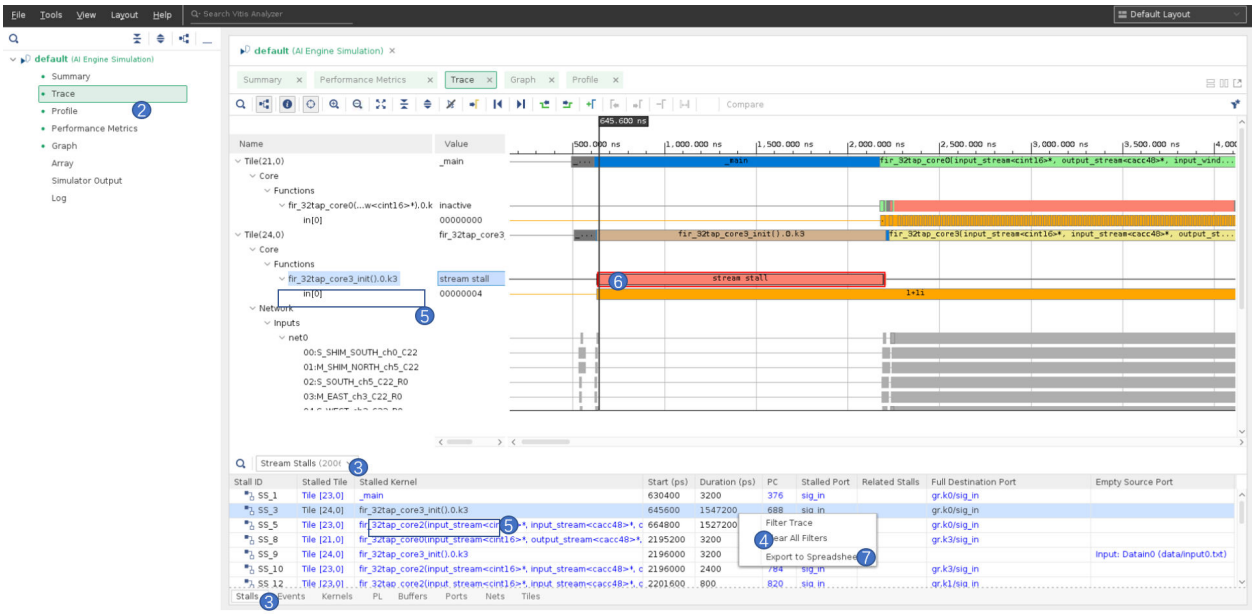
1. 在“Performance Metrics”视图中，选中“Stream Stall Time (%)”（串流停滞时间 (%)）即可查看所有拼块间的串流停滞。识别要分析的拼块。请注意，“Performance Metrics”视图中的对象可与“Trace”（追踪）视图、“Graph”视图和“Array”（阵列）视图中的对象进行交叉探测。例如，选中“Performance Metrics”视图中的拼块即可在“Trace”视图中高亮此拼块，这样有助于快速定位此拼块。

图 27：**“Performance Metrics”视图中的串流停滞**



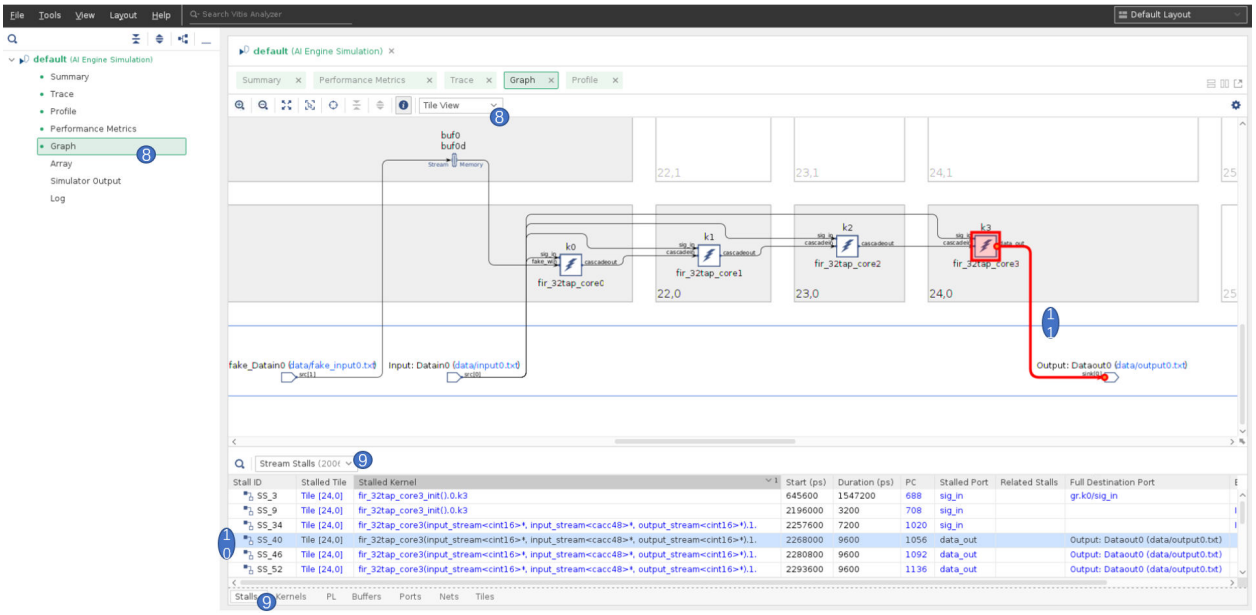
2. 选择“Trace”（追踪）视图。

图 28：**“Trace”视图中的串流停滞**



3. 在“Stalls”（停滞）视图中，选中下拉列表中的“Stream Stalls”（串流停滞）。在“Stalls”视图中，串流停滞具有以下信息，单击蓝色对象即可与其它视图进行交叉探测。
   - “Stall ID”（停滞 ID）：此串流停滞名为 `SS_<NUM>`。停滞发生时间越早，编号越小。此编号在所有类型的停滞中都是唯一的。
   - “Stalled Tile”（停滞的拼块）：已停滞的内核所在的 AI 引擎 tile。
   - “Stalled Kernel”（停滞的内核）：已停滞的内核。此内核名为 `<Kernel_function_name>.<Schedule_ID>.<Graph_instance_name>`。有时，它显示为 `_main`，随后，需通过交叉探测来查找实际的内核函数。
   - “Start (ns)”（开始 (ns)）：停滞的开始时间。
   - “Duration (ns)”（持续时间 (ns)）：停滞的持续时间。
   - PC：发生停滞时的程序计数器。
   - “Stalled Port”（停滞的端口）：已停滞的内核的端口。
   - “Related Stalls”（相关停滞）：可能导致此停滞的其它停滞。
   - “Full Destination Port”（目标端口已满）：此端口已满，导致已停滞的内核无法写入。
   - “Empty Source Port”（源端口为空）：此端口为空，导致已停滞的内核无法从中读取。
4. 单击“Stalls”视图中的串流停滞将转至“Trace”视图中此停滞的起始位置。右键单击此停滞，然后按需选择“Filter Trace”（筛选追踪）。筛选追踪后，与停滞相关的信号都会显示在“Trace”视图中。不相关的信号则隐藏。如果采用大型设计，那么使用筛选追踪来浏览追踪更清晰。
5. “Stalls”视图中的蓝色对象可供点击和交叉探测。例如，单击“Stalls”视图中的内核将在“Trace”视图中高亮此内核。
6. 缩放“Trace”视图即可浏览停滞。根据停滞的位置、类似停滞发生的频率、停滞和相关停滞（如有）发生前的事件等信息，可以为您提供有关发生停滞的原因的提示。
7. 要清除先前筛选的追踪，请右键单击并选中“Clear All Filters”（清除所有筛选）。
8. 在“Graph”视图中显示停滞路径的概览是很有帮助的。选中“Graph”视图，然后从下拉列表中选中“Tile View”（拼块视图）。

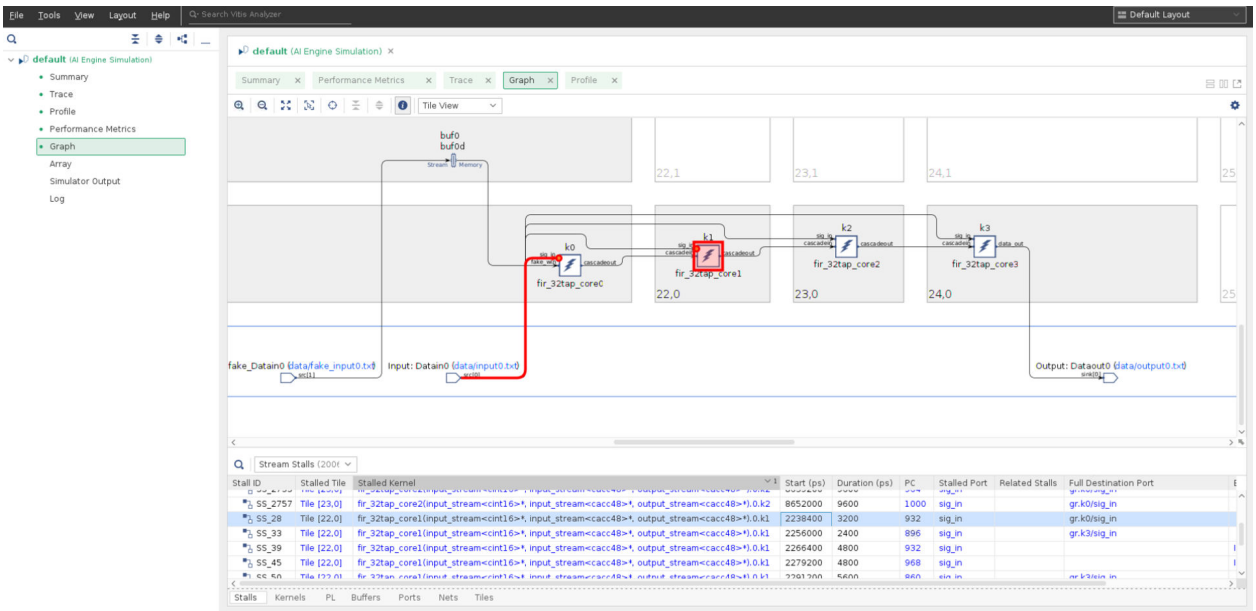
图 29：**“Graph”视图中的停滞路径**



9. 选中“Stalls”视图，然后从下拉列表中选中“Stream Stalls”。
10. 浏览“Stalls”视图中的串流停滞。单击“Stalls”视图中的串流停滞即可查看 graph 中的停滞概览。红色路径显示的是停滞发生的位置。此路径可能是从已停滞的内核到已满的目标端口，或者从空的源端口到已停滞的内核。

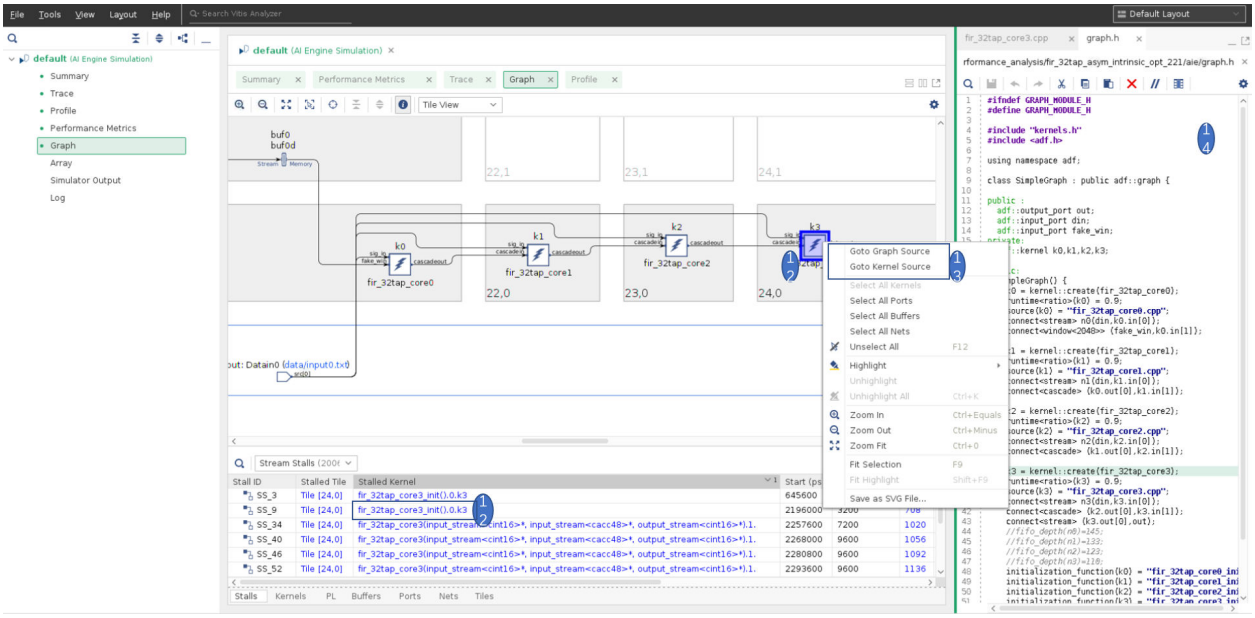
> **提示：** 如果任一串流多播至多个目标，并且由于此串流没有足够 FIFO 可满足所有目标而导致串流停滞时，高亮的停滞内核与停滞的信号线可能处于未连接状态（分离并显示为红色）。这意味着必须将多播串流的所有目标作为一个整体进行串流停滞分析。下图中显示了“Graph”视图中的多播串流停滞示例。

图 30：**多播串流停滞路径**



11. 可从“Graph”视图或“Array”视图打开 graph 源代码或内核源代码。单击“Stalls”视图中的内核对象或者单击“Graph”视图中的内核即可选中内核实例。

图 31：**查看 Graph 代码和内核代码**



12. 右键单击“Graph”视图中的内核实例，然后选中“Goto Graph Source”（转至 graph 源代码）或“Goto Kernel Source”（转至内核源代码）。这样将打开 graph 源代码或内核源代码。
13. 将 graph 源代码和内核源代码与分析的停滞加以关联，按需编辑源代码。

下表列出了可能导致串流停滞的部分场景以及可能的解决方案。

表 39：**串流停滞场景和解决方案**

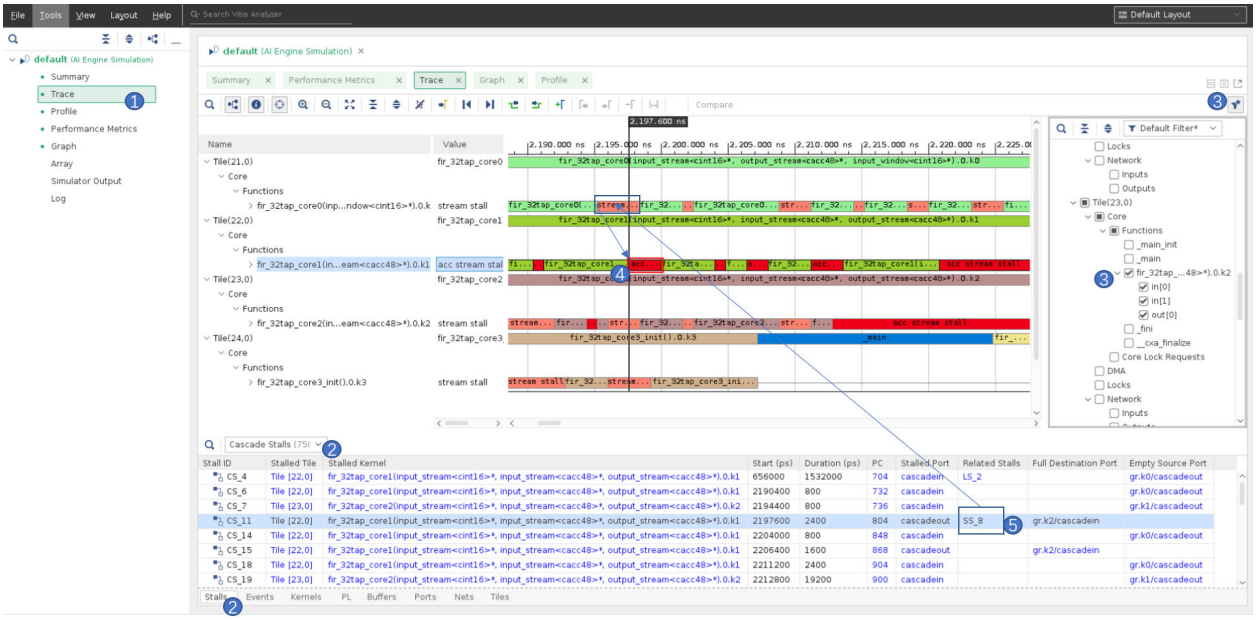
来源	目标对象	停滞类型	可能的解决方案	注释
串流	串流	串流停滞	· 增加 FIFO 深度。请参阅《AI 引擎内核与 Graph 编程指南》(UG1079) 中的 FIFO 深度约束。 · 调整源内核或目标内核中的串流读取和写入指令。	
串流	多条串流	串流停滞	· 增加 FIFO 深度。请参阅《AI 引擎内核与 Graph 编程指南》(UG1079) 中的 FIFO 深度约束。 · 为不同目标信号线插入 DMA FIFO 或者设置不同 FIFO 深度。请参阅 FIFO 深度约束。	多播
串流	同一 AI 引擎内存在多个内核的多条串流	串流停滞	· 将多个内核置于不同 AI 引擎内 · 给内核串流添加足够的 FIFO。	多播
多条串流	多条串流	串流停滞	· 调整指令，使不同串流间匹配。 · 增加 FIFO 深度（ssFIFO 或 DMA FIFO）。	
PLIO	串流	串流停滞	· 尽可能增大 AI 引擎到 PL 接口带宽。例如，64 位接口，采用最高 PL 频率（1/2 AI 引擎频率）。或者 128 位接口（请注意，这表示对一个 128 位接口使用两条 64 位通道）。请参阅《AI 引擎内核与 Graph 编程指南》(UG1079) 中的 AI 引擎到 PL 的接口编程。	
串流	PLIO	串流停滞	同上。	
串流（每次迭代 32 位）	PLIO	串流停滞	为每个 32 位发送 TLAST。请参阅《AI 引擎内核与 Graph 编程指南》(UG1079) 中的 AI 引擎到 PL 的接口编程。	

级联停滞分析

您可通过“Performance Metrics”（性能指标）分析来识别需分析的级联停滞。

1. 选中“Trace”（追踪）视图。
2. 选中“Stalls”（停滞）视图，然后从下拉列表中选择“Cascade Stalls”（级联停滞）。
3. 右上角的筛选按钮可用于选择“Trace”视图中的信号。选中任一窗口中的所有级联停滞有助于更好地了解在时间线中发生这些停滞的原因，因为在级联链中，级联停滞通常彼此之间存在关联。单击筛选按钮，取消选中顶部的“All”（全部），然后选中感兴趣的所有内核。
4. 选择“Cascade stall”即可在“Trace”视图中高亮停滞。

图 32：**“Trace”视图中的级联停滞**



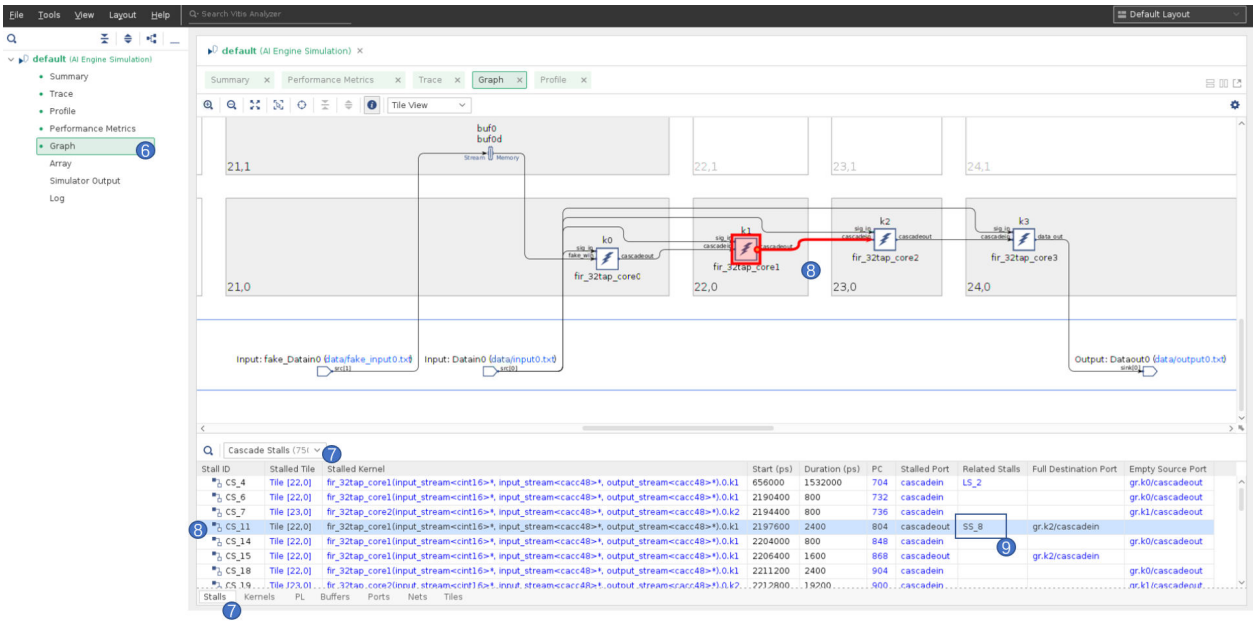
每项级联停滞都包含下列信息。

- “Stall ID”（停滞 ID）：串流停滞名为 CS_<NUM>。停滞发生时间越早，编号越小。此编号在所有类型的停滞中都是唯一的。
- “Stalled Tile”（停滞的拼块）：已停滞的内核所在的 AI 引擎 tile。
- “Stalled Kernel”（停滞的内核）：已停滞的内核。此内核名为 `<Kernel_function_name>.<Schedule_ID>.<Graph_instance_name>`。有时，它显示为 `_main`，随后，需通过交叉探测来查找实际的内核函数。
- “Start (ns)”（开始 (ns)）：发生停滞的开始时间
- “Duration (ns)”（持续时间 (ns)）：停滞的持续时间。
- PC：发生停滞时的程序计数器。
- “Stalled Port”（停滞的端口）：已停滞的内核所在的端口。
- “Related Stalls”（相关停滞）：可能导致此停滞的其它停滞。
- “Full Destination Port”（目标端口已满）：此端口已满，导致已停滞的内核无法写入。
- “Empty Source Port”（源端口为空）：此端口为空，导致已停滞的内核无法从中读取。

浏览“Trace”视图中的级联停滞，查看其彼此之间的关联。

5. 可能是由于其它类型的停滞导致了级联停滞。浏览其它类型的停滞，分析发生停滞的原因。“Stalls”视图中的相关停滞可显示哪项停滞导致了级联停滞。
6. 查看“Graph”视图中发生的停滞有助于识别导致停滞的原因。选中“Graph”视图。
7. 选中“Stalls”视图，然后从下拉列表中选中“Cascade Stalls”。
8. 单击“Stalls”视图中的路径，这样就会在“Graph”视图中以红色显示停滞路径。
9. 浏览级联停滞及其关联停滞，这样有助于查找有关停滞原因的提示。单击“Related Stalls”也能以红色显示相关停滞。例如，单击“Related Stalls”中的 SS_16 即可查看红色路径，这样可以提供有关停滞开始位置的提示。

图 33：**“Graph”视图中的级联停滞**



下表列出了导致级联停滞的部分可能场景以及可能的解决方案。

表 40：**级联停滞场景和解决方案**

来源	目标对象	停滞类型	可能的解决方案
级联串流	级联串流	级联停滞	· 调整内核执行周期中的级联串流读/写指令，使源内核与目标内核之间相匹配 · 分析导致级联停滞的其它源
串流 + 级联串流	串流 + 级联串流	级联停滞	· 调整指令，使不同串流间匹配 · 分析导致级联停滞的其它源

存储器停滞分析

AI 引擎可在每个周期内执行多次矢量加载或存储操作。但这些操作必须以不同存储体为目标才能并行执行加载或存储操作。如果在同一周期内多次访问同一存储体，就会发生存储器停滞。

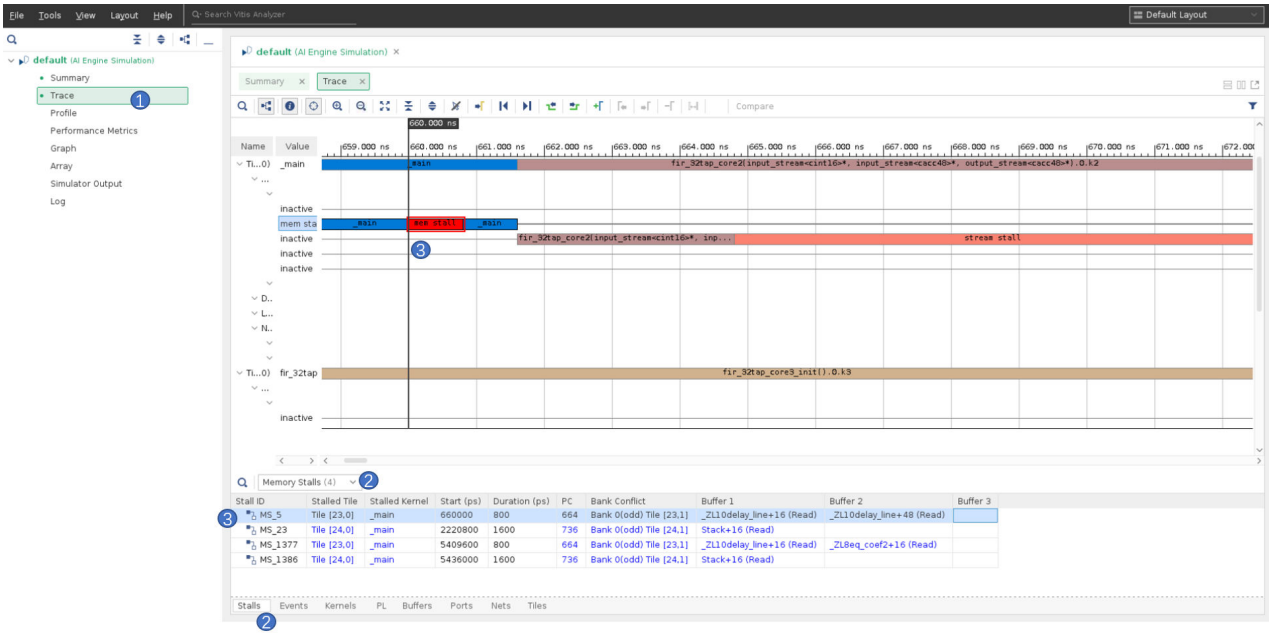
存储器种类包括内核、RTP 缓冲器与系统存储器之间的窗口缓冲器和 DMA FIFO。系统存储器包括前 32 字节、栈和堆之间的内核同步信息。静态变量位于堆内，而函数控制逻辑则位于栈内。系统存储器占据连续存储体。工具可以在特定存储体上自动或手动完成窗口缓冲器、RTP 缓冲器、DMA FIFO 和系统存储器的布局。为缓解这些存储器之间的存储器停滞，请尝试尽可能将其布局到独立存储体内。但如果无法为所有存储器都找到独立的存储体，或者在同一个存储器上发生多次访问，那么仍可能发生存储器停滞。

总之，编译器会尝试尽可能在同一个周期内调度多次存储器访问，但也存在例外情况。源自同一指针的多次存储器访问调度为在不同周期内执行。如果编译器在同一个周期内对多个变量或指针调度执行多项操作，则可能发生存储体冲突。每个存储体都有其自己的仲裁器用于在所有请求之间执行仲裁，且仲裁采用循环方式执行。解决所有请求后，就会释放存储器停滞。

您可通过“Performance Metrics”（性能指标）分析来识别是否需要对存储器停滞执行分析。

1. 选择“Trace”（追踪）视图。
2. 选择底部的“Stalls”（停滞）视图，然后从下拉列表中选择“Memory Stalls”（存储器停滞）。

图 34：**“Trace”视图中的“Memory Stall”**



此处停滞名为 MS_<NUM>。编号随时间增加。每次停滞都包含下列关联信息。

- “Stall ID”（停滞 ID）：存储器停滞 ID。停滞发生时间越早，编号越小。此编号在所有类型的停滞中都是唯一的。
- “Stalled Tile”（停滞的拼块）：已停滞的内核所在的 AI 引擎 tile。
- “Stalled Kernel”（停滞的内核）：已停滞的内核。此内核名为 `<Kernel_function_name>.<Schedule_ID>.<Graph_instance_name>`。有时，它显示为 `_main`，随后，需通过交叉探测来查找实际的内核函数。
- “Start (ns)”（开始 (ns)）：发生停滞的开始时间
- “Duration (ns)”（持续时间 (ns)）：停滞的持续时间。
- PC：发生停滞时的程序计数器。
- “Bank Conflict”（存储体冲突）：发生停滞的存储器。

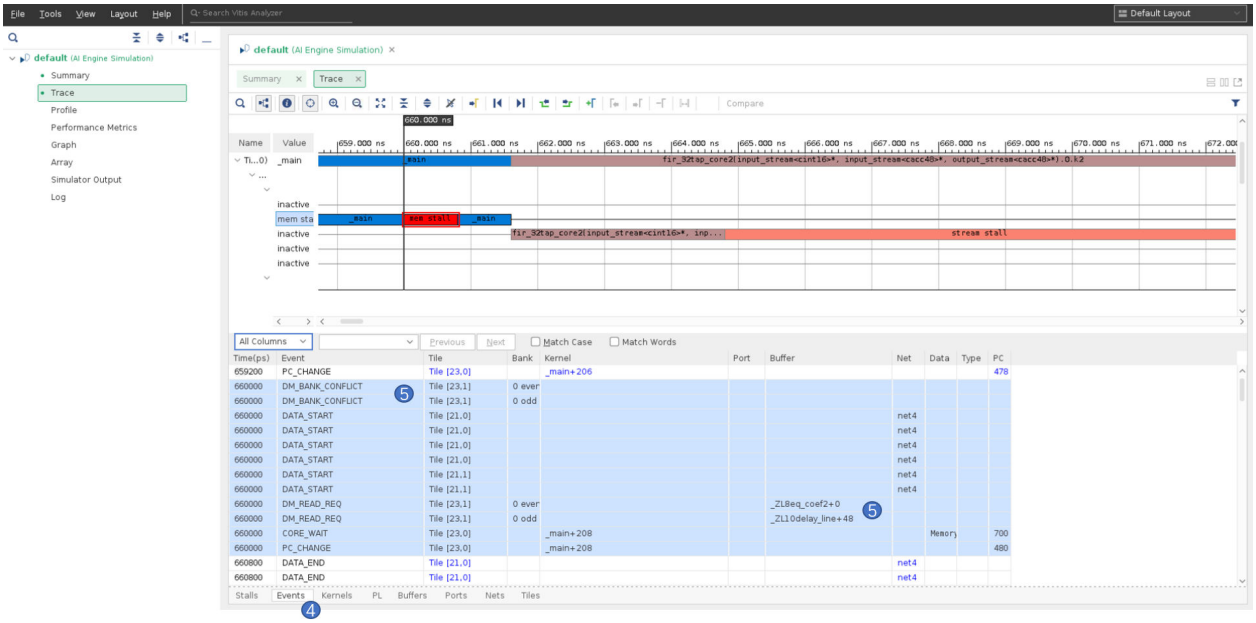
- “Buffer 1”、“Buffer 2”、“Buffer 3”：导致存储器停滞的缓冲器。可能仅有一个缓冲器，也可能有多个缓冲器导致停滞。

3. 单击“Stalls”（停滞）视图中的每一行停滞时，它会转至“Trace”（追踪）视图中存储器停滞的起始位置。通过缩放“Trace”（追踪）视图即可观察存储器停滞的发生频率，以及运行中的内核中发生停滞的位置。

   注释：如果在运行中的内核中重复发生大量存储器停滞，这表明停滞可能循环发生。最好对此问题进行调查并解决。如果仅在内核开始运行时发生一次存储器停滞，或者在内核运行期间仅发生少量停滞，通常可将其忽略。根据导致停滞的缓冲器名称可以识别出导致停滞的是窗口缓冲器、系统缓冲器或其它缓冲器。如果是可在 graph 中加以控制的窗口缓冲器或 RTP 缓冲器，那么可以使用约束对其进行手动布局，前提是能够找到更好的布局方法。如果是系统存储器（名为 `system<NUM>+<NUM>`），则需识别停滞中所涉及的变量。

4. 单击特定停滞对应的行，然后切换至“Events”（事件）视图。

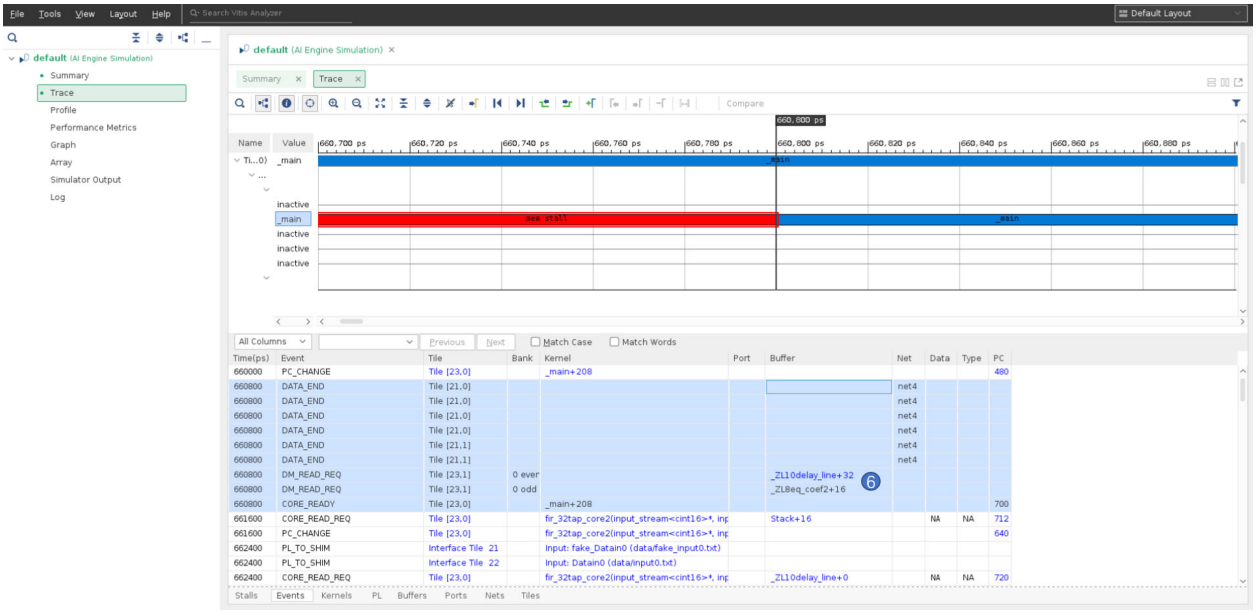
图 35：**“Memory Stall”的“Events”视图**



5. “Events”视图可显示器件中发生的事件。发生存储器停滞的周期会高亮显示。您可看到发生 DM_BANK_CONFLICT 事件的 tile（拼块），以及正在读取或写入的变量。在前图中，同一周期内正在同时读取 `delay_line` 变量和 `eq_coef1` 变量。

6. 请尝试浏览停滞周期之前或之后的其它周期以寻找线索。例如，下图显示了前一个停滞周期之后发生的事件。此外还能看到，在同一周期内的不同部分（128 位）同时也在读取 `delay_line` 和 `eq_coef1`。通过检验源代码和汇编代码可以发现，在同一周期内对 `delay_line` 和 `eq_coef1` 发出了 256 位，因而导致存储器停滞。由于存储器停滞，两次 256 位存储器访问被拆分到两个周期中。

图 36： **“Memory Stall”的“Events”视图**



在《AI 引擎内核与 Graph 编程指南》(UG1079) 的“含虚拟资源注解的加载和存储”中记录了一种解决方案。例如，重定义指向 `eq_coef0` 和 `delay_line` 的指针，并为其添加注解 `__aie_dm_resource_a`。改用下列新指针。

```
const v8cint16 __aie_dm_resource_a* __restrict coeff = (v8cint16
__aie_dm_resource_a*) eq_coef0; const v8cint16 coe = *coeff;
```

```
v16cint16 __aie_dm_resource_a* __restrict p_buff = (v16cint16
__aie_dm_resource_a*) &delay_line; v16cint16 buff=*p_buff;
```

下表列出了导致存储器停滞的部分可能场景以及可能的解决方案。

表 41：**存储器停滞场景和解决方案**

来源	目标	停滞类型	可能的解决方案	注释
单个内核	单个存储体上的缓冲器	存储器停滞	· 将存储器分派至不同存储体（存储器包括系统存储器、RTP、窗口缓冲器、DMA 和 FIFO）。请参阅 存储器停滞。 · 利用虚拟存储器注解来为编译器调度提供指引。请参阅《AI 引擎内核与 Graph 编程指南》(UG1079) 中的“含虚拟资源注解的加载和存储”部分	单个内核访问同一存储体上的多个存储器。 或者，单个内核对同一存储体上的单个存储器进行多次访问。 （每个周期均可包含两次加载和一次存储）
相邻 AI 引擎 tile 上的多个内核	单个存储体上的多个缓冲器	存储器停滞	· 将存储器分派至不同存储体（存储器包括系统存储器、RTP、窗口缓冲器、DMA 和 FIFO）。 · BufferOptLevel。请参阅 映射器和布线器选项。 · 如果存储体耗尽，请执行剖析和 AI 引擎停滞分析，寻找内核执行时间更短或者停滞比例更低的更优解决方案。	多个内核访问同一存储体上的多个存储器。

Vitis 分析器中的 FIFO 深度可视化

在 Vitis 分析器的时间线中可使用基于 VCD 的分析来查看 DMA FIFO 大小。Vitis 分析器可显示实时使用的 FIFO 深度。根据使用的 FIFO 深度，Vitis 分析器可帮助分析设计停滞问题、最优化所需 FIFO 大小，然后最优化设计性能。

注释：不支持串流开关 FIFO 大小。

为了启用 DMA FIFO 大小可视化，应启用 `aiesimulator` 的 VCD 转储选项。并且还可在 Vitis 分析器中打开仿真运行结果，例如：

```
aiesimulator --pkg-dir=./Work --online -wdb -ctf
vitis_analyzer aiesimulator_output/default.aierun_summary
```

如需了解有关如何在 Vitis 分析器中运行仿真器和打开运行结果的其它选项的信息，请参阅 在 Vitis 分析器中执行 AI 引擎停滞分析。

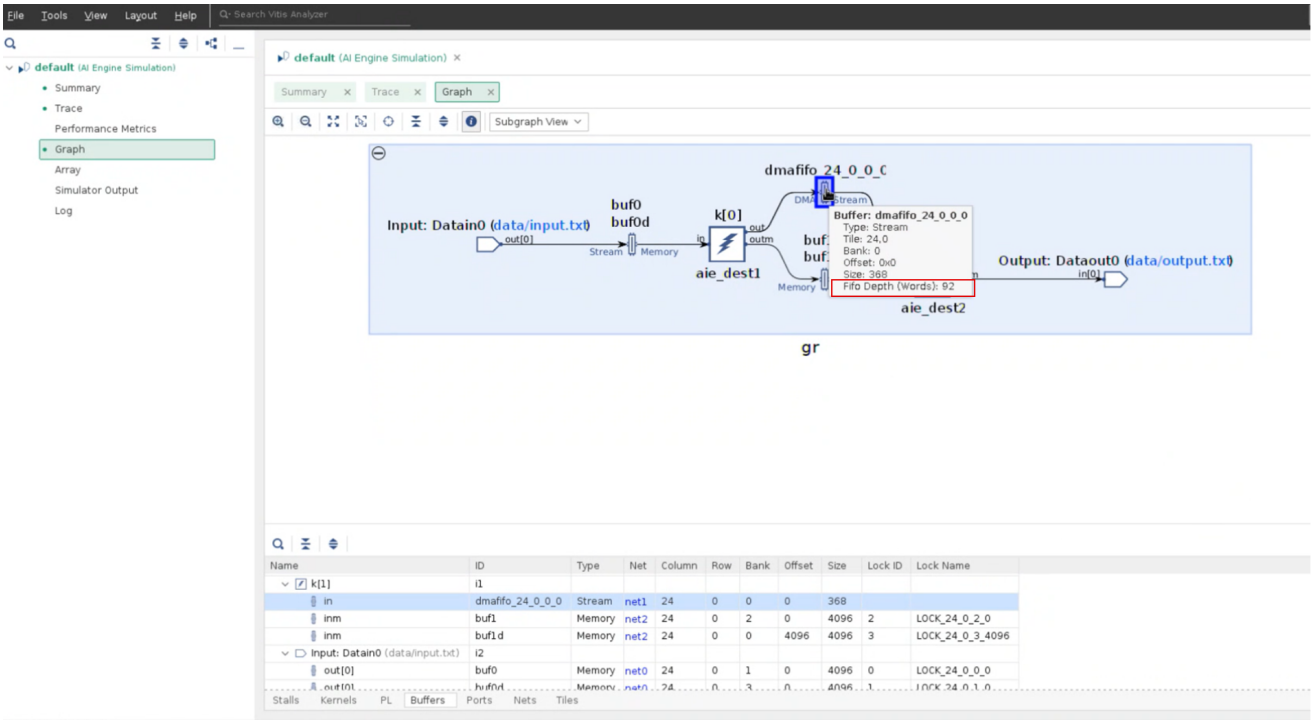
提示：如果设计仿真时挂起，请使用 `--simulation-cycle-timeout=<cycles>` 选项在建立时间停止 `aiesimulator` 仿真。

在 Vitis 分析器中，选中“Graph”视图，然后选中视图中的 DMA FIFO。

提示：将鼠标悬停在 DMA FIFO 上即可显示有关 DMA FIFO 的信息，例如，FIFO 深度（以码字数来表示）。

下图显示了 graph 视图中的 DMA FIFO 示例：

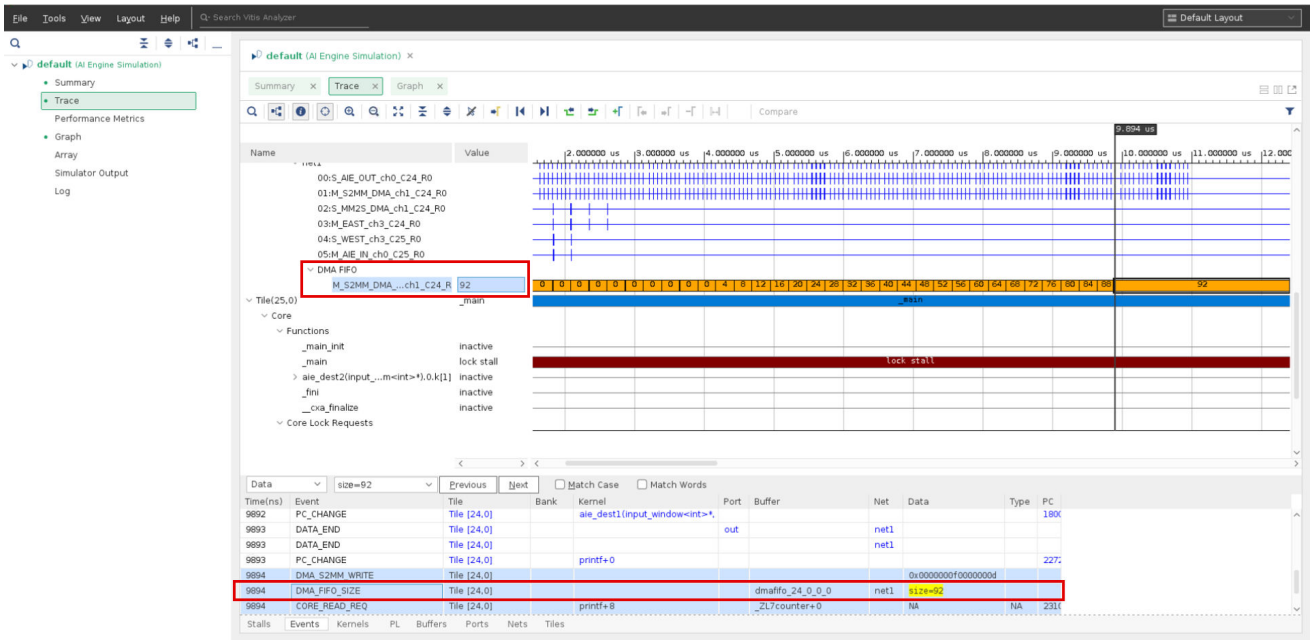
图 37：**graph 视图中的 DMA FIFO**



graph 视图中选定的 DMA FIFO 也会同时在“Trace”视图中高亮显示。您必须切换至“Trace”视图。

在时间线中会以码字数来表示 DMA FIFO 大小，如下图所示。在“Events”（事件）视图中，它包含 `DMA_FIFO_SIZE` 事件和大小信息

图 38：**“Trace”视图中的 DMA FIFO**



注释：DMA FIFO 大小将显示为 4 的倍数，因为仅当它在存储体中一次性成功写入 4 个码字（128 位）后，才会递增。如果最后写入的字数小于 4，那么最后写入的码字会暂存在 S2MM 通道内，使用者从不读取这些码字。为确保没有任何数据暂存在 S2MM 通道内，串流必须连续或者为 4 个码字的倍数。

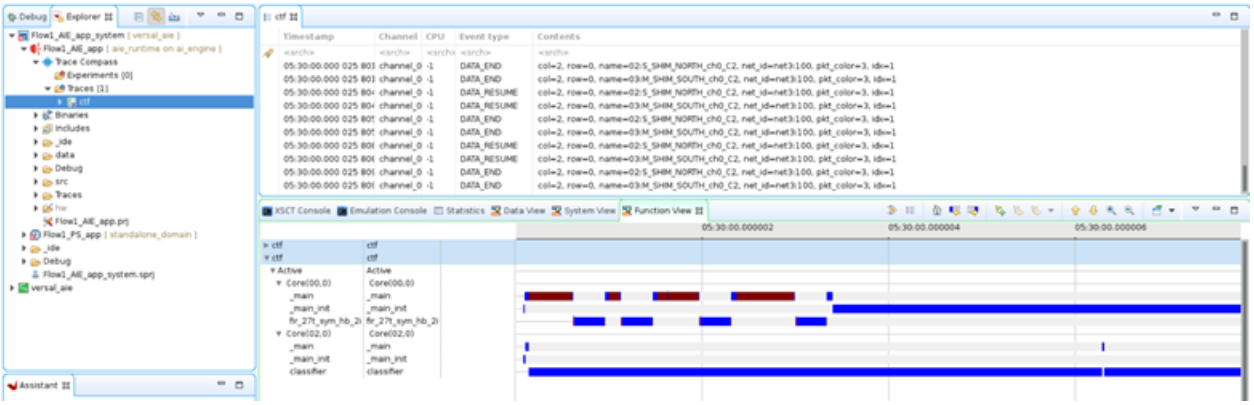
DMA FIFO 信息可搭配其它设计分析技巧一起使用来帮助设计最优化。如需了解其它设计分析技巧，请参阅 在 Vitis 分析器中执行 AI 引擎停滞分析。

使用 Trace Compass 来直观显示 AI 引擎追踪

基于 CTF 的 AI 引擎追踪可使用名为 Trace Compass 的免费 Eclipse 工具来直观显示。该工具已作为插件集成到 Vitis IDE 中，支持您在自己的 Vitis IDE 主面板中直观显示追踪。

注释：您可从 Trace Compass 网站 http://tracecompass.org 上下载其独立版本和文档。

1. 在 Vitis IDE 中，仿真期间捕获追踪数据后，您可右键单击自己的工程，然后选择“Analyze AIE Events”（分析 AI 引擎事件）。这样即可从仿真导入事件数据，并创建各种视图以对其进行分析。您的屏幕应如下所示：



2. 要查看各视图，请在“Statistics”（统计数据）、“Data View”（数据视图）、“System View”（系统视图）和“Function View”（函数视图）选项卡之间进行切换。

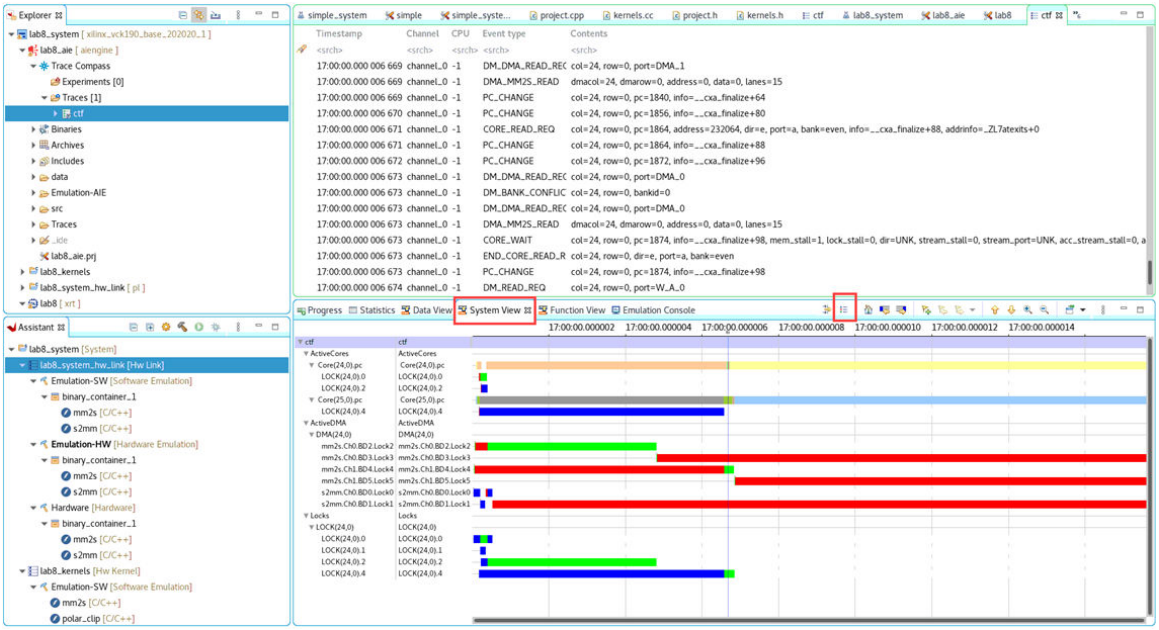
追踪视图

追踪报告支持多个视图：

- 顶层窗口按时间顺序显示事件的文本列表，以及各种事件类型信息和其它相关信息。每一列中的顶层行都允许您基于文本模式筛选事件。在底部窗口中，有多个选项卡用于提供执行相关的各种视图。
- “Statistics”（统计数据）选项卡基于选定的一组事件或时间切片 (slice) 来汇总事件统计数据。
- “System View”（系统视图）选项卡可显示系统资源状态，例如，AI 引擎、锁定和 DMA。
- “Function View”（函数视图）选项卡可显示 AI 引擎（核）上执行的各内核的状态。
- “Data View”（数据视图）选项卡表示流经串流开关网络的数据的状态。

以下分别显示了是函数视图、系统视图和数据视图的截屏。视图的顶部工具栏中包含多个选项：解释颜色的图例、放大和缩小、转至状态开始和结束位置，以及将其关联至导致状态更改的文本事件。每个视图均由一系列已对齐的时间线组成，这些时间线表示某种资源或程序对象的状态。每条时间线上各显示一种事件。您可将鼠标悬停在时间线上以查看所收集的信息。单击某一视图中的时间线就会创建一个时间条，以供您在其它视图中查看该时间的对应事件。

图 39：**含 AI 引擎、锁定和 DMA 的系统视图**



如系统视图中所示，其中有 3 个部分：“ActiveCores”（活动的核）、“ActiveDMA”（活动的 DMA）和“Locks”（锁定）。如果应用中使用了 PL 块，那么系统视图还将显示“ActivePLPorts”（活动的 PL 端口）。您可使用“ActiveCores”、“ActiveDMA”和“Locks”部分中的锁定 ID 来识别各 AI 引擎和 DMA 通过获取和释放锁定来彼此交互的方式。悬停在 `Core(0,0).pc` 条上时，就会显示当前执行的函数名称。单击图例图标即可打开图例，其中会显示颜色编码，位于主页图标左侧的 即图例图标，主页图标用于将时间刻度复位至默认值。单击左箭头或右箭头可分别转至状态开始和结束位置。文本窗口会为您显示导致状态更改的事件。在此示例中，所有锁定均已正确获取和释放。如果锁定未释放，则会出现一长条红色条形，延申至仿真时间末。

图 40：**图例**

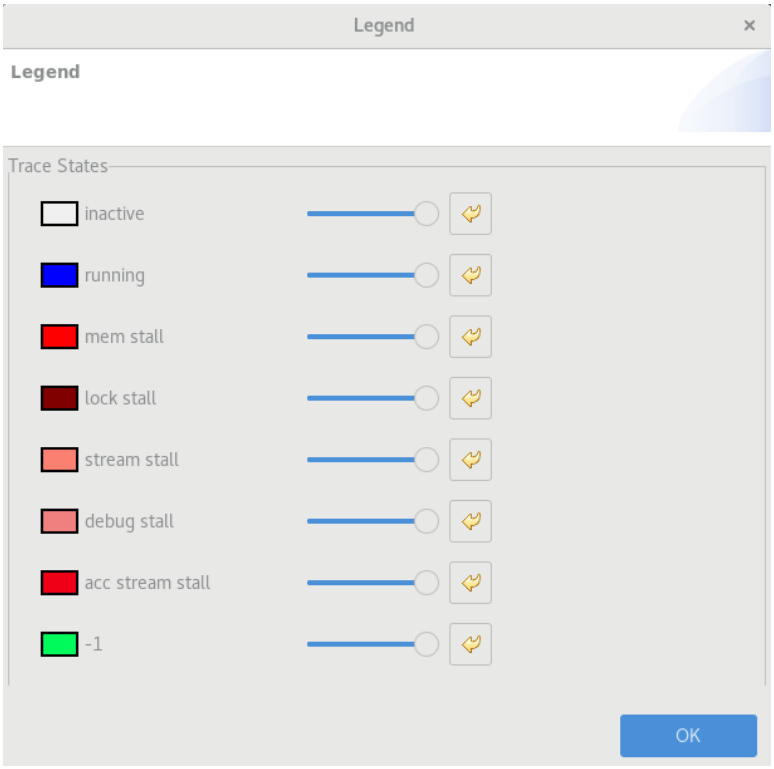
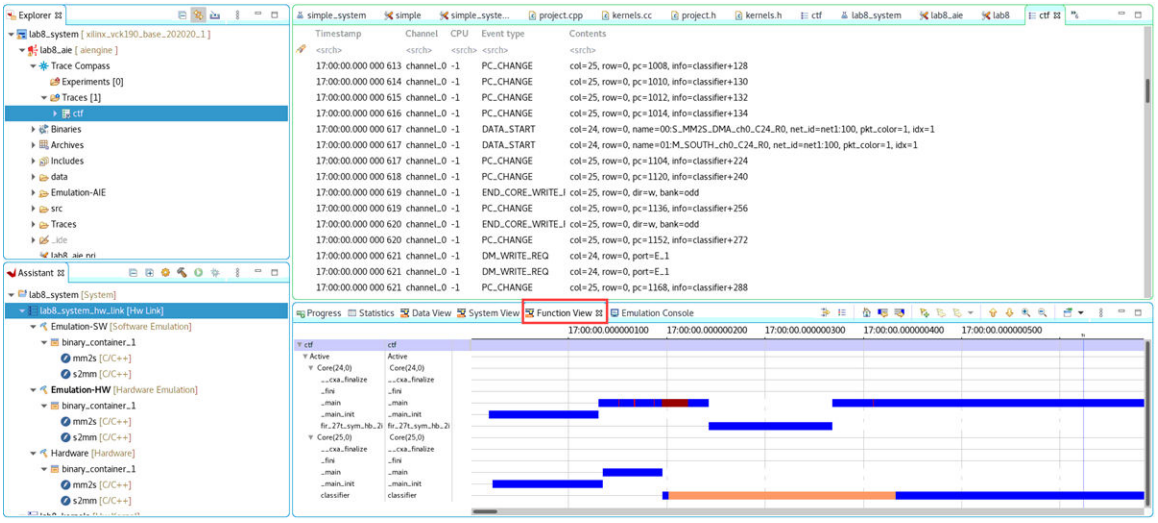
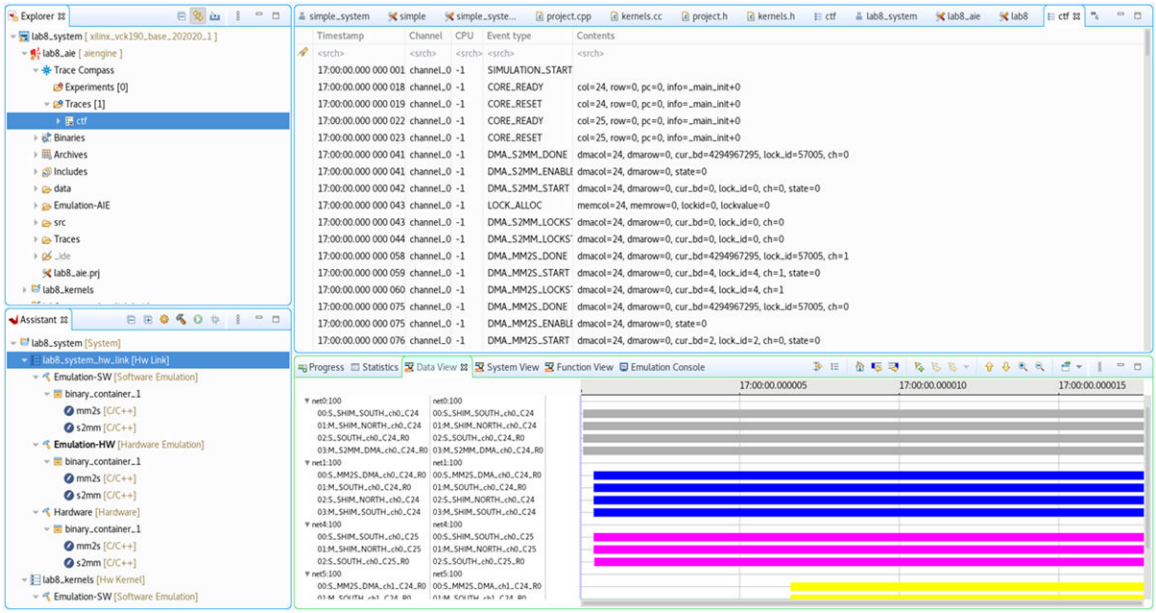


图 41：**显示每个 AI 引擎上正在运行和已停滞的内核以及 main 函数的函数视图**



函数视图最适合用于从程序角度来分析应用。对于映射到 AI 引擎（核）的每个内核都各有一条时间线，此视图会显示内核执行的时间（蓝色）或停滞的时间。将鼠标悬停在函数视图上时，会出现详细的弹出窗口，其中包含停滞类型和持续时间等详细信息。

图 42：**显示流经串流开关网络的数据的数据视图**



数据视图可显示流经串流开关网络的数据，在每个中继段中都带有从入口点和主出口点。这对于查找布线延迟以及因包切换而产生的网络拥塞影响最有用，在此类情况下，如果两个包共享相同串流通道，那么其中一个包可能延迟至另一个包之后。

AMD XILINX

第 6 章

在硬件上对 AI 引擎 graph 应用进行性能分析

程序执行的系统级视图有助于识别程序执行期间的问题，包括功能正确与否以及是否存在性能相关难题。虽然系统级别硬件和软件仿真有助于系统的开发阶段的执行，但赛灵思还提供了围绕硬件中的功能和性能调试的流程。AI 引擎架构支持在硬件执行期间，对剖析相关的数据以及事件（作为追踪数据）执行生成、收集和串流。

分析硬件中的 AI 引擎状态

AI 引擎可提供输出 AI 引擎状态汇总信息的功能，其中包括已发生的错误事件。硬件中检测到死锁时，也会发出警告。在 Vitis 分析器中可进一步分析状态和警告以供调试。

在硬件中运行设计时，有两种方法可用于输出 AI 引擎状态。

- 自动的定期 AI 引擎状态输出：在 `xrt.ini` 中完成初始设置后，此方法只需最低限度的用户干预，因为该工具将按指定时间间隔输出定期状态。
- 手动输出 AI 引擎状态：您每次想要查看状态输出报告时都必须运行命令。

随后，您即可在 Vitis 分析器中打开状态输出，进行进一步分析。

本节涵盖了获取状态输出和分析输出所需的步骤。如需了解有关停滞的更多详细信息，请参阅 在 Vitis 分析器中执行 AI 引擎停滞分析。

生成 AI 引擎状态

使用 XRT 执行定期 AI 引擎状态输出

您可使用 `xrt.ini` 文件启用运行时死锁检测和状态输出。这是一次性设置，它将导致定期输出状态数据，包括死锁检测。要开启此功能特性，请在 `xrt.ini` 文件中添加：

```
[Debug]
aie_status=true
```

要指定 AI 引擎状态探测和分析时间间隔，请执行以下操作：

```
[Debug]
aie_status=true
aie_status_interval_us=1000
```

运行主机程序时，将把 AI 引擎状态复制到下列文件：

- `xrt.run_summary`：运行汇总，包含可供 Vitis 分析器使用的文件列表信息。
- `aie_status_edge.json`：AI 引擎状态和 AI 引擎存储器。
- `aieshim_status_edge.json`：AI 引擎接口 tile （拼块）状态。
- `summary.csv`：始终创建此文件，用于其它剖析功能，例如，提供指导信息。

如果检测到死锁，则会报告如下警告信息：

```
[XRT] WARNING: Potential deadlock/hang found in AI Engines. Graph : gr
[XRT] WARNING: Potential stuck cores found in AI Engines. Graph : gr Tile :
(25,1) Status 0x81 : Enable,Lock_Stall_W
```

除 AI 引擎状态外，如果发生任何错误事件，则会将其记录在 JSON 文件中。

您负责判定错误严重性以及是否可恢复。使用汇总文件和 JSON 文件在 Vitis 分析器中进行分析。

手动 AI 引擎状态输出

您可以在器件完成加载后，随时将 AI 引擎状态的单一快照输出到 JSON 文件。这是 AI 引擎运行状态的静态快照，并包含在此之前发生的事件。

使用 `xbutil` 命令搭配 `-f json` 选项即可将状态输出至 JSON 文件。例如：

```
xbutil examine -r aie -d 0 -f json -o aie_status_xbutil.json
```

稍后，可遵循特定步骤将此 JSON 文件导入 Vitis 分析器。

在 Vitis 分析器中分析 AI 引擎状态

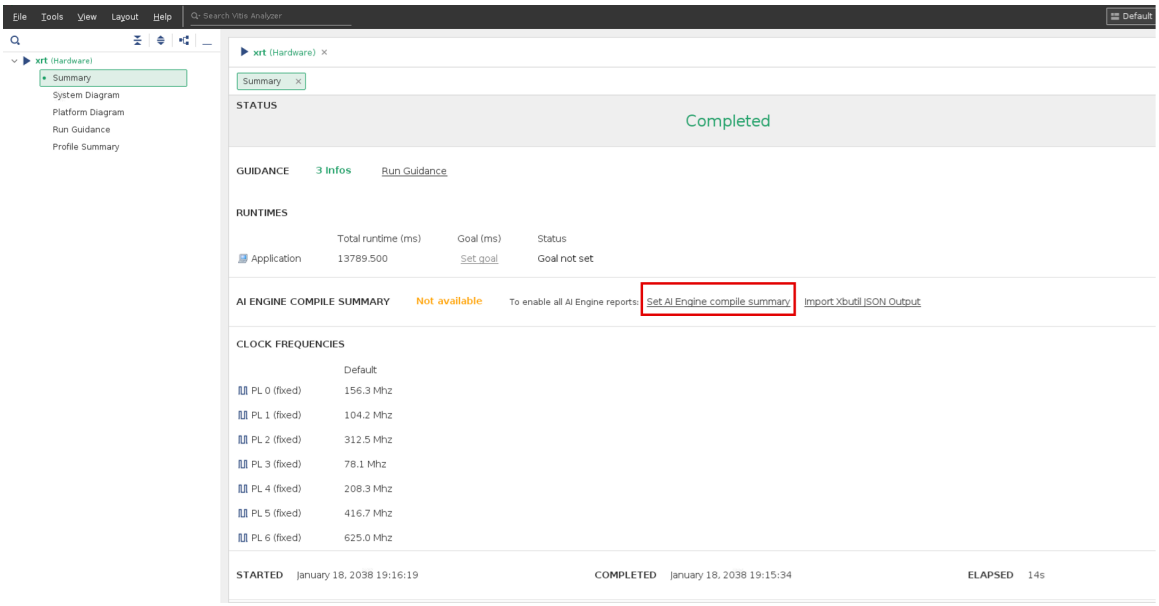
分析自动状态输出

在 Vitis 分析器中，您可分析自动生成的 AI 引擎状态输出，如下所示：

1. 使用以下命令打开运行汇总文件 `xrt.run_summary`：

   ```
   vitis_analyzer xrt.run_summary
   ```

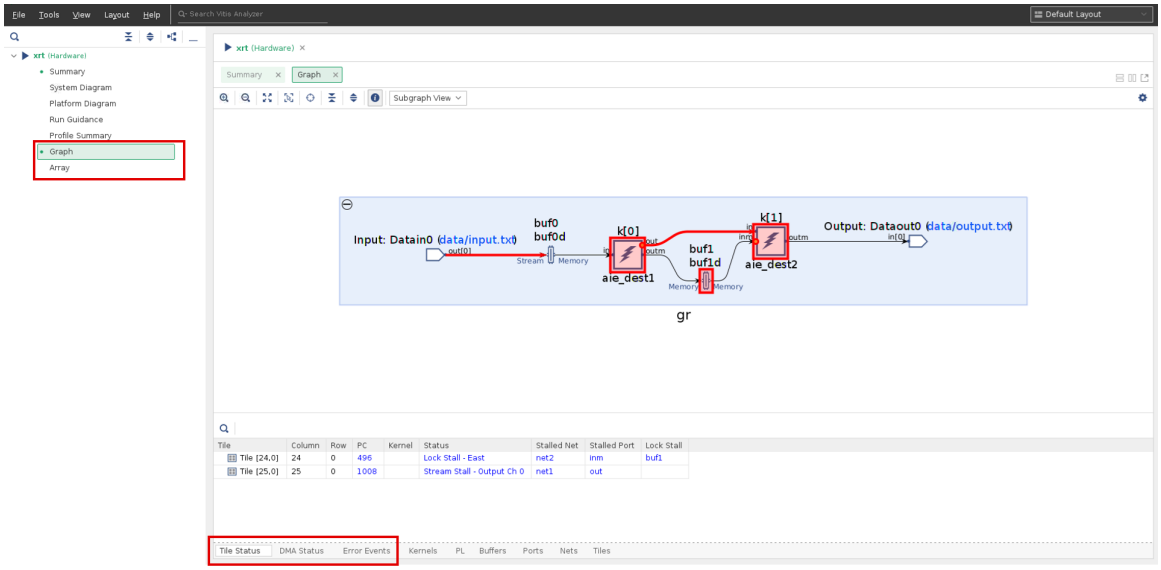
2. 在 Vitis 分析器中，单击汇总视图中的“Set AI Engine compile summary”（设置 AI 引擎编译汇总）以设置 AI 引擎编译汇总，如下所示。



3. 在弹出的对话框中，单击“...”按钮，并选择 AI 引擎编译汇总，如 `./Work/graph.aiecompile_summary`。汇总数据显示在 Vitis 分析器中。
4. 单击“Graph”或“Array”即可显示所期望的汇总数据。注意，所有停滞均以红色显示。
5. 您可在汇总数据底部访问与 AI 引擎状态相关的关联视图，其中部分视图定义如下。
   - “Tile Status”（拼块状态）：显示 AI 引擎状态，包括串流停滞、级联停滞和锁定停滞。各列解释如下：
     - “PC”：显示当前 PC 值。单击该值即可通过源代码对其进行交叉探测。
     - “Status”（状态）：显示 AI 引擎状态。可能的值包括：
       - “Done”（完成）：全部完成。
       - “Memory Stall - South”（存储器停滞 - 南）：AI 引擎在访问位于 AI 引擎南侧的存储器时已停滞。请参阅“Array”（阵列）视图，获取 AI 引擎的物理布局和缓冲器的物理布局。
       - “Memory Stall - West”（存储器停滞 - 西）：AI 引擎在访问位于 AI 引擎西侧的存储器时已停滞。
       - “Memory Stall - North”（存储器停滞 - 北）：AI 引擎在访问位于 AI 引擎北侧的存储器时已停滞。
       - “Memory Stall - East”（存储器停滞 - 东）：AI 引擎在访问位于 AI 引擎东侧的存储器时已停滞。
       - “Lock Stall - South”（锁定停滞 - 南）：AI 引擎在获取位于 AI 引擎南侧的缓冲器的锁定时已停滞。
       - “Lock Stall - West”（锁定停滞 - 西）：AI 引擎在获取位于 AI 引擎西侧的缓冲器的锁定时已停滞。
       - “Lock Stall - North”（锁定停滞 - 北）：AI 引擎在获取位于 AI 引擎北侧的缓冲器的锁定时已停滞。
       - “Lock Stall - East”（锁定停滞 - 东）：AI 引擎在获取位于 AI 引擎东侧的缓冲器的锁定时已停滞。
       - “Stream Stall - Input Ch 0”（串流停滞 - 输入通道 0）：AI 引擎在读取串流输入通道 0 时已停滞。
       - “Stream Stall - Input Ch 1”（串流停滞 - 输入通道 1）：AI 引擎在读取串流输入通道 1 时已停滞。
       - “Stream Stall - Output Ch 0”（串流停滞 - 输出通道 0）：AI 引擎在写入串流输出通道 0 时已停滞。
       - “Stream Stall - Output Ch 1”（串流停滞 - 输出通道 1）：AI 引擎在写入串流输出通道 1 时已停滞。
       - “Cascade Stall - Input”（级联停滞 - 输入）：AI 引擎在读取级联输入串流时已停滞。
       - “Cascade Stall - Output”（级联停滞 - 输出）：AI 引擎在写入级联输出串流时已停滞。
       - “Debug Halt”（调试中止）：AI 引擎在启用调试模式时已停滞。
       - “ECC Error Stall”（ECC 错误停滞）：AI 引擎在发生 ECC 错误时已停滞。
       - “ECC Scrubbing Stall”（ECC 清理停滞）：AI 引擎在发生 ECC 清理错误时已停滞。
       - “Error Halt”（错误中止）：AI 引擎在发生错误时已停滞。
       - “Enabled”（启用）：AI 引擎已启用。

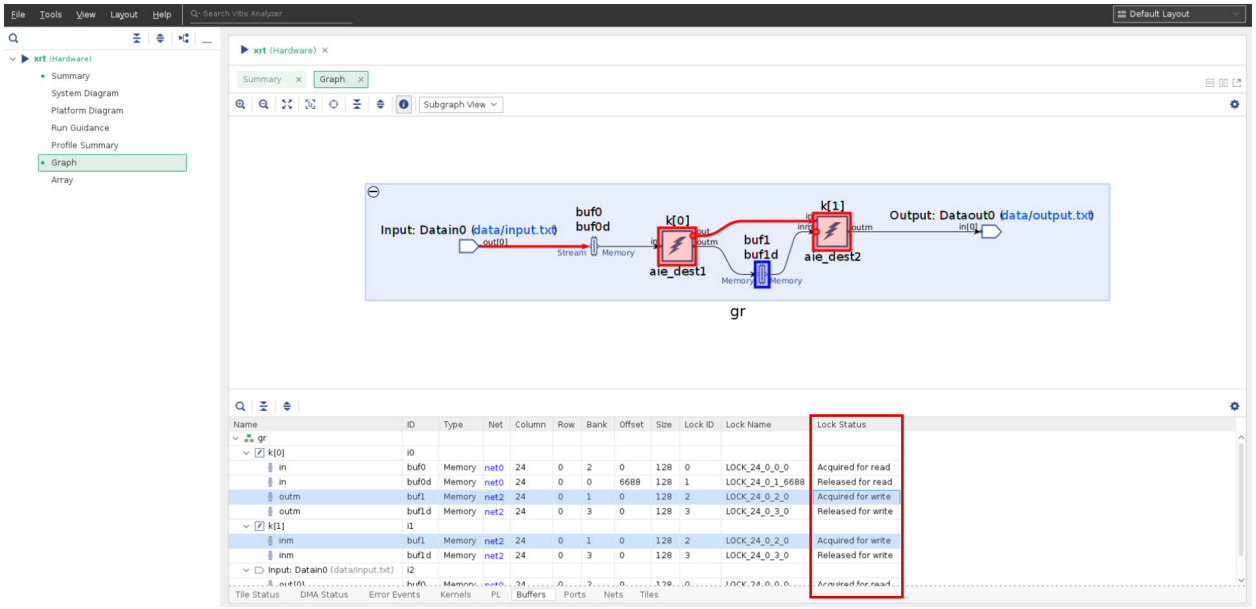
- “Reset”（复位）：处于复位状态。

图 43：**“Graph”视图中的停滞和状态**



- “DMA Status”（DMA 状态）：显示设计中的存储器模块 DMA 通道状态。
- “Error Events”（错误事件）：显示生成状态前已发生的错误事件。在下列各表中提供了更多详细信息：表 50：floating_point 指标、表 72：核模块错误事件 和 表 73：存储器模块错误事件。
- “Buffers”（缓冲器）：显示设计中的缓冲器对象（含当前状态）。请注意：
  - “Lock Status”（锁定状态）：缓冲器对象的锁定状态：
    - “Acquired for read”（已获取，可供读取）：已获取缓冲器以供使用者内核读取。
    - “Released for read”（已释放，可供读取）：已释放缓冲器以供生产者内核读取。
    - “Acquired for write”（已获取，可供写入）：已获取缓冲器以供生产者内核写入。
    - “Released for write”（已释放，可供写入）：已释放缓冲器以供使用者内核写入。

图 44：**含锁定状态的缓冲器**



提示：将鼠标悬停在“Graph”视图中的对象上。它将显示有关对象的更多信息。

如需了解有关如何分析停滞的信息，请参阅 在 Vitis 分析器中执行 AI 引擎停滞分析。

分析手动状态输出

手动生成 AI 引擎状态（如 生成 AI 引擎状态 中所述）后，您可在 Vitis 分析器中分析数据，如下所示：

1. 在 Vitis 分析器中，单击“File” → “Import Xbutil/Xsdb JSON output”（文件 > 导入 Xbutil/Xsdb JSON 输出）
2. 在弹出的窗口中，设置下列选项。
   - “Xbutil JSON File”（Xbutil JSON 文件）：选择使用 `xbutil` 命令手动生成的 JSON 文件。例如，选中 `aie_status_xbutil.json` 文件。
   - “AI Engine Compile Summary”（AI 引擎编译汇总）：选择 AI 引擎编译汇总文件。例如：

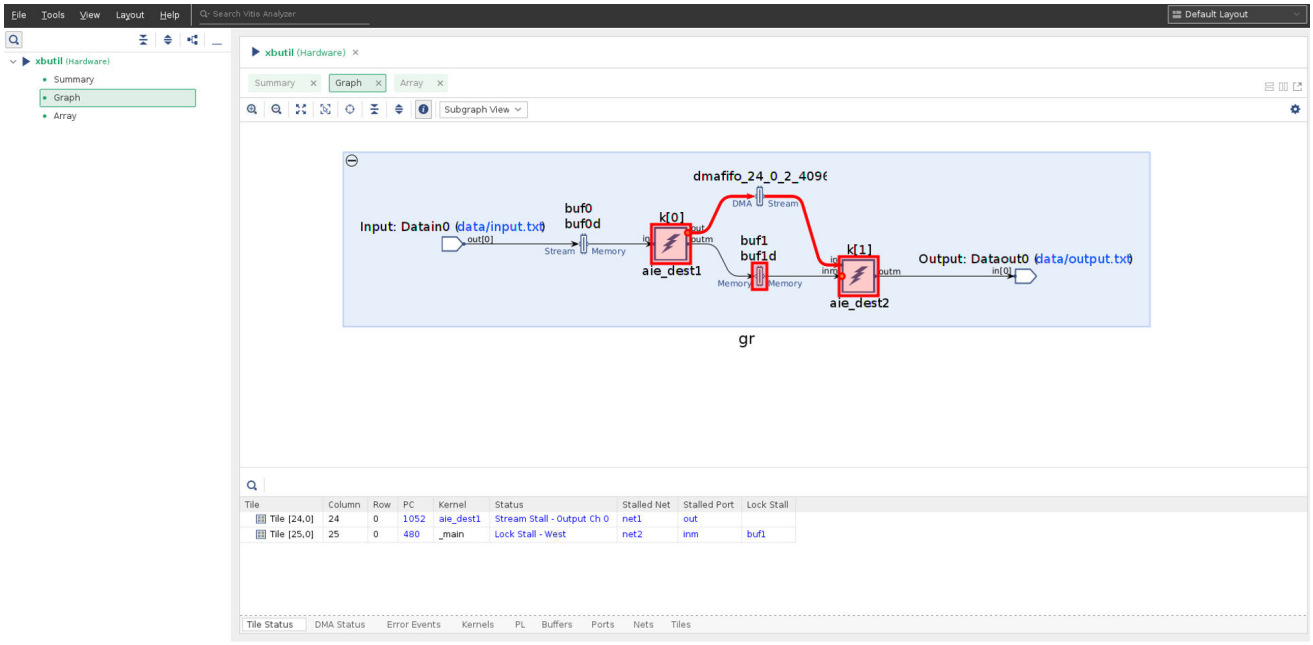
     ```
     ./Work/graph.aiecompile_summary
     ```

   - “Run Summary”（运行汇总）：要写入的运行汇总。此处已提供默认名称。运行汇总可用于下次通过如下方式重新加载分析：“File” → “Open Summary”（文件 > 打开汇总）或“File” → “Open Recent” → “Run Summary”（文件 > 打开最近 > 运行汇总）。

“Graph”视图和“Array”视图会显示在 Vitis 分析器中。下图显示了 Vitis 分析器中的手动 AI 引擎状态示例。

注释：如果先加载 JSON 文件而后加载编译工作目录，那么将不会显示此信息，因为并没有 graph 报告可供显示。导入的 JSON 文件会覆盖先前设置的任何其它 JSON 文件。

图 45：**Vitis 分析器中的 AI 引擎状态示例**



如需了解 Vitis 分析器中的视图的详细信息，请参阅 分析自动状态输出。如需了解有关如何分析停滞的信息，请参阅 在 Vitis 分析器中执行 AI 引擎停滞分析。

使用 XSDB 执行死锁检测

在 Linux 和裸机操作系统上均可使用 XSDB 检验 AI 引擎的状态。该功能特性允许您在开发板处于死锁或挂起状态时调试应用并检测 AI 引擎的状态。不同于需要 XRT 的 `xbutil` 命令，XSDB 命令独立于 XRT 来运行。

XSDB 能使用 `aiestatus examine` 命令以 `.json` 文件格式报告 AI 引擎状态。您可在运行应用之前、期间和之后使用此命令。

以下提供了有关 XSDB 的 `aiestatus examine` 命令和选项的详细解释。

命令

```
aiestatus examine
```

支持的选项

```
aiestatus examine [-graphs] <graph-list> [-work-dir] <dir-path> [-aie-
version] <version> [-file] <file-name> [-run-summary] [-target-name]
<target-name> [-tiles] <tile-list>
```

支持的选项定义如下：

表 42：**支持的选项**

选项	描述
`-graphs`	指定 AI 引擎应用中使用的一个或多个 graph 的列表。如果不指定该选项，那么此命令会将目标设为设计中的所有 graph。
`-work-dir`	指定 AI 引擎工程工作目录。这是必需选项。
`-aie-version`	指定 AI 引擎硬件生成数，可作为 `hw_gen` 值包含在 `Work/ps/c_rts/aie_control_config.json` 文件内。 **注释：**仅当不提供工作目录时才需要该选项。所使用的默认值为 1。
`-file`	以 `.json` 格式指定此命令所生成的输出文件的名称。这并非必需选项。
`-run-summary`	启用该选项即可在生成 `.json` 文件的同时，生成运行汇总文件。这并非必需选项。
`-target-name`	指定要连接的目标的名称。 **注释：**默认目标为 `Versal`。要查看可用的不同目标，请在 XSDB 终端内输入 `connect` 命令，后接 `targets`。
`-tiles`	指定一个或多个拼块的列表，这些拼块将供 AI 引擎应用使用并且将由此命令进行监控。拼块可采用 col,row 格式（例如，0,0; 0,1）。任一行或列中的所有拼块均可使用通配符“*”来指定（例如，0,*; *,1; *,*）

命令示例

使用 source 命令找到 XSDB 终端内的 `aie_status` Tcl 脚本，如以下 2 个示例所示。

```
xsdb% source $XILINX_VITIS/scripts/vitis/util/aie_status.tcl
```

- 示例 1：指定 graph 和工作目录。

  ```
  aiestatus examine -work-dir ./Work -graphs dut
  ```

  此示例会检验并报告名为 `dut` 的 graph 的状态。

- 示例 2：指定拼块。

  ```
  aiestatus examine -tiles 10,2;5,*
  ```

  此示例会检验并报告拼块 (10,2) 的状态以及列 5 中所有拼块的状态。

以上命令可在运行应用之前、期间和之后随时运行，前提是开发板正在运行设计或者处于死锁或挂起状态。此命令会在运行命令的目录内生成 JSON 文件 `<aie_status_<date_time>.json>`。文件名示例：`aie_status_2022_08_11_113542.json`。此文件可在 Vitis™ 分析器内打开，可供您查看和分析状态。要在运行应用之前读取 AI 引擎拼块的状态寄存器，请在开发板上运行主机应用之前运行 `aiestatus` 命令。

运行 `aiestatus examine` 命令之后的 XSDB 控制台样本如下所示。

图 46：**XSDB 控制台**

```
xsdb% aiestatus examine -work-dir ./Work
INFO: Targeting all graphs in design.
Reading status registers for AI Engine tiles in graph all...
AIE Tiles: 24,1 24,2 25,1 25,3 25,2
```

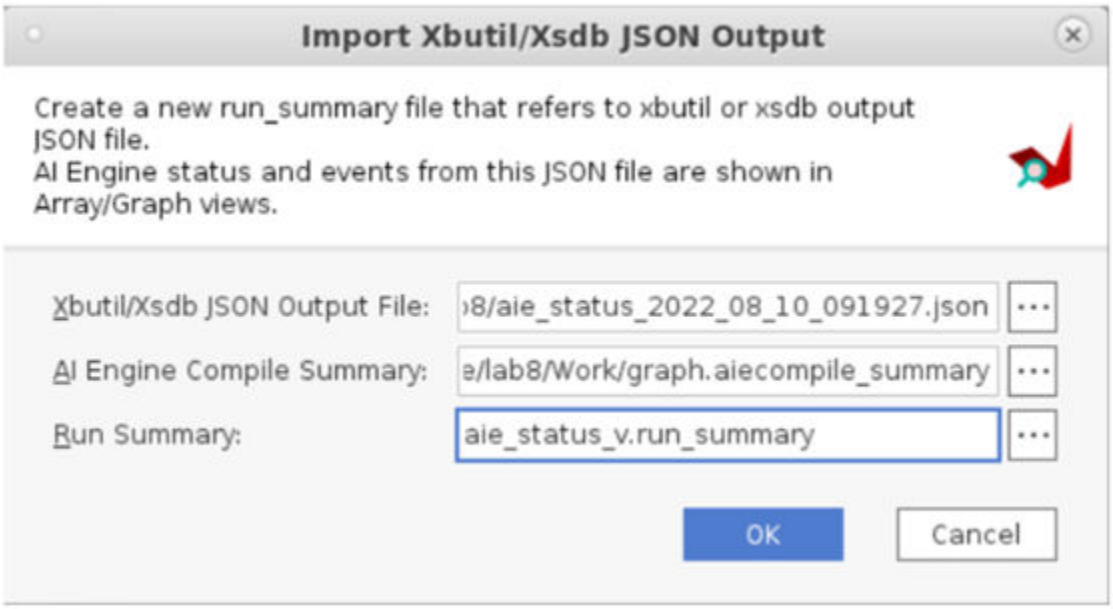
注释：在以上 XSDB 控制台样本中，并未指定 `-graphs` 选项，此命令的目标为 AI 引擎应用中的所有 graph。

此外还可观察到此命令返回与命令中指定的 graph 对应的所有 AI 引擎拼块的状态寄存器。

在 Vitis 分析器中打开 `.json` 文件

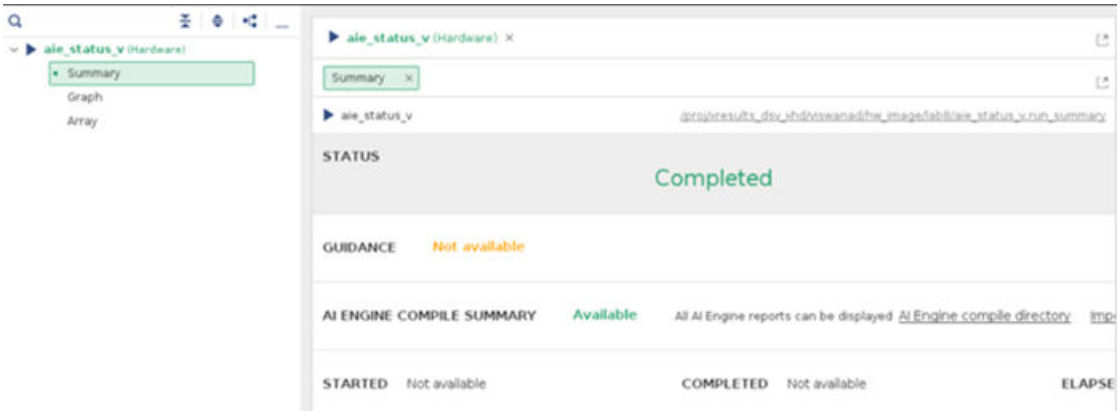
1. 打开 Vitis™ 分析器，并选中“File” → “Import Xbutil/Xsdb JSON output”（文件 > 导入 Xbutil/Xsdb JSON 输出）。
2. 在对话框中，指定 `.json` 文件以及要生成的 `AI Engine Compile Summary` 文件和 `Run Summary` 文件。

图 47：**导入 Xbutil/Xsdb JSON 输出**



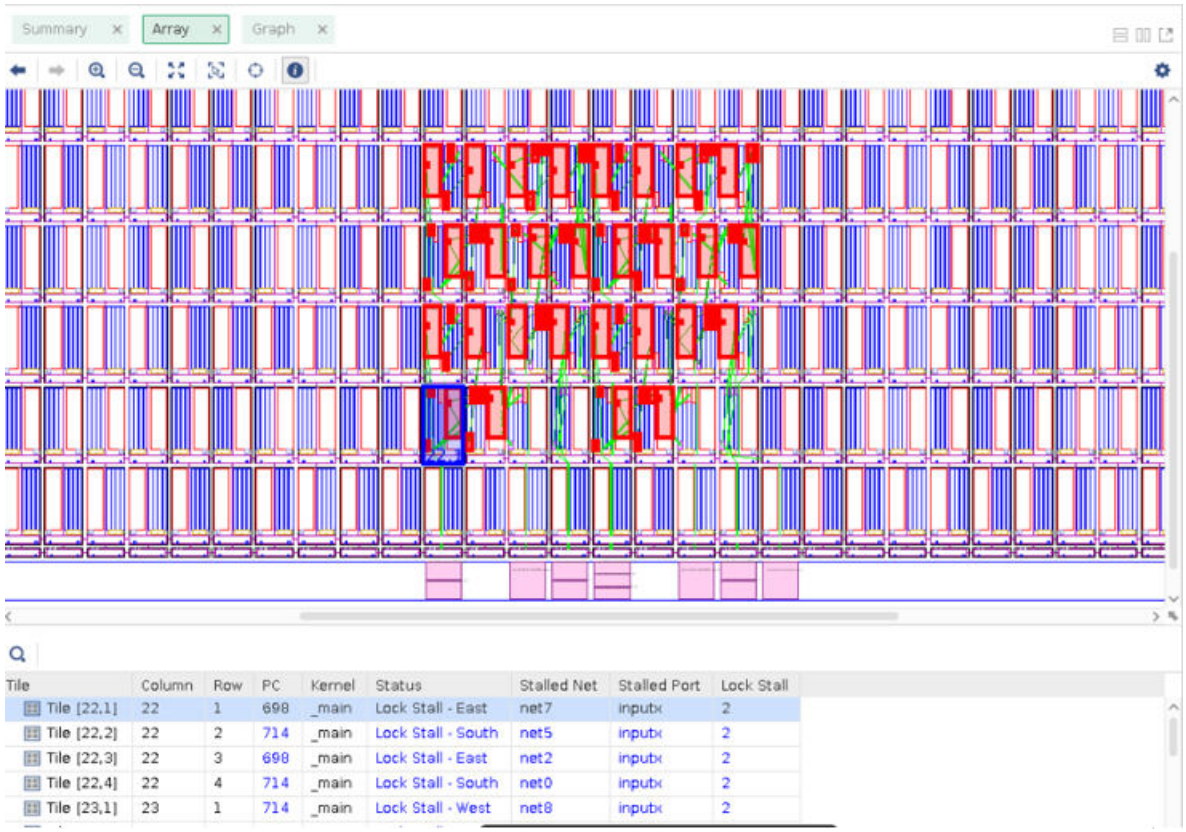
3. 单击 **OK**（确定）后，会在当前工作目录内生成新的汇总文件。“Vitis Analyzer”（Vitis 分析器）窗口会打开此汇总文件，如下所示。

图 48：**Vitis 分析器中的汇总视图**



您可导航到“Graph”或“Array”（阵列）视图，并观察 AI 引擎拼块的状态。下图显示了已停滞的 AI 引擎拼块的状态。

图 49：**阵列视图**



`aie_status.run_summary` 文件可用于后续在 Vitis 分析器中查看和分析状态结果。

或者，您也可以运行 `aiestatus` 命令搭配 `-run-summary` 选项，如下所示。

```
aiestatus examine -work-dir ./Work -graphs dut -run-summary
```

此命令会生成 `aie_status.run_summary` 文件和 `.json` 文件，这些文件均可在 Vitis 分析器中打开。

建议：使用 `-run-summary` 选项搭配 `-work-dir` 选项即可在 Vitis 分析器内无缝打开和分析此汇总文件。

AI 引擎剖析

剖析是一种动态程序分析形式，用于测量输入/输出吞吐量、空间（存储器）、程序的时间复杂性、特定指令的使用情况或者函数调用的频率和持续时间。剖析信息最常用于辅助程序最优化，具体来说就是性能调优。

在仿真中运行设计时或者在硬件仿真中的运行时，可获取剖析数据。分析此数据有助于您测量内核效率、与每个 AI 引擎关联的停滞和活动时间，并确定性能可能未处于最优状态的 AI 引擎内核。您还可在自己的 PS 主机代码中使用运行时事件 API 来收集有关设计时延、吞吐量和带宽的数据。

有两个选项可供您用于收集此信息；第一个选项是基于第二个选项构建的。

1. 使用硬件内建的性能计数器来监控 AI 引擎和存储器模块事件。此功能特性可在硬件和硬件仿真中使用。
2. 使用事件 API 剖析来自 graph 主机应用的 AI 引擎 graph 输入和输出。此功能特性可在仿真流程和硬件流程中使用。

剖析硬件中的 AI 引擎

AI 引擎事件可提供有关特定时刻的系统信息。与时间戳、类型和一系列数据值关联的事件被称为有效载荷。有效载荷的解读取决于事件类型。时间戳支持事件排序、计算因果关系和对一系列事件执行验证器的实现。

为便于事件建模，AI 引擎阵列的关键模块包括处理器、DMA、锁定模块、存储器和 I/O 串流。每个模块均可视作为事件生成器/响应器。每个模块均可接收事件和响应事件。可生成新事件作为响应。事件基于事件生成器来分类。在事件定义中并不会显式提及时间戳。每起事件均以有效载荷来描述，有效载荷即多个与该事件关联的值。每个 AI 引擎、存储器、DMA 或锁定均可按二维索引 <col, row> 进行寻址，二维索引即 AI 引擎阵列中的列索引和行索引。下表显示了部分 AI 引擎事件。

下表列出了由 AI 引擎生成的事件及其对应的 ID。

表 43：**AI 引擎事件**

事件编号	事件名称	注释
22	Group Core Stall	从事件编号 23 到 31 范围内的任意或所有事件均可触发此事件
23	Memory Stall	如果由于存储器冲突导致核停滞，则会生成此类事件
24	Stream Stall	如果由于输入处无数据或者由于来自核的串流输出上存在反压而导致核停滞，则会生成此类事件
25	Cascade Stall	如果由于输入处无数据或者由于来自核的串流输出上存在反压而导致核停滞，则会生成此类事件
26	Lock Stall	如果由于当前正在获取锁定而导致核停滞，则会生成此类事件

表 43：**AI 引擎事件** (续)

事件编号	事件名称	注释
28	Active	当核状态从 disabled（禁用）更改为 active（活动）时所生成的事件
32	Group Core Program Flow	从事件编号 33 到 45 范围内的任意或所有事件均可触发此事件
37	Instr Vector	当 AI 引擎核执行“vector”（矢量）指令时所生成的事件
38	Instr Load	当 AI 引擎核执行“load”（加载）指令时所生成的事件
39	Instr Store	当 AI 引擎核执行“store”（存储）指令时所生成的事件
40	Instr Stream Get	当 AI 引擎核执行“read from stream”（读取串流）指令时所生成的事件
41	Instr Stream Put	当 AI 引擎核执行“write to stream”（写入串流）指令时所生成的事件
42	Instr Cascade Get	当 AI 引擎核执行“read from cascade stream”（读取级联串流）指令时所生成的事件
43	Instr Cascade Put	当 AI 引擎核执行“write to cascade stream”（写入级联串流）指令时所生成的事件
50	FP Overflow	当浮点上溢异常标志位发生置位时所生成的事件
51	FP Underflow	当浮点下溢异常标志位发生置位时所生成的事件
52	FP Invalid	当浮点无效异常标志位发生置位时所生成的事件
53	FP Div by Zero	当浮点除以零异常标志位发生置位时所生成的事件

下表列出了由 AI 引擎存储器模块生成的事件及其对应的 ID。

表 44：**存储器模块事件**

事件编号	事件名称	注释
20	Group DMA Activity	从事件编号 21 到 40 范围内的任意或所有事件均可触发此事件
25	DMA S2MM 0 finished packet	当 S2MM 通道 0 完成 DMA 包的传输时所生成的事件
26	DMA S2MM 1 finished packet	当 S2MM 通道 1 完成 DMA 包的传输时所生成的事件
27	DMA MM2S 0 finished packet	当 MM2S 通道 0 完成 DMA 包的传输时所生成的事件
28	DMA MM2S 0 finished packet	当 MM2S 通道 1 完成 DMA 包的传输时所生成的事件
33	DMA S2MM 0 stalled lock acquire	当 S2MM 通道 0 在锁定获取状态下发生停滞时所生成的事件
34	DMA S2MM 1 stalled lock acquire	当 S2MM 通道 1 在锁定获取状态下发生停滞时所生成的事件

表 44：**存储器模块事件** (续)

事件编号	事件名称	注释
35	DMA MM2S 0 stalled lock acquire	当 MM2S 通道 0 在锁定获取状态下发生停滞时所生成的事件
36	DMA MM2S 1 stalled lock acquire	当 MM2S 通道 1 在锁定获取状态下发生停滞时所生成的事件
43	Group Lock	从事件编号 44 到 75 范围内的任意或所有事件均可触发此事件
76	Group Memory Conflict	从事件编号 77 到 84 范围内的任意或所有事件均可触发此事件
86	Group Errors	从事件编号 87 到 100 范围内的任意或所有事件均可触发此事件

下表列出了由 AI 引擎接口生成的事件及其对应的 ID。

表 45：**AI 引擎接口事件**

事件编号	事件名称	注释
74	Port_Idle_0	当指定 PLIO 端口处于 Idle（空闲）状态时所生成的事件。 通过监控 AI Engine Tile Stream Switch（AI 引擎拼块串流开关）的任意主端口或从端口均可查看 Port Idle（端口空闲）状况。`Stream_Switch_Event_Port_Selection_0` 寄存器位 5:0 必须加以配置。如果已配置的 Port ID（端口 ID）处于 idle（空闲）状态，则将触发此事件。
75	Port_Running_0	当指定 PLIO 端口处于 Running（运行）状态时所生成的事件。 通过监控 AI Engine Tile Stream Switch（AI 引擎拼块串流开关）的任意主端口或从端口均可查看 Port running（端口运行）状况。`Stream_Switch_Event_Port_Selection_0` 寄存器位 5:0 必须加以配置。如果已配置的 Port ID（端口 ID）处于 running（运行）状态，则将触发此事件。
76	Port_Stalled_0	当指定 PLIO 端口处于 Stalled（停滞）状态时所生成的事件。 通过监控 AI Engine Tile Stream Switch（AI 引擎拼块串流开关）的任意主端口或从端口均可查看 Port stalled（端口停滞）状况。`Stream_Switch_Event_Port_Selection_0` 寄存器位 5:0 必须加以配置。如果已配置的 Port ID（端口 ID）处于 stalled（停滞）状态，则将触发此事件。

表 45：**AI 引擎接口事件** (续)

事件编号	事件名称	注释
77	Port_TLAST_0	当指定 PLIO 端口正在发射 TLAST 信号时所生成的事件。 通过监控 AI Engine Tile Stream Switch（AI 引擎拼块串流开关）的任意主端口或从端口均可查看 Port TLAST（端口 TLAST）状况。 `Stream_Switch_Event_Port_Selection_0` 寄存器位 5:0 必须加以配置。如果已配置的 Port ID（端口 ID）处于 TLAST 状态，则将触发此事件。

剖析 AI 引擎、存储器模块和接口拼块

性能计数器分三种类型：用于 AI 引擎模块的运行时事件性能计数器、用于存储器模块的运行时存储器计数器，以及用于 AI 引擎到 PL 接口拼块的运行时接口计数器。这些性能计数器可配置为跟踪 AI 引擎、存储器模块和接口拼块中的各种事件。纠错码 (ECC) 清理、事件追踪和剖析等各种功能特性均可使用这些性能计数器。性能计数器会对剖析配置中给定事件出现的次数进行计数。剖析功能特性可为这些性能计数器提供多种不同配置，可在运行时动态应用这些配置来收集各种剖析统计数据。

使用性能计数器时，PS 主机代码中无需任何更改。在硬件中执行设计时，可在运行时对这些计数器进行配置、读取和收集。下表列出了不同配置中可用的性能计数器的数量。

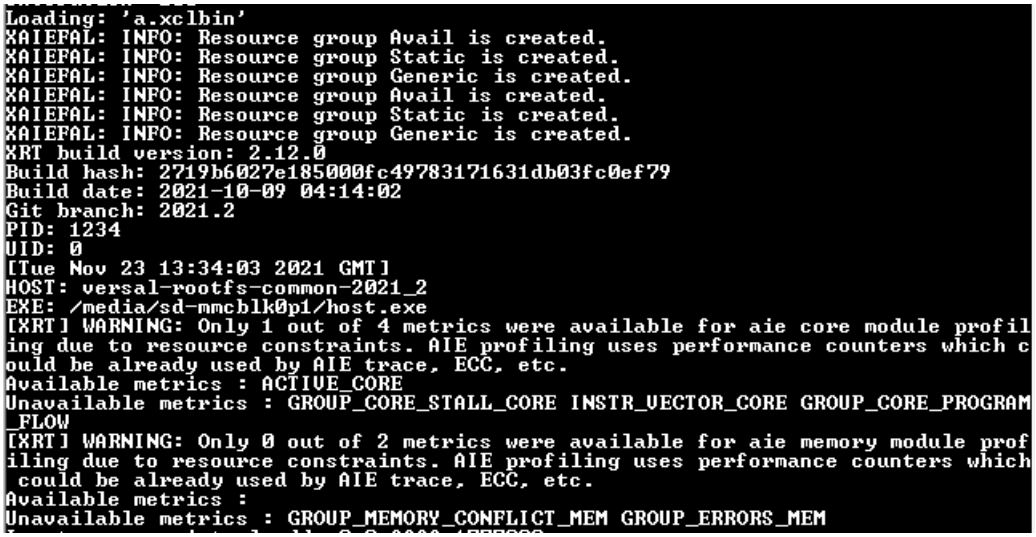
表 46：**可用性能计数器**

是否使用事件追踪?	是否使用 ECC 清理?	可用于剖析的计数器		
		核模块	存储器模块	PL 接口
不支持	不支持	4	2	2
不支持	支持	3	2	2
支持	不支持	3	1	2
支持	支持	2	1	2

ECC 清理默认开启，可使用 AI 引擎编译器选项来将其开启或关闭。如需了解更多信息，请参阅 AI 引擎编译器选项。启用 ECC 清理时，有三个计数器可用于剖析。

在同一次执行中将性能计数器用于 ECC 清理、事件追踪和剖析时，分配的性能计数器无法同时满足请求的所有功能特性的要求。以下警告消息即表示出现此状况。

图 50：**警告消息**



选中的所有指标集都应组合到 `xrt.ini` 文件中，并置于 XRT 流程的 `[Debug]` 和 `[AIE_profile_settings]` 关键字下，或者在 XSDB 流程中使用 `aieprofile` 时将所选指标集指定为标记。

AI 引擎剖析

下表列出了可用于 AI 引擎的预定义指标集配置，所列配置按分配给可用计数器的优先顺序排序。在 `xrt.ini` 文件中，所有这些指标名称均应采用小写，且分配到指标选择器 `aie_profile_core_metrics`。

表 47：**heat_map 指标**

指标名称	事件 ID	描述
Active Time	28	AI 引擎启用后变为活动状态的时间。
Stall Time	22	AI 引擎变为停滞状态的时间。此停滞包括 AI 引擎存储器停滞、串流停滞、级联停滞和锁定停滞。
Vector Instruction Time	37	AI 引擎在矢量处理器中执行指令的时间。
Cumulative Instruction Time	32	AI 引擎执行 load/store（加载/存储）、stream get/put（串流获取/放置）、lock acquire/release（锁定获取/释放）指令。
Active Utilization	衍生	AI 引擎保持主动执行指令而不停滞的时间。此百分比与 Active Time（活动时间）有关。
Vector Instruction Utilization	衍生	AI 引擎执行矢量指令的时间。此百分比与 Active Utilization（主动使用）时间（活动 - 停滞）有关。

这些指示符有助于您了解 AI 引擎中实现的内核的效率。您可将停滞时间与活动时间进行比较，以判定每个 AI 引擎是否存在数据通信问题。

表 48：**stalls 指标**

指标名称	事件 ID	描述
Memory Stall Time	23	AI 引擎因存储器停滞而变为不活动的时间。
Stream Stall Time	24	AI 引擎因串流停滞而变为不活动的时间。
Lock Stall Time	26	AI 引擎处于锁定停滞的时间。

表 48：**stalls 指标** (续)

指标名称	事件 ID	描述
Cascade Stall Time	25	AI 引擎处于级联停滞的时间。

AI 引擎在多种情况下都有可能发生停滞：

- 如果单个核、多个核和/或 DMA 请求对同一个存储体进行多次访问，则可能发生存储器停滞。
- 如果串流上的数据生成与数据耗用速率不同，导致输入串流发生数据匮乏或者输出串流发生上溢，则可能发生串流停滞。
- 如果级联写入程序与级联读取程序的速率不同，则可能发生级联停滞。
- 如果窗口数据生产者与窗口数据使用者的迭代速率不同，则可能发生锁定停滞。

表 49：**execution 指标**

指标名称	事件 ID	描述
Vector Instruction Time	37	AI 引擎用于执行矢量指令的时间：矢量处理器指令和矢量数据加载/存储
Load Instruction Time	38	AI 引擎用于执行加载指令（将数据从存储器移至寄存器）的时间
Store Instruction Time	39	AI 引擎用于执行存储指令（将数据从寄存器移至存储器）的时间
Cumulative Instruction Time	32	AI 引擎用于存储器和串流访问以及锁定获取/释放的时间

所有这些指示符都允许您估算内核的效率。为了提升效率，您应最优化数据访问、多使用矢量指令替代标量指令，并尽可能对串流使用 128 位访问。

表 50：**floating_point 指标**

指标名称	事件 ID	描述
Floating-Point Overflow Exception	50	由 AI 引擎生成的浮点上溢异常的数量
Floating-Point Underflow Exception	51	由 AI 引擎生成的浮点下溢异常的数量
Floating-Point Invalid Exception	52	由 AI 引擎生成的浮点无效异常的数量
Floating-point Divide by Zero Exception	53	由 AI 引擎生成的浮点除以零 (0) 值异常的数量

浮点异常导致错误结果。如果异常数量过多，或者代码关键区域内有单个异常，您可能就必须对浮点算法进行重新编码。

表 51：**aie_trace 指标**

指标名称	事件 ID	描述
AI Engine Trace Word Count	75	生成的 AI 引擎追踪量。
AI Engine Trace Stall Count	76	生成的 AI 引擎追踪反压事件量。
Memory Module Trace Word Count	79	生成的存储器模块追踪量。
Memory Module Trace Stall Count	80	生成的存储器模块追踪反压事件量。

这些指标（尤其是停滞计数）有助于定义正确数量的串流，以便将 AI 引擎和存储器模块事件发射到可编程逻辑。

表 52：**write_bandwidths 指标**

指标名称	事件 ID	描述
Active Time	28	AI 引擎启用后变为活动状态的时间。
Stream Write Instruction Time	41	AI 引擎用于对数据串流执行写入指令的时间。
Cascade Write Instruction Time	43	AI 引擎用于对级联串流执行写入指令的时间。
Stall Time	22	AI 引擎变为停滞状态的时间。此停滞包括 AI 引擎存储器停滞、串流停滞、级联停滞和锁定停滞。
Stream Write Bandwidth (MB/s)	衍生	串流端口的写入带宽（以 MB/s 为单位）
Cascade Write Bandwidth (MB/s)	衍生	级联端口的写入带宽（以 MB/s 为单位）

这些指标适用于对系统的总体输出写入带宽进行求值。

表 53：**read_bandwidths 指标**

指标名称	事件 ID	描述
Active Time	28	AI 引擎启用后变为活动状态的时间。
Stream Read Instruction Time	40	AI 引擎用于对数据串流执行读取指令的时间。
Cascade Read Instruction Time	42	AI 引擎用于对级联串流执行读取指令的时间。
Stall Time	22	AI 引擎变为停滞状态的时间。此停滞包括 AI 引擎存储器停滞、串流停滞、级联停滞和锁定停滞。
Stream Read Bandwidth (MB/s)	衍生	串流端口的读取带宽（以 MB/s 为单位）
Cascade Read Bandwidth (MB/s)	衍生	级联端口的读取带宽（以 MB/s 为单位）

这些指标适用于对系统的总体输出读取带宽进行求值。

存储器模块剖析

下表列出了可用于存储器模块的预定义指标集配置。在 `xrt.ini` 文件中，所有这些指标名称均应采用小写，且分配到指标选择器 `aie_profile_memory_metrics`。

表 54：**conflicts**

指标名称	事件 ID	描述
Memory Conflict	76	由于存储器模块中的 8 个 bank 上的任一 bank 存在数据存储器冲突而耗用的时间。 **注释：**硬件视图是位宽为 128 位的 8 个 bank。软件视图是位宽为 256 位的 4 个 bank。

表 54：**conflicts** (续)

指标名称	事件 ID	描述
Cumulative Memory Errors	86	由于任意数据存储体中以及 2x MM2S 和 2x S2MM DMA 中存在 ECC 错误而耗用的时间。

当两个存储器区块驻留在相同存储体内，并且供相同 AI 引擎（使用两个读取端口）访问或者供两个不同 AI 引擎访问时，就会发生存储器冲突。可能的解决方案是将这些存储器的位置约束到不同 bank。为了获取有关哪个 bank 导致这些冲突的更多详细信息，应对来自仿真（AI 引擎仿真）的事件进行分析。

表 55：**dma_locks**

指标名称	事件 ID	描述
Cumulative DMA Activity	20	由于在 DMA 的 MM2S 和 S2MM 通道上都发生已停滞的锁定获取而耗用的时间。
Cumulative DMA Lock Count	43	DMA 通道上的锁定停滞计数。

4 条 DMA 通道（2xS2MM 和 2xMM2S）均由缓冲器描述符 (BD) 来驱动。“Cumulative DMA Activity”是由于所有通道上存在已停滞的锁定获取事件而导致耗费的时间的计数。所有这些 DMA 事件将帮助您了解穿过器件的部分连接速度低于期望速度的原因。

表 56：**dma_stalls_s2mm**

指标名称	事件 ID	描述
S2MM Channel 0 Stalls	33	S2MM 通道 0 在锁定获取状态下发生停滞的时间。
S2MM Channel 1 Stalls	34	S2MM 通道 1 在锁定获取状态下发生停滞的时间。

表 57：**dma_stalls_mm2s**

指标名称	事件 ID	描述
MM2S Channel 0 Stalls	35	MM2S 通道 0 在锁定获取状态下发生停滞的时间。
MM2S Channel 1 Stalls	36	MM2S 通道 1 在锁定获取状态下发生停滞的时间。

表 58：**write_bandwidths**

指标名称	事件 ID	描述
DMA S2MM Channel 0 Packet Count	25	通过 DMA S2MM 通道 0 写入的包数。
DMA S2MM Channel 1 Packet Count	26	通过 DMA S2MM 通道 1 写入的包数。
Bandwidth of DMA S2MM channel 0	衍生	通过 DMA S2MM 通道 0 的写入带宽。此带宽是根据活动时间来计算所得的。
Bandwidth of DMA S2MM channel 1	衍生	通过 DMA S2MM 通道 1 的写入带宽。此带宽是根据活动时间来计算所得的。

这些指标允许您了解所使用的 DMA S2MM 的效率。

注释：如果在 DMA FIFO 模式下使用 DMA S2MM，那么该指标集无法提供实用结果。

表 59：**read_bandwidths**

指标名称	事件 ID	描述
DMA MM2S Channel 0 Packet Count	27	从 DMA MM2S 通道 0 读取的包数。
DMA MM2S Channel 1 Packet Count	28	从 DMA MM2S 通道 1 读取的包数。
Bandwidth of DMA MM2S channel 0	衍生	通过 DMA MM2S 通道 0 的读取带宽。此带宽是根据活动时间来计算所得的。
Bandwidth of DMA MM2S channel 1	衍生	通过 DMA MM2S 通道 1 的读取带宽。此带宽是根据活动时间来计算所得的。

这些指标允许您了解所使用的 DMA MM2S 的效率。

注释：如果在 DMA FIFO 模式下使用 DMA MM2S，那么该指标集无法提供实用结果。

接口拼块剖析

下表列出了可用于接口拼块的预定义指标集配置。在 `xrt.ini` 文件中，所有这些指标名称均应采用小写，且分配到指标选择器 `aie_profile_interface_metrics`。

所有这些指标集的目标均为每个 PLIO 中的单一通道，此通道在指定指标集之后立即指定。例如，`aie_profile_interface_metrics=input_bandwidths:2` 将剖析所有输入 PLIO 中使用的所有编号为 2 的通道。

表 60：**input_bandwidths**

指标名称	事件 ID	描述
PLIO channel transfer time	75	PLIO 通道用于数据传输的时间。
PLIO channel stalls time	76	PLIO 通道处于停滞状态的时间：使用者（AI 引擎阵列）无法接受新数据，生产者 (PL) 能够发送新数据。
PLIO channel idle time	衍生	PLIO 通道处于空闲状态的时间：使用者（AI 引擎阵列）就绪，生产者 (PL) 空闲。
PLIO channel bandwidth	衍生	PLIO 使用的带宽。

这些值引用 PLIO 的特定输入通道（PL 到 AI 引擎），共有 8 条通道（0 到 7）可用。

表 61：**output_bandwidths**

指标名称	事件 ID	描述
PLIO channel transfer time	75	PLIO 通道用于数据传输的时间。
PLIO channel stalls time	76	PLIO 通道处于停滞状态的时间：使用者 (PL) 无法接受新数据，生产者（AI 引擎阵列）能够发送新数据。
PLIO channel idle time	衍生	PLIO 通道处于空闲状态的时间：使用者 (PL) 就绪，生产者（AI 引擎阵列）空闲。
PLIO channel bandwidth	衍生	PLIO 使用的带宽。

这些值引用 PLIO 的特定输出通道（AI 引擎 到 PL），共有 6 条通道（0 到 5）可用。

表 62：**packets**

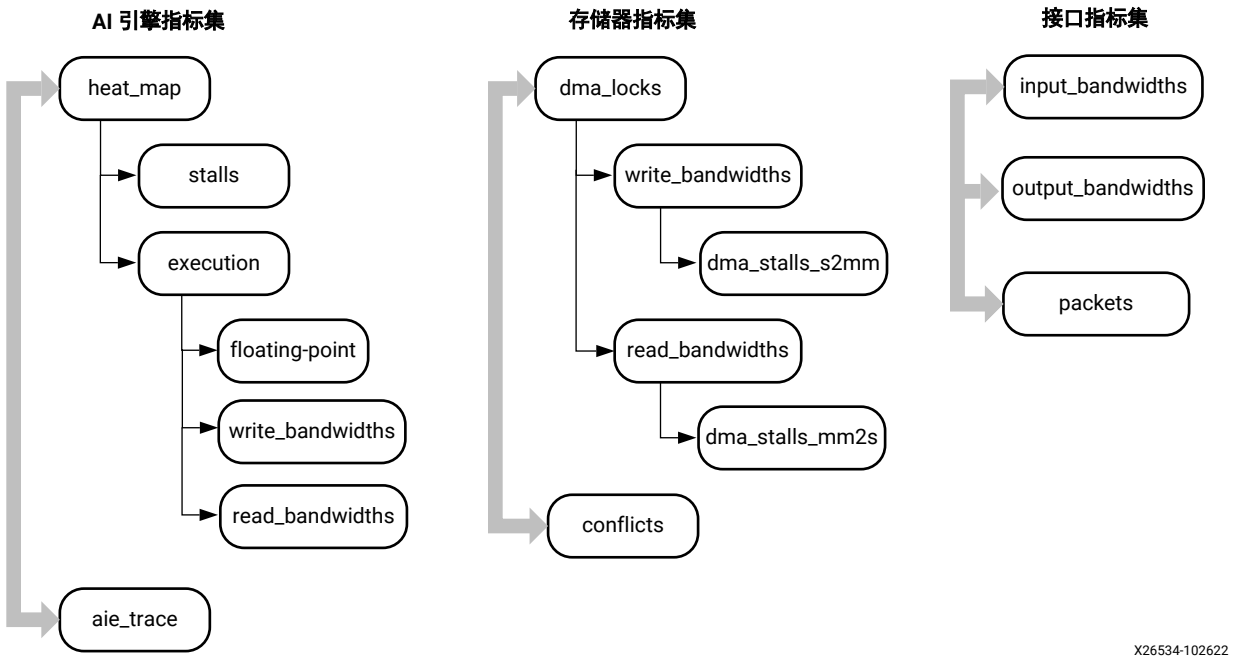
指标名称	事件 ID	描述
TLAST event count on port 0	77	报告输入数据包的数量
TLAST event count on port 1	81	报告输出数据包的数量

这些值引用所有 PLIO 的特定输入通道（PL 到 AI 引擎），共有 8 条通道（0 到 7）可用。

指标集

下图显示了 AI 引擎（核）模块、存储器模块和接口模块的指标集。这些指标集互斥。根据您的设计要求，来自某一个模块类型（例如，AI 引擎）的某一个指标可以与来自另一个模块类型（例如，存储器）的某一个指标组合使用。最初，赛灵思建议您将 `heat_map` 指标用于 AI 引擎模块，将 `conflicts` 指标用于存储器模块。这些设置是在运行时指定的，因此，运行次数无限制。但由于某些指标集使用组合事件，某些指标集则使用个别事件，因此其中部分指标集已互连。例如，`heat_map` 指标集中某一个指标将各种类型的停滞事件组合在单一指标内，其它指标则将数据传输事件（加载/存储、串流、级联等）与矢量指令组合在一起。为了更清晰地查看哪个（哪些）停滞类型更为普遍，请以 `stalls` 指标集重新运行。为了更好地了解执行情况，请以 `execution` 指标集重新运行。

图 51：**指标集**



流程剖析

以下提供了用于配置和捕获 AI 引擎剖析计数器的两种不同流程：

- XSDB 流程：此流程提供了与开发板进行通信的低开销方法，此方法对主机上使用的任意操作系统都适用。
- XRT 流程：此流程已无缝集成到赛灵思的 Xilinx Runtime (XRT) 中。

XSDB 流程

如下步骤中所述，设置 `xsdb` 以连接到器件硬件。

1. 运行应用时，会在计数器中捕获剖析数据，通过调试和剖析 IP 即可检索此数据。要捕获此数据并对其进行求值，必须使用 `xsdb` 连接到硬件器件。此命令通常用于对器件和调试应用进行编程。通过 JTAG 将您的系统连接到硬件平台或器件、在命令 shell 中启动 `xsdb` 命令，然后运行以下命令序列：

```
xsdb% connect
xsdb% ta 1
xsdb% source $::env(XILINX_VITIS)/scripts/vitis/util/aie_profile.tcl
xsdb% aieprofile start -graphs myGraph -work-dir ./Work \
      -graph-based-aie-metrics "dut:kernel1:heat_map" \
      -tile-based-aie-metrics "all:stalls" \
      -graph-based-aie-memory-metrics "dut:all:write_bandwidths" \
      -tile-based-aie-memory-metrics "{4,1}:{6,2}:conflicts;
{8,3}:dma_locks" \
      -tile-based-interface-tile-metrics "2:10:input_bandwidths:3" \
      -interval 20  -samples 100
```

   其中：

   - `connect`：启动 hw_server 并将 `xsdb` 连接到器件。
   - `source $::env(XILINX_VITIS)/scripts/vitis/util/aie_profile.tcl`：使用 source 命令运行 Tcl trace 命令以设置 `xsdb` 环境。
   - `aieprofile start`：指示 DPA IP 基于配置开始捕获剖析数据。
     - `-graph`：要捕获的 graph 剖析数据。
     - `-work-dir`：aiecompiler 工作目录的路径
     - `-graph-based-aie-metrics`：针对选定的 AI 引擎，将捕获此处选定的指标。基于 graph 名称与内核名称来确定所选项。
     - `-tile-based-aie-metrics`：针对基于拼块选定的 AI 引擎，将捕获此处指定的指标。
     - `-graph-based-aie-memory-metrics`：针对选定的 AI 引擎存储器，将捕获此处指定的指标。基于 graph 名称与内核名称来确定所选项。
     - `-tile-based-aie-memory-metrics`：针对基于拼块选定的 AI 引擎存储器，将捕获此处指定的指标。
     - `-tile-based-interface-tile-metrics`：针对基于拼块选定的 AI 引擎阵列接口和通道，将捕获此处指定的指标。
     - `-interval`：采样时间间隔，以毫秒为单位，默认值为 20。
     - `-samples`：计数器样本数（默认为 100）。
2. 在硬件上运行设计以生成硬件剖析数据。当设计应用运行时，XSDB `aieprofile` 脚本仍应保持运行并执行采样，直至应用完成为止，以便捕获有效的统计数据。为了确保如此，可相应调整 `-samples` 与 `-interval` 开关。
3. 使用 `vitis_analyzer` 导入和分析数据，如 使用 Vitis 分析器查看剖析结果 中所述。

aieprofile 命令综合

```
aieprofile start [options] -tiles <tile-list>
```

```
aieprofile start [options] -graphs <graph-list>
```

表 63：**aieprofile 选项**

选项名称	描述
`-config-file <json-file>`	由 AI 引擎编译器生成的 JSON 文件，位于 `Work/ps/c_rts/aie_control_config.json`，其中包含剖析配置数据。创建此文件也可以获取定制剖析配置。或者，您也可以指定 `-work-dir` 选项以替代 `config-file`。
`-graph-based-aie-metrics <graph name|all>:<kernel name|all>:<metric_set>`	该选项允许您为指定的 graph 与内核设置 AI 引擎计数器指标集。有效值包括 `off`、`heat_map`、`stalls`、`execution`、`floating-point`、`write_bandwidths`、`read_bandwidths` 和 `aie_trace`。
`-tile_based_aie_metrics <{<column>,<row>}|all>:<metric_set>`	用于为指定拼块设置 AI 引擎计数器指标。有效指标如上所述。
`-tile_based_aie_metrics {<mincolumn,<minrow>}:{<maxcolumn>,<maxrow>}:<metric_set>`	用于为某一范围内指定拼块设置 AI 引擎计数器指标集。有效指标如上所述。
`-graph_based_aie_memory_metrics <graph name|all>:<kernel name|all>:<metric_set>`	该选项允许您为指定的 graph 与内核设置 AI 引擎存储器模块计数器指标集。有效值包括 `off`、`conflicts`、`dma_locks`、`dma_stalls_s2mm`、`dma_stalls_mm2s`、`write_bandwidths` 和 `read_bandwidths`。
`-tile_based_aie_memory_metrics = <{<column>,<row>}|all>:<metric_set>`	该选项允许您为指定拼块设置 AI 引擎存储器模块计数器指标集。有效指标如上所述。
`-tile_based_aie_memory_metrics = {<mincolumn,<minrow>}:{<maxcolumn>,<maxrow>}:<metric_set>`	用于为某一范围内指定拼块设置 AI 引擎存储器模块计数器指标集。有效指标如上所述。
`-tile_based_interface_tile_metrics <column|all>:<metric_set>[:<channel>]`	用于为指定列和通道设置 AI 引擎阵列接口计数器指标集。有效指标包括 `off`、`input_bandwidths`、`output_bandwidths` 和 `packets`
`-tile_based_interface_tile_metrics <mincolumn>:<maxcolumn>:<metric_set>[:<channel>]`	用于为某一范围和通道内指定的列设置 AI 引擎阵列接口计数器指标集。有效指标如上所述。
`-graphs <graph-list>`	该选项允许您设置在 AI 引擎应用内使用的一个或多个 graph。

`-tiles <tile-list>`	该选项允许您选择对一个或多个拼块进行剖析。拼块可采用 col, row 格式（例如，0,1 或 2,5）。任一行或列中的所有拼块均可使用通配符“*”来指定（例如，0,* 表示选择列 0 中的所有拼块）。
	`# This example profiles tile (10,2) and all tiles in column 5`
	`xsdb% aieprofile start [options] -tiles 10,2;5,*`
	注释：列索引从 0 开始。行索引从 1（而不是 0）开始。

`-work-dir <dir-path>`	该选项允许您设置工作目录。
`-interval`	该选项允许您设置采样时间间隔，此时间间隔以毫秒为单位且默认值为 20。
`-samples`	该选项允许您设置应用剖析中所使用的计数器采样的精确数量（默认值为 100）。

XRT 流程

1. 将生成的 `sd_card.img` 烧写到物理 SD 卡上。

2. 在 PS 主机应用所在位置创建 `xrt.ini` 文件。`xrt.ini` 文件示例如下所示：

```
[Debug]
#
# Profile Counters
#
aie_profile = true

[aie_profile_settings]
# Sample interval (in usec)
aie_profile_interval_us = 100
#   All tiles
tile_based_aie_metrics = all:heat_map
tile_based_aie_memory_metrics = all:conflicts
tile_based_interface_tile_metrics = input_bandwidths:0
```

其中：

- `[Debug]`：指定 XRT 的调试部分，区分大小写。
- `aie_profile`：启用剖析配置。
- `[aie_profile_settings]`：指定 XRT 的剖析设置。
- `aie_profile_interval_us`：剖析数据收集时间间隔（以微秒为单位）。
- `tile_based_aie_metrics`：配置要基于拼块应用于 AI 引擎的指标。
- `tile_based_aie_memory_metrics`：配置要基于拼块来应用的存储器指标。
- `tile_based_interface_tile_metrics`：配置要基于拼块来应用的接口指标。

3. 在硬件上运行设计以捕获剖析数据。
4. 将生成的剖析文件 `aie_profile_*.csv`、`summary.csv` 和 `xrt.run_summary` 从 SD 卡复制到设计上与 `Work` 目录同级的位置。这些生成的文件与 SD 卡上主机应用所在位置相同。
5. 使用 `vitis_analyzer` 导入和分析数据，如 使用 Vitis 分析器查看剖析结果 中所述输入。

xrt.ini 规范

```
[Debug]
#
# Profile Counters
#
aie_profile = true

# Subsection for AIE profile settings
[AIE_profile_settings]
# Interval in between reading counters (in us)
interval_us = 1000

# Graph/Kernel name
graph_based_aie_metrics = <graph name|all>:<kernel name|all>:<off|heat_map|
stalls|execution|floating_point|write_bandwidths|read_bandwidths|aie_trace>
graph_based_aie_memory_metrics = <graph name|all>:<kernel name|all>:<off|
conflicts|dma_locks|dma_stalls_s2mm|dma_stalls_mm2s|write_bandwidths|
read_bandwidths>

# AI Engine Core Metrics : Configuration can be used only once : Multiple
values can be specified on a single line separated with ';'
# Single or all tiles
tile_based_aie_metrics = <{<column>,<row>}|all>:<off|heat_map|stalls|
execution|floating_point|write_bandwidths|read_bandwidths|aie_trace>
```

```
# Range of tiles
tile_based_aie_metrics = {<mincolumn,<minrow>}:{<maxcolumn>,<maxrow>}:<off|
heat_map|stalls|execution|floating_point|write_bandwidths|read_bandwidths|
aie_trace>

# AI Engine Memory Metrics : Configuration can be used only once : Multiple
values can be specified on a single line separated with ';'
# Single or all tiles
tile_based_aie_memory_metrics = <{<column>,<row>}|all>:<off|conflicts|
dma_locks|dma_stalls_s2mm|dma_stalls_mm2s|write_bandwidths|read_bandwidths>
# Range of tiles
tile_based_aie_memory_metrics = {<mincolumn,<minrow>}:
{<maxcolumn>,<maxrow>}:<off|conflicts|dma_locks|dma_stalls_s2mm|
dma_stalls_mm2s|write_bandwidths|read_bandwidths>

# Interface Tiles
# Single or all columns
tile_based_interface_tile_metrics = <column|all>:<off|input_bandwidths|
output_bandwidths|packets>[:<channel>]
# Range of columns
tile_based_interface_tile_metrics = <mincolumn>:<maxcolumn>:<off|
input_bandwidths|output_bandwidths|packets>[:<channel>]
```

表 64：**xrt.ini 选项**

选项名称	描述
`[Debug]`	该选项用于指定 XRT 的调试部分，区分大小写。
`aie_profile`	启用剖析配置。
`[AIE_profile_settings]`	该选项用于指定 XRT 的剖析设置，区分大小写。
`interval_us`	该选项允许您设置采样时间间隔，此时间间隔以微秒为单位且默认值为 20。
`graph_based_aie_metrics = <graph name|all>:<kernel name|all>:`	该选项用于配置 AI 引擎指标，该指标将应用于所有 graph 或特定 graph 内的所有内核或特定内核。该指标将应用于拼块，即使内核拼块内不只包含内核也是如此。 **注释：**在 2022.2 中，`<kernel_name>` 始终被替换为 `all`。
`graph_based_aie_memory_metrics = <graph name|all>:<kernel name|all>:`	该选项用于配置存储器指标，该指标将应用于所有 graph 或特定 graph 内的所有内核或特定内核。该指标将应用于拼块，即使内核拼块内不只包含内核也是如此。 **注释：**在 2022.2 中，`<kernel_name>` 始终被替换为 `all`。
`tile_based_aie_metrics = <{<column>,<row>}|all>:`	该选项用于配置 AI 引擎指标，该指标将应用于单个拼块或所有拼块
`tile_based_aie_metrics = {<mincolumn,<minrow>}:{<maxcolumn>,<maxrow>}:`	该选项用于配置 AI 引擎指标，该指标将应用于某一范围内的所有拼块。
`tile_based_aie_memory_metrics = <{<column>,<row>}|all>:`	该选项用于配置 AI 引擎存储器指标，该指标将应用于单个拼块或所有拼块
`tile_based_aie_memory_metrics = {<mincolumn,<minrow>}:{<maxcolumn>,<maxrow>}:`	该选项用于配置 AI 引擎存储器指标，该指标将应用于某一范围内的所有拼块。
`tile_based_interface_tile_metrics = <column|all>:<metric>[:<channel>]`	该选项用于配置 AI 引擎接口指标，该指标将应用于指定通道 ID 上的单个拼块或所有拼块。如果省略通道 ID，则将取默认通道。
`tile_based_interface_tile_metrics = <mincolumn>:<maxcolumn>:<metric>[:<channel>]`	该选项用于配置 AI 引擎接口指标，该指标将应用于指定通道 ID 的某一范围内的所有拼块。如果省略通道 ID，则将取默认通道。

xrt.ini 示例

在此首个示例中，抽取剖析信息前并未执行任何拼块或 graph 选择。将使用默认指标集来剖析所有已用的拼块：

```
[AIE_profile_settings]
tile_based_aie_metrics = all:heat_map
tile_based_aie_memory_metrics = all:conflicts
tile_based_interface_tile_metrics = input_bandwidths:0
```

在此第二个示例中，演示了边界框和拼块选择：

```
[AIE_profile_settings]
tile_based_aie_metrics = {4,1}:{6,2}:stalls; {10,4}:execution
tile_based_aie_memory_metrics = {4,1}:dma_locks
```

在此最后一个示例中，基于 graph 名称选择内核。多项选择必须包含在一行内，并以分号分隔：

```
[AIE_profile_settings]
graph_based_aie_metrics = tx_chain_0:all:execution;
tx_chain_1:all:floating_point
graph_based_aie_memory_metrics = tx_chain_2:all:write_bandwidths
```

注释：如果使用的性能计数器总数超过可用性能计数器的数量，那么 API 将不会获取性能计数器，运行失败，并在控制台上显示以下错误消息。

```
[AIE ERROR]:    Failed to request resource 0
[AIE WARNING]:  Unable to request resources. RscType: 0
XAIEFAL: WARN: perfcount _reserve (6,1) Expect Mod= 1 resource not
available.
...
```

注释：如果必须对各内核/拼块选择应用不同的指标集，这些指标集必须在同一行内并置并以分号分隔。

使用 Vitis 分析器查看剖析结果

要启动 `vitis_analyzer` 以查看 XRT 流程中的剖析信息，请使用以下命令。

```
vitis_analyzer xrt.run_summary
```

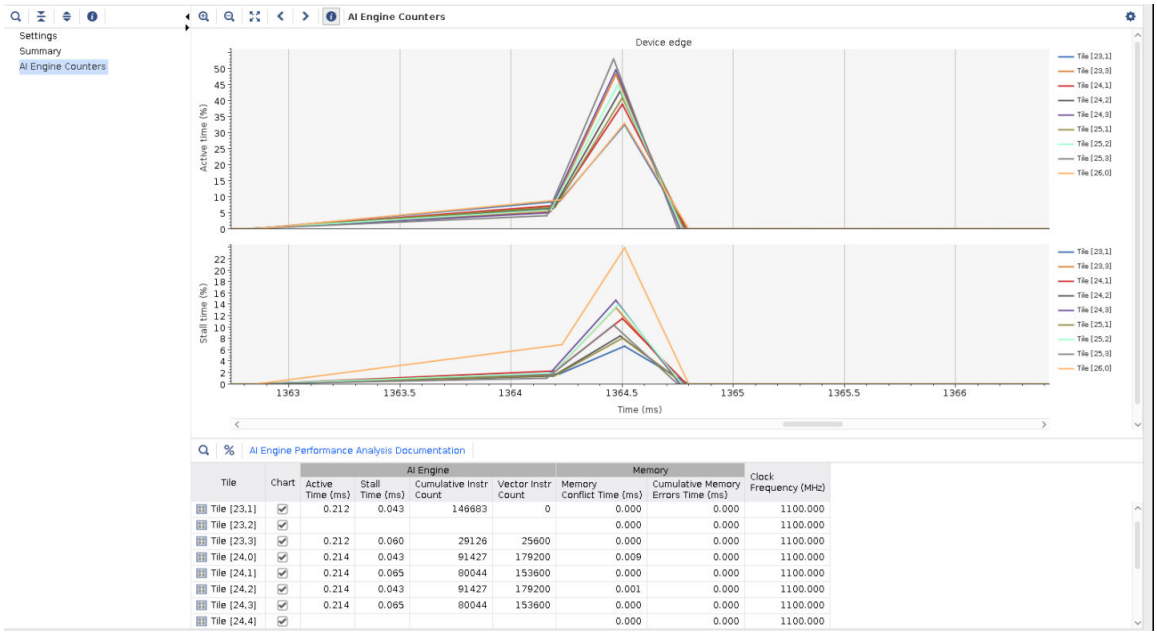
要启动 `vitis_analyzer` 以查看 XSDB 流程中的剖析信息，请使用以下命令。

```
vitis_analyzer aie_trace_profile.run_summary
```

heat_map 核指标与 conflicts 存储器指标的示例

下图显示了 `heat_map` 指标和存储器冲突时间所涵盖的设计活动时间、停滞时间、累积指令计数和 vector_instruction_count，以及对应设计示例的十个拼块的 `conflicts` 指标的累积存储器错误时间。

图 52：**heat_map 指标和 conflicts 指标的示例**



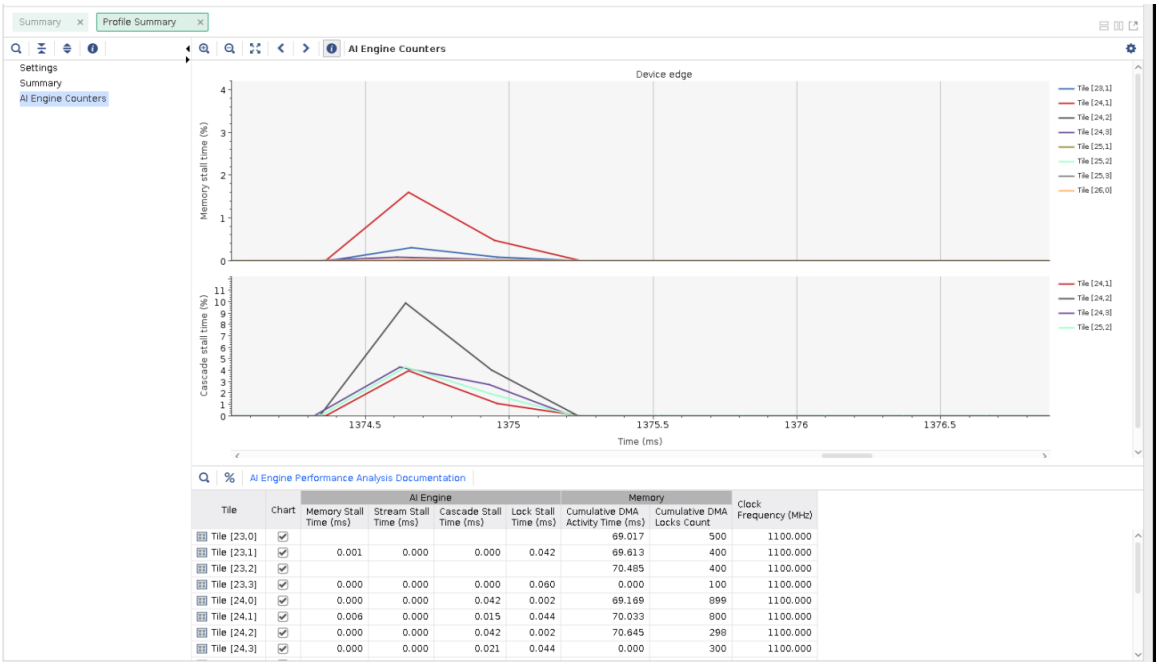
注释：请单击右上角的此  图标以启用/禁用图表。

以位于 (24,2) 的 AI 引擎为例。停滞时间 (.043 ms) 为活动时间 (.214 ms) 的 20%。在此活动时间期间，它会执行 179200 条矢量指令，占活动时间的 95%。这表示性能卓越，即核的最优化十分有效。

stalls 核指标与 dma_locks 存储器指标的示例

下图显示了 `stalls` 指标和累积 DMA 活动时间所涵盖的设计存储器停滞时间、串流停滞时间、级联停滞时间和锁定停滞时间，以及对应设计示例的十个拼块的 `dma_locks` 指标的累积 DMA 锁定计数。

图 53：**stalls 指标和 dma_locks 指标的示例**



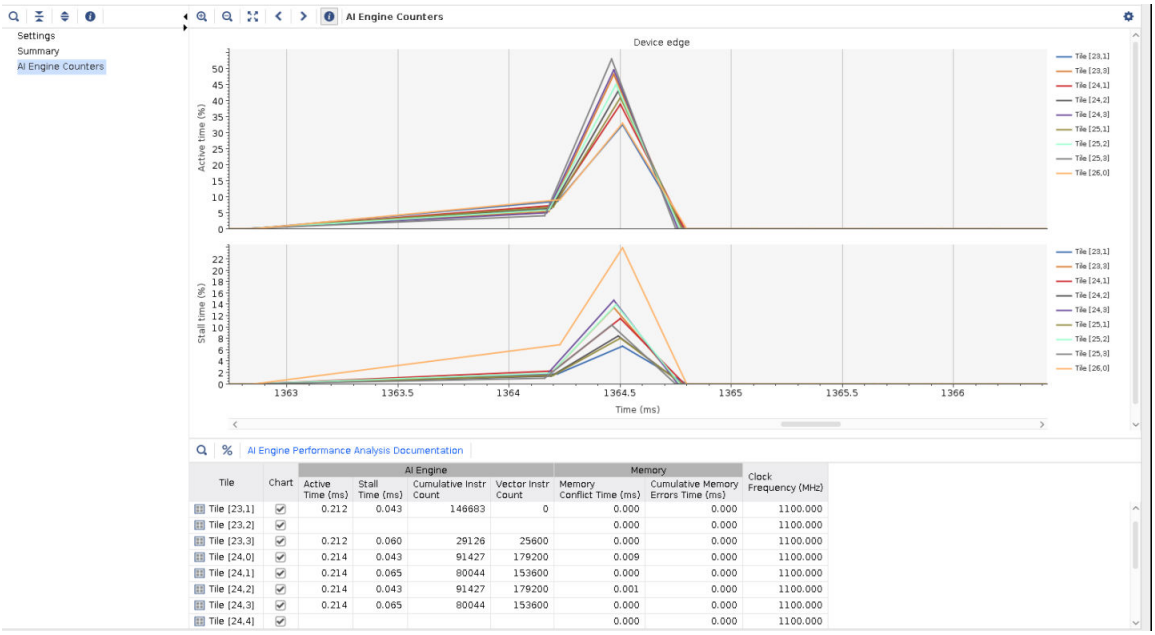
Tile	Chart	AI Engine				Memory		Clock Frequency (MHz)
		Memory Stall Time (ms)	Stream Stall Time (ms)	Cascade Stall Time (ms)	Lock Stall Time (ms)	Cumulative DMA Activity Time (ms)	Cumulative DMA Locks Count	
Tile [23,0]	☑					69.017	500	1100.000
Tile [23,1]	☑	0.001	0.000	0.000	0.042	69.613	400	1100.000
Tile [23,2]	☑					70.485	400	1100.000
Tile [23,3]	☑	0.000	0.000	0.000	0.060	0.000	100	1100.000
Tile [24,0]	☑	0.000	0.000	0.042	0.002	69.169	899	1100.000
Tile [24,1]	☑	0.006	0.000	0.015	0.044	70.033	800	1100.000
Tile [24,2]	☑	0.000	0.000	0.042	0.002	70.645	298	1100.000
Tile [24,3]	☑	0.000	0.000	0.021	0.044	0.000	300	1100.000

在核 (24,2) 上，DMA 已保持活动达 70.645 ms（7780 万条指令），但发生 298 次停滞。

execution 核指标与 conflicts 存储器指标的示例

下图显示了 `execution` 指标和存储器冲突时间所涵盖的设计累积指令计数、矢量指令计数、加载指令计数和存储指令计数，以及对应设计示例的十个拼块的 `conflicts` 指标的累积存储器错误时间。

图 54：**execution 与 conflicts 指标的示例**



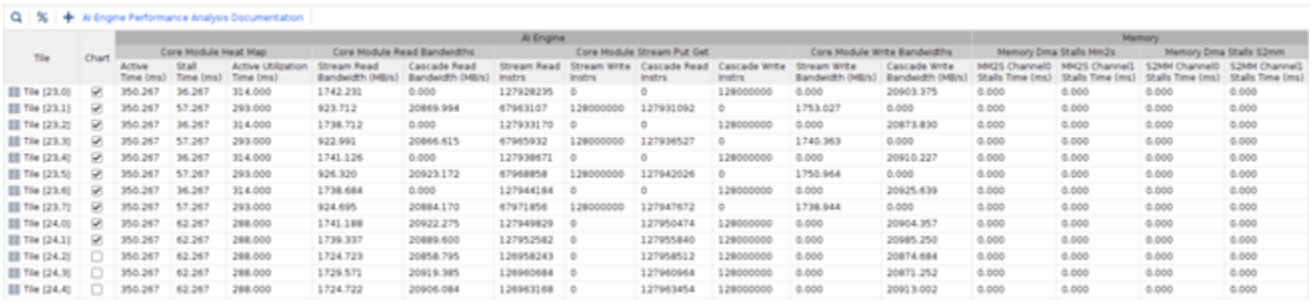
Tile	Chart	AI Engine				Memory		Clock Frequency (MHz)
		Active Time (ms)	Stall Time (ms)	Cumulative Instr Count	Vector Instr Count	Memory Conflict Time (ms)	Cumulative Memory Errors Time (ms)	
Tile [23,1]	☑	0.212	0.043	146683	0	0.000	0.000	1100.000
Tile [23,2]	☑					0.000	0.000	1100.000
Tile [23,3]	☑	0.212	0.060	29126	25600	0.000	0.000	1100.000
Tile [24,0]	☑	0.214	0.043	91427	179200	0.009	0.000	1100.000
Tile [24,1]	☑	0.214	0.065	80044	153600	0.000	0.000	1100.000
Tile [24,2]	☑	0.214	0.043	91427	179200	0.001	0.000	1100.000
Tile [24,3]	☑	0.214	0.065	80044	153600	0.000	0.000	1100.000
Tile [24,4]	☑					0.000	0.000	1100.000

核 (24,2) 存在某些存储器冲突，虽然这些只是次要冲突，但也必须明确。这些冲突较为罕见，原因可能是因为存在某些 DMA 或某些其它内核访问干扰。

AI 引擎指标 read_bandwidths 和 write_bandwidth 以及 AI 引擎存储器指标 dma_stalls_s2mm 和 dma_stalls_mm2s 的示例

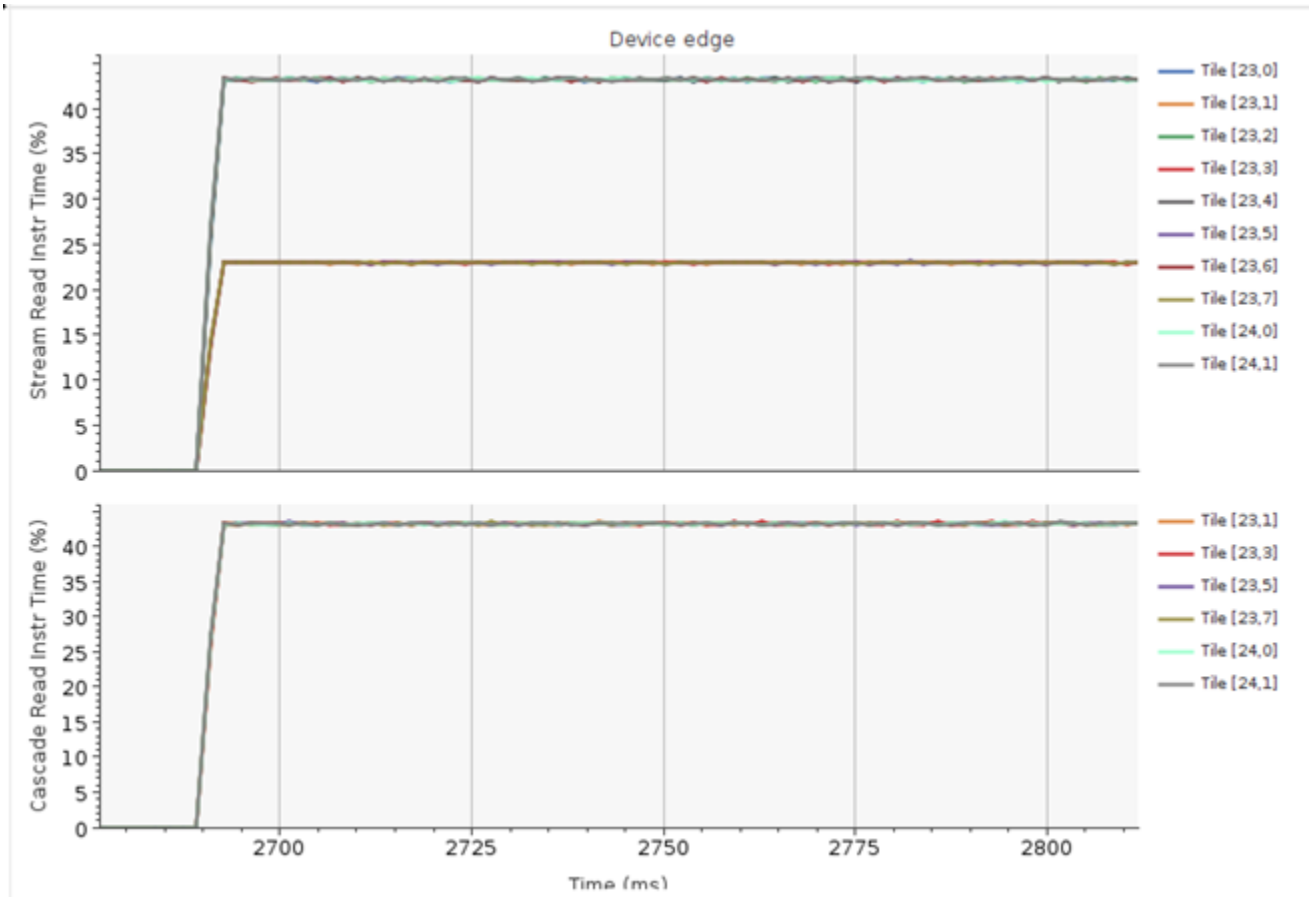
下图显示了设计示例的 10 个拼块的 `read_bandwidths` 和 `write_bandwidths` 指标中所包含的设计串流和级联的读取和写入指令计数，以及 `dma_stalls_s2mm` 和 `dma_stalls_mm2s` 指标的 s2mm 和 mm2s channel0 和 channel1 停滞时间。

图 55：**此数据表包含 AI 引擎的读取和写入带宽以及 AI 引擎存储器的所有 mm2s 和 s2mm 通道的停滞**



Tile	Chart	AI Engine										Memory			
		Core Module Heat Map			Core Module Read Bandwidths		Core Module Stream Put Get			Core Module Write Bandwidths		Memory Dma Stalls Mm2s		Memory Dma Stalls S2mm	
		Active Time (ns)	Stall Time (ns)	Active Utilization Time (ns)	Stream Read Bandwidth (MB/s)	Cascade Read Bandwidth (MB/s)	Stream Read Instrs	Stream Write Instrs	Cascade Read Instrs	Cascade Write Instrs	Stream Write Bandwidth (MB/s)	Cascade Write Bandwidth (MB/s)	MM2S Channel0 Stalls Time (ns)	MM2S Channel1 Stalls Time (ns)	S2MM Channel0 Stalls Time (ns)	S2MM Channel1 Stalls Time (ns)
Tile [23,0]	☑	350.267	36.267	314.000	1742.231	0.000	127928235	0	0	128000000	0.000	20903.375	0.000	0.000	0.000	0.000
Tile [23,1]	☑	350.267	57.267	293.000	923.712	20869.994	67963107	128000000	127931092	0	1753.027	0.000	0.000	0.000	0.000	0.000
Tile [23,2]	☑	350.267	36.267	314.000	1738.712	0.000	127933170	0	0	128000000	0.000	20873.830	0.000	0.000	0.000	0.000
Tile [23,3]	☑	350.267	57.267	293.000	922.991	20866.615	67965932	128000000	127936527	0	1740.363	0.000	0.000	0.000	0.000	0.000
Tile [23,4]	☑	350.267	36.267	314.000	1741.126	0.000	127938671	0	0	128000000	0.000	20910.227	0.000	0.000	0.000	0.000
Tile [23,5]	☑	350.267	57.267	293.000	926.320	20923.172	67968858	128000000	127942026	0	1750.964	0.000	0.000	0.000	0.000	0.000
Tile [23,6]	☑	350.267	36.267	314.000	1738.684	0.000	127944184	0	0	128000000	0.000	20925.639	0.000	0.000	0.000	0.000
Tile [23,7]	☑	350.267	57.267	293.000	924.695	20884.170	67971856	128000000	127947672	0	1738.944	0.000	0.000	0.000	0.000	0.000
Tile [24,0]	☑	350.267	62.267	288.000	1741.189	20922.275	127949829	0	127950474	128000000	0.000	20904.357	0.000	0.000	0.000	0.000
Tile [24,1]	☑	350.267	62.267	288.000	1739.337	20889.600	127952582	0	127955840	128000000	0.000	20985.250	0.000	0.000	0.000	0.000
Tile [24,2]	☐	350.267	62.267	288.000	1724.723	20858.795	126958243	0	127958512	128000000	0.000	20874.684	0.000	0.000	0.000	0.000
Tile [24,3]	☐	350.267	62.267	288.000	1729.571	20919.385	126960684	0	127960964	128000000	0.000	20871.252	0.000	0.000	0.000	0.000
Tile [24,4]	☐	350.267	62.267	288.000	1724.722	20906.084	126963168	0	127963454	128000000	0.000	20913.002	0.000	0.000	0.000	0.000

图 56：**此图表按百分比形式显示级联读取和串流读取指令时间**

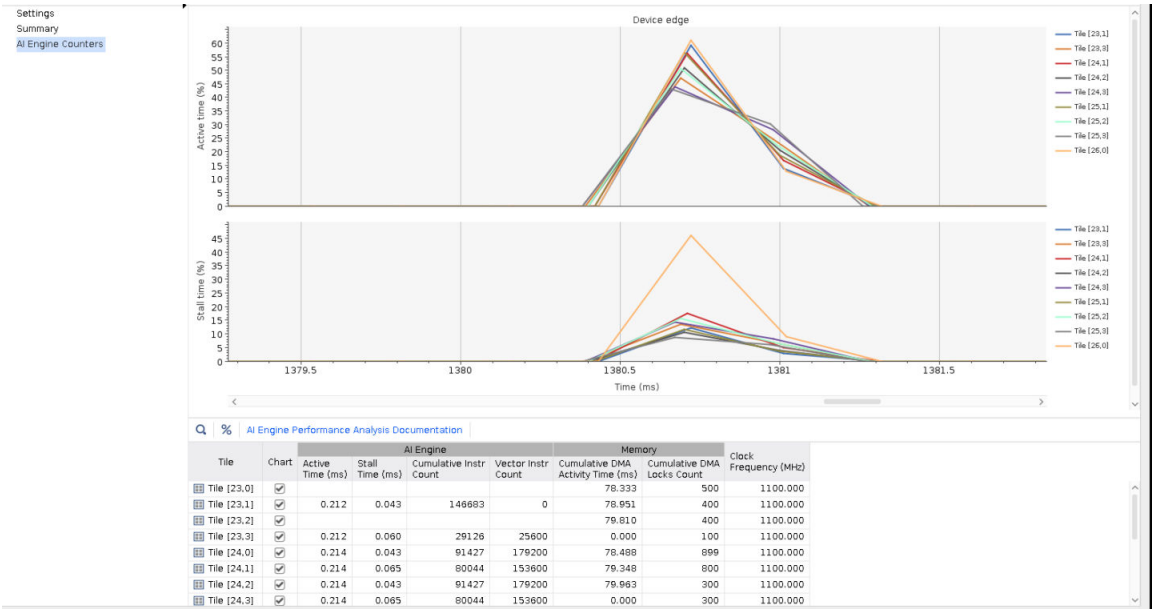


此处可以看到，在 AI 引擎内核中，超过 45% 的时间内都有一次级联读取和串流读取。这是使用 AI 引擎保持活动状态所必需的，因为串流带宽远小于存储器带宽。

heat_map 核指标与 dma_locks 存储器指标的示例

下图显示了 `heat_map` 指标和累积 DMA 活动时间所涵盖的设计活动时间、停滞时间、累积指令计数和 vector_instruction_count，以及对应设计示例的十个拼块的 `dma_lock` 指标的累积 DMA 锁定计数。

图 57：**heat_map 指标和 dma_locks 指标的示例**

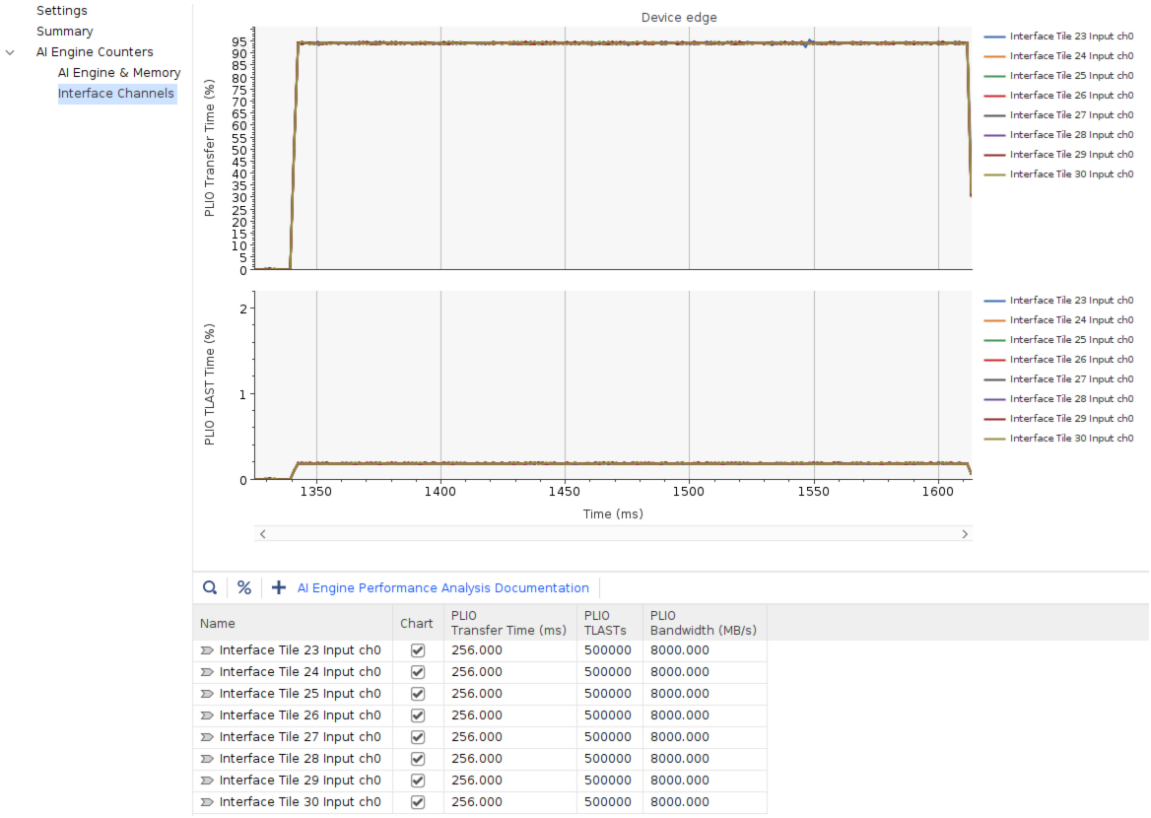


累积 DMA 活动时间与累积 DMA 锁定计数相结合即可便于您查看在锁定获取数量与通过 DMA 传输的数据量之间是否存在任何不一致。锁定计数的相对数量同样可用于解读每个核的相对迭代次数。

input_bandwidths 接口指标的示例

下图显示了 8 x 8 级联拼块设计内的 `input_bandwidths:0` 指标所涵盖的 PLIO 级设计输入带宽。

图 58：**input_bandwidths:0 接口指标示例**



Name	Chart	PLIO Transfer Time (ms)	PLIO TLASTs	PLIO Bandwidth (MB/s)
Interface Tile 23 Input ch0	☑	256.000	500000	8000.000
Interface Tile 24 Input ch0	☑	256.000	500000	8000.000
Interface Tile 25 Input ch0	☑	256.000	500000	8000.000
Interface Tile 26 Input ch0	☑	256.000	500000	8000.000
Interface Tile 27 Input ch0	☑	256.000	500000	8000.000
Interface Tile 28 Input ch0	☑	256.000	500000	8000.000
Interface Tile 29 Input ch0	☑	256.000	500000	8000.000
Interface Tile 30 Input ch0	☑	256.000	500000	8000.000

在此 graph 中，所有输入 PLIO 的通道 0 带宽约为 95%，接近可达成的最大值。经此剖析步骤后，请验证 AI 引擎并未发生数据匮乏。

Vitis 分析器中的报告整合

在剖析阶段，运行时期间无法同时使用所有指标。您可通过重启开发板来多次运行硬件中的设计，每次运行都使用 `xrt.ini` 中的不同剖析指标集。通常对于 AI 引擎接口带宽剖析，运行时期间可剖析一条通道（对于所有 PLIO 都是如此）。要进行多通道剖析，需多次运行。

`vitis_analyzer` 能够对涉及同一设计的不同运行轮次的多份报告进行整合。举例来说，这样您即可显示多条接口通道的带宽。虽然 `vitis_analyzer` 是搭配设计的特定运行的 `xrt.run_summary` 来运行的，但仍可通过单击主工具栏和窗口工具栏中的 **+** 来打开其它 `xrt.run_summary` 报告，如下所示。

图 59：**主工具栏中的“添加”(+) 按钮**

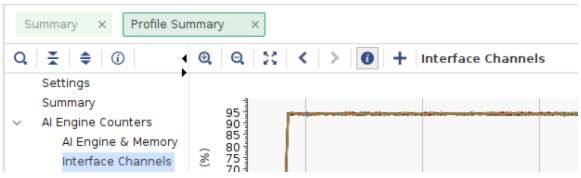
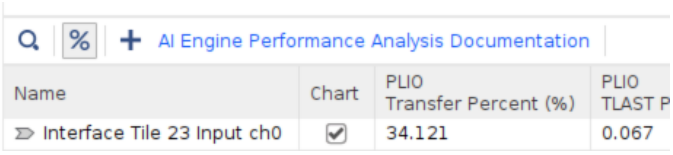


图 60：**窗口工具栏中的“添加”(+) 按钮**



为输入 PLIO 通道 0 和 4 及输出 PLIO 通道 0 整合剖析数据后，`vitis_analyzer` 即可显示下表：

图 61：**通道 0 和 4 输入及通道 0 输出 PLIO 带宽**

Name	Chart	PLIO Transfer Time (ms)	PLIO TLASTs	PLIO Bandwidth (MB/s)
Interface Tile 23 Input ch0	☑	256.000	500000	8000.000
Interface Tile 23 Input ch4	☐	256.000	500000	7586.198
Interface Tile 24 Input ch0	☑	256.000	500000	8000.000
Interface Tile 24 Input ch4	☐	256.000	500000	7583.512
Interface Tile 25 Input ch0	☑	256.000	500000	8000.000
Interface Tile 25 Input ch4	☐	256.000	500000	7586.771
Interface Tile 25 Output ch0	☐	256.000	0	7604.266
Interface Tile 26 Input ch0	☑	256.000	500000	8000.000
Interface Tile 26 Input ch4	☐	256.000	500000	7585.827
Interface Tile 27 Input ch0	☑	256.000	500000	8000.000
Interface Tile 27 Input ch4	☐	256.000	500000	7589.033
Interface Tile 28 Input ch0	☑	256.000	500000	8000.000
Interface Tile 28 Input ch4	☐	256.000	500000	7593.244
Interface Tile 29 Input ch0	☑	256.000	500000	8000.000
Interface Tile 29 Input ch4	☑	256.000	500000	7588.190
Interface Tile 30 Input ch0	☑	256.000	500000	8000.000
Interface Tile 30 Input ch4	☑	256.000	500000	7588.719
Interface Tile 21 Output ch0	☐	256.000	0	7583.766

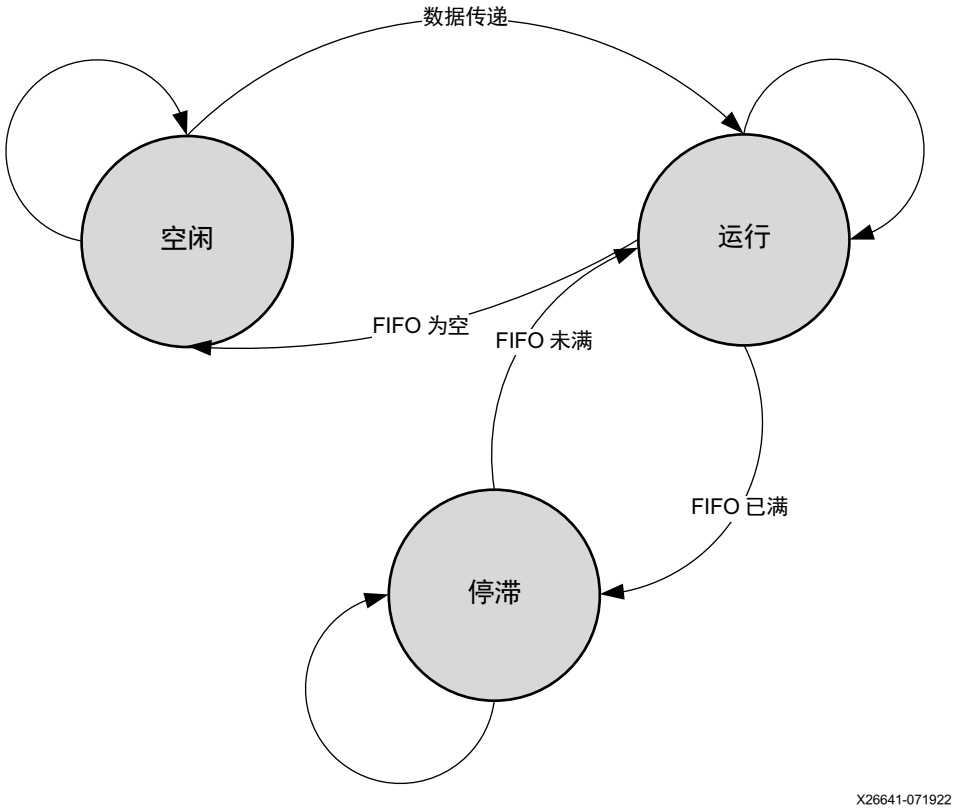
用于 graph 输入和输出的事件剖析 API

您可通过在自己的 PS 主机代码中调用事件 API 来收集设计的剖析统计数据。这些事件 API 在仿真期间以及在硬件中运行设计时均可用。

AI 引擎具有硬件性能计数器，可通过配置来对硬件事件进行计数，以测量性能指标。您可将事件 API 与 graph 控制 API 搭配使用，在 graph 执行的受控时间段内剖析某些性能指标。事件 API 仅支持平台 I/O 端口 (PLIO & GMIO) 以测量各项性能指标，如平台 I/O 端口带宽、graph 吞吐量和 graph 时延。

事件 API 会跟踪跨 AI 引擎到 PL 接口的信号线的串流开关上发生的事件。信号线的串流开关上的事件包括 `idle`、`running` 和 `stall`，如下图所示。

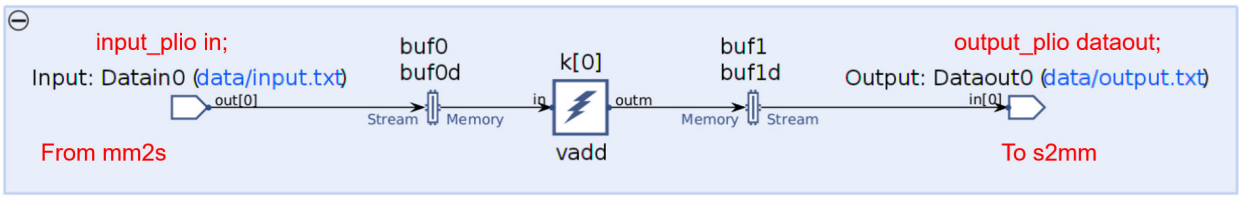
图 62：**信号线上的事件**



- 如果没有数据经过串流开关，则串流开关处于 `idle` 状态。
- 如有数据经过串流开关，则串流开关处于 `running` 状态。
- 当信号线上的所有 FIFO 都已满时，串流开关处于 `stall` 状态。
- 当数据传输恢复时，串流开关会返回 `running` 状态。

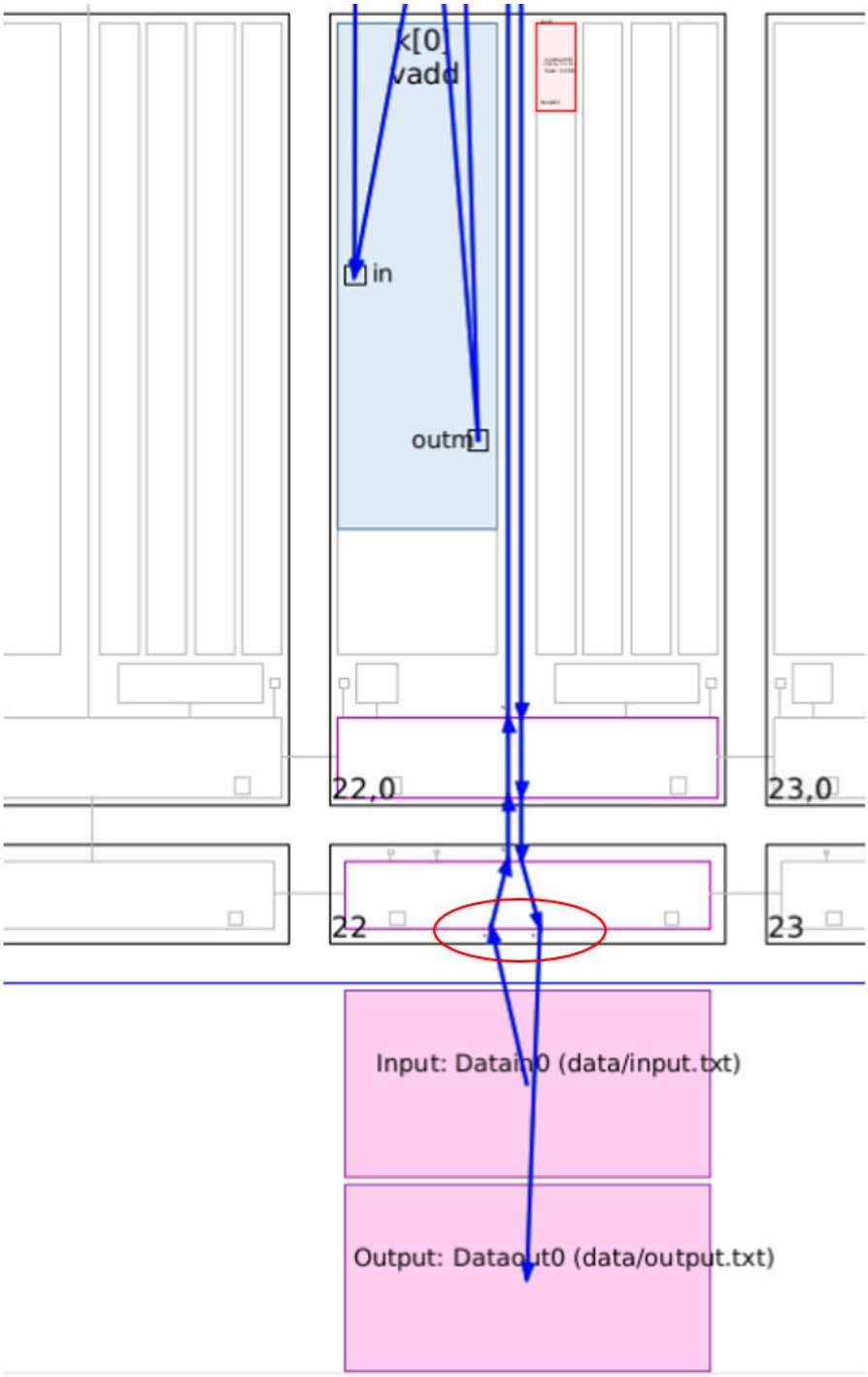
以下 graph 显示了从 `mm2s` PL 内核发送到 AI 引擎的数据示例。它还显示了 graph 将数据从 AI 引擎发送到 `s2mm` PL 内核的过程。

图 63：**graph 示例**



不同端口可经过相同的 AI 引擎到 PL 接口列，并共享接口中的性能计数器。您可检查 Vitis 分析器中的“array”（阵列）视图，查看这些端口布线经过的列。下图显示了以上示例的“array”视图，请注意，红色圆圈中的串流开关就是事件 API 正在监控的部分。

图 64：**阵列视图示例**


k[0]
vadd
in
outm
22,0
23,0
22
23
Input: Datain0 (data/input.txt)
Output: Dataout0 (data/output.txt)

注释：

- 当 graph 已完成初始化后，AI 引擎中的输入缓冲器 `buf0` 即已准备好接受来自 `mm2s` PL 内核的数据。一旦 `mm2s` PL 内核启动，它将按顺序填满连接到 `buf0` 的串流开关内的乒乓缓冲器和 FIFO。往来这些缓冲器传输的数据不依赖于 `graph.run()`。
- AI 引擎到 PL 接口的每个列都具有两个性能计数器。由于性能计数器数量有限，`event::stop_profiling()` 可用于释放性能计数器。
- 调用 graph 和剖析 API 时都存在一些开销。剖析结果可使用 `event::read_profiling()` 来读取。如果在 `event::read_profiling()` 前不停止性能计数器，那么可能产生不同的剖析结果。

重要提示！这些 API 可在 AI 引擎仿真流程、硬件仿真流程和硬件流程中使用。

剖析运行中的事件和 graph 吞吐量

赛灵思提供了 `event::io_stream_running_event_count` 枚举，用于对运行中的事件进行计数，其数量对应于通过信号线发送的样本数。运行 `event::start_profiling()` 时，性能计数器会启动。每次有数据样本穿过 AI 引擎到 PL 接口时，性能计数器就会递增。`event::read_profiling()` 读回的值是已穿过该接口发送的样本数量。

已发送和已接受的样本的计数方法

此方法可用于对发生 graph 停滞之前已发送或已接收的样本数量进行计数。以下示例可用于对 graph 停滞之前接收到来自 AI 引擎端口的样本数量进行计数：

```
event::handle handle = event::start_profiling(gr_pl.dataout,
event::io_stream_running_event_count);
if(handle==event::invalid_handle){
    printf("ERROR:Invalid handle. Only two performance counter in a AIE-PL
interface tile\n");
    return 1;
}
gr_pl.run(iterations);
sleep(2);//Wait for enough time
long long cycle_count = event::read_profiling(handle);
printf("Sample number: %d\n", cycle_count);
event::stop_profiling(handle);//Performance counter is released and cleared
```

以下示例可用于对发生 graph 停滞之前发送至 AI 引擎端口的样本数量进行计数：

```
event::handle handle = event::start_profiling(gr_pl.in,
event::io_stream_running_event_count);
if(handle==event::invalid_handle){
    printf("ERROR:Invalid handle. Only two performance counter in a AIE-PL
interface tile\n");
    return 1;
}
auto mm2s_run = mm2s(nullptr, OUTPUT_SIZE_MM2S);//After start profiling,
send data from mm2s
gr_pl.run(iterations);

sleep(2);//Wait for enough time
long long cycle_count = event::read_profiling(handle);
printf("Sample number: %d\n", cycle_count);
event::stop_profiling(handle);//Performance counter is released and cleared
```

端口吞吐量剖析

端口吞吐量定义为特定时间段内传输的样本数。运行 graph 后，可在主机代码中插入以下代码以测量端口吞吐量。为了剖析稳定状态下设计的端口吞吐量，您必须确保数据传输处于稳定状态后才能剖析端口吞吐量。

```
gr_pl.run(iterations); // The graph may also have been started during
device boot-up
usleep(100); // Wait enough time (here 100us) to be in a steady state IO
activity
int wait_time_us=20000;
event::handle handle = event::start_profiling(gr_pl.dataout,
event::io_stream_running_event_count);
if(handle==event::invalid_handle){
    printf("ERROR:Invalid handle. Only two performance counter in a AIE-PL
interface tile\n");
    return 1;
}
long long count0 = event::read_profiling(handle);
usleep(wait_time_us);
long long count1 = event::read_profiling(handle);
event::stop_profiling(handle);
long long samples = count1 - count0;
std::cout << "num runnning samples: " << samples << std::endl;
std::cout << "Throughput: " << (double)samples / wait_time_us << " MSPS "
<< std::endl;
```

赛灵思建议您在硬件中运行设计并多次迭代以确保准确性。此方法的准确性可能因硬件仿真而异。

对于 AI 引擎仿真器，此剖析方法也同样适用。在 SystemC 中您需要将 `usleep` 替换为 `wait` 函数，根据仿真时间，等待时间可能更短，因为在仿真下它运行速度更慢。例如，对于 AI 引擎仿真器，前述代码中的 `usleep` 函数可替换为以下函数调用，这样即可缩短仿真时间。

```
wait(20,SC_US);
```

graph 吞吐量剖析

graph 吞吐量可定义为每秒生成（或耗用）的平均字节数。`event::io_stream_start_to_bytes_transferred_cycles` 枚举可用于记录传输一定量的数据所耗费的周期。

执行 `event::start_profiling()` 后，`performance counter 0` 和 `performance counter 1` 这两个性能计数器即可协同工作。`performance counter 0` 在接收到第一条数据后就会开始递增计数器。`performance counter 1` 在接收到数据后会递增。当 `performance counter 1` 与 `event::start_profiling` 中指定的数据量相等时，它会生成事件以通知 `performance counter 0` 停止。`event::read_profiling()` 所读取的值是 `performance counter 0` 值。在 `performance counter 0` 停止后，计数器的值表示传输数据所耗费的周期数。

如果 `event::start_profiling` 中指定的数据量未传输完成，`performance counter 0` 就不会停止。完成指定量的数据传输后，`performance counter 0` 将停止。如想要剖析传输已知量的数据所耗费的时间，此技巧很有用。但如果传输更多数据，那么性能计数器 0 会持续计数。

警告！对于任意 graph 吞吐量剖析方法，如果迭代次数太小，graph 时延或 graph 与内核 API 调用的开销可能无法忽略。建议运行大量迭代，尽可能减小此类开销的影响。

使用 graph 输出来剖析 graph 吞吐量

以下示例演示了如何使用 graph 输出来剖析 graph 吞吐量：

```
auto s2mm_run = s2mm(out_bo, nullptr, OUTPUT_SIZE);
const int WINDOW_SIZE_in_bytes=8192;
int iterations=999;
//Third parameter is the amount of data to be transferred (in bytes).
event::handle handle = event::start_profiling(gr_pl.dataout,
event::io_stream_start_to_bytes_transferred_cycles,
WINDOW_SIZE_in_bytes*iterations);
if(handle==event::invalid_handle){
    printf("ERROR:Invalid handle. Only two performance counter in a AIE-PL
interface tile\n");
    return 1;
}
gr_pl.run(iterations);
s2mm_run.wait();//performance counter 0 stops, assumming s2mm able to
receive all data
long long cycle_count = event::read_profiling(handle);
double throughput = (double)WINDOW_SIZE_in_bytes*iterations / (cycle_count
* 1e-9); //bytes per second
event::stop_profiling(handle);//Performance counter is released and cleared
```

请注意，在以上代码中，运行会等待 `s2mm` 完成，以确保所有数据都通过 PLIO 完成传输。

在 AI 引擎仿真流程中使用 API 时，可改用 `graph.wait()`。请注意，执行 `graph.wait()` 后，API 仍将需要额外周期以将数据从窗口缓冲器传输至 PLIO。有一种解决方案是使用足够多的迭代次数，使开销尽可能小且可忽略。另一种解决方案是使用 `graph.wait(<NUM_CYCLES>)` 并运行多个周期，使其足以确保所有数据都通过 PLIO 完成传输。

使用 graph 输入来剖析 graph 吞吐量

从 PLIO 到内核的串流以及输入缓冲器的 DMA 完成配置后即可立即接收数据。当 PL 内核 `mm2s` 断言有效时，输入信号线即可开始接收数据，即使在 `graph::run` 之前也是如此。有一种 PLIO 输入剖析方法是在 `event::start_profiling()` 后断言 PL 有效。以下示例演示了如何使用 graph 输入来剖析 graph 吞吐量：

```
const int WINDOW_SIZE_in_bytes=8192;
int iterations=999;
//Third parameter is the amount of data to be transferred (in bytes).
event::handle handle = event::start_profiling(gr_pl.in,
event::io_stream_start_to_bytes_transferred_cycles,
WINDOW_SIZE_in_bytes*iterations);
if(handle==event::invalid_handle){
    printf("ERROR:Invalid handle. Only two performance counter in a AIE-PL
interface tile\n");
    return 1;
}
gr_pl.run(iterations);
auto mm2s_run = mm2s(nullptr, OUTPUT_SIZE_MM2S);//After start profiling,
send data from mm2s
gr_pl.wait();//performance counter 0 stops, assumming s2mm able to receive
all data
long long cycle_count = event::read_profiling(handle);
double throughput = (double)WINDOW_SIZE_in_bytes*iterations / (cycle_count
* 1e-9); //bytes per second
event::stop_profiling(handle);//Performance counter is released and cleared
```

如果传输的数据量未知，也可以采用此方法来估算 graph 吞吐量。例如，如果 PL 内核自由运行并且 graph 输出 AI 引擎到 PL 接口列已没有性能计数器可用，那么我们仍可通过 graph 输入来剖析 graph 吞吐量：

```
const int WINDOW_SIZE_in_bytes=8192;
int iterations=999;
//Third parameter is the amount of data to be transferred (in bytes).
event::handle handle = event::start_profiling(gr_pl.in,
event::io_stream_start_to_bytes_transferred_cycles,
WINDOW_SIZE_in_bytes*iterations);
if(handle==event::invalid_handle){
    printf("ERROR:Invalid handle. Only two performance counter in a AIE-PL
interface tile\n");
    return 1;
}
gr_pl.run(iterations);
gr_pl.wait();//performance counter 0 does not stop
//Read performance counter value immediately
//Assuming that overhead can be negligible if iteration is large enough
long long cycle_count = event::read_profiling(handle);
double throughput = (double)WINDOW_SIZE_in_bytes*iterations / (cycle_count
* 1e-9); //bytes per second
event::stop_profiling(handle);//Performance counter is released and cleared
```

graph 时延剖析

`event::io_stream_start_difference_cycles` 枚举可用于测量两个 PLIO 或 GMIO 端口之间的时延。执行 `event::start_profiling()` API 后，两个性能计数器会开始逐个周期递增，等待两条独立信号线接收其第一项数据。当第一项数据经过任一信号线后，对应的性能计数器将停止。由 `event::read_profiling()` 读回的值表示两个性能计数器之间的差值。

执行 `event::stop_profiling()` 后，性能计数器会清零并释放。

graph 时延剖析

graph 时延可定义为从接收到第一项输入数据到生成第一项输出数据之间所耗费的时间。它与 graph 运行的迭代次数无关。以下示例演示了如何在 AI 引擎仿真流程和硬件流程或硬件仿真流程中使用事件 API。

注释： `event::start_profiling()` 具有两个不同的 PLIO 参数。

在 AI 引擎仿真中：

```
event::handle handle = event::start_profiling(gr_pl.in, gr_pl.dataout,
event::event::io_stream_start_difference_cycles);
if(handle==event::invalid_handle){
    printf("ERROR:Invalid handle. Only two performance counter in a AIE-PL
interface tile\n");
    return 1;
}
gr_pl.run(iterations); //Data transfer starts after graph.run()
gr_pl.wait();
long long cycle_count = event::read_profiling(handle);
printf("Latency cycles=: %d\n", cycle_count);
event::stop_profiling(handle);//Performance counter is released and cleared
```

在硬件流程和硬件仿真流程中：

```
auto s2mm_run = s2mm(out_bo, nullptr, OUTPUT_SIZE);
event::handle handle = event::start_profiling(gr_pl.in, gr_pl.dataout,
event::event::io_stream_start_difference_cycles);
if(handle==event::invalid_handle){
    printf("ERROR:Invalid handle. Only two performance counter in a AIE-PL
interface tile\n");
    return 1;
}
gr_pl.run(iterations);
auto mm2s_run = mm2s(nullptr, OUTPUT_SIZE_MM2S); //input data transfer
starts
s2mm_run.wait();//make sure both ports have data transferred
long long cycle_count = event::read_profiling(handle);
printf("Latency cycles=: %d\n", cycle_count);
event::stop_profiling(handle);//Performance counter is released and cleared
```

注释：在 PL 内核 `mm2s` 启动后，输入数据传输就会立即启动。为了在 graph 时延剖析中避免 `graph.run()` 可能引入的任何开销，请在剖析代码中先启动 `event::start_profiling` 和 `graph.run()` 之后再启动 PL 内核 `mm2s`。

剖析两个端口之间的时延差

此方法不限于剖析相同 graph 的输入端口和输出端口之间的时延。它可用于剖析任意两个端口之间的时延。例如，它可剖析具有公用输入端口的两个输出端口之间的时延。

“AI Engine Simulation”（AI 引擎仿真）

此仿真流程中的代码示例如下所示：

```
event::handle handle = event::start_profiling(gr_pl.dataout,
gr_pl.dataout2, event::event::io_stream_start_difference_cycles);
if(handle==event::invalid_handle){
    printf("ERROR:Invalid handle. Only two performance counter in a AIE-PL
interface tile\n");
    return 1;
}
gr_pl.run(iterations);
gr_pl.wait();
long long cycle_count = event::read_profiling(handle);
printf("Latency cycles=: %d\n", cycle_count);
event::stop_profiling(handle);//Performance counter is released and cleared
```

“Hardware Emulation and Hardware”（硬件仿真和硬件）

这些流程中的代码示例如下所示：

```
auto s2mm_run = s2mm(out_bo, nullptr, OUTPUT_SIZE);event::handle handle =
event::start_profiling(gr_pl.dataout, gr_pl.dataout2,
event::event::io_stream_start_difference_cycles);
if(handle==event::invalid_handle){
    printf("ERROR:Invalid handle. Only two performance counter in a AIE-PL
interface tile\n");
    return 1;
}
gr_pl.run(iterations);
auto mm2s_run = mm2s(nullptr, OUTPUT_SIZE_MM2S);
s2mm_run.wait();//make sure both ports have data transferred
long long cycle_count = event::read_profiling(handle);
printf("Latency cycles=: %d\n", cycle_count);
event::stop_profiling(handle);//Performance counter is released and cleared
```

其中，正值表示数据到达 `gr_pl.dataout2` 的时间晚于 `gr_pl.dataout`，负值则表示数据到达 `gr_pl.dataout2` 的时间早于 `gr_pl.dataout`。

graph 带宽剖析

`event::io_total_stream_running_to_idle_cycles` 剖析事件可用于跟踪已剖析的 AI 引擎到 PL 接口上发生的运行事件和停滞事件。这意味着，它将会跟踪接口处于活动状态（数据流经接口）周期数和接口处于停滞状态的周期数。它并不会跟踪接口处于空闲状态的周期数。

执行 `event::start_profiling()` 后，性能计数器会等待运行数据开始，如果串流处于空闲状态，那么它将暂停。性能计数器暂停后，会随数据流一起恢复。执行 `event::stop_profiling()` 后，性能计数器将清零并释放。此 API 会报告 AI 引擎和 PL 内核利用可用带宽的情况。它并非用于测量端口带宽的最佳利用率。

使用输入端口剖析 graph 带宽

graph 带宽可定义为 graph 可接受数据的时间的百分比。

以下提供了通过 graph 输入端口测量 graph 带宽的代码示例：

```
const int WINDOW_SIZE_in_bytes=8192;
int iterations=999;
event::handle handle = event::start_profiling(gr_pl.in,
event::io_total_stream_running_to_idle_cycles);
if(handle==event::invalid_handle){
    printf("ERROR:Invalid handle. Only two performance counter in a AIE-PL
interface tile\n");
    return 1;
}
auto mm2s_run = mm2s(nullptr, OUTPUT_SIZE_MM2S);
gr_pl.run(iterations);
gr_pl.wait();
long long cycle_count = event::read_profiling(handle);
double bandwidth = (double) (WINDOW_SIZE_in_bytes*iterations/4) /
cycle_count;
event::stop_profiling(handle);//Performance counter is released and cleared
```

其中，运行总周期数可根据传输的字节数来计算。如果每个周期传输 4 字节，那么运行总周期数即为 `WINDOW_SIZE_in_bytes*iterations/4`。`event::read_profiling()` 会从性能计数器读取运行中和已停滞的总周期数。

如果剖析的带宽为 1，这表示 graph 运行速度高于 PL 内核 `mm2s`，输入端口尚未停滞。

如果剖析的带宽小于 1，这表示 PL 内核 `mm2s` 发送数据的速度可能高于 graph 或 PL 内核 `s2mm` 的接收速度。您可能需要评估带宽下降是否是由 graph 或 PL 内核 `s2mm` 导致的。

使用输出端口剖析 graph 带宽

graph 带宽可定义为 graph 可发送数据的时间的百分比。如果剖析的带宽为 1，这表示 PL 内核 `s2mm` 并未阻塞 graph。如果剖析的带宽小于 1，这表示由于反压，`s2mm` 给 graph 造成了一定比例的阻塞。以下提供了通过 graph 输出端口剖析 graph 带宽的代码示例：

```
const int WINDOW_SIZE_in_bytes=8192;
int iterations=999;
event::handle handle = event::start_profiling(gr_pl.dataout,
event::io_total_stream_running_to_idle_cycles);
if(handle==event::invalid_handle){
    printf("ERROR:Invalid handle. Only two performance counter in a AIE-PL
interface tile\n");
    return 1;
}
gr_pl.run(iterations);
gr_pl.wait();
long long cycle_count = event::read_profiling(handle);
double bandwidth = (double) (WINDOW_SIZE_in_bytes*iterations/4) /
cycle_count;
event::stop_profiling(handle);//Performance counter is released and cleared
```

运行时事件 API 性能计数器使用汇总

运行时事件 API 在 AI 引擎到 PL 接口拼块和 AI 引擎到 NoC 接口拼块中使用性能计数器。在每一列接口拼块中均有 2 个性能计数器。本节列出了每个运行时事件 API 所使用的性能计数器数量。如果所使用的性能计数器总数超过接口拼块的列中可用的性能计数器数量，那么无法获取性能计数器的 API 将运行失败，并在 AI 引擎仿真器中显示以下错误消息。

```
[AIE WARNING]: Unable to request resources. RscType: 0
ERROR: event::start_profiling: Failed to request performance counter
resources.
```

对于硬件仿真或硬件流程，使用的错误消息如下。

```
[XRT] ERROR: ERROR: event::start_profiling: Failed to request performance
counter resources.: Resource temporarily unavailable
```

表 65：**运行时事件 API 性能计数器使用情况**

运行时事件枚举	性能计数器的数量
`event::io_total_stream_running_to_idle_cycles`	1
`event::io_stream_start_to_bytes_transferred_cycles`	2
`event::io_stream_start_difference_cycles`	1 个用于输入端口，1 个用于输出端口
`event::io_stream_running_event_count`	1

注释：完成 `event::stop_profiling` 后会释放性能计数器。性能计数器释放后，运行时事件 API 即可获取同样的性能计数器。

注释：如有多个 graph 端口映射到同一个接口拼块，如果在这些端口上运行时事件 API，那么这些 API 将争用相同接口拼块内的性能计数器。

硬件中的事件追踪

AI 引擎事件追踪工具可以提供设计操作和性能方面的深度调查。为支持该功能，需执行多项设置以捕获运行时的追踪数据。在硬件中，您必须在编译 AI 引擎 graph 应用时准备设计，以确保 `libadf.a` 支持在运行时捕获追踪数据。硬件构建中的事件追踪以使用 `--event-trace` 选项的 `aiecompiler` 命令开始。该选项用于设置硬件器件以捕获 AI 引擎运行时追踪数据。

事件追踪流程包含三个部分，如下所示。

- 事件追踪构建流程
- 在硬件中运行设计并在运行时捕获追踪数据
- 使用 Vitis 分析器查看和分析追踪数据

事件追踪构建流程

事件追踪构建流程如下所示。

1. 使用 `--event-trace` 和其它相应的标志来编译 graph。

   用于事件追踪的 AI 引擎编译器命令示例如下所示：

   ```
   aiecompiler --verbose --pl-freq=100 --workdir=./myWork \
   --event-trace-port=gmio --event-trace=runtime \
   --num-trace-streams=1 --xlopt=0 --include="./" \
   --include="./src" --include="./src/kernels" --include="./data" \
   ./src/graph.cpp
   ```

   注释：

   - 前述示例演示了使用 `--event-trace=runtime` 配置编译设计的操作。该选项可用于配置 AI 引擎在运行时期间捕获的事件的类型。
   - `--event-trace-port=gmio` 选项使用 GMIO 来捕获事件追踪数据。该选项使用 AI 引擎到 NoC 事件追踪路径。替代方法是使用 PLIO，它使用 AI 引擎到 PL 路径来捕获事件追踪数据。此方法使用编程逻辑资源捕获从 AI 引擎到 DDR 的数据。

     建议：赛灵思建议使用 gmio 选项作为 event-trace-port（事件追踪端口）配置。这样即可避免使用编程逻辑资源，并避免由于 PL 资源使用而导致的时序错误。

     - `gmio` 是 AI 引擎 NoC 事件路径。GMIO 是默认事件追踪端口配置。
     - `plio` 是 AI 引擎到 PL 事件追踪路径。事件数据会被传输到 PL，并存储在 BRAM 和 URAM 资源内。此附加 PL 资源可能在硬件设计侧引发时序错误。

2. 使用 Vitis 编译器来编译和链接设计。

编译 AI 引擎 graph 应用后，必须构建系统的其它要素，如 第 8 章：使用 Vitis 工具流程来集成应用 中所述。在 `libadf.a` 文件中从 AI 引擎编译器启用 `--event-trace` 后，由 Vitis 编译器生成的系统硬件会包含为 PS 应用编译的 ELF 文件、为 AI 引擎处理器编译的 ELF 文件以及 PL 的 XCLBIN 文件。这些都是在硬件上运行系统所需的要素。

3. 完成链接并创建器件二进制文件后，运行 Vitis 编译器 `--package` 步骤即可创建启动器件所需的 `sd_card` 文件夹和文件，如 封装 中所述。此步骤会对为系统构建 `BOOT.BIN` 文件所需的所有一切进行封装。为器件封装启动文件时，还必须指定 `--package.defer_aie_run` 以伴随 ELF 文件一起加载 AI 引擎应用，但等待至 `graph.run` 发出指令后再运行该应用，如《AI 引擎内核与 Graph 编程指南》(UG1079) 的 graph 执行控制中所述。

   `aiecompiler --event-trace` 选项具有一项功能特性，您可使用 `runtime` 实参编译自己的设计以捕获数据。您可使用该选项来编译 AI 引擎 graph，对其进行设置用于事件追踪，并指定要在运行时捕获的剖析数据的类型：`functions`、`functions_partial_stalls` 和 `functions_all_stalls`。因此，您无需重新编译设计以捕获不同类型的数据。使用该功能特性可以减少重新编译 graph 和重新封装设计的需求。

表 66：**预定义的事件追踪级别中支持的事件**

事件类型	预定义的事件追踪级别		
	`functions`	`functions_partial_stalls`	`functions_all_stalls`
函数调用/返回	已捕获	已捕获	已捕获
串流停滞	不适用	已捕获	已捕获
级联停滞	不适用	已捕获	已捕获
锁定停滞	不适用	已捕获	已捕获
存储器停滞	不适用	不适用	已捕获

- 函数调用/返回：调用和返回内核函数时生成的事件。
- 串流停滞：当核停滞时生成的事件。停滞原因可能是输入处无数据或者串流输出处存在来自核的反压。
- 级联停滞：当核停滞时生成的事件。停滞原因可能是输入处无数据或者串流输出处存在来自核的反压。
- 锁定停滞：如果由于当前正在获取锁定而导致核停滞，则会生成此类事件。
- 存储器停滞：如果由于存储器冲突导致核停滞，则会生成此类事件。

在硬件中运行设计并在运行时捕获追踪数据

赛灵思的 Xilinx Runtime (XRT) 和赛灵思的 Xilinx Software Debugger (XSDB) 这两种方式均可用于在硬件中的 Arm® 处理器上运行设计并在运行时捕获追踪数据。XRT 在 Linux 平台上受支持，而 XSDB 则在裸机和 Linux 平台上均受支持。下表突出显示了两个流程中支持的功能特性。

表 67：**XRT 对比 XSDB**

	裸机	PetaLinux	每条追踪串流的带宽（位数/s）	易用性
XSDB	支持	支持	PL 时钟速率 * W（其中 W=32、64 或 128）	此流程涉及手动设置环境、获取脚本源码和运行事件追踪流。（欲知详情，请参阅以下 XSDB 章节）
XRT	不支持	支持	PL 时钟速率 * W（其中 W=32、64 或 128）	此流程较之 XSDB 更为简单，因为可在 `xrt.ini` 文件中设置事件追踪。

注释：该表引用 PLIO 数据传输。

XSDB 流程

XSDB 流程如下所示：

1. 如下步骤中所述，设置 `xsdb` 以连接到器件硬件。

   运行应用时，调试和剖析 IP 会将追踪数据存储在 DDR 存储器内。要捕获此数据并对其进行求值，必须使用 `xsdb` 连接到硬件器件。此命令通常用于执行器件编程和调试裸机应用。通过 JTAG 将您的系统连接到硬件平台或器件、在命令 shell 中启动 `xsdb` 命令，然后运行以下命令序列：

   ```
   xsdb% connect
   xsdb% ta
   xsdb% ta 1
   xsdb% source $::env(XILINX_VITIS)/scripts/vitis/util/aie_trace.tcl
   xsdb% aietrace start -graphs mygraph -work-dir ./Work -link-summary
   $PROJECT/xsa.link_summary -base-address 0x900000000 -depth 0x800000 -
   tile-based-aie-tile-metrics "all:functions:memory_stalls; {4,1}:
   {6,2}:functions_all_stalls"

   # Execute the PS host application (.elf) on Linux
   ## After the application completes processing.
   xsdb% aietrace stop
   ```

   其中：

   - `connect`：启动 `hw_server` 并将 `xsdb` 连接到器件。
   - `source $::env(XILINX_VITIS)/scripts/vitis/util/aie_trace.tcl`：使用 source 命令运行 Tcl trace 命令以设置 `xsdb` 环境。
   - `aietrace start -graphs mygraph -link-summary PROJECT/xsa.link_summary -base-address 0x900000000 -depth 0x800000 -tile-based-aie-tile-metrics "all:functions:memory_stalls; {4,1}:{6,2}:functions_all_stalls"`：初始化 DPA IP，开始捕获追踪数据。`-graphs` 会指定一个或多个 graph 以收集事件追踪数据。如果倾向于处理特定拼块，请指定 `-tiles` 和拼块列表，以收集追踪数据。`-tile-based-aie-tile-metrics` 指定要在其中捕获 `memory_stalls` 或 `functions_all_stalls` 事件追踪级别的拼块。值 `-base-address 0x900000000 -depth 0x800000` 用于指定将追踪数据写入 AI 引擎的起始地址和要存储的数据量。

     重要提示！ `-base-address 0x900000000` 中使用的 DDR 存储器地址必须为高位地址，以限制 `xilinx_vck190_base_202220_1` 平台上的操作系统或应用发生存储器冲突的几率。对于定制平台，请确保您已知当前使用的 DDR 存储器量并制定相应计划。

   - `aietrace stop`：指令 DPA IP 从 DDR 存储器卸载追踪事件数据。此命令必须等待至应用完成后才有效。数据写入当前工作目录中的 `event_trace<N>.txt` 文件，`xsdb` 同样是在此目录中启动的。此外还会创建 `aie_trace_profile.run_summary` 文件。在 Vitis 分析器中可打开此文件，如 在 Vitis 分析器中查看运行汇总 中所述。

     提示： 如果再次运行 graph 时不移除 `event_trace<N>.txt`，那么旧文件将被新的运行结果覆盖。

2. 在硬件上运行设计以追踪硬件事件。
3. 卸载捕获的追踪数据。
4. 启动 Vitis 分析器以使用此命令导入和分析数据。

   ```
   vitis_analyzer aie_trace_profile.run_summary
   ```

表 68：**XSDB 追踪选项**

选项	描述
start	启动事件追踪
stop	停止事件追踪
-work-dir <Work Directory>	指定工作目录
-graphs <Graph Name>	该选项允许您指定将受事件追踪影响的一个或多个 graph。
-base-address <address>	表示 DDR 中的起始地址，将存储来自该地址的追踪数据。
-depth <size>	表示将在 DDR 中使用的存储器的长度。
-tile_based_aie_tile_metrics	用于设置要为指定拼块存储的 AI 引擎事件。
-graph_based_aie_tile_metrics	用于设置要为指定内核/graph 存储的 AI 引擎事件。
-tile_based_interface_tile_metrics	用于设置要为指定拼块存储的接口拼块事件。

对于指定的事件，每个选项实质上都是一个组合字符串，该字符串由所有设置组成，各项设置以分号分隔。例如：

```
# Sets event trace to "functions_all_stall:memory_stalls" for the tile
range column 4 to 6 and row 1 to 2, and overides to "functions" for tile
(4,1)
-tile_based_aie_metrics "{4,1}:{6,2}:functions_all_stalls:memory_stalls;
{4,1}:functions"

# Sets event trace to "functions" for kernel k1 in graph myGraph
-graph_based_aie_tile_metrics "myGraph:k1:functions"
```

XRT 流程

XRT 流程如下所示：

1. 将生成的 sd_card.img 烧写到物理 SD 卡上。
2. 按本节中所述方式，在 sd_card 文件夹中创建 xrt.ini 文件以启用 xrt 流程。

   xrt.ini 文件示例如下所示。

   ```
   # Main switch to turn on aie trace
   [Debug]
   aie_trace = true
   # Continuous trace knobs
   [AIE_trace_settings]
   reuse_buffer = true
   periodic_offload = true
   # Time to wait between trace reads
   buffer_offload_interval_us = 100
   # Total amount of device memory shared between trace streams
   buffer_size = 16M
   # granularity
   graph_based_aie_tile_metrics = all:all:functions
   ```

3. 在硬件上运行设计以追踪硬件事件。
4. 将捕获的追踪数据从 sd_card 文件夹复制到设计内与设计的 Work 目录相同的层次。追踪数据的生成位置与 SD 卡上主机应用所在位置相同。这些文件为 xrt.run_summary、aie_event_runtime_config.json 和 aie_trace_N.txt。

5. 使用 Vitis 分析器，通过此命令导入和分析数据。

```
vitis_analyzer xrt.run_summary
```

注释： .ini 文件中的 aie_trace_* 设置将在后续版本中弃用。建议使用 [AIE_trace_settings] 部分中的等效设置，如下所示。

“XRT.INI 规范”

```
[Debug]
aie_trace = true

# Section for AIE trace settings
[AIE_trace_settings]

# Size of AIE trace buffer in DDR (Format: <Integer>[K|k|M|m|G|g]; Default:
1M)
buffer_size = 100M

# Graph/Kernel name
graph_based_aie_tile_metrics = <graph name|all>:all:<off|functions|
functions_partial_stalls|functions_all_stalls>[:memory_stalls|stream_stalls|
cascasde_stalls|lock_stalls>]

# AI Engine Tiles
# Single or all tiles
tile_based_aie_tile_metrics = <{<column>,<row>}|all>:<off|functions|
functions_partial_stalls|functions_all_stalls>[:<memory_stalls|
stream_stalls|cascasde_stalls|lock_stalls>]

# Range of tiles
tile_based_aie_tile_metrics = {<mincolumn,<minrow>}:
{<maxcolumn>,<maxrow>}:<off|functions|functions_partial_stalls|
functions_all_stalls>[:<memory_stalls|stream_stalls|cascasde_stalls|
lock_stalls>]
```

表 69：**XRT 追踪选项**

选项	描述
aie_trace=true	在应用执行期间启用 AI 引擎事件追踪。
trace_buffer_size=100M	在 DDR 存储器内设置事件追踪缓冲器的大小。
reuse_buffer = true	在 DDR 存储器内启用事件追踪缓冲器的复用。启用该选项时，会将 DDR 追踪缓冲器作为圆形缓冲器来处理，并从 XRT 持续卸载追踪数据。该选项仅适用于使用 PLIO 追踪捕获的事件追踪数据。默认设为 false。
periodic_offload = true / false	在应用运行时，启用定期将追踪数据从 DDR 到卸载 XRT 的功能。默认选项设为 true。该选项与“reuse_buffer”选项组合使用即可启用连续卸载追踪数据，同时可避免追踪缓冲器存储器空间不足。该选项仅适用于使用 PLIO 追踪捕获的事件追踪数据。
offload_interval_us = 10	指定将追踪事件数据从 DDR 存储器中的追踪缓冲器读取到 XRT 中的缓冲器的频率（以毫秒为单位）。仅当 periodic_offload=true 时，该选项才有效。该选项仅适用于使用 PLIO 追踪捕获的事件追踪数据。默认设为 100
file_dump_interval_s = 3	指定将追踪事件数据从 XRT 中的追踪缓冲器读取到 SD 卡中的事件追踪文件的频率（以秒为单位）。仅当 periodic_offload=true 时，该选项才有效。默认设为 5

表 69：**XRT 追踪选项** (续)

选项	描述
`graph_based_aie_tile_metrics = <graph name|all>:all:<off|functions|functions_partial_stalls|functions_all_stalls>`	该选项用于配置 AI 引擎事件追踪指标，该指标将应用于所有 graph 或特定 graph 中的所有内核。该选项会被应用于拼块，即使在该拼块上有多个内核正在运行也是如此。
`tile_based_aie_tile_metrics = <{<column>,<row>}|all>:<off|functions|functions_partial_stalls|functions_all_stalls>`	该选项用于配置 AI 引擎事件追踪指标，该指标将应用于单个拼块或所有拼块。
`tile_based_aie_tile_metrics = {<mincolumn,<minrow>}:{<maxcolumn>,<maxrow>}:<off|functions|functions_partial_stalls|functions_all_stalls>`	该选项用于配置 AI 引擎事件追踪指标，该指标将应用于某一范围内的所有拼块。

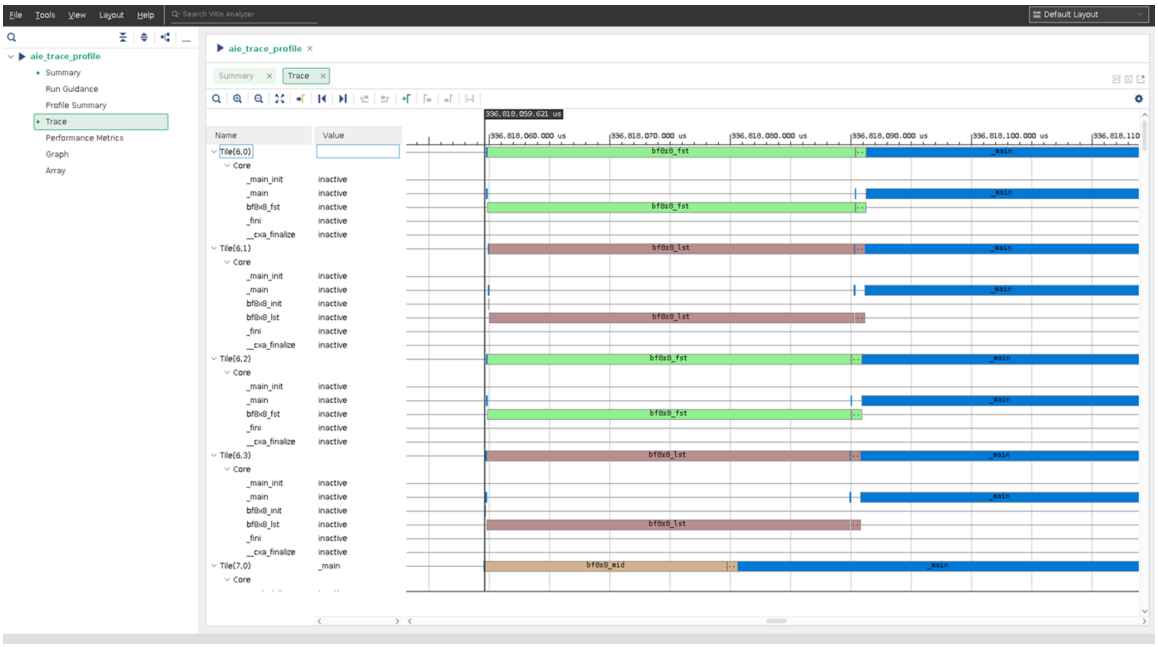
“XRT.INI 示例”

```
# Example 1 : trace function events for all tiles used in the AI Engine
array
[AIE_trace_settings]
tile_based_aie_tile_metrics = all:functions

# Example 2 : trace function events in all the kernels in all graphs
(Similar to the example above)
[AIE_trace_settings]
graph_based_aie_tile_metrics = all:all:functions

# Example 3 : trace all function stalls events on all used tiles
[AIE_trace_settings]
tile_based_aie_tile_metrics = all:functions:functions_all_stalls

# Example 4 : trace function events within the bounding box of tiles or on
specific tiles
[AIE_trace_settings]
tile_based_aie_tile_metrics = {4,1}:{6,2}:functions_all_stalls;
{4,1}:functions

# Example 5 : trace events specified by graph name
[AIE_trace_settings]
graph_based_aie_tile_metrics = chain_0:all:functions;
chain_1:all:functions_all_stalls; chain_2:all:functions_partial_stall

# Example 6 : trace functions_all_stalls events all kernels in all graphs
and trace function events occuring in all kernels  in the graph chain_0
[AIE_trace_settings]
graph_based_aie_tile_metrics = all:all:functions_all_stalls;
chain_0:all:functions

# Example 7 : Trace graphs and all kernels function events and override
tile 4,1 with functions all stalls
[AIE_trace_settings]
graph_based_aie_tile_metrics = all:all:functions
tile_based_aie_tile_metrics = {4,1}:functions_all_stalls

# Example 8 : Trace all graphs and all kernels function events and turn off
trace in tile 4,1
[AIE_trace_settings]
graph_based_aie_tile_metrics = all:all:functions
tile_based_aie_tile_metrics = {4,1}:off
```

使用 Vitis 分析器查看和分析追踪数据

必须使用 Vitis 分析器才能查看和分析追踪数据。使用 XRT 或 XSDB 捕获追踪数据后，您应已获得所需的所有数据，可在 Vitis 分析器中打开“Event Trace”（事件追踪）视图进行查看。

图 65：**Vitis 分析器中的事件追踪**



使用 Vitis 分析器打开运行汇总文件以查看事件追踪数据。以下提供了 XSDB 流程示例：

```
vitis_analyzer ./aie_trace_profile.run_summary
```

以下提供了 XRT 流程示例：

```
vitis_analyzer ./xrt.run_summary
```

限制

- 由于资源有限，在事件追踪中可看到溢出。请参阅 使用多个事件追踪串流，了解如何配置追踪串流数量以尽可能减少溢出问题。
- 如需获取详细的事件追踪信息，必须在编译中指定 `--xlopt=0` 选项。如果省略，则使用默认设置 `--xlopt=1`，这可能导致函数发生内联，从而限制调试功能。
- 调用 `graph.end()` 时，需要主机代码以确保正确完成 XRT 流程。

使用多个事件追踪串流

随着 AI 引擎设计增大，运行设计的同时对生成的事件进行追踪有助于识别性能瓶颈和理解设计中 AI 引擎的总体工作状况。当然，设计越大，生成的事件也会越来越多，导致使用的追踪 IP 所记录的事件出现瓶颈。为了有效捕获所有这些数据，您应考虑例化多个事件追踪串流。这些串流将分散来自 AI 引擎的事件数据，使其能够及时准确存储这些数据。

要增加设计中的追踪串流，请使用 `aiecompiler --num-trace-streams` 选项，该选项可取 1 到 16 之间的值。下表提供了有关根据设计大小来选择所使用的追踪串流数量的指导信息。

表 70：**事件追踪串流数量方法论**

AI 引擎数量	建议的串流数量
小于 10	1
介于 10 到 20 之间	2
介于 20 到 40 之间	4
介于 40 到 80 之间	8
大于 80	16

注释：

1. 鉴于资源使用对 PL 和 DMA 通道资源产生的影响，建议最多使用 16 条事件串流。

更改 AI 引擎编译器选项之后，请使用 Vitis 编译器搭配 `config` 文件来重新编译和重新链接 XCLBIN 文件和 `libadf.a`，如 系统链接 中所述。

```
v++ -l --config system.cfg ...
```

在硬件中对事件追踪进行故障排除

表 71：**在硬件中对事件追踪进行故障排除**

问题	解决办法
在某些串流中，追踪包被丢弃。（这可通过观察 Vitis 分析器 GUI 中的事件追踪串流上的黑条来确定。）	选项 1：在构建时间，使用以下选项增大 AI 引擎传出的追踪串流数量。 `aiecompiler --num-trace-streams=<N>` 选项 2：如果使用 PLIO 事件追踪串流，请使用 `aiecompiler` 标记 `--trace-plio-width=128`，它使用的 PLIO 串流位宽为 128 位。这样可确保 PLIO 串流位宽与 AI 引擎数据速率相匹配。 选项 3：减少每条追踪串流连接的拼块数。您可配置要应用于特定 graph 或拼块的 AI 引擎事件追踪指标。如需了解有关这些选项的更多详细信息，请参阅 表 68：XSDB 追踪选项 和 表 69：XRT 追踪选项。 选项 4：`aiecompiler` 会以水平方式为每个拼块分配追踪串流。为确保最优追踪信号线，建议使用位置约束来确保 AI 引擎尽可能垂直布局，使所有活动状态的 AI 引擎都均等分布到各追踪串流。
在追踪中不显示内核函数名称。	可能原因是该内核是由编译器或者通过属性直接插入的内联内核。要禁用内核内联，请应用 `--xlopt=0` 来编译设计，或者指定含 `__attribute__((noinline))` 属性的内核函数。
内核启动时间偏移超过 100 个周期。	将 `--broadcast-enable-core=true` 选项应用于编译器，以确保设计在若干个时钟周期范围内启动所有内核。

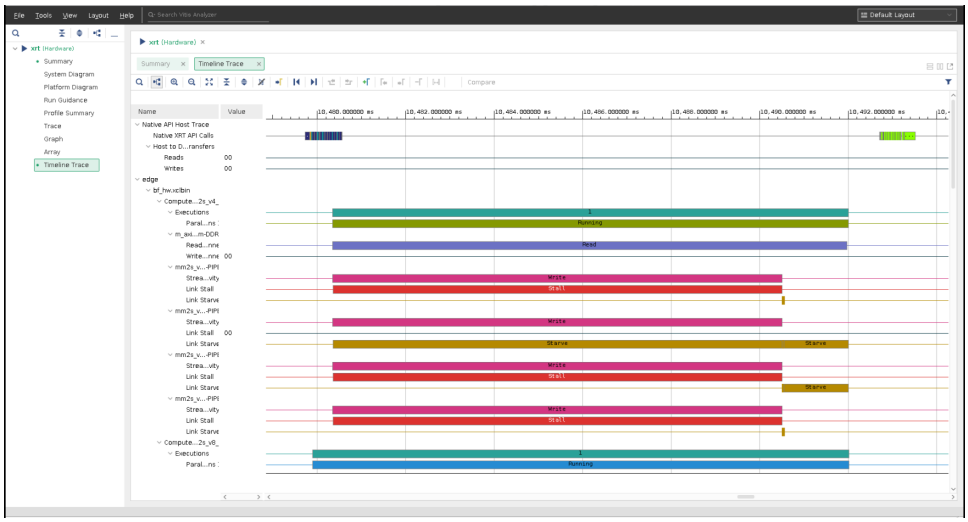
表 71：**在硬件中对事件追踪进行故障排除** (续)

问题	解决办法

工具发出警告消息，指示追踪缓冲器已满。	增大事件追踪缓冲器大小。
	对于 XSDB 流程，在 AI 引擎追踪启动命令中，使用 `-depth` 选项指定更大的值。
	`%xsdb aietrace start -graphs dut -config-level functions_all_stalls -work-dir ./Work -link-summary ./bf_hw.xsa.xclbin.link_summary -base-address 0x900000000 -depth 0x8000000`
	对于 XRT 流程，更新 `xrt.ini` 文件和 `aie_trace_buffer_size` 行。
	`[Debug]`
	`aie_trace=true`
	`aie_trace_buffer_size=100M`
	`aie_trace_metrics = functions_all_stalls`
	注释：追踪缓冲器已满时，XSDB 和 XRT 会发出以下警告消息。
	警告消息：
	`AI 引擎 Trace Buffer size is full, Device trace could be incomplete.`

表 71：**在硬件中对事件追踪进行故障排除** (续)

问题	解决办法

需要确定 PL 与 AI 引擎之间是否正在按期望方式发送或接收数据。	在 PL 内核及其 AXI4 存储器映射主接口上添加监控器。
	`v++ -l --profile_kernel <data:[kernel_name|all]:[compute_unit_name|all]:[interface_name|all](:[counters|all])><[stall|exec]:[kernel_name|all]:[compute_unit_name|all](:[counters|all])>`
	例如，要监控每个内核主接口，请添加：
	`v++ -l --profile_kernel data:all:all:all`
	在硬件上运行时，请在 `xrt.ini` 文件中添加以下行。
	`[Debug]`
	`profile=true`
	`native_xrt_trace=true`
	`data_transfer_trace=coarse`
	在硬件上运行应用后，就会在 sd_card 上生成 `device_trace_0.csv`、`native_trace.csv`、`summary.csv` 和 `xrt.run_summary` 文件。将这些文件复制到您的工程所在位置，与工程的 Work 目录位于同一级。发出 `vitis_analyzer xrt.run_summary` 命令，选择时间线“Trace”视图，以检验 PL 追踪。
	 

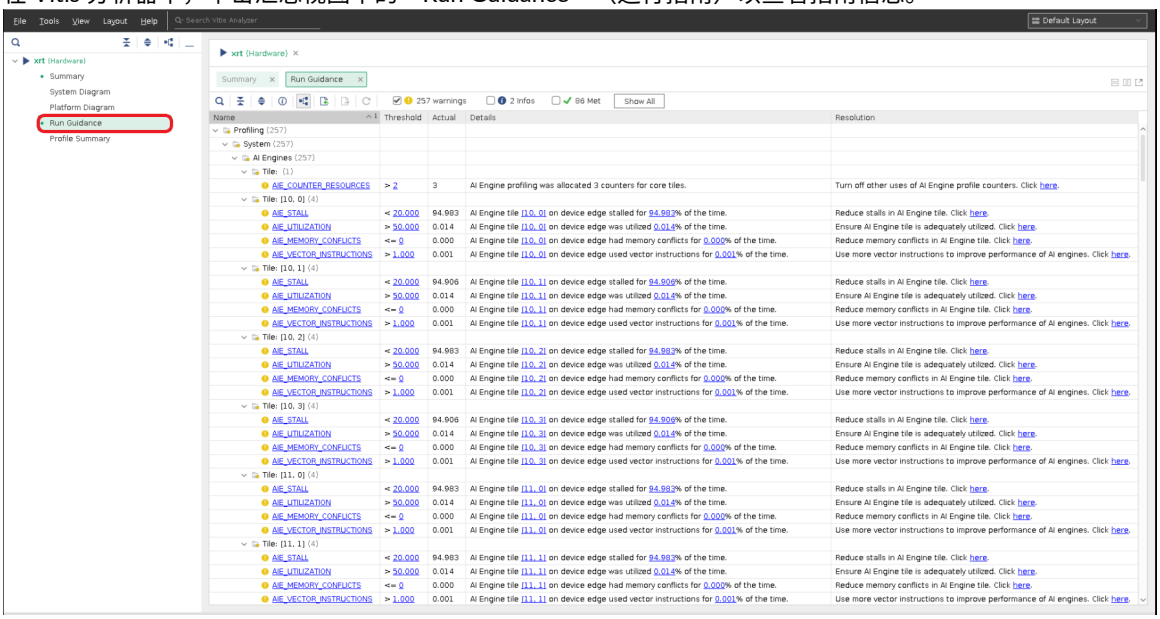
使用 Vitis 分析器查看指南

查看设计指南有助于评估设计性能，并突出显示设计中建议执行性能调优的区域。在 Vitis 分析器中，您可以查看工具所生成的设计指南，如下所示：

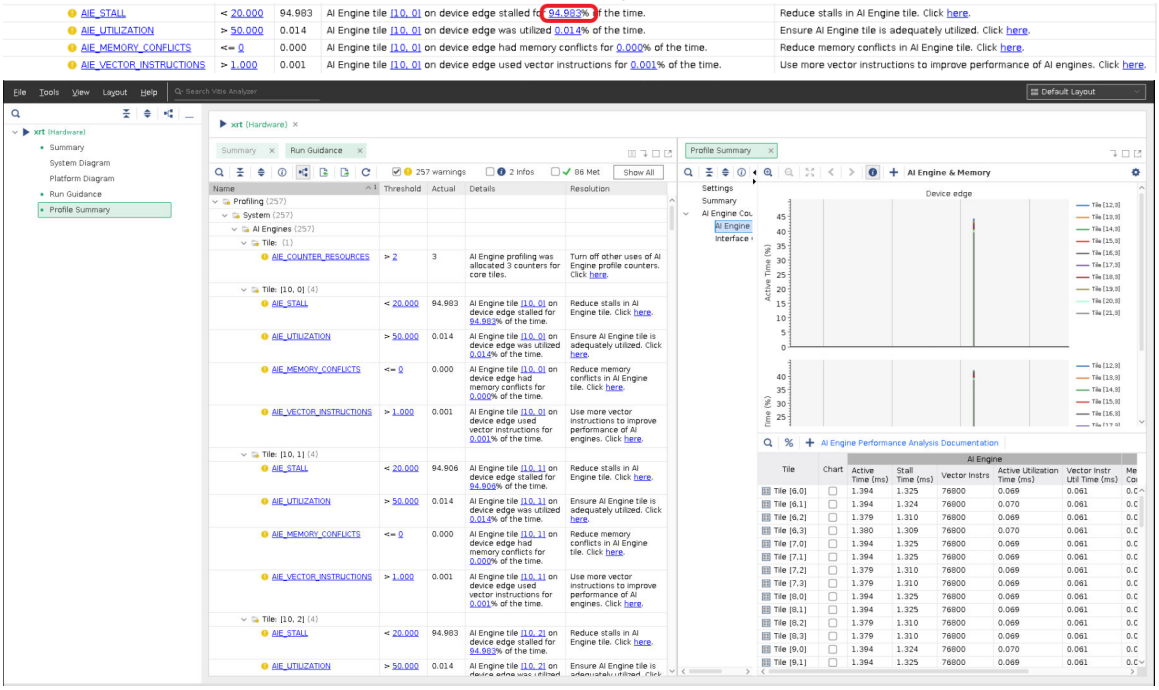
1. 使用以下命令打开运行汇总文件 `xrt.run_summary`：

```
vitis_analyzer xrt.run_summary
```

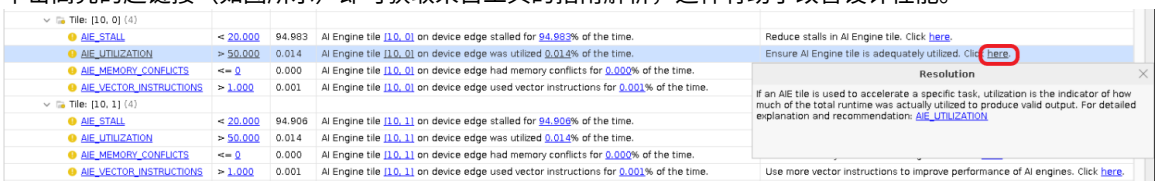
2. 在 Vitis 分析器中，单击汇总视图中的“Run Guidance”（运行指南）以查看指南信息。



3. “Run Guidance”（运行指南）中提供了建议的预定义阈值和来自带复查设计的实际结果。
4. 单击指南名称即可获取有关指南规则的更详细的描述。例如，“AIE_STALL” 的描述为 `The rule checks the stall percentage of AI Engine tiles`（此规则用于检查 AI 引擎拼块的停滞百分比）。
5. 要调整预定义的值，只需单击显示的蓝色预定义值并更新首选值即可。这样即可筛选掉设计的指南视图中的规则。
6. 单击指南规则的百分比值可以打开设计的“Profile Summary”（剖析汇总）。



7. 单击高亮的超链接（如图所示）即可获取来自工具的指南解析，这样有助于改善设计性能。



PS 主机应用编程

在《AI 引擎内核与 Graph 编程指南》(UG1079) 的创建数据流 graph（包含内核）中，主要围绕非常简单的 AI 引擎 graph 应用展开讨论。其中通过顶层应用来对 graph 进行了初始化、运行和终止。但对于实际 AI 引擎 graph 应用，主机代码所做的远不止于这些简单的任务。Cortex®-A72 上运行的顶层 PS 应用能够控制 graph 和 PL 内核：管理 graph 的数据输入、处理 graph 的数据输出以及控制搭配 graph 工作的任何 PL 内核。

此外，AI 引擎 graph 应用可在 Linux 操作系统或裸机系统上运行。这两种系统中的编程要求存在显著差异，如下列主题中所述。赛灵思提供的驱动程序可供主机程序中的 API 调用用于基于操作系统来控制 graph 和 PL 内核。在 Linux 中，这是由 `libadf_api_xrt` 库提供的，在裸机中，AI 引擎内核是使用 graph API 来控制的，PL 内核则是使用 `libUIO` 驱动程序调用来控制的。

防止执行多个 graph

如果您的 graph 是在基于 PS 的主机应用中实现的，那么您必须为自己的 `graph.cpp` 代码定义有条件的编译指示 (`#ifdef`)，以确保 graph 仅初始化一次或者仅运行一次。以下代码示例是《AI 引擎内核与 Graph 编程指南》(UG1079) 的创建数据流 graph（包含内核）中定义的简单应用，并具有额外的守卫宏 `__AIESIM__` 和 `__X86SIM__`。

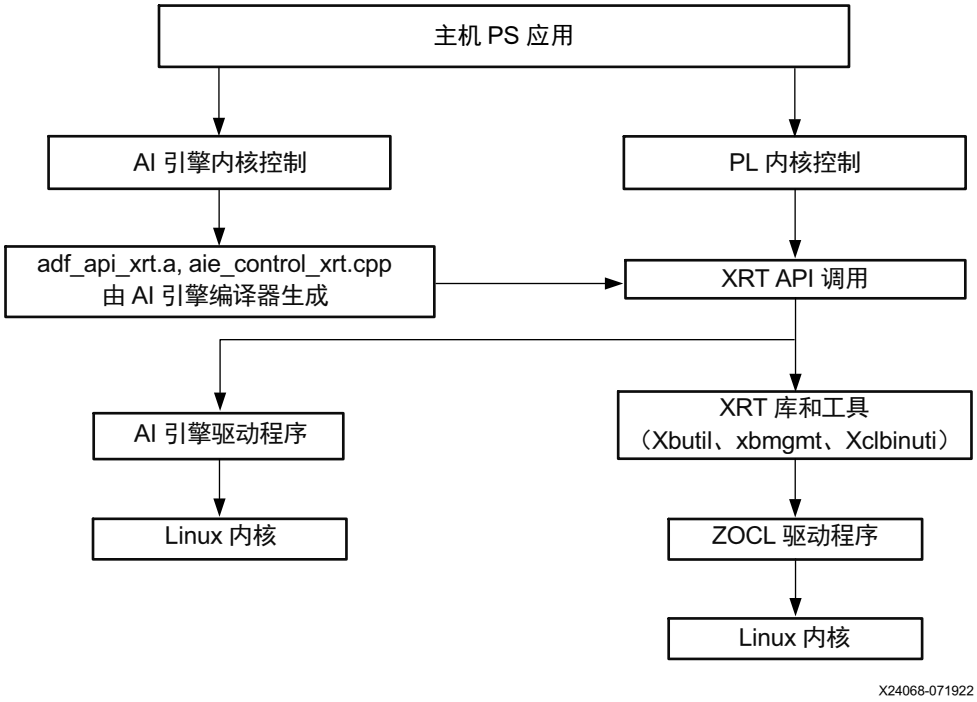
```
#include "project.h"

simpleGraph mygraph;


#if defined(__AIESIM__) || defined(__X86SIM__)

int main(void) {
  mygraph.init();
  mygraph.run(<number_of_iterations>);
  mygraph.end();
  return 0;
}
#endif
```

此有条件指令将该应用编译为仅供 AI 引擎仿真器使用。它会阻止从 graph 和 PS 主机应用多次初始化或多次运行该 graph。该指令允许在系统设计的仿真或硬件仿真中运行 `graph.cpp`，使其同样可在 AI 引擎仿真器和 x86 仿真器中运行。但在硬件中运行时，会从 PS 应用而不是 `graph.cpp` 来初始化和运行此 graph。

Linux 上的主机编程

在 Linux 操作系统中，ADF API 用于控制 AI 引擎 graph。赛灵思的 Xilinx Runtime (XRT) API 则用于控制 PL 内核。赛灵思的 Xilinx Runtime (XRT) API 还可用于控制 AI 引擎 graph。下图显示此系统中所需的 API 和驱动程序。

图 66：**AI 引擎 XRT 软件栈**



利用 ADF API 控制 AI 引擎 graph

ADF API 用于控制顶层应用或主机代码中的 graph 执行，如《AI 引擎内核与 Graph 编程指南》(UG1079) 的 graph 编程简介中所述。例如，以下代码对用于 graph 中的 AI 引擎内核运行时参数进行同步更新：

```
// ADF API:run and update graph parameters (RTP)
gr.run(4);
gr.update(gr.trigger,10);
gr.update(gr.trigger,10);
gr.update(gr.trigger,100);
gr.update(gr.trigger,100);
gr.end();
```

提示：使用 `graph.end()` 会终止此 graph。调用 `end()` 后，此 graph 无法恢复。

在主机应用 (`host.cpp`) 中，通过调用 `graph.update()` 函数即可更新 RTP，调用 `graph.run()` 即可在 graph 中启动 AI 引擎内核。在硬件仿真流程和硬件流程中，ADF API 会调用 XRT API，而 `adf::registerXRT()` 则用于管理两者之间的关系。

重要提示！必须先调用 `adf::registerXRT()`，然后才能调用任何 ADF API 控制 graph 或者与 graph 交互。

以下代码示例显示的是 RTP 更新以及 ADF API 的执行过程。

```
// update graph parameters (RTP) & run
adf::registerXRT(dhdl, uuid);
gr.update(gr.size, 1024);//update RTP
gr.run(16);//start AIE kernel
gr.wait();
```

在前述示例中，`gr.run(16)` 指定运行 16 次迭代。

在 `gr.wait()` 中，应用会等待 AI 引擎内核完成。

此代码示例显示 `adf::registerXRT()` 需要 XCLBIN 镜像的器件句柄 (`dhdl`) 和 `UUID`。这些均可使用 XRT API 来获取：

```
auto dhdl = xrtDeviceOpen(0);//device index=0
xrtDeviceLoadXclbinFile(dhdl,xclbinFilename);
xuid_t uuid;
xrtDeviceGetXclbinUUID(dhdl, uuid);
```

利用 XRT API 控制 PL 内核

赛灵思提供了 OpenSource XRT API 用于在对 Linux 主机代码进行编程时控制 PL 内核的执行。

用于控制 PL 内核的 XRT API 的执行模型如下：

1. 获取器件句柄并加载 XCLBIN。按需获取 `uuid`。
2. 分配缓冲器对象并映射到主机存储器。处理来自主机存储器的数据，并将其传输到器件存储器。
3. 获取内核与运行句柄、设置内核实参并启动内核。
4. 等待内核完成。
5. 将数据从器件中的全局存储器传回主机存储器。
6. 主机代码继续处理主机存储器中的新数据。

使用本机 XRT API 时，主机应用如下所示。

```
1.// Open device, load xclbin, and get uuid

auto dhdl = xrtDeviceOpen(0);//device index=0

xrtDeviceLoadXclbinFile(dhdl,xclbinFilename);
xuid_t uuid;
xrtDeviceGetXclbinUUID(dhdl, uuid);

2. Allocate output buffer objects and map to host memory

xrtBufferHandle out_bohdl = xrtBOAlloc(dhdl, output_size_in_bytes, 0, /
*BANK=*/0);
std::complex<short> *host_out = (std::complex<short>*)xrtBOMap(out_bohdl);

3. Get kernel and run handles, set arguments for kernel, and launch kernel.
xrtKernelHandle s2mm_khdl = xrtPLKernelOpen(dhdl, top->m_header.uuid,
"s2mm"); // Open kernel handle
xrtRunHandle s2mm_rhdl = xrtRunOpen(s2mm_khdl);
xrtRunSetArg(s2mm_rhdl, 0, out_bohdl); // set kernel arg
xrtRunSetArg(s2mm_rhdl, 2, OUTPUT_SIZE); // set kernel arg
xrtRunStart(s2mm_rhdl); //launch s2mm kernel

// ADF API:run, update graph parameters (RTP) and so on
......

4. Wait for kernel completion.
auto state = xrtRunWait(s2mm_rhdl);

5. Sync output device buffer objects to host memory.

xrtBOSync(out_bohdl, XCL_BO_SYNC_BO_FROM_DEVICE , output_size_in_bytes,/
*OFFSET=*/ 0);

//6. post-processing on host memory - "host_out"
```

完成数据后处理之后，释放已分配的对象：

```
graph.end();
xrtRunClose(s2mm_rhdl);
xrtKernelClose(s2mm_khdl);

xrtBOFree(out_bohdl);
xrtDeviceClose(dhdl);
```

重要提示！完成 `graph.end()` 后，AI 引擎内核将不再恢复。`graph.end()` API 会等待 graph 终止。当 graph 的所有活动处理器都退出其 main 线程并禁用自身后，此 graph 即被视为已终止。这对于 PS 应用属于阻塞运算。`graph.end()` 还会清除 graph 的状态，例如强制释放所有锁定和清除 graph 中使用的串流开关配置。要多次运行 graph，请将 `graph.end()` 替换为 `graph.wait()`。

利用 XRT C API 控制 AI 引擎 graph

为 Linux 进行主机代码编程时，赛灵思提供的开源 XRT API 同样可用于控制 AI 引擎 graph 的执行。为控制 AI 引擎 graph，XRT 会通过头文件 `experimental/xrt_graph.h` 来提供 API。

提示：头文件 `experimental/xrt_graph.h` 包含在头文件 `experimental/xrt_kernel.h` 中。因此，只需包含 `experimental/xrt_kernel.h` 即可使用 XRT API 来控制 AI 引擎 graph。

XRT graph API 包含 C 版本和 C++ 版本。以下提供了使用 XRT C API 来控制 AI 引擎 graph 的代码示例：

```
int narrow_filter[12] = {180, 89, -80, -391, -720, -834, -478, 505, 2063,
3896, 5535, 6504};
    int wide_filter[12] = {-21, -249, 319, -78, -511, 977, -610, -844,
2574, -2754, -1066, 18539};
    xrtGraphOpen(dhdl,top->m_header.uuid,"gr");
    if(!ghdl){
    int size=1024;
    xrtGraphUpdateRTP(ghdl,"gr.fir24.in[1]",
(char*)narrow_filter,12*sizeof(int));
    xrtGraphRun(ghdl,16);
    xrtGraphWait(ghdl,0);
    xrtGraphUpdateRTP(ghdl,"gr.fir24.in[1]",
(char*)wide_filter,12*sizeof(int));
    xrtGraphRun(ghdl,16);
    ......
    xrtGraphEnd(ghdl,0);
    xrtGraphClose(ghdl);
```

提示：`Work/ps/c_rts/aie_control_xrt.cpp` 文件包含有关 graph、RTP、GMIO 和初始化配置的信息。您可按需查找有关这些 XRT API 的信息。

利用 XRT C++ API 控制 AI 引擎 graph

如前一节中所述，XRT 可通过头文件 `experimental/xrt_graph.h` 提供 C 和 C++ API 来控制 AI 引擎 graph。

XRT 在名称空间 `xrt` 及其成员函数内提供 `graph` 类，以控制 graph。以下提供了使用 XRT C++ API 来控制 AI 引擎 graph 的代码示例：

```
using namespace adf;
// Open xclbin
auto device = xrt::device(0); //device index=0
auto uuid = device.load_xclbin(xclbinFilename);
auto dhdl = xrtDeviceOpenFromXcl(device);
...
int coeffs_readback[12];
int narrow_filter[12] = {180, 89, -80, -391, -720, -834, -478, 505, 2063,
3896, 5535, 6504};
int wide_filter[12] = {-21, -249, 319, -78, -511, 977, -610, -844, 2574,
-2754, -1066, 18539};
auto ghdl=xrt::graph(device,uuid,"gr");
ghdl.update("gr.fir24.in[1]",narrow_filter);
ghdl.run(16);
ghdl.wait();
ghdl.read("gr.fir24.inout[0]",coeffs_readback);//Read after graph::wait.
RTP update effective
ghdl.update("gr.fir24.in[1]",wide_filter);
ghdl.run(16);
ghdl.read("gr.fir24.inout[0]", coeffs_readback);//Async read
ghdl.end();
```

注释： `Work/ps/c_rts/aie_control_xrt.cpp` 文件包含有关 graph、RTP、GMIO 和初始化配置的信息。例如，在以上代码中，您可在 `aie_control_xrt.cpp` 的 `RTP Configurations` 部分中找到 RTP 端口名称，名为 `gr.fir24.in[1]`。

控制 AI 引擎 GMIO 传输

AI 引擎 GMIO 支持同步和异步传输，ADF API 和 XRT API 也都支持同步和异步传输。以下代码示例显示了利用 XRT API 执行的异步 GMIO 实现：

```
char* xclbinFilename = argv[1];
// Open xclbin
auto device = xrt::device(0); //device index=0
auto uuid = device.load_xclbin(xclbinFilename);

auto din_buffer = xrt::aie::bo (device, BLOCK_SIZE_in_Bytes,
xrt::bo::flags::normal, /*memory group*/0);
int* dinArray= din_buffer.map<int*>();
auto dout_buffer = xrt::aie::bo (device, BLOCK_SIZE_in_Bytes,
xrt::bo::flags::normal, /*memory group*/0);
int* doutArray= dout_buffer.map<int*>();

int ret=0;
int error=0;
//Initialization
for(int i=0;i<ITERATION*1024/4;i++){
  dinArray[i]=i;
}

din_buffer.async("gr.gmioIn",XCL_BO_SYNC_BO_GMIO_TO_AIE,BLOCK_SIZE_in_Bytes,
/*offset*/0);

auto ghdl=xrt::graph(device,uuid,"gr");
ghdl.run(ITERATION);

dout_buffer.async("gr.gmioOut",XCL_BO_SYNC_BO_AIE_TO_GMIO,BLOCK_SIZE_in_Byte
```

```
s,/*offset*/0);

ghdl.wait();

//Post-processing
...
```

用于控制 AI 引擎 graph 的多进程和多线程支持

赛灵思 XRT API 提供多进程支持，用于控制 AI 引擎阵列和 graph。它支持 AI 引擎阵列和 graph 上的 3 种操作模式。

- Exclusive Mode（专享模式）：全权访问 AI 引擎阵列或 graph。没有任何其它进程可对其进行访问。
- Primary Mode（基准模式）：全权访问 AI 引擎阵列或 graph。其它进程可对 AI 引擎阵列或 graph 进行非破坏性访问。
- Shared Mode（共享模式）：仅限对 AI 引擎阵列或 graph 进行非破坏性访问。

赛灵思 XRT 提供了下列 API，用于在 3 种模式下打开 AI 引擎阵列。

- `xrtAIEDeviceOpenExclusive`（专享模式）
- `xrtAIEDeviceOpen`（基准模式）
- `xrtAIEDeviceOpenShared`（共享模式）

注释：如果应用不调用 `xrtAIEDeviceOpen*` 以获取器件句柄，那么默认情况下它将尝试获取基准上下文，同时尝试通过 XRT API 来访问 AI 引擎阵列。

赛灵思 XRT 还提供了以下 API，以便在打开 AI 引擎阵列后打开 graph。

- `xrtGraphOpenExclusive`（专享模式）
- `xrtGraphOpen`（基准模式）
- `xrtGraphOpenShared`（共享模式）

AI 引擎阵列上的下列 API 在专享模式和基准模式下允许执行加载、复位和 GMIO 数据传输。

- `xrtDeviceLoadXclbin`
- `xrtAIEResetArray`
- `xrtAIESyncBO`
- `xrtDeviceClose`

AI 引擎阵列上的下列 API 在共享模式下允许执行非破坏性操作。

- `xrtDeviceLoadXclbin`：从 xclbin 读取元数据。
- `xrtDeviceClose`

AI 引擎 graph 上的专享模式和基准模式下允许的 API 包括：

- `xrtGraphRun`
- `xrtGraphWait`
- `xrtGraphEnd`
- `xrtGraphUpdateRTP`

- `xrtGraphReadRTP`
- `xrtGraphTimeStamp`
- `xrtGraphClose`

AI 引擎阵列和 graph 上的共享模式下允许的 API 包括：

- `xrtGraphUpdateRTP`（异步模式）
- `xrtGraphReadRTP`
- `xrtGraphTimeStamp`
- `xrtGraphClose`

以下规则适用于 AI 引擎阵列多进程支持。

- 在专享模式下，仅有一个进程能打开 AI 引擎阵列。如果 AI 引擎阵列在专享模式下打开，那么无论是在同一进程内还是在其它进程内，都无法在任何其它模式下再次打开。
- 在基准模式下，仅有一个进程能打开 AI 引擎阵列。如果 AI 引擎阵列在基准模式下打开，则在专享模式下或基准模式下无法再次打开，但在共享模式下可以将其再次打开。

以下规则适用于 AI 引擎 graph 多进程支持。

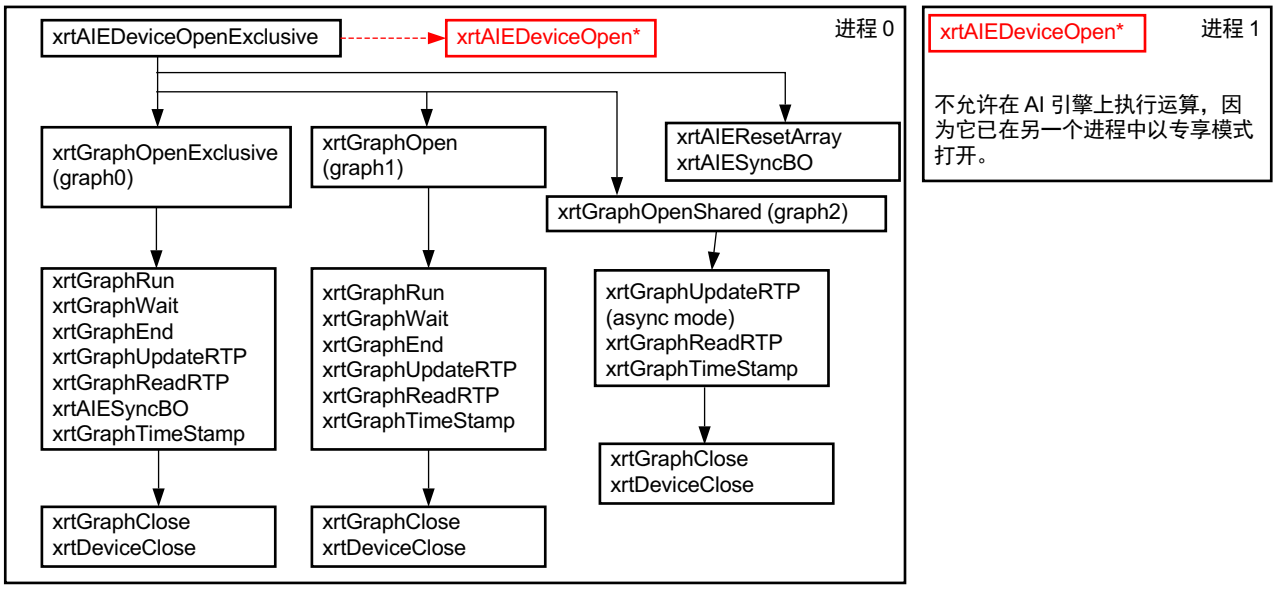
- 在任一进程内，graph 不应多次打开。
- 如果某个 AI 引擎 graph 已在专享模式下打开，则无法在任何模式下再次打开。
- 在共享模式下，可有多个进程打开 graph，但在基准模式下，仅允许在单个进程内打开此 graph。

必须先关闭 graph，然后才能关闭阵列。关闭所有已打开的 AI 引擎阵列和 graph 后，即可再次打开这些 AI 引擎阵列和 graph，并且前述规则再次适用。

注释：当某个 graph 或 AI 引擎阵列由 `xrtGraphClose` 或 `xrtDeviceClose` 关闭后，仅关闭已打开的上下文。它并不会清除 AI 引擎阵列中的数据存储器、程序存储器、FIFO 和 DMA 状态。

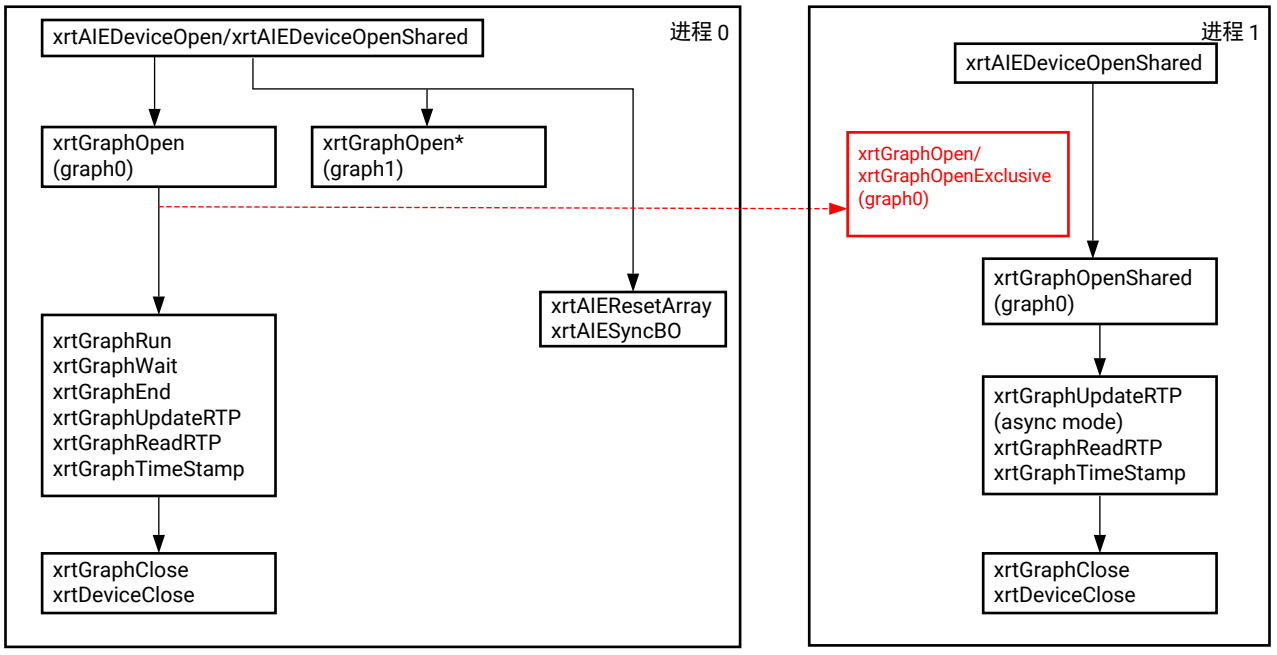
下图汇总了专享模式下的多进程支持（黑色：受支持，红色：不受支持，“*”：任意模式）。

图 67：**专享模式下的多进程支持**



下图汇总了基准模式和共享模式下的多进程支持（黑色：受支持，红色：不受支持）。

图 68：**基准模式和共享模式下的多进程支持**



建议多线程使用与多进程相同的模型。但由于 AI 引擎器件句柄和 graph 句柄可在线程之间共享，因此在线程之间使用相同的器件句柄或 graph 句柄是合规的。主机应用负责在线程之间同步 AI 引擎阵列状态和 graph 状态，当多线程是 AI 引擎阵列或 graph 的专享或基准所有者时，尤其如此。

以下提供了使用多进程的样本代码。

```
#include <stdlib.h>
#include <fstream>
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>
#include "adf/adf_api/XRTConfig.h"
#include "experimental/xrt_kernel.h"

#include "graph.cpp"

//8192 matches 32 iterations of graph::run
#define OUTPUT_SIZE 8192
int value1[16] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
int value2[16] = {-1,-2,-3,-4,-5,-6,-7,-8,-9,-10,-11,-12,-13,-14,-15,-16};

using namespace adf;

int run(int argc, char* argv[],int id){
    std::cout<<"Child process "<<id<<" start"<<std::endl;

    //TARGET_DEVICE macro needs to be passed from gcc command line
    if(argc != 2) {
        std::cout << "Usage: " << argv[0] <<" <xclbin>" << std::endl;
        return EXIT_FAILURE;
    }
    char* xclbinFilename = argv[1];
    std::string graph_name=std::string("gr[")+std::to_string(id)+"]";
    std::string rtp_inout_name=std::string("gr[")+std::to_string(id)
+std::string("].k.inout[0]");

    int ret;
    int value_readback[16]={0};
    if(fork()==0){//child child process
        xrtDeviceHandle dhdl2=xrtAIEDeviceOpenShared(0);
        ret=xrtDeviceLoadXclbinFile(dhdl2,xclbinFilename);
           if(ret){
            printf("child child Xclbin Load fail\n");
            }
        if(!dhdl2){
            std::cout<<"child child device open error"<<std::endl;
            return 1;
        }else{
            std::cout<<"child child device open pass"<<std::endl;
        }
        xuid_t uuid2;
            ret=xrtDeviceGetXclbinUUID(dhdl2, uuid2);
        if(ret){
            std::cout<<"child child get xclbin uuid error"<<std::endl;
            return 1;
        }else{
            std::cout<<"child child get xclbin uuid pass"<<std::endl;
        }
        auto ghdl2=xrtGraphOpenShared(dhdl2,uuid2,graph_name.data());
        if(!ghdl2){
            std::cout<<"child child graph open error"<<std::endl;
            return 1;
        }else{
            std::cout<<"child child graph open pass"<<std::endl;
        }

        ret=xrtGraphReadRTP(ghdl2, rtp_inout_name.data(), (char*)value_readback,
16*sizeof(int));
        if(ret){
            std::cout<<"child child Graph RTP read fail"<<std::endl;
            return 1;
        }
        std::cout<<"Add value read back are:";
        for(int i=0;i<16;i++){
```

```
            std::cout<<value_readback[i]<<",\t";
        }
        std::cout<<std::endl;
        xrtGraphClose(ghdl2);
        xrtDeviceClose(dhdl2);
        std::cout<<"child child process exit"<<std::endl;
        exit(0);
    }

    xrtDeviceHandle dhdl=xrtAIEDeviceOpen(0);
    ret=xrtDeviceLoadXclbinFile(dhdl,xclbinFilename);
       if(ret){
        printf("Xclbin Load fail\n");
        }
    xuid_t uuid;
        xrtDeviceGetXclbinUUID(dhdl, uuid);

    auto ghdl=xrtGraphOpen(dhdl,uuid,graph_name.data());
    if(!ghdl){
        std::cout << "Graph Open error" << std::endl;
    }else{
        std::cout << "Graph Open ok" <<std::endl;
    }
    std::string rtp_in_name=std::string("gr[")+std::to_string(id)+std::string("].k.in[1]");
    ret=xrtGraphUpdateRTP(ghdl,rtp_in_name.data(),(char*)value1,16*sizeof(int));
    if(ret){
        std::cout<<"Graph RTP update fail"<<std::endl;;
        return 1;
    }
    xrtGraphRun(ghdl,16);

    xrtGraphWait(ghdl,0);
    std::cout<<"Graph wait done"<<std::endl;

    //second run
    ret=xrtGraphUpdateRTP(ghdl,rtp_in_name.data(),(char*)value2,16*sizeof(int));
    if(ret!=0){
        std::cout<<"Graph RTP update fail"<<std::endl;
        return 1;
    }else{
        std::cout<<"Graph RTP update pass"<<std::endl;
    }
    xrtGraphRun(ghdl,16);

    while(wait(NULL)>0){//Wait for child child process
    }

    ret=xrtGraphWait(ghdl,0);
    if(ret){
        std::cout << "Graph wait error" << std::endl;
    }else{
        std::cout<<"Graph done"<<std::endl;
    }
    xrtGraphClose(ghdl);
    xrtDeviceClose(dhdl);
    std::cout<<"Child process:"<<id<<" done"<<std::endl;
    return 0;
}

int main(int argc, char* argv[])
{
    try {
        for(int i=0;i<GRAPH_NUM;i++){
            if(fork()==0){//child
                auto match = run(argc, argv,i);
                std::cout << "TEST child " <<i<< (match ? " FAILED" : " PASSED") << "\n";
                return (match ? EXIT_FAILURE :  EXIT_SUCCESS);
            }else{
                size_t output_size_in_bytes = OUTPUT_SIZE * sizeof(int);
                //TARGET_DEVICE macro needs to be passed from gcc command line
                if(argc != 2) {
```

```
                    std::cout << "Usage: " << argv[0] <<" <xclbin>" << std::endl;
                    return EXIT_FAILURE;
                }
                char* xclbinFilename = argv[1];

                int ret;
                // Open xclbin
                auto device = xrt::device(0); //device index=0
                auto uuid = device.load_xclbin(xclbinFilename);

                // s2mm & data_generator kernel handle
                std::string s2mm_kernel_name=std::string("s2mm:{s2mm_")+std::to_string(i
+1)+std::string("}");
                xrt::kernel s2mm = xrt::kernel(device, uuid, s2mm_kernel_name.data());
                std::string data_generator_kernel_name=std::string("data_generator:
{data_generator_")+std::to_string(i+1)+std::string("}");
                xrt::kernel data_generator = xrt::kernel(device, uuid,
data_generator_kernel_name.data());

                // output memory
                auto out_bo=xrt::bo(device, output_size_in_bytes,s2mm.group_id(0));
                auto host_out=out_bo.map<int*>();
                auto s2mm_run = s2mm(out_bo, nullptr, OUTPUT_SIZE);//1st run for s2mm has
started
                auto data_generator_run = data_generator(nullptr, OUTPUT_SIZE);

                // wait for s2mm done
                std::cout<<"Waiting s2mm to complete"<<std::endl;
                auto state = s2mm_run.wait();
                std::cout << "s2mm "<<" completed with status(" << state << ")"<<std::endl;
                out_bo.sync(XCL_BO_SYNC_BO_FROM_DEVICE);

                int match = 0;
                int counter=0;
                for (int i = 0; i < OUTPUT_SIZE/2/16; i++) {
                    for(int j=0;j<16;j++){
                        if(host_out[i*16+j]!=counter+value1[j]){
                            std::cout<<"ERROR: num="<<i*16+j<<"
out="<<host_out[i*16+j]<<std::endl;
                            match=1;
                            break;
                        }
                        counter++;
                    }
                }
                for(int i=OUTPUT_SIZE/2/16;i<OUTPUT_SIZE/16;i++){
                    for(int j=0;j<16;j++){
                        if(host_out[i*16+j]!=counter+value2[j]){
                            std::cout<<"ERROR: num="<<i*16+j<<"
out="<<host_out[i*16+j]<<std::endl;
                            match=1;
                            break;
                        }
                        counter++;
                    }
                }

                std::cout << "TEST " <<i<< (match ? " FAILED" : " PASSED") << "\n";
                while(wait(NULL)>0){//Wait for all child process
                }
                std::cout<<"all done"<<std::endl;
                return (match ? EXIT_FAILURE :  EXIT_SUCCESS);
            }
        }
    }
        catch (std::exception const& e) {
        std::cout << "Exception: " << e.what() << "\n";
        std::cout << "FAILED TEST\n";
        return 1;
    }
}
```

C++ XRT API 提供多进程支持

XRT C++ API 将 `xrt::aie::device` 类扩展为支持 https://github.com/Xilinx/XRT/blob/master/src/runtime_src/core/include/xrt/xrt_aie.h 中的访问模式，如以下代码所示。

```
namespace xrt { namespace aie {

/**
* @enum access_mode - AIE array access mode
*
* @var exclusive
* Exclusive access to AIE array. No other process will have
* access to the AIE array.
* @var primary
* Primary access to AIE array provides same capabilities as exclusive
* access, but other processes will be allowed shared access as well.
* @var shared
* Shared none destructive access to AIE array, a limited number of APIs
* can be called.
* @var none
* For internal use only, to be removed.
*
* By default the AIE array is opened in primary access mode.
*/
enum class access_mode : uint8_t { exclusive = 0, primary = 1, shared = 2,
none = 3 };

class device : public xrt::device
{
public:
  using access_mode = xrt::aie::access_mode;

  /**
  * device() - Construct device with specified access mode
  *
  * @param args
  * Arguments to construct a device (xrt_device.h).
  * @param am
  * Open the AIE device is specified access mode (default primary)
  *
  * The default access mode is primary.
  */
  template
  device(ArgType&& arg, access_mode am = access_mode::primary)
    : xrt::device(std::forward(arg))
  {
    open_context(am);
  }
...
};

}} // namespace aie, xrt
```

XRT C++ API 将 `xrt::graph` 类扩展为支持 https://github.com/Xilinx/XRT/blob/master/src/runtime_src/core/include/xrt/xrt_graph.h 中的访问模式，如以下代码所示。

```
namespace xrt {

class graph
{
public:
/**
* @enum access_mode - graph access mode
*
* @var exclusive
* Exclusive access to graph and all graph APIs. No other process
* will have access to the graph.
* @var primary
* Primary access to graph provides same capabilities as exclusive
* access, but other processes will be allowed shared access as well.
* @var shared
* Shared none destructive access to graph, a limited number of APIs
* can be called.
*
* By default a graph is opened in primary access mode.
*/
  enum class access_mode : uint8_t { exclusive = 0, primary = 1, shared =
2 };

  /**
  * graph() - Constructor from a device, xclbin and graph name
  *
  * @param device
  * Device on which the graph should execute
  * @param xclbin_id
  * UUID of the xclbin with the graph
  * @param name
  * Name of graph to construct
  * @param am
  * Open the graph with specified access (default primary)
  */
  graph(const xrt::device& device, const xrt::uuid& xclbin_id, const
std::string& name,
    access_mode am = access_mode::primary);

...
};

} // namespace xrt
```

此代码样本的对应 C++ 版本如下所示。

```
#include <stdlib.h>
#include <fstream>
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>
#include "adf/adf_api/XRTConfig.h"
#include "experimental/xrt_aie.h"
#include "experimental/xrt_graph.h"
#include "experimental/xrt_kernel.h"

#include "graph.cpp"

//8192 matches 32 iterations of graph::run
#define OUTPUT_SIZE 8192
int value1[16] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
```

```
int value2[16] = {-1,-2,-3,-4,-5,-6,-7,-8,-9,-10,-11,-12,-13,-14,-15,-16};

using namespace adf;

int run(int argc, char* argv[],int id){
    std::cout<<"Child process "<<id<<" start"<<std::endl;

    //TARGET_DEVICE macro needs to be passed from gcc command line
    if(argc != 2) {
        std::cout << "Usage: " << argv[0] <<" <xclbin>" << std::endl;
        return EXIT_FAILURE;
    }
    char* xclbinFilename = argv[1];
    std::string graph_name=std::string("gr[")+std::to_string(id)+"]";
    std::string rtp_inout_name=std::string("gr[")+std::to_string(id)
+std::string("].k.inout[0]");

    int ret;
    int value_readback[16]={0};
    if(fork()==0){//child child process
        xrt::aie::device device{0, xrt::aie::device::access_mode::shared};
        auto uuid = device.load_xclbin(xclbinFilename);
        xrt::graph graph{device, uuid, graph_name, xrt::graph::access_mode::shared};

        graph.read(rtp_inout_name, value_readback);
        std::cout<<"Add value read back are:";
        for(int i=0;i<16;i++){
            std::cout<<value_readback[i]<<",\t";
        }
        std::cout<<std::endl;
        std::cout<<"child child process exit"<<std::endl;
        exit(0);
    }

    xrt::aie::device device{0};   // default primary context
    auto uuid = device.load_xclbin(xclbinFilename);
    xrt::graph graph{device, uuid, graph_name}; // default primary context

    std::string rtp_in_name=std::string("gr[")+std::to_string(id)+std::string("].k.in[1]");
    graph.update(rtp_in_name, value1);
    graph.run(16); // 16 iterations

    graph.wait(0); // wait 0 => wait till graph is done
    std::cout<<"Graph wait done"<<std::endl;

    //second run
    graph.update(rtp_in_name.data(), value2);
    graph.run(16); // 16 iterations;

    while(wait(NULL)>0){//Wait for child child process
    }

    graph.wait(0); // wait 0 => wait till graph is done
    std::cout<<"Child process:"<<id<<" done"<<std::endl;
    return 0;
}

int main(int argc, char* argv[])
{
    try {
        for(int i=0;i<GRAPH_NUM;i++){
            if(fork()==0){//child
                auto match = run(argc, argv,i);
                std::cout << "TEST child " <<i<< (match ? " FAILED" : " PASSED") << "\n";
                return (match ? EXIT_FAILURE :  EXIT_SUCCESS);
            }else{
                size_t output_size_in_bytes = OUTPUT_SIZE * sizeof(int);
                //TARGET_DEVICE macro needs to be passed from gcc command line
                if(argc != 2) {
                    std::cout << "Usage: " << argv[0] <<" <xclbin>" << std::endl;
                    return EXIT_FAILURE;
```

```
                }
                char* xclbinFilename = argv[1];

                int ret;
                // Open xclbin
                auto device = xrt::device(0); //device index=0
                auto uuid = device.load_xclbin(xclbinFilename);

                // s2mm & data_generator kernel handle
                std::string s2mm_kernel_name=std::string("s2mm:{s2mm_")+std::to_string(i
+1)+std::string("}");
                xrt::kernel s2mm = xrt::kernel(device, uuid, s2mm_kernel_name.data());
                std::string data_generator_kernel_name=std::string("data_generator:
{data_generator_")+std::to_string(i+1)+std::string("}");
                xrt::kernel data_generator = xrt::kernel(device, uuid,
data_generator_kernel_name.data());

                // output memory
                auto out_bo=xrt::bo(device, output_size_in_bytes,s2mm.group_id(0));
                auto host_out=out_bo.map<int*>();
                auto s2mm_run = s2mm(out_bo, nullptr, OUTPUT_SIZE);//1st run for s2mm has
started
                auto data_generator_run = data_generator(nullptr, OUTPUT_SIZE);

                // wait for s2mm done
                std::cout<<"Waiting s2mm to complete"<<std::endl;
                auto state = s2mm_run.wait();
                std::cout << "s2mm "<<" completed with status(" << state << ")"<<std::endl;
                out_bo.sync(XCL_BO_SYNC_BO_FROM_DEVICE);

                int match = 0;
                int counter=0;
                for (int i = 0; i < OUTPUT_SIZE/2/16; i++) {
                    for(int j=0;j<16;j++){
                        if(host_out[i*16+j]!=counter+value1[j]){
                            std::cout<<"ERROR: num="<<i*16+j<<"
out="<<host_out[i*16+j]<<std::endl;
                            match=1;
                            break;
                        }
                        counter++;
                    }
                }
                for(int i=OUTPUT_SIZE/2/16;i<OUTPUT_SIZE/16;i++){
                    for(int j=0;j<16;j++){
                        if(host_out[i*16+j]!=counter+value2[j]){
                            std::cout<<"ERROR: num="<<i*16+j<<"
out="<<host_out[i*16+j]<<std::endl;
                            match=1;
                            break;
                        }
                        counter++;
                    }
                }

                std::cout << "TEST " <<i<< (match ? " FAILED" : " PASSED") << "\n";
                while(wait(NULL)>0){//Wait for all child process
                }
                std::cout<<"all done"<<std::endl;
                return (match ? EXIT_FAILURE :  EXIT_SUCCESS);
            }
        }
    }
        catch (std::exception const& e) {
        std::cout << "Exception: " << e.what() << "\n";
        std::cout << "FAILED TEST\n";
        return 1;
    }
}
```

通过 XRT API 报告错误

XRT 提供了错误报告 API。错误报告 API 可分类为两种类型：同步 API 和异步 API。同步错误是在 XRT 运行时函数调用期间检测到的错误。它符合 POSIX 标准。例如：

```
rval = xclSyncBO(devHandle, boHandle, XCL_BO_SYNC_BO_TO_DEVICE, size, offset);
if (rval != 0) {
    /* code to handle xclSyncBO fail */
}
```

```
// Syncronous error captured by xclSyncBO call
//    -EINVAL:      Invalid arguments, e.g. invalid sync dir, invalid size or offset
//    -ENOENT:      No such file or directory, e.g. invalid bo handle
//    -EOPNOTSUPP:  Operation not supported, e.g. BO is not syncable
//    -EBUSY:       Device or resource busy, e.g. No available DMA channel
//    -ENOMEM:      Out of memory, e.g. No available free memory for sync BO
//    -EIO:         I/O error, e.g. DMA error
// "dmesg" or look at system log file (/var/log/syslog) might give you more information
```

异步错误可能与当前 XRT 函数调用或正在运行的应用无关。异步错误缓存在驱动程序子系统内，可供用户应用通过异步错误报告 API 来访问。缓存的错误将长久保存直至被显式清除为止。持久存在的错误并不一定表示当前系统状态，例如，开发板可能已复位且正常工作，而先前缓存的错误仍可用。为避免混淆当前状态，异步错误附有时间戳以指示错误发生时间。例如，此时间戳可与最近的 `xbutil` 复位时间戳进行比较。

驱动程序缓存的错误包含系统错误代码和 `xrt_error_code.h` 中定义的额外元数据，此元数据在用户空间与内核空间之间共享。

异步错误的错误代码格式如下所示：

```
/**
 * xrtErrorCode layout
 *
 * This layout is internal to XRT (akin to a POSIX error code).
 *
 * The error code is populated by driver and consumed by XRT
 * implementation where it is translated into an actual error / info /
 * warning that is propagated to the end user.
 *
 * 63 - 48  47 - 40   39 - 32   31 - 24   23 - 16    15 - 0
 * ---------------------------------------------------------
 * |    |    |    |    |    |    |    |    |    |----| xrtErrorNum
 * |    |    |    |    |    |    |    |    |----|---------- xrtErrorDriver
 * |    |    |    |    |    |    |----|--------------------- xrtErrorSeverity
 * |    |    |    |    |----|-------------------------------- xrtErrorModule
 * |    |    |----|-------------------------------------------- xrtErrorClass
 * |----|---------------------------------------------------- reserved
 *
 */
typedef uint64_t xrtErrorCode;
typedef uint64_t xrtErrorTime;

#define XRT_ERROR_NUM_MASK         0xFFFFUL
#define XRT_ERROR_NUM_SHIFT         0
#define XRT_ERROR_DRIVER_MASK         0xFUL
#define XRT_ERROR_DRIVER_SHIFT         16
#define XRT_ERROR_SEVERITY_MASK         0xFUL
#define XRT_ERROR_SEVERITY_SHIFT    24
#define XRT_ERROR_MODULE_MASK         0xFUL
#define XRT_ERROR_MODULE_SHIFT         32
#define XRT_ERROR_CLASS_MASK         0xFUL
#define XRT_ERROR_CLASS_SHIFT         40

#define    XRT_ERROR_CODE_BUILD(num, driver, severity, module, eclass) \
    ((((num) & XRT_ERROR_NUM_MASK) << XRT_ERROR_NUM_SHIFT) | \
    (((driver) & XRT_ERROR_DRIVER_MASK) << XRT_ERROR_DRIVER_SHIFT) | \
    (((severity) & XRT_ERROR_SEVERITY_MASK) << XRT_ERROR_SEVERITY_SHIFT) | \
    (((module) & XRT_ERROR_MODULE_MASK) << XRT_ERROR_MODULE_SHIFT) | \
    (((eclass) & XRT_ERROR_CLASS_MASK) << XRT_ERROR_CLASS_SHIFT))
```

```
#define XRT_ERROR_NUM(code) (((code) >> XRT_ERROR_NUM_SHIFT) &
XRT_ERROR_NUM_MASK)
#define XRT_ERROR_DRIVER(code) (((code) >> XRT_ERROR_DRIVER_SHIFT) &
XRT_ERROR_DRIVER_MASK)
#define XRT_ERROR_SEVERITY(code) (((code) >> XRT_ERROR_SEVERITY_SHIFT) &
XRT_ERROR_SEVERITY_MASK)
#define XRT_ERROR_MODULE(code) (((code) >> XRT_ERROR_MODULE_SHIFT) &
XRT_ERROR_MODULE_MASK)
#define XRT_ERROR_CLASS(code) (((code) >> XRT_ERROR_CLASS_SHIFT) &
XRT_ERROR_CLASS_MASK)

/**
 * xrt_error_num - XRT specific error numbers
 */

enum xrtErrorNum {
XRT_ERROR_NUM_FIRWWALL_TRIP = 1,
XRT_ERROR_NUM_TEMP_HIGH,
XRT_ERROR_NUM_AIE_SATURATION,
XRT_ERROR_NUM_AIE_FP,
XRT_ERROR_NUM_AIE_STREAM,
XRT_ERROR_NUM_AIE_ACCESS,
XRT_ERROR_NUM_AIE_BUS,
XRT_ERROR_NUM_AIE_INSTRUCTION,
XRT_ERROR_NUM_AIE_ECC,
XRT_ERROR_NUM_AIE_LOCK,
XRT_ERROR_NUM_AIE_DMA,
XRT_ERROR_NUM_AIE_MEM_PARITY,
XRT_ERROR_NUM_UNKNOWN
};

enum xrtErrorDriver {
  XRT_ERROR_DRIVER_XOCL,
  XRT_ERROR_DRIVER_XCLMGMT,
  XRT_ERROR_DRIVER_ZOCL,
  XRT_ERROR_DRIVER_AIE,
  XRT_ERROR_DRIVER_UNKNOWN
};

enum xrtErrorSeverity {
  XRT_ERROR_SEVERITY_EMERGENCY = 0,
  XRT_ERROR_SEVERITY_ALERT,
  XRT_ERROR_SEVERITY_CRITICAL,
  XRT_ERROR_SEVERITY_ERROR,
  XRT_ERROR_SEVERITY_WARNING,
  XRT_ERROR_SEVERITY_NOTICE,
  XRT_ERROR_SEVERITY_INFO,
  XRT_ERROR_SEVERITY_DEBUG,
  XRT_ERROR_SEVERITY_UNKNOWN
};

enum xrtErrorModule {
  XRT_ERROR_MODULE_FIREWALL = 0,
  XRT_ERROR_MODULE_CMC,
  XRT_ERROR_MODULE_AIE_CORE,
  XRT_ERROR_MODULE_AIE_MEMORY,
  XRT_ERROR_MODULE_AIE_SHIM,
  XRT_ERROR_MODULE_AIE_NOC,
  XRT_ERROR_MODULE_AIE_PL,
  XRT_ERROR_MODULE_AIE_UNKNOWN
};

enum xrtErrorClass {
```

```
XRT_ERROR_CLASS_FIRST_ENTRY = 1,
XRT_ERROR_CLASS_SYSTEM = XRT_ERROR_CLASS_FIRST_ENTRY,
XRT_ERROR_CLASS_AIE,
XRT_ERROR_CLASS_HARDWARE,
XRT_ERROR_CLASS_UNKNOWN,
XRT_ERROR_CLASS_LAST_ENTRY = XRT_ERROR_CLASS_UNKNOWN
};
```

API 头文件 `experimental/xrt_error.h` 用于定义 API 以访问当前缓存的错误。它可提供 `xrtErrorGetLast()` 和 `xrtErrorGetString()` API 以检索系统级异步错误。

```
/**
 * xrtErrorGetLast - Get the last error code and its timestamp of a given
error class.
 *
 * @handle:       Device handle.
 * @class:        Error Class for the last error to get.
 * @error:        Returned XRT error code.
 * @timestamp:    The timestamp when the error generated
 *
 * Return:        0 on success or appropriate XRT error code.
 */
int
xrtErrorGetLast(xrtDeviceHandle handle, xrtErrorClass ecl, xrtErrorCode*
error, uint64_t* timestamp);

/**
 * xrtErrorGetString - Get the description string of a given error code.
 *
 * @handle:       Device handle.
 * @error:        XRT error code.
 * @out:          Preallocated output buffer for the error string.
 * @len:          Length of output buffer.
 * @out_len:      Output of length of message, ignored if null.
 *
 * Return:        0 on success or appropriate XRT error code.
 *
 * Specifying out_len while passing nullptr for output buffer will
 * return the message length, which can then be used to allocate the
 * output buffer itself.
 */
int
xrtErrorGetString(xrtDeviceHandle, xrtErrorCode error, char* out, size_t
len, size_t* out_len);
```

应用可按给定错误类来调用 `xrtErrorGetLast()` 以获取最新错误代码。应用可按给定错误代码调用 `xrtErrorGetString()` 以获取对应于此错误代码的错误字符串。XRT 会维护每个类的最新代码和关联的时间戳（指示错误生成时间）。

`xbutil` 可用于报告错误。错误报告会累积来自先前各类的所有错误，并按时间戳对其进行排序。此报告会查询驱动程序，了解上次请求复位的时间。此复位将合并（使用时间戳）到报告列表中。

```
$ xbutil examine -r error -d 0000:00:00.0
Asynchronous Errors
  Time                            Class
Module              Driver             Severity            Error
Code
  2020-Oct-08 16:40:02              CLASS_SYSTEM
MODULE_FIREWALL     DRIVER_XOCL        SEVERITY_EMERGENCY  FIREWALL_TRIP
```

```
$ xbutil2 examine -r error -f JSON-2020.2 -o <OUTPUT_FILE> -d 0000:00:00.0
{
    "schema_version": {
        "schema": "JSON",
        "creation_date": "Fri Oct  9 11:04:24 2020 GMT"
    },
    "devices": [
        {
            "asynchronous_errors": [
                {
                    "timestamp": "1602175202572070700",
                    "class": "CLASS_SYSTEM",
                    "module": "MODULE_FIREWALL",
                    "severity": "SEVERITY_EMERGENCY",
                    "driver": "DRIVER_XOCL",
                    "error_code": {
                        "error_id": "1",
                        "error_msg": "FIREWALL_TRIP"
                    }
                }
            ]
        }
    ]
}
```

`xbutil` 还可用于报告 AI 引擎运行状态和读取寄存器以便调试。例如，以下命令会在执行 graph 后读取内核状态。

```
$ xbutil examine -r aie -d 0000:00:00.0

---------------------------
1/1 [0000:00:00.0] : edge
---------------------------
Aie
  Aie_Metadata
  GRAPH[ 0] Name : gr
            Status : running
    SNo. Core [C:R] Iteration_Memory [C:R] Iteration_Memory_Addresses
    [ 0] 23:1 23:1 16388
    [ 1] 23:2 23:0 6980
    [ 2] 23:3 23:1 4
    [ 3] 24:1 24:0 4
    [ 4] 24:2 24:2 4
    [ 5] 24:3 24:1 4
    [ 6] 25:1 25:1 4


Core [ 0]
  Column : 23
  Row : 1
  Core:
    Status : core_done
    Program Counter : 0x00000308
    Link Register : 0x00000290
    Stack Pointer : 0x000340a0
  DMA:
    MM2S:
      Channel:
        Id : 0
        Channel Status : idle
        Queue Size : 0
        Queue Status : okay
        Current BD : 0
```

```
        Id : 1
        Channel Status : idle
        Queue Size : 0
        Queue Status : okay
        Current BD : 0

    S2MM:
      Channel:
        Id : 0
        Channel Status : idle
        Queue Size : 0
        Queue Status : okay
        Current BD : 0

        Id : 1
        Channel Status : idle
        Queue Size : 0
        Queue Status : okay
        Current BD : 0

  Locks:
    0 : released_for_write
    1 : released_for_write
    2 : released_for_write
    3 : released_for_write
    4 : released_for_write
    5 : released_for_write
    6 : released_for_write
    7 : released_for_write
    8 : released_for_write
    9 : released_for_write
    10 : released_for_write
    11 : released_for_write
    12 : released_for_write
    13 : released_for_write
    14 : released_for_write
    15 : released_for_write


  Events:
    core : 1, 2, 5, 22, 23, 24, 28, 29, 31, 32, 35, 36, 38, 39, 40, 44, 45,
47, 68
    memory : 1, 43, 44, 45, 106, 113

......


Core [ 6]
  Column : 25
  Row : 1
  Core:
    Status : enabled, east_lock_stall
    Program Counter : 0x000001e6
    Link Register : 0x000000b0
    Stack Pointer : 0x00030020
  DMA:
    MM2S:
      Channel:
        Id : 0
        Channel Status : stalled_on_requesting_lock
        Queue Size : 0
        Queue Status : okay
        Current BD : 2
```

```
        Id : 1
        Channel Status : idle
        Queue Size : 0
        Queue Status : okay
        Current BD : 0

    S2MM:
      Channel:
        Id : 0
        Channel Status : running
        Queue Size : 0
        Queue Status : okay
        Current BD : 0

        Id : 1
        Channel Status : idle
        Queue Size : 0
        Queue Status : okay
        Current BD : 0


  Locks:
    0 : acquired_for_write
    1 : released_for_write
    2 : released_for_write
    3 : released_for_write
    4 : released_for_write
    5 : released_for_write
    6 : released_for_write
    7 : released_for_write
    8 : released_for_write
    9 : released_for_write
    10 : released_for_write
    11 : released_for_write
    12 : released_for_write
    13 : released_for_write
    14 : released_for_write
    15 : released_for_write

  Events:
    core : 1, 2, 5, 22, 26, 28, 29, 31, 32, 35, 38, 39, 44
    memory : 1, 20, 21, 23, 35, 43, 44, 106, 113
```

以下命令可用于读取特定寄存器以便调试。

```
$ xbutil advanced --read-aie-reg -d 0000:00:0 0 25 Core_Status
Register Core_Status Value of Row:0 Column:25 is 0x00000201
```

如需了解 AI 引擎寄存器定义，请参阅《Versal ACAP AI 引擎寄存器参考资料》(AM015)。如需了解有关 `xbutil` 命令使用的详细信息，请参阅 Xilinx Runtime (XRT) 架构。

AI 引擎错误事件

本节提供了有关先前所述使用 XRT 错误报告 API 获取的错误以及这些错误的相关调试信息。这些错误是由 AI 引擎阵列传出的，可用于调试硬件中应用特有的错误。对于 XRT_ERROR_CLASS_AIE 类的错误，您可启用 `dmesg` log 日志来获取更多信息，该日志可提供错误原因，如下表所述。log 日志示例如下：

```
[ 6616.963964] aie aie0: Asserted tile error event 56 at col 6 row 7
[DLBF] Completed reading 4 iterat[ 6616.970234] aie aie0: Asserted tile
error event 56 at col 7 row 8
[ 6616.979187] aie aie0: Asserted tile error event 56 at col 8 row 5
```

注释：请注意，tile（拼块）的位置是以 `col` 和 `row` 编号来表示的。行 0 是 SHIM tile（SHIM 接口拼块），AI 引擎从行 1 开始。

下表列出了各种错误类别、精确的错误数量、错误描述以及有关调试并解决错误的后续步骤的技巧提示。

表 72：**核模块错误事件**

错误分组	编号	名称	描述	调试技巧
Instruction Errors（指令错误）	59	Instruction Decompression Error（指令解压错误）	当 AI 引擎无法解压所提取的指令时发生的事件。如果程序指令损坏，就会发生此错误。确认 ELF 生成。	使用 Vitis 编译器 (V++) `--package` 命令重新生成 ELF 文件。如果问题仍然存在，请联系赛灵思技术支持。
Access Errors（访问错误）	55	PM Reg Access Failure（PM 寄存器访问故障）	AXI 存储器映射接口和 AI 引擎访问 PM 时，如发生 bank 访问冲突，就会出现此错误。	联系赛灵思技术支持。
	60	DM address out of range（DM 地址超出范围）	如果 AI 引擎尝试访问的存储器位置超出 0x20000 – 0x3FFFF 的范围，就会生成此事件。	请运行 AI 引擎仿真器 (`aiesimulator`) 并搭配 `--enable-memory-check`，这将标记所有访问违例。 或者运行 `x86simulator` 搭配 `--valgrind`，这将标记所有访问违例。
	65	PM address out of range（PM 地址超出范围）	PC 超出范围时，就会生成此事件	请运行 AI 引擎仿真器 (`aiesimulator`) 并搭配 `– enable-memory-check`，这将标记所有访问违例。或者运行 `x86simulator` 搭配 `--valgrind`，这将标记所有访问违例。
	66	DM access to unavailable（DM 访问不可用位置）	如果 AI 引擎对近邻中的隔离 tile 发出访问，就会生成此事件。	请检查 AI 引擎上运行的内核是否正在访问另一分区内的隔离 tile 的数据存储器。 如果问题仍然存在，请联系赛灵思技术支持。
Bus Errors（总线错误）	58	AXI MM Slave Error（AXI MM 从接口错误）	如果 AXI 存储器映射接口的从读/写请求的对象是 AI 引擎 tile 中不存在的地址，则会生成此事件。	如果 PL IP 当前使用 AXI 存储器映射接口来访问 AI 引擎寄存器，请检查 PL IP，查看它是否正在访问无效的寄存器。 如果问题仍然存在，请联系赛灵思技术支持。
Stream Errors（串流错误）	54	TLAST in WSS words 0-2（WSS 码字 0-2 中存在 TLAST）	如果 TLAST 并非位于宽串流的第 4 个码字处，就会生成此事件。	如果使用 PL IP 生成串流，请检查它是否正确生成 TLAST。 如果问题仍然存在，请联系赛灵思技术支持。
	56	Stream Pkt Parity Error（串流包奇偶校验错误）	如果包的报头中存在任何奇偶校验错误，就会生成此事件。	请检查数据源（例如，生成包的 PL IP），确认包是否有效，以及是否能正确计算得出奇偶校验位。如果数据来自 PL IP，请检查从该 PL IP 生成的包报头。
	57	Control Pkt Error（控制包错误）	控制包错误	请检查数据源（例如，生成包的 PL IP），确认它是否能正确生成包。 如果问题仍然存在，请联系赛灵思技术支持。

表 72：**核模块错误事件** (续)

错误分组	编号	名称	描述	调试技巧
ECC Errors（ECC 错误）	64	PM ECC Error 2bit（PM ECC 2 位错误）	当检测到 2 位 ECC 错误时，就会生成此事件	重新运行该应用。 如果问题仍然存在，请联系赛灵思技术支持。
	62	PM ECC Error Scrub 2bit（PM ECC 2 位清理）	如果 ECC 清理程序检测到 2 位 ECC 错误，就会生成此事件	重新运行该应用。 如果问题仍然存在，请联系赛灵思技术支持。
Lock Errors（锁定错误）	67	Lock Access to unavailable（锁定访问不可用位置）	如果 AI 引擎对近邻中的隔离 tile 发出访问，就会生成此事件。	请运行 AI 引擎仿真器 (`aiesimulator`) 并搭配 `--enable-memory-check`，这将标记所有访问违例。如果问题仍然存在，请联系赛灵思技术支持。或者运行 `x86simulator` 搭配 `--valgrind`，这将标记所有访问违例。

注释：

1. CORE（核）指的是 AI 引擎 tile 中的 AI 引擎。

表 73：**存储器模块错误事件**

错误分组	编号	名称	描述	调试技巧
ECC Errors（ECC 错误）	88	DM ECC Error Scrub 2bit（DM ECC 2 位清理）	当 ECC 清理程序在 DM 的 bank 0 或 bank 1 中检测到 2 位 ECC 错误时，就会生成此错误。	重新运行该应用。 如果问题仍然存在，请联系赛灵思技术支持。
	90	DM ECC Error 2bit（DM ECC 2 位错误）	在访问 DM 的 bank 0 或 1 期间检测到 2 位 ECC 错误时，就会生成此事件。从 AI 引擎、tile DMA 或 AXI 存储器映射接口访问 DM 都可能导致发生此数据存储器 ECC 错误。	重新运行该应用。 如果问题仍然存在，请联系赛灵思技术支持。

表 73：**存储器模块错误事件** (续)

错误分组	编号	名称	描述	调试技巧
Memory Parity Errors（存储器奇偶校验错误）	91	DM Parity Error Bank 2（DM 奇偶校验错误 bank 2）	在访问 DM bank 2 期间检测到奇偶校验错误时，就会生成此事件。 从 AI 引擎、tile DMA 或 AXI 存储器映射接口访问 DM 都可能导致发生此数据存储器奇偶校验错误。	重新运行该应用。 如果问题仍然存在，请联系赛灵思技术支持。
	92	DM Parity Error Bank 3（DM 奇偶校验错误 bank 3）	在访问 DM bank 3 期间检测到奇偶校验错误时，就会生成此事件。 从 AI 引擎、tile DMA 或 AXI 存储器映射接口访问 DM 都可能导致发生此数据存储器奇偶校验错误。	重新运行该应用。 如果问题仍然存在，请联系赛灵思技术支持。
	93	DM Parity Error Bank 4（DM 奇偶校验错误 bank 4）	在访问 DM bank 4 期间检测到奇偶校验错误时，就会生成此事件。 从 AI 引擎、tile DMA 或 AXI 存储器映射接口访问 DM 都可能导致发生此数据存储器奇偶校验错误。	重新运行该应用。 如果问题仍然存在，请联系赛灵思技术支持。
	94	DM Parity Error Bank 5（DM 奇偶校验错误 bank 5）	在访问 DM bank 5 期间检测到奇偶校验错误时，就会生成此事件。 从 AI 引擎、tile DMA 或 AXI 存储器映射接口访问 DM 都可能导致发生此数据存储器奇偶校验错误。	重新运行该应用。 如果问题仍然存在，请联系赛灵思技术支持。
	95	DM Parity Error Bank 6（DM 奇偶校验错误 bank 6）	在访问 DM bank 6 期间检测到奇偶校验错误时，就会生成此事件。 从 AI 引擎、tile DMA 或 AXI 存储器映射接口访问 DM 都可能导致发生此数据存储器奇偶校验错误。	重新运行该应用。 如果问题仍然存在，请联系赛灵思技术支持。
	96	DM Parity Error Bank 7（DM 奇偶校验错误 bank 7）	在访问 DM bank 7 期间检测到奇偶校验错误时，就会生成此事件。 从 AI 引擎、tile DMA 或 AXI 存储器映射接口访问 DM 都可能导致发生此数据存储器奇偶校验错误。	重新运行该应用。 如果问题仍然存在，请联系赛灵思技术支持。
DMA Errors（DMA 错误）	97	DMA S2MM 0 Error（DMA S2MM 0 错误）	写入 S2MM 通道 0 的 BD 任务队列时，如果此队列已满，则可能导致此错误。	如果您负责管理应用中的缓冲器描述符，请验证您在队列已满时未推送新的缓冲器描述符。 如果问题仍然存在，请联系赛灵思技术支持。
	98	DMA S2MM 1 Error（DMA S2MM 1 错误）	写入 S2MM 通道 1 的 BD 任务队列时，如果此队列已满，则可能导致此错误。	如果您负责管理应用中的缓冲器描述符，请验证您在队列已满时未推送新的缓冲器描述符。 如果问题仍然存在，请联系赛灵思技术支持。
	99	DMA MM2S 0 Error（DMA MM2S 0 错误）	写入 MM2S 通道 0 的 BD 任务队列时，如果此队列已满，则可能导致此错误。	如果您负责管理应用中的缓冲器描述符，请验证您在队列已满时未推送新的缓冲器描述符。 如果问题仍然存在，请联系赛灵思技术支持。
	100	DMA MM2S 1 Error（DMA MM2S 1 错误）	写入 MM2S 通道 1 的 BD 任务队列时，如果此队列已满，则可能导致此错误。	如果您负责管理应用中的缓冲器描述符，请验证您在队列已满时未推送新的缓冲器描述符。 如果问题仍然存在，请联系赛灵思技术支持。

表 74：**SHIM 模块错误事件**

错误分组	编号	名称	描述	调试技巧
Bus Errors（总线错误）	62	AXI MM Slave Tile Error（AXI MM 从拼块错误）	如果 AXI 存储器映射接口从请求进入接口拼块，但地址无效，则会生成此事件。	如果使用 PL IP 通过 AXI 存储器映射接口访问 AI 引擎寄存器，请检查此 IP 是否会尝试访问错误的地址。 如果问题仍然存在，请联系赛灵思技术支持。
	64	AXI MM Decode NSU Error（AXI MM 解码 NSU 错误）	AXI 存储器映射接口流量的内部响应含有 DECERR。例如，如果某一列中的多个 tile 均已采用时钟门控，那么在内部会生成解码器错误并穿越 AXI 存储器映射接口进入接口 tile，从而生成此事件。	如果使用 PL IP 通过 AXI 存储器映射接口访问 AI 引擎寄存器，请检查此 IP 是否会尝试访问已采用门控的 tile。 如果问题仍然存在，请联系赛灵思技术支持。
	65	AXI MM Slave NSU Error（AXI MM 从 NSU 错误）	AXI 存储器映射接口流量的内部响应含有 SLVERR。例如，该接口 tile 列内的某个 AI 引擎 tile 的响应中包含从错误。该从错误将作为从错误，穿越 AXI 存储器映射接口到达接口 tile。	如果使用 PL IP 通过 AXI 存储器映射接口访问 AI 引擎寄存器，请检查此 IP 是否会尝试访问错误的地址。 如果问题仍然存在，请联系赛灵思技术支持。
	66	AXI MM Unsupported Traffic（AXI MM 不受支持的流量）	源自 NoC 的 AXI 存储器映射接口提交了 AI 引擎不支持的请求。	如果使用 PL IP 通过 AXI 存储器映射接口访问 AI 引擎，请检查此 IP 是否会生成不受支持的 AXI 存储器映射接口请求。
	67	AXI MM Unsecure Access in Secure Mode（安全模式下的 AXI MM 不安全访问）	源自 NoC 的 AXI 存储器映射接口当前发生安全模式违例（在 AI 引擎仅支持安全流量的情况下，尝试路由不安全的流量）。	检查 AI 引擎阵列是否是在安全模式下配置的。
	68	AXI MM Byte Strobe Error（AXI MM 字节选通错误）	源自 NoC 的 AXI 存储器映射接口正在以不完整的 32 位码字执行写入（任一 32 位码字内的所有字节选通都必须置位）。	如果 PL IP 正在使用 AXI 存储器映射接口访问 AI 引擎，请检查是否每个 32 位码字中的所有字节选通都已置位。
Stream Error（串流错误）	63	Control Pkt Error（控制包错误）	控制包错误	如果 PL IP 正在生成控制包，请检查此 IP 是否正确生成这些包。 如果问题仍然存在，请联系赛灵思技术支持。

表 74：**SHIM 模块错误事件** (续)

错误分组	编号	名称	描述	调试技巧
DMA Error（DMA 错误）	69	DMA S2MM 0 Error（DMA S2MM 0 错误）	此 DMA 错误对应于 DMA S2MM 通道 0。可能的原因有： · 写入已满的 BD 任务队列； · 尝试访问存储器时发生解码错误 · 尝试访问存储器时发生从错误	如果您负责管理应用中的缓冲器描述符，请验证您在队列已满时未推送新的缓冲器描述符。 如果您负责管理应用中的缓冲器描述符，请检查发送到接口 tile DMA 缓冲器描述符的存储器地址是否有效。 如果问题仍然存在，请联系赛灵思技术支持。
	70	DMA S2MM 1 Error（DMA S2MM 1 错误）	此 DMA 错误对应于 DMA S2MM 通道 1。可能的原因有： · 写入已满的 BD 任务队列； · 尝试访问存储器时发生解码错误 · 尝试访问存储器时发生从错误	如果您负责管理应用中的缓冲器描述符，请验证您在队列已满时未推送新的缓冲器描述符。 如果您负责管理应用中的缓冲器描述符，请检查发送到接口 tile DMA 缓冲器描述符的存储器地址是否有效。 如果问题仍然存在，请联系赛灵思技术支持。
	71	DMA MM2S 0 Error（DMA MM2S 0 错误）	此 DMA 错误对应于 DMA MM2S 通道 0。可能的原因有： · 写入已满的 BD 任务队列； · 尝试访问存储器时发生解码错误 · 尝试访问存储器时发生从错误	如果您负责管理应用中的缓冲器描述符，请验证您在队列已满时未推送新的缓冲器描述符。 如果您负责管理应用中的缓冲器描述符，请检查发送到接口 tile DMA 缓冲器描述符的存储器地址是否有效。 如果问题仍然存在，请联系赛灵思技术支持。
	72	DMA MM2S 1 Error（DMA MM2S 1 错误）	此 DMA 错误对应于 DMA MM2S 通道 1。可能的原因有： · 写入已满的 BD 任务队列； · 尝试访问存储器时发生解码错误 · 尝试访问存储器时发生从错误	如果您负责管理应用中的缓冲器描述符，请验证您在队列已满时未推送新的缓冲器描述符。 如果您负责管理应用中的缓冲器描述符，请检查发送到接口 tile DMA 缓冲器描述符的存储器地址是否有效。 如果问题仍然存在，请联系赛灵思技术支持。

注释：

1. SHIM 是指 AI 引擎阵列中的接口 tile（拼块）。

含 ADF API 和 XRT API 的主机代码参考

本节提供了 XRT API 的汇总信息，这些 API 用于控制 PL 内核与 graph 以及 ADF API 与 XRT API 之间的映射关系。此外还提供了完整的主机代码以供参考，这些主机代码使用 ADF API 或 XRT API 来控制 graph。

注释：本节仅列出部分 API。如需了解有关 XRT API 的最新详细信息，请访问 https://github.com/xilinx/xrt。

表 75：**XRT API**

XRT API	描述
类别：器件句柄 (`experimental/xrt_device.h`)	
`xrtDeviceHandle xrtDeviceOpen(unsigned int index);`	打开器件并获取其句柄。
`xrtDeviceHandle xrtDeviceOpenFromXcl(xclDeviceHandle xhdl);`	从 `xclDeviceHandle` 获取器件处理。
`int xrtDeviceClose(xrtDeviceHandle dhdl);`	关闭已打开的器件。

表 75：**XRT API** (续)

XRT API	描述
`int xrtDeviceLoadXclbinFile(xrtDeviceHandle dhdl, const char* xclbin_fnm);`	读取和加载 XCLBIN 文件。
`void xrtDeviceGetXclbinUUID(xrtDeviceHandle dhdl, xuid_t out);`	获取器件上加载的 XCLBIN 镜像的 UUID。
类别：PL 内核句柄 (`experimental/xrt_kernel.h`)	
`xrtKernelHandle xrtPLKernelOpen(xrtDeviceHandle deviceHandle, const xuid_t xclbinId, const char *name);`	打开 PL 内核并获取其句柄。
`int xrtKernelClose(xrtKernelHandle kernelHandle);`	关闭已打开的内核。
`xrtRunHandle xrtKernelRun(xrtKernelHandle kernelHandle, ...);`	启动内核执行。
`xrtRunHandle xrtRunOpen(xrtKernelHandle kernelHandle);`	为内核打开新的运行句柄，但不启动内核。
`int xrtRunSetArg(xrtRunHandle rhdl, int index, ...);`	为这轮运行设置特定的内核实参。
`int xrtRunUpdateArg(xrtRunHandle rhdl, int index, ...);`	异步更新内核实参。
`int xrtRunStart(xrtRunHandle rhdl);`	启动现有运行句柄。
`enum ert_cmd_state xrtRunWait(xrtRunHandle rhdl);`	等待运行完成。
`int xrtRunClose(xrtRunHandle rhdl);`	关闭运行句柄。
类别：graph 句柄 (`experimental/xrt_graph.h`)	
`xrtGraphHandle xrtGraphOpen(xrtDeviceHandle handle, const uuid_t xclbinUUID, const char *graphName);`	打开 graph 并获取其句柄。
`void xrtGraphClose(xrtGraphHandle gh);`	关闭已打开的 graph。
`int xrtGraphRun(xrtGraphHandle gh, int iterations);`	启动 graph 执行。
`int xrtGraphWait(xrtGraphHandle gh, uint64_t cycle);`	等待指定的 AI 引擎周期数（从上次 `xrtGraphRun` 起），然后停止 graph。如果 `cycle` 为 0，则等待至 graph 完成。如果 graph 运行周期数已超过指定周期数，则立即停止 graph。
`int xrtGraphResume(xrtGraphHandle gh);`	恢复暂挂的 graph。
`int xrtGraphEnd(xrtGraphHandle gh, uint64_t cycle);`	等待指定的 AI 引擎周期数（从上次 `xrtGraphRun` 起），然后终止 graph。如果 `cycle` 为 0，则等待至 graph 完成，然后再将其终止。如果 graph 运行周期数已超过指定周期和结束数，则立即停止 graph，然后将其终止。
`int xrtGraphUpdateRTP(xrtGraphHandle gh, const char *hierPathPort, const char *buffer, size_t size);`	以分层名称更新端口的 RTP 值。
`int xrtGraphReadRTP(xrtGraphHandle gh, const char *hierPathPort, char *buffer, size_t size);`	以分层名称读取端口的 RTP 值。
类别：AIE 句柄 (`experimental/xrt_aie.h`)	
`int xrtAIESyncBO(xrtDeviceHandle handle, xrtBufferHandle bohdl, const char *gmioName, enum xclBOSyncDirection dir, size_t size, size_t offset);`	在 DDR 存储器与接口拼块 DMA 通道之间传输数据。
类别：缓冲器对象句柄 (`experimental/xrt_bo.h`)	
`xrtBufferHandle xrtBOAlloc(xrtDeviceHandle dhdl, size_t size, xrtBufferFlags flags, xrtMemoryGroup grp);`	以相应的标志按请求大小分配 BO。
`int xrtBOFree(xrtBufferHandle bhdl);`	清空/取消分配已分配的 BO。
`int xrtBOSync(xrtBufferHandle bhdl, enum xclBOSyncDirection dir, size_t size, size_t offset);`	按请求方向同步缓冲器内容。
`void* xrtBOMap(xrtBufferHandle bhdl);`	存储器映射 BO 到用户地址空间内。

表 75：**XRT API** (续)

XRT API	描述
类别：错误报告 (`experimental/xrt_error.h`)	
`int xrtErrorGetLast(xrtDeviceHandle handle, xrtErrorClass ecl, xrtErrorCode* error, uint64_t* timestamp);`	获取最新错误代码及其给定错误类的时间戳。
`int xrtErrorGetString(xrtDeviceHandle, xrtErrorCode error, char* out, size_t len, size_t* out_len);`	获取给定错误代码的描述字符串。

下表列出了 ADF API 与 XRT API 之间的映射。`xrtGraphOpen()`、`xrtPLKernelOpen()`、`xrtRunOpen()` 和 `xrtKernelClose()` XRT API 按需在 ADF API 内部进行调用，且不列出对应的映射。

表 76：**ADF API 和 XRT API 映射**

Graph API	XRT API
`graph::run()`	`xrtGraphRun(xrtGraphHandle, 0)`，用于 AI 引擎。
`graph::run(iterations)`	`xrtGraphRun(xrtGraphHandle, iterations)`，用于 AI 引擎。
`graph::wait()`	`xrtGraphWait(xrtGraphHandle, 0)`，用于 AI 引擎。
`graph::wait(aie_cycles)`	`xrtGraphWait(xrtGraphHandle aie_cycles)`，用于 AI 引擎。
`graph::resume()`	`xrtGraphResume(xrtGraphHandle)`
`graph::end()`	`xrtGraphEnd(xrtGraphHandle, 0)`，随后 `xrtGraphClose(xrtGraphHandle)`，用于 AI 引擎。
`graph::end(aie_cycles)`	`xrtGraphEnd(xrtGraphHandle, aie_cycles)`，随后 `xrtGraphClose(xrtGraphHandle)`，用于 AI 引擎。
`graph::update()`	`xrtGraphUpdateRTP()`，用于 AI 引擎；
`graph::read()`	`xrtGraphReadRTP()`，用于 AI 引擎；
`GMIO::malloc()`	`xrtBOAlloc()`，`xrtBOMap()`
`GMIO::free()`	`xrtBOFree()`
`GMIO::gm2aie_nb()`	不适用
`GMIO::aie2gm_nb()`	不适用
`GMIO::wait()`	不适用
`GMIO::gm2aie()`	`xrtSyncBOAIE(...,XCL_BO_SYNC_BO_GMIO_TO_AIE,...)`
`GMIO::aie2gm()`	`xrtSyncBOAIE(...,XCL_BO_SYNC_BO_AIE_TO_GMIO,...)`
`adf::event` API，用于剖析和事件追踪	不适用

以下是使用 ADF API 和 XRT API 的主机代码，以供参考。__USE_ADF_API__ 是代码中用户定义的宏，可用于在 ADF API 与 XRT API 之间进行切换，以控制 AI 引擎 graph。

```
#include <stdlib.h>
#include <fstream>
#include <iostream>
#include "host.h"
#include <unistd.h>
#include <complex>
#include "adf/adf_api/XRTConfig.h"
#include "experimental/xrt_kernel.h"

#include "graph.cpp"

#define OUTPUT_SIZE 2048
```

```
using namespace adf;

int main(int argc, char* argv[]) {

    size_t output_size_in_bytes = OUTPUT_SIZE * sizeof(int);

    //TARGET_DEVICE macro needs to be passed from gcc command line
    if(argc != 2) {
    printf("Usage: %d <xclbin>\r\n",argv[0]);
    return EXIT_FAILURE;
    }
    char* xclbinFilename = argv[1];

    int ret;
    // Open xclbin
    auto dhdl = xrtDeviceOpen(0);//device index=0
    if(!dhdl){
    printf("Device open error\n");
    }
    ret=xrtDeviceLoadXclbinFile(dhdl,xclbinFilename);
    if(ret){
    printf("Xclbin Load fail\n");
    }
    xuid_t uuid;
    xrtDeviceGetXclbinUUID(dhdl, uuid);

    // output memory
    xrtBufferHandle out_bohdl = xrtBOAlloc(dhdl, output_size_in_bytes, 0, /*BANK=*/0);
    std::complex<short> *host_out = (std::complex<short>*)xrtBOMap(out_bohdl);

    // s2mm ip
    xrtKernelHandle s2mm_khdl = xrtPLKernelOpen(dhdl, uuid, "s2mm");
    xrtRunHandle s2mm_rhdl = xrtRunOpen(s2mm_khdl);
    xrtRunSetArg(s2mm_rhdl, 0, out_bohdl);
    xrtRunSetArg(s2mm_rhdl, 2, OUTPUT_SIZE);
    xrtRunStart(s2mm_rhdl);
    printf("run s2mm\n");

#if __USE_ADF_API__
    // update graph parameters (RTP) & run
    adf::registerXRT(dhdl, uuid);
    printf("Register XRT\r\n");
    int narrow_filter[12] = {180, 89, -80, -391, -720, -834, -478, 505, 2063, 3896, 5535,
6504};
    int wide_filter[12] = {-21, -249, 319, -78, -511, 977, -610, -844, 2574, -2754, -1066,
18539};
    gr.run(16);//start AIE kernel
    gr.update(gr.fir24.in[1], narrow_filter, 12);//update AIE kernel RTP
    printf("Update fir24 done\r\n");
    printf("Graph run done\r\n");
    gr.wait(); // wait for AIE kernel to complete
    printf("Graph wait done\r\n");
    gr.update(gr.fir24.in[1], wide_filter, 12);//Update AIE kernel RTP
    printf("Update fir24 done\r\n");
    gr.run(16);//start AIE kernel
    printf("Graph run done\r\n");
#else
    int narrow_filter[12] = {180, 89, -80, -391, -720, -834, -478, 505, 2063, 3896, 5535,
6504};
    int wide_filter[12] = {-21, -249, 319, -78, -511, 977, -610, -844, 2574, -2754, -1066,
18539};
    auto ghdl=xrtGraphOpen(dhdl,uuid,"gr");
    if(!ghdl){
    printf("Graph Open error\r\n);;
    }else{
    printf("Graph Open ok\r\n");;
    }
    int size=1024;
    xrtKernelHandle noisegen_khdl = xrtPLKernelOpen(dhdl, uuid, "random_noise");
    xrtRunHandle noisegen_rhdl = xrtRunOpen(noisegen_khdl);
```

```
    xrtRunSetArg(noisegen_rhdl, 1, size);
    xrtRunStart(noisegen_rhdl);
    printf("run noisegen\n");
    ret=xrtGraphUpdateRTP(ghdl,"gr.fir24.in[1]",(char*)narrow_filter,12*sizeof(int));
    if(ret!=0){
    printf("Graph RTP update fail\n");
    return 1;
    }
    ret=xrtGraphRun(ghdl,16);
    if(ret){
    printf("Graph run error\r\n");
    }else{
    printf("Graph run ok\r\n");
    }
    ret=xrtGraphWait(ghdl,0);
    if(ret){
    printf("Graph wait error\r\n");
    }else{
    printf("Graph wait ok\r\n");
    }
    xrtRunWait(noisegen_rhdl);
    xrtRunSetArg(noisegen_rhdl, 1, size);
    xrtRunStart(noisegen_rhdl);
    printf("run noisegen\n");
    ret=xrtGraphUpdateRTP(ghdl,"gr.fir24.in[1]",(char*)wide_filter,12*sizeof(int));
    if(ret!=0){
    printf("Graph RTP update fail\n");
    return 1;
    }
    ret=xrtGraphRun(ghdl,16);
    if(ret){
    printf("Graph run error\r\n");
    }else{
    printf("Graph run ok\r\n");
    }
#endif
    // wait for s2mm done
    auto state = xrtRunWait(s2mm_rhdl);
    printf("s2mm completed with status %d\r\n",state);

    xrtBOSync(out_bohdl, XCL_BO_SYNC_BO_FROM_DEVICE , output_size_in_bytes,/*OFFSET=*/ 0);

    std::ofstream out("out.txt",std::ofstream::out);
    std::ifstream golden("data/filtered.txt",std::ifstream::in);
    short g_real=0,g_imag=0;
    int match = 0;
    for (int i = 0; i < OUTPUT_SIZE; i++) {
    golden >> std::dec >> g_real;
    golden >> std::dec >> g_imag;
    if(g_real!=host_out[i].real() || g_imag!=host_out[i].imag()){

          printf("ERROR: i=%d gold.real=%d gold.imag=%d out.real=%d out.imag=%d\r
\n",i,g_real,g_imag,host_out[i].real(),host_out[i].imag());
        match=1;
    }
    out<<host_out[i].real()<<" "<<host_out[i].imag()<<" "<<std::endl;
    }
    out.close();
    golden.close();

#if __USE_ADF_API__
    gr.end();
#else
    ret=xrtGraphEnd(ghdl,0);
    if(ret){
    printf("Graph end error"\r\n);;
    }
    xrtRunClose(noisegen_rhdl);
    xrtKernelClose(noisegen_khdl);
    xrtGraphClose(ghdl);
#endif
```

```
    xrtRunClose(s2mm_rhdl);
    xrtKernelClose(s2mm_khdl);
    xrtBOFree(out_bohdl);
    xrtDeviceClose(dhdl);


    char pPass[]= "PASSED";
    char pFail[]= "FAILED";
    char* presult;
    presult = (match ? pFail : pPass);
    printf("TEST %s\r\n",presult);

    return (match ? EXIT_FAILURE :  EXIT_SUCCESS);
}
```

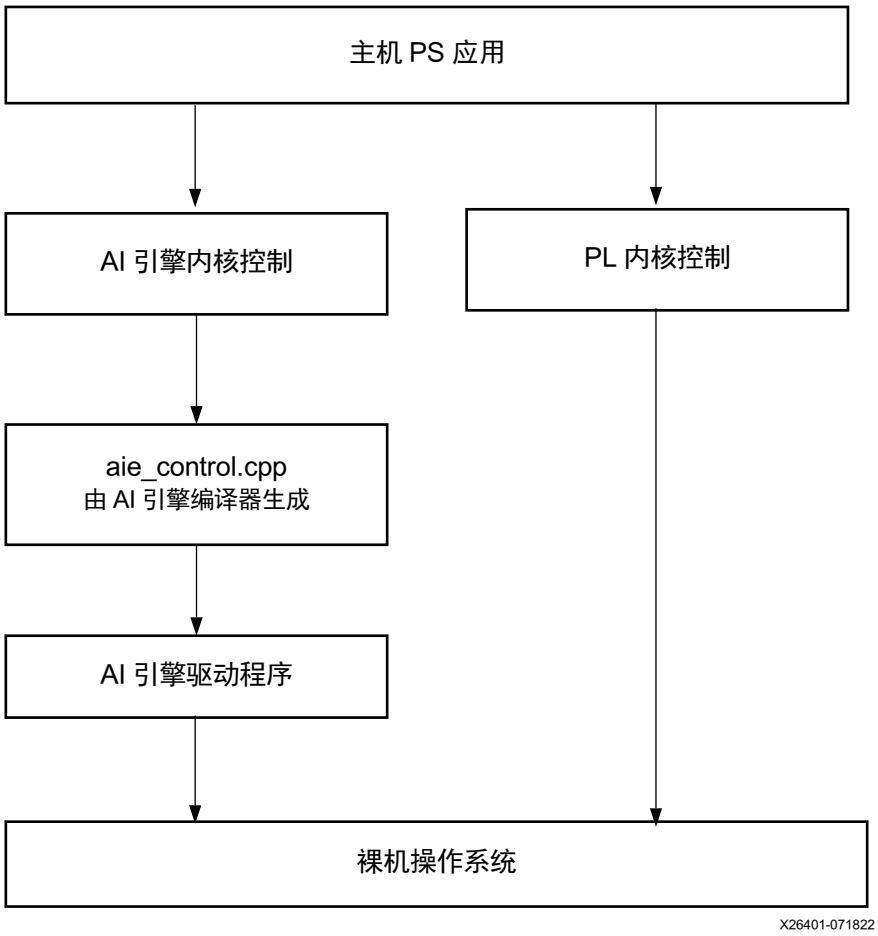
XRT API 具有 C 语言和 C++ 语言版本，用于控制 PL 内核。如需了解有关 XRT API 的 C++ 语言版本的更多信息，请参阅 XRT 本机 API。

裸机的主机编程

在裸机/独立环境中，赛灵思提供了独立的板级支持包 (BSP)、驱动程序和库，以供应用程序用于减少开发工作量。如 Linux 上的主机编程 中所述，裸机系统的顶层应用还必须集成和管理 AI 引擎 graph 与 PL 内核。

提示：如需了解将裸机系统与 AI 引擎 graph 和 PL 内核相集成的步骤，请参阅 构建裸机系统 或 在 Vitis IDE 中构建裸机 AI 引擎。

图 69：**AI 引擎裸机软件栈**



以下提供了裸机系统的顶层应用 (`main.cpp`) 示例：

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <unistd.h>
#include "platform.h"
#include "xparameters.h"
#include "xil_io.h"
#include "xil_cache.h"
#include "input.h"
#include "golden.h"
...
void InitData(int32_t** out, int size)
{
    int i;
    *out = (int32_t*)malloc(sizeof(int32_t) * size);

    if(!out) {
        printf("Allocation of memory failed \n");
        exit(-1);
    }

    for(i = 0; i < size; i++) {
        (*out)[i] = 0xABCDEF00;
    }
}
```

```
int RunTest(uint64_t mm2s_base, uint64_t s2mm_base, int32_t* in, int32_t* golden,
    int32_t* out, int input_size, int output_size)
{
    int i;
    int errCount = 0;
    uint64_t memAddr = (uint64_t)in;
    uint64_t mem_outAddr = (uint64_t)out;

    printf("Starting test w/ cu\n");

    Xil_Out32(mm2s_base + MEM_OFFSET, (uint32_t) memAddr);
    Xil_Out32(mm2s_base + MEM_OFFSET + 4, 0);
    Xil_Out32(s2mm_base + MEM_OFFSET, (uint32_t) mem_outAddr);
    Xil_Out32(s2mm_base + MEM_OFFSET + 4, 0);
    Xil_Out32(mm2s_base + SIZE_OFFSET, input_size);
    Xil_Out32(s2mm_base + SIZE_OFFSET, output_size);
    Xil_Out32(mm2s_base + CTRL_OFFSET, 1);
    Xil_Out32(s2mm_base + CTRL_OFFSET, 1);

    printf("GRAPH INIT\n");
    clipgraph.init();

    printf("GRAPH RUN\n");
    clipgraph.run();

    while(1) {
        uint32_t v = Xil_In32(s2mm_base + CTRL_OFFSET);
        if(v & 6) {
            break;
        }
    }

    printf("PLIO IP DONE!\n");

    for(i = 0; i < output_size; i++) {
        if((((int32_t*)out)[i] != ((int32_t*)golden)[i]) ) {
            printf("Error found in sample %d != to the golden %d\n", i+1, ((int32_t*)out)
[i], ((int32_t*)golden)[i]);
            errCount++;
        }
        else
            printf("%d\n ",((int32_t*)out)[i]);
    }

    printf("Ending test w/ cu\n");
    return errCount;
}

int main()
{
    int i;
    int32_t* out;
    int errCount;

    Xil_DCacheDisable();
    init_platform();
    sleep(1);

    printf("Beginning test\n");
    InitData(&out, OUTPUT_SIZE);
    errCount = RunTest(MM2S_BASE, S2MM_BASE, (int32_t*)cint16input, int32golden, out,
INPUT_SIZE, OUTPUT_SIZE);

    if(errCount == 0)
        printf("Test passed. \n");
    else
```

```
        printf("Test failed! Error count: %d \n",errCount);

    cleanup_platform();
    return errCount;
}
```

此代码示例中的步骤如下所示：

- `main()` 函数用于初始化平台、数据、运行测试、验证返回代码和返回错误代码。
- `InitData()` 用于分配存储器空间的大小 (`size`)，并将成功分配的存储器空间初始化至已知数据。
- `RunTest()` 用于将内核必要的数据传递至进程并返回结果。
- `clipgraph.init()` 用于初始化拼块，内核将在这些拼块上运行。
- `clipgraph.run()` 用于启动在关联拼块上运行内核。

前述代码示例引用从裸机 BSP 自动生成的 `xparameters.h`。应用需确保正确生成裸机 BSP，以便为所有驱动程序正确分配存储器映射地址。

`xil_io.h` 包含通用的驱动程序 I/O API。这是访问驱动程序的首选方法。

在裸机应用内进行内核寻址

对于裸机应用，从嵌入式应用进行 PL 内核寻址时，必须使用控制寄存器，或者在硬件中实现内核的相应基址和偏移处读取和写入该内核。通过观察前述应用，嵌入式应用可将数据交付给 `MM2S` 内核，将其引入 AI 引擎 graph 以供 `Interpolator` 和 `Classifier` 内核使用，并从 `S2MM` 内核读取数据以便继续在嵌入式应用中进行处理。在此例中，按固定平台上 PL 区域中实现 `MM2S` 和 `S2MM` 内核的地址来对这些内核进行寻址。

此示例的 `main.cpp` 显示了特定寄存器的内核基址和地址偏移的 `#define` 语句。例如：

```
#define MM2S_BASE XPAR_MM2S_S_AXI_CONTROL_BASEADDR
#define S2MM_BASE XPAR_S2MM_S_AXI_CONTROL_BASEADDR

#define MEM_OFFSET 0x10
#define SIZE_OFFSET 0x1C
#define CTRL_OFFSET 0x0
```

要判定内核的地址和偏移，请检验固定平台中的部分文档。已实现的内核的基址位置位于固定平台的 `xparameters.h` 文件中，此文件位于 `<platform_name>/standalone_domain/bspinclude/include` 文件夹下。对于设计示例，请在 `xparameters.h` 中使用以下条目来判定这些内核的基址。

```
/* Definitions for peripheral MM2S */
#define XPAR_MM2S_S_AXI_CONTROL_BASEADDR 0xA4020000
#define XPAR_MM2S_S_AXI_CONTROL_HIGHADDR 0xA402FFFF

/* Definitions for peripheral S2MM */
#define XPAR_S2MM_S_AXI_CONTROL_BASEADDR 0xA4030000
#define XPAR_S2MM_S_AXI_CONTROL_HIGHADDR 0xA403FFFF
```

注释： `xparameters.h` 文件采用动态生成和寻址。引用内核的地址宏比硬编码效果更好。

地址偏移的位置位于 Vitis™ 编译器生成的已编译内核的 `_x/<kernel>/<kernel>/<kernel>/solution/impl/ip/drivers/<kernel>_v1_0/src` 文件夹下的 `<kernel_driver>_hw.h` 文件内。例如，`MM2S` 内核驱动程序 `xmm2s_mm2s_hw.h` 会显示以下数据。

```
#define XMM2S_MM2S_CONTROL_ADDR_AP_CTRL     0x00
#define XMM2S_MM2S_CONTROL_ADDR_GIE         0x04
#define XMM2S_MM2S_CONTROL_ADDR_IER         0x08
#define XMM2S_MM2S_CONTROL_ADDR_ISR         0x0c
#define XMM2S_MM2S_CONTROL_ADDR_MEM_V_DATA 0x10
#define XMM2S_MM2S_CONTROL_BITS_MEM_V_DATA 64
#define XMM2S_MM2S_CONTROL_ADDR_SIZE_DATA  0x1c
#define XMM2S_MM2S_CONTROL_BITS_SIZE_DATA  32
```

读取或写入内核时，请使用以下偏移。例如，在 `main.cpp` 应用文件示例中，使用以下语句来写入存储器位置。

```
Xil_Out32(MM2S_BASE + MEM_OFFSET, (uint32_t) memAddr);
```

Linux 与裸机之间的主机编程支持比较

下表对 PetaLinux 和裸机操作系统支持的主机编程功能特性进行了比较。

表 77：**主机编程支持比较**

特征	裸机	PetaLinux
主机应用堆/栈大小配置	是否需要手动调整	由操作系统自动调整
直接访问器件寄存器	受支持	受支持
主机应用 printf() 支持	受支持	受支持
软件仿真支持	不支持	受支持
XRT 支持	不支持	受支持
Xbutil 支持	不支持	受支持
Sysfs 支持	不支持	受支持
多线程主机应用支持	不支持	受支持
多进程主机应用支持	不支持	受支持
事件追踪/剖析 XSDB 流程	受支持	受支持
事件追踪/剖析 XRT 流程	不支持	受支持
操作系统支持	不支持	受支持

AMD XILINX

第 8 章

使用 Vitis 工具流程来集成应用

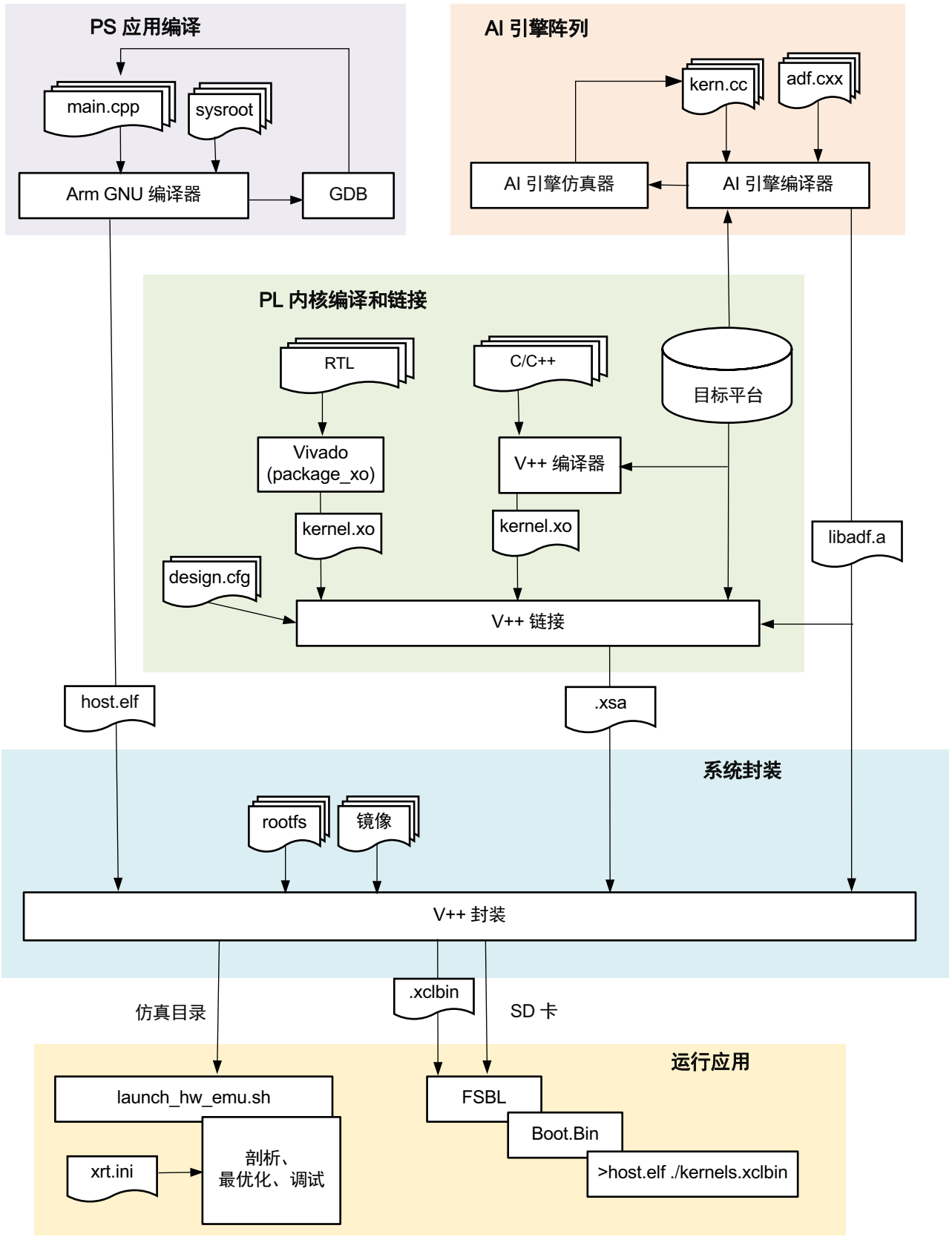
开发 AI 引擎设计 graph 时，通常使用 AI 引擎编译器或 AI 引擎仿真器工具来执行大量设计迭代。此方法能够提供快速设计迭代，同时聚焦 AI 引擎应用的开发。准备就绪后，即可使用本章中所述流程将 AI 引擎设计集成到更大的系统设计中。

Vitis™ 工具流可以凭借类似软件的编译和链接流程来集成 Versal® 器件的下列三大领域，从而简化硬件设计和集成：AI 引擎阵列、可编程逻辑 (PL) 区域和处理器系统 (PS)。Vitis 编译器流程允许您将自己已编译的 AI 引擎设计 graph (`libadf.a`) 与器件的 PL 区域内实现的其它内核（包括 HLS 和 RTL 内核）加以集成，并将其链接在一起以供在目标平台上使用。您可从 Versal 器件内的 Arm® 处理器中运行的主机程序来调用这些已编译的硬件函数。Vitis 编译器会提供抽象指令，用于访问系统存储器、CPU 控制核串流 I/O，因此，通常可以在标准开发平台上开发 AI 引擎 graph 与内核，并将 AI 引擎代码适用目标快速重新调整为针对您的特定应用而开发的定制平台。

下图显示了使用 Vitis 工具流程集成应用所需的高层次步骤。此处描述了用于运行此流程的命令行进程。

注释：您也可以在 Vitis IDE 内使用此流程，如 第 9 章：使用 Vitis IDE 中所述。

图 70：**Vitis 工具流程**



X24782-071922

重要提示！使用 Vitis 工具和 AI 引擎工具需要完成 设置 Vitis 工具环境 中所述的设置。

以下步骤可按 Versal 器件中的任意 AI 引擎设计需求加以调整。

1. 如 第 3 章：对 AI 引擎 Graph 应用进行编译 中所述，第一步是创建 AI 引擎 graph 并使用 AI 引擎编译器将其编译到 `libadf.a` 文件内。您可以在 AI 引擎编译器与 AI 引擎仿真器之间执行迭代以开发 graph，直至您准备好继续为止。
2. PL 内核编译：PL 内核经编译后即可使用 `v++ --compile` 命令在目标平台的 PL 区域中实现。这些内核可以是 C/C++ 内核或 RTL 内核并采用已编译的赛灵思对象 (`xo`) 形式。
3. 系统链接：将已编译的 AI 引擎 graph 与 C/C++ 内核和 RTL 内核链接到目标平台上。此进程会创建 XSA 文件以封装系统。
4. 为 Cortex-A72 处理器编译嵌入式应用：（可选）编译主机应用以在 Cortex®-A72 核处理器上运行，此类处理器使用 GNU Arm 交叉编译器来创建 ELF 文件。该主机程序与 AI 引擎内核和 PL 区域内的内核进行交互。此编译步骤是可选步骤，因为有多种方式可用于部署 AI 引擎内核并与之交互，PS 中运行的主机程序只是其中一种方式。
5. 为硬件封装系统：使用 `v++ --package` 进程收集所需的文件以配置和启动系统、加载和运行应用，包括 AI 引擎 graph 和 PL 内核。这样即可构建必要的封装以运行仿真和调试，或者在硬件上运行您的应用。

平台

平台是全包式镜像，其中定义了硬件 (XSA) 和软件（裸机和/或 Linux）。XSA 包含平台的硬件描述，其描述在 Vivado Design Suite 内定义，软件则使用裸机设计来定义或者使用通过 PetaLinux 定义的 Linux 镜像来定义。

平台类型

下列不同类型的平台针对 CPU 控制、外部存储起、串流 I/O 和时钟设置共享相同的连接抽象层：

- 基础平台：基础平台是由赛灵思提供的平台（例如，`xilinx_vck190_base_202220_1`），通常目标为赛灵思开发板。如果平台被称为基础平台，这表示它是静态平台。对于静态平台，硬件链接结果包含完整的比特配置。系统启动期间，会加载此完整比特。当主机应用执行 XCLBIN 文件时，它仅从 XCLBIN 读取元数据。
- DFX 平台：DFX 平台可在运行时期间动态加载内核 PL 配置。当系统启动时，仅加载硬件平台。当主机应用下载 XCLBIN 文件后，它会运行 DFX 区域的部分重配置，并从 XCLBIN 读取元数据。例如，`xilinx_vck190_base_dfx_202220_1` 平台包含一个 DFX 区域，其中包含 AI 引擎和 PL 内核。
- 定制平台：您可通过扩展或者重新自定义基础平台或基础 DFX 平台来创建定制平台，也可以创建新平台作为定制平台。开始平台开发时，使用来自赛灵思的基础平台或基础 DFX 平台作为参考开发平台有助于创建您的定制平台。

本章以基础平台 `xilinx_vck190_base_202220_1` 为例，为您展示命令的使用方式。如以 DFX 平台 `xilinx_vck190_base_dfx_202220_1` 为目标，请参阅 以 DFX 平台为目标。

定制平台

您可通过自定义现有基本平台（例如，更改 AI 引擎时钟频率、可编程逻辑 (PL) 中可用的时钟、更改存储器控制器设置）来创建平台，或者也可以创建以赛灵思开发板或非赛灵思开发板为目标的新平台。创建平台允许您提供自己的 IP 或子系统以满足您的需求。如需了解有关平台创建流程的信息，请参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的在 Vitis 中创建嵌入式平台。

平台时钟设置

平台具有各种时钟设置：处理器、PL 和 AI 引擎时钟设置。下表显示了上述各种时钟设置。

表 78：**平台时钟设置**

时钟	描述
AI 引擎	可在平台内通过 AI 引擎 IP 来配置。
处理器	可在平台内通过 CIPS IP 来配置。
可编程逻辑 (PL)	可包含多个时钟，并且可在平台内配置。
NoC	与器件相关，可在平台内通过 CIPS 和 NoC IP 来配置。

注释：

1. 这些时钟衍生自平台，并且受到器件、速度等级和工作电压的影响。

如需了解有关平台时钟设置的更多信息，请参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393)。如需了解有关 Versal 器件时钟的信息，请参阅《Versal AI Core 系列数据手册：DC 和 AC 开关特性》(DS957)。

PL 内核

PL 内核可表现为两种形式：以 C/C++ 编写的 HLS 内核，或者封装在 Vivado Design Suite 内的 RTL 内核。这些内核必须单独编译才能生成赛灵思对象文件 (XO)，用于在目标平台上集成系统设计。

以 C/C++ 编写的 HLS 内核可在 Vitis HLS 工具内直接进行编写和编译，或者在 Vitis 应用加速开发流程内进行编写和编译。

如需获取有关创建和构建 RTL 内核的信息，请参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的 RTL 内核。

PL 内核编译

要按《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的使用 Vitis 编译器来编译内核中所述使用 Vitis 编译器命令编译内核，请使用以下命令语法：

```
v++ --compile -t hw_emu --platform xilinx_vck190_base_202220_1 -g \
-k <kernel_name> <kernel>.cpp -o <kernel_name>.xo --save-temps
```

`v++` 命令可使用下表所述的选项。

表 79：**Vitis 编译器选项**

选项	描述
`--compile`	指定编译模式。
`-t hw_emu`	指定编译进程的构建目标。如需了解更多信息，请参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的《构建目标》章节。
`--platform`	指定目标平台的路径和名称。对于此命令行选项示例，假定“PLATFORM_REPO_PATHS”设为正确的平台路径。

表 79：**Vitis 编译器选项** (续)

选项	描述
`-g`	启用调试功能特性。这是仿真模式必需的选项。
`-k`	指定内核名称。此名称必须与指定的内核源文件中的函数名称相匹配。
`-o`	指定已编译的赛灵思对象文件 (`.xo`) 的输出文件名。
`--save-temps`	保存编译进程中生成的临时文件。这是可选项。

PL 内核时钟设置

PLIO 表示对接 PL 的 ADF graph 接口。此 PL 可以是 PL 内核、表示信号源或宿端的平台 IP，或者也可以是用于将 ADF graph 对接到存储器的数据移动器。您应为这些接口提供时钟频率值，以确保仿真结果与在硬件中运行设计的结果相匹配。此外，将 ADF graph 链接到平台时，在 Vitis 连接器 (`v++ -link`) 步骤中，您可指示各工具生成应用所需的精确时钟频率。PL 内核可以单独进行时钟设置，并且如果需要，v++ 连接器将自动确保在设计中插入时钟域交汇电路。如果您不在意具体 PL 时钟频率，则无需指定时钟，各工具将自动选择特定于平台的默认时钟。要在 graph 中设置 PLIO 接口的精确频率以及对应 PL 内核的时钟频率，您必须在三处位置指定时钟频率：

- ADF graph（可选）
- 在 Vitis 编译 PL 内核 (`v++ -c`) 时
- 在 Vitis 链接 (`v++ -l`) 时

您必须根据内核所在位置指定时钟设置。下表描述了基于内核位置的默认时钟。

表 80：**默认内核时钟**

内核位置	描述
AI 引擎内核	按 AI 引擎时钟频率进行时钟设置。所有核都按相同时钟频率来运行。
PL 内核连接到 AI 引擎 graph	HLS：所有 HLS 内核的默认频率为 150 MHz RTL：频率设置为编译 XO 文件时的频率。 AI 引擎：在 AI 引擎 graph 的 PLIO 构造函数内设置。此处设置频率为可选操作。欲知详情，请参阅《AI 引擎内核与 Graph 编程指南》(UG1079) 中的自适应数据流 graph 规范引用。[1]
使用 Vitis 连接器添加到平台的 PL 内核	平台具有默认时钟。如果在命令行或配置文件中未设置任何时钟选项，则使用默认时钟。根据设计和所需时钟值，可覆盖此默认设置，如下表所示。

注释：

1. 如果在 PLIO 构造函数中未提供 PLIO 频率，那么 AI 引擎编译器默认会将频率设为 AI 引擎时钟频率的四分之一。当您确定目标平台后，赛灵思建议您显式设置 PLIO 时钟频率，使 AI 引擎仿真对于您的应用而言更具有代表性。

注释：根据器件速度等级，受支持的最大 PLIO 接口时钟频率是 AI 引擎时钟频率的一半。如果您指定更高的频率，Vitis 连接器将把频率限制为受支持的最大频率，并且将在连接器阶段发出严重警告。

在 Vitis 连接器步骤中设置时钟支持您基于平台选择频率。下表描述了链接步骤期间的 Vitis 编译器时钟设置选项。

表 81：**Vitis 链接时钟选项**

`[clock]` 选项	描述
`--clock.defaultFreqHz arg`	指定要使用的默认时钟频率 (Hz)。
`--clock.defaultId arg`	指定要使用的默认时钟参考 ID。
`--clock.defaultTolerance arg`	指定要使用的默认时钟容限。

表 81：**Vitis 链接时钟选项** (续)

[clock] 选项	描述
`--clock.freqHz arg`	`<frequency_in_Hz>:<cu_0>[.<clk_pin_0>][,<cu_n>[.<clk_pin_n>]]` 指定时钟频率（以 Hz 为单位）、关联计算单元名称列表及其时钟管脚（可选）。
`--clock.id arg`	`<reference_ID>:<cu_0>[.<clk_pin_0>][,<cu_n>[.<clk_pin_n>]]` 指定时钟参考 ID、关联计算单元名称列表及其时钟管脚（可选）。
`--clock.tolerance arg`	`<tolerance>:<cu_0>[.<clk_pin_0>][,<cu_n>[.<clk_pin_n>]]` 指定时钟容限、关联计算单元名称列表及其时钟管脚（可选）。

下表描述了为对接平台的 PLIO 设置时钟频率的步骤，包括与 ADF graph 范围外指定的 PL 内核对接的 PLIO。

表 82：**使用非默认时钟设置编译 PL 内核**

PL 内核位置	时钟规格
ADF graph 中指定的 PLIO 接口	按 graph 中的 PLIO 接口指定时钟频率。 对于 PLIO 接口，您可指定 `FreqMHz`（可选）。 `adf::input_plio <interface_name> = adf::input_plio::create(<logical_name>, <plio_width>, <file>, <FreqMHz>);`

HLS 内核	使用 Vitis 编译器编译 HLS 代码。
	`v++ -c -k kernelName kernel.cpp --hls.clock freqHz:kernelName`
	要更改已编译的 HLS 内核的频率，请使用 `--hls.clock arg:kernelName`。
	`arg` 必须以 Hz 为单位（例如，`250000000Hz` 表示 250 MHz）。
	按内核，在 Vitis 连接器中指定时钟。
	`v++ -l ... --clock.freqHz <freqHz>:kernelName.ap_clk`

RTL 内核	按内核，在 Vitis 连接器中指定时钟。 `v++ -l ... --clock.freqHz <freqHz>:kernelName.ap_clk`

注释： Vitis 连接器阶段的 PL 内核的时钟频率优先于 Vitis 编译时时钟频率值。但 Vitis 连接器阶段所指定的时钟频率值不应明显超过 Vitis 编译器时钟频率过多，因为 Vitis 编译器会基于指定的目标频率来生成 RTL。

如需了解有关如何为特定平台时钟编译内核的更多详细信息以及相关时钟设置信息，请参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393)。

系统链接

完成 AI 引擎 graph 和 C/C++ 内核的编译并封装所有 RTL 内核后，Vitis `v++ --link` 命令会将其与目标平台加以链接，以构建平台文件 (XSA)，用于对设计进行封装。

如需了解更多信息，请参阅《Vitis 统一软件平台文档》的“应用加速开发”流程中的链接内核。

以下提供了 AI 引擎设计流程中的 Vitis 编译器链接命令示例。

```
v++ --link -t hw_emu --platform xilinx_vck190_base_202220_1 -g \
<pl_kernel1>.xo <pl_kernel2>.xo ../libadf.a -o vck190_aie_graph.xsa \
--config ../system.cfg --save-temps
```

v++ 命令可使用下表中的选项。

表 83：**Vitis 编译器链接选项**

选项	描述
--link	指定链接进程。
-t hw_emu	指定链接进程的构建目标。对于 AI 引擎内核流程，目标可设为 hw_emu 用于仿真和测试，或者设为 hw 用于构建系统硬件。 **重要提示！** v++ 编译命令和链接命令必须使用相同构建目标 (-t) 和相同目标平台 (--platform)。
--platform	指定到目标平台的路径。
-g	指定添加调试逻辑以启用调试（用于硬件仿真）和捕获波形数据。
<pl_kernel1>.xo <pl_kernel2>.xo	指定输入已编译的 PL 内核对象文件 (.xo)，与 AI 引擎 graph 和目标平台相链接。
../libadf.a	指定输入已编译的 AI 引擎 graph 应用，与 PL 内核与目标平台相链接。
-o	指定平台 (XSA) 文件，此文件即链接进程的输出。
--config	指定配置文件，以定义部分编译或链接选项。[1]
--save-temps	指示构建进程期间创建的临时文件应予以保留以供后续检验或使用。这包括由 Vitis HLS 和 Vivado Design Suite 创建的输出文件。

注释：

1. --config 选项用于简化 v++ 命令行，方法是将含扩展语法的大量命令移入可从命令行指定的文件内。如需了解更多信息，请参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的 Vitis 编译器配置文件。

提示： config 文件对于命令行的要求不同于 Vitis IDE 的要求，如 配置硬件链接工程 中所述。

对于 AI 引擎内核流程，Vitis 编译器需在配置文件中包含一个特殊小节：[connectivity]。以下是配置文件示例。

```
[connectivity]
nk=mm2s:1:mm2s
nk=s2mm:1:s2mm
stream_connect=mm2s.s:ai_engine_0.DataIn1
stream_connect=ai_engine_0.DataOut1:s2mm.s
```

配置文件的 [connectivity] 节包含下表所述的选项。

表 84：**Connectivity 节的选项**

选项	描述
nk	指定 v++ 命令添加到平台 (XSA) 文件的内核实例或计算单元的数量。 nk 选项可指定内核名称、实例数量或者该内核的计算单元数量以及每个实例的 CU 名称。在此示例中，nk=mm2s:1:mm2s 用于指定 mm2s 内核应仅包含 1 个实例，且该实例应名为 mm2s。 多个内核实例指定为 nk=mm2s:2:mm2s_1.mm2s_2。这表示 mm2s 应包含 2 个计算单元，分别名为 mm2s_1 和 mm2s_2。如需了解更多信息，请参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的创建内核的多个实例。

表 84：**Connectivity 节的选项** (续)

选项	描述
sc	定义 AI 引擎 graph 的端口与 PL 内核的串流端口之间的 AXI4-Stream 连接。连接可定义为将某一个内核的串流输出连接到另一个内核的串流输入，或者连接到目标平台中实现的 IP 的串流输入端口。如需了解更多信息，请参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的 --connectivity 选项。 示例 stream_connect=mm2s.s:ai_engine_0.DataIn1 源自 config 文件，定义的是 mm2s PL 内核的串流输出与 AI 引擎 graph 的 DataIn1 输入之间的连接。 示例 stream_connect=ai_engine_0.DataOut1:s2mm.s 定义的是 AI 引擎 graph 的 DataOut1 输出端口到 PL 内核 s2mm 的输入端口 s 之间的连接。如需了解更多信息，请参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的在计算单元之间指定串流连接。
sp	指定存储器与系统端口之间的连接。该选项可用于指定 GMIO 到特定存储器的连接。sp=toai_engine_0.DataOut2:LPDDR 示例用于指定 GMIO 输出 DataOut2 与 LPDDR 存储器之间的连接。 要获取平台内可用的存储器信息，请运行 platforminfo 命令行实用工具，该工具可用于以结构化格式来报告平台元数据，包括有关接口、时钟、有效的 SLR 和已分配的资源的信息以及有关存储器的信息。

在链接进程期间，Vitis 编译器会调用 Vivado Design Suite 来为目标平台生成平台文件 (XSA)。XSA 文件用于封装设计，并包含下列信息。

- PDI：AI 引擎阵列的编程信息
- Debug data（调试数据）：构建中包含的调试信息
- Memory topology（存储器拓扑结构）：定义目标平台的存储器资源和结构
- IP Layout（IP 布局）：定义已实现的硬件设计的布局信息
- 元数据：平台元数据的各种元素，用于支持工具在平台上加载和运行 XCLBIN 文件

如需了解有关 XRT 使用 XCLBIN 文件的更多信息，请参阅 XRT。

为 Cortex-A72 处理器编译嵌入式应用

完成 AI 引擎 graph 与 PL 内核的链接后，工作焦点转向嵌入式应用，此应用在与 AI 引擎 graph 及内核交互的 PS 中运行。此 PS 应用是以 C/C++ 编写的，它使用 API 调用来控制 AI 引擎 graph 的初始化、运行和关闭，如《AI 引擎内核与 Graph 编程指南》(UG1079) 的运行时 graph 控制 API 中所述。

您遵循适用于 Arm Cortex-A72 处理器的典型交叉编译流程来编译嵌入式应用。以下提供了编译和链接 PS 应用的命令示例：

```
aarch64-xilinx-linux-g++ -std=c++17 -O0 -g -Wall -c \
-I<platform_path>/sysroots/aarch64-xilinx-linux/usr/include/xrt \
--sysroot=<platform_path>/sysroots/aarch64-xilinx-linux/ \
-I./ -I./src -I${XILINX_HLS}/include/ -I${XILINX_VITIS}/aietools/include -o
sw/host.o sw/host.cpp

aarch64-xilinx-linux-g++ -std=c++17 -O0 -g -Wall -c \
-I<platform_path>/sysroots/aarch64-xilinx-linux/usr/include/xrt \
--sysroot=<platform_path>/sysroots/aarch64-xilinx-linux/ \
-I./ -I./src -I${XILINX_HLS}/include/ -I${XILINX_VITIS}/aietools/include -o
sw/aie_control_xrt.o Work/ps/c_rts/aie_control_xrt.cpp
```

前述命令中的诸多选项均为标准选项，可在 g++ 命令描述中找到。以下罗列了更重要的选项。

- `-std=c++17`
- `-I<platform_path>/sysroots/aarch64-xilinx-linux/usr/include/xrt`
- `--sysroot=<platform_path>/sysroots/aarch64-xilinx-linux/`
- `-I./ -I./src`
- `-I${XILINX_HLS}/include/`
- `-I${XILINX_VITIS}/aietools/include`
- `-o sw/host.o sw/host.cpp`

交叉编译器 `aarch64-xilinx-linux-g++` 用于编译 Linux 主机代码。`aie_control_xrt.cpp` 是从 `Work/ps/c_rts` 目录复制的。

```
aarch64-xilinx-linux-g++ -ladf_api_xrt -lgcc -lc -lpthread -lrt -ldl \
-lcrypt -lstdc++ -lxrt_coreutil \
-L<platform_path>/sysroots/aarch64-xilinx-linux/usr/lib \
--sysroot=<platform_path>/sysroots/aarch64-xilinx-linux \
-L${XILINX_VITIS}/aietools/lib/aarch64.o -o sw/host.exe sw/host.o sw/
aie_control_xrt.o
```

请注意，在前述连接器脚本中，链接的是 `adf_api_xrt` 库，这些是 ADF API 搭配 XRT API 一起使用所必需的库。

`xrt_coreutil` 是 XRT 和 XRT API 所必需的库。

虽然在 `g++` 命令描述中可以找到许多选项，在下表中列出了部分更重要的选项。

表 85：**命令选项**

选项	描述
`-ladf_api_xrt`	这是 ADF API 必需的选项。如需了解更多信息，请参阅 Linux 上的主机编程。 它用于通过 XRT 控制 AI 引擎。如果不使用 XRT 进行控制，请使用 `-ladf_api` 搭配 `-L${XILINX_VITIS}/aietools/lib/aarch64none.so` 路径。如需了解更多信息，请参阅 裸机的主机编程。
`-lxrt_coreutil`	这是 XRT API 必需的选项。
`-L<platform_path>/sysroots/aarch64-xilinx-linux/usr/lib`	
`--sysroot=<platform_path>/aarch64-xilinx-linux`	
`-L${XILINX_VITIS}/aietools/lib/aarch64.o`	
`-o sw/host.exe`	

为 x86 处理器编译嵌入式应用

要在软件仿真中为 PS 的 x86 模型编译嵌入式应用，必须使用 GCC 或 g++ 编译器的 x86 版本。

在软件仿真流程中支持使用本机 x86 编译来执行 x86 编译，并且需要 GCC 8.3 或更高版本。此功能特性需要安装 XRT 的 x86 版本，如《Vitis 统一软件平台文档：应用加速开发》(UG1393) 的安装 Xilinx Runtime 和平台中所述。

用于编译主机应用的命令

```
g++ -Wall -c -std=c++17 -Wno-int-to-pointer-cast -I${XILINX_XRT}/include \
-I./src/aie -I./ -I${XILINX_VITIS}/aietools/include \
-o aie_control_xrt.o ./Work/ps/c_rts/aie_control_xrt.cpp $(HOST_SRCS) -o
main.o
```

基于所需 XRT API 链接主机应用并生成可执行文件的命令

```
g++ *.o -lxrt_coreutil -ladf_api_xrt -L${XILINX_VITIS}/aietools/lib/lnx64.o
\
-L${XILINX_XRT}/lib -o $(EXECUTABLE)
```

注释：如上所述，您需要安装 x86 XRT，它会将 `LD_LIBRARY_PATH` 变量自动设置为指向 XRT 库。

如果要在相同设置（终端）上的嵌入式 (ARM-GCC) 流程与 x86 编译流程之间进行切换，需要显式指定 `ARM-GCC` 和 `SYSROOT` 路径，用于为基于 ARM-GCC 的流程编译和链接应用。如果您尝试使用主机应用的交叉编译来运行仿真，同时想要在同一个 shell（终端）上使用主机应用的本机 x86 编译运行软件仿真，则适用此场景。

注释：运行本机 x86 编译后，如果您尝试在同一个终端内使用 ARM-GCC 编译器来编译主机应用，那么您将使用 source 命令找到环境设置脚本 `xilinx-versal-common-v2022.2/environment-setup-cortexa72-cortexa53-xilinx-linux`。确保其中并未设置 `LD_LIBRARY_PATH` 变量，因为如果设置了此变量，则将发出警告。随后，您必须取消设置 `LD_LIBRARY_PATH`，然后重新运行该环境设置脚本。

如需了解有关如何为 AI 引擎内核运行 PS on x86 的信息，请参阅 GitHub 上的使用 x86 PS 执行 AI 引擎加法器嵌入式软件仿真示例。

限制

运行 x86 编译时，在主机代码中不支持仅限 Arm 的数据类型或库。

以下是使用 _fp16（浮点 16）数据类型的主机代码示例，此示例仅在基于 ARM-GCC 的编译器中受支持。x86 编译器在编译此主机代码时会发出错误。在此类情况下，我们建议使用 PS 的 QEMU 模型并使用基于 ARM GCC 的编译器来编译 PS 应用。

```
#define LENGTH (1024)
#define HALF __fp16
int main(int argc, char* argv[])
{
unsigned fileBufSize;
std::string binaryFile = argv[1];
size_t vector_size_bytes = sizeof(HALF) * LENGTH;
//Source Memories
std::vector<HALF> source_a(LENGTH);
std::vector<HALF> source_b(LENGTH);
std::vector<HALF> result_sim (LENGTH);
std::vector<HALF> result_krnl(LENGTH);
/* Create the test data and golden data locally */
for(int i=0; i < LENGTH; i++){
source_a[i] = i;
source_b[i] = i*2;
result_sim[i] = source_a[i] + source_b[i];
}
```

提示：如 为 Cortex-A72 处理器编译嵌入式应用 中所述，您也可以 PS 的 QEMU 模型作为目标，方法是在软件仿真流程中使用 ARM-GCC 编译器来编译嵌入式应用。

迭代 AI 引擎应用编译

AI 引擎应用开发可从系统开发阶段早期开始。AI 引擎开发团队和硬件开发团队能够逐渐合流至可编程逻辑与 AI 引擎阵列之间的接口上。此接口会在某个时间点达到固定状态且不应进行更改，但只要此接口保持不变，AI 引擎就仍能继续演变。

只要此接口不变，硬件和 AI 引擎开发团队就能执行独立的开发进程。完成 AI 引擎应用编译后，编译器会在 `Work` 目录中生成众多文件，用于识别整个进程中的决策。其中，名为 `Work/temp/graph_aie_routed.aiecst` 的文件包含 AI 引擎阵列与 PL 和 PS 之间的所有接口规范（JSON 格式）以及其它信息。`NodeConstraints` 区域包含 graph 中定义的所有接口的描述及其列位置和通道选择。

1. 您可提取此数据并将其存储在另一个 JSON 格式的文件内：

```
{
 "NodeConstraints": {
   "DataIn1": {
     "shim": {
       "column": 24,
       "channel": 0
       }
     },
   "clip_in": {
     "shim": {
       "column": 24,
       "channel": 0
       }
     },
   "clip_out": {
     "shim": {
       "column": 25,
       "channel": 0
     }
   },
   "DataOut1": {
     "shim": {
       "column": 25,
       "channel": 0
     }
   }
 }
}
```

2. 通过将先前提取的接口约束文件提供给编译器，即可对 AI 引擎应用进行修改和重新编译：

```
aiecompiler $(AIE_FLAGS) --workdir=./Work2 --
constraints=interface.aiecst graph.cpp
```

除非您指定其它名称，否则会创建一个新的 `libadf.a`，并且可将其与先前链接阶段由主机可执行文件生成的 `xsa` 直接封装在一起。

您可手动更改此约束文件，无需等待新的系统链接，随后可将 `libadf.a` 文件提供给硬件团队。

如需获取包含执行此任务的所有步骤的教程，请访问：https://github.com/Xilinx/Vitis-Tutorials/tree/2022.1/AI_Engine_Development/Feature_Tutorials/15-post-link-recompile

使用固定硬件平台开发 AI 引擎软件

构建硬件设计之后，AI 引擎/可编程逻辑 (PL) 接口会变为固定状态，但您可基于此固定硬件对 AI 引擎 graph 与内核进行重新编译，且重新编译的频率并无限制。实际上，在此类仅限软件的 AI 引擎编译之间，内核与 graph 可以存在较大差异，前提是 AI 引擎/可编程逻辑接口保持固定不变即可。如果您尝试执行的仅限软件的 AI 引擎编译无法符合固定的 AI 引擎/PL 接口要求，那么 AI 引擎编译器将会发出错误。在系统链接阶段之后，Vitis™ 会生成含 `.xsa` 扩展名的新平台。此文件包含除比特流之外的很多其它信息，尤其是有关 AI 引擎-PL 接口约束的信息。AI 引擎开发者可以基于此软件平台来开发软件。此文件可供 `aiecompiler` 用于自动集成所有约束：

```
aiecompiler -include ... -platform=LinkedPlatform.xsa  newgraph.cpp
```

主机代码也可根据新的 graph 进行调整，经编译后能与新的 `libadf.a` 封装在一起，以生成镜像。

注释：Vitis v++ 连接器和平台接口会抽取 CPU 控制、DDR 存储器和串流 I/O，以便于以标准开发板为目标来构建特定于应用的硬件测试工具，这样您即可按硬件的速度来开发、编译和运行 AI 引擎代码。要将 AI 引擎应用代码目标改为特定于应用的定制平台，只需在设定定制平台目标时，为 AI 引擎重新应用 v++ 连接器指令即可。

封装

AI 引擎编译器以库文件 `libadf.a` 形式生成输出，此库文件中包含 ELF 文件和 CDO 文件以及工具专用的数据和元数据，以供硬件流程和硬件仿真流程使用。要创建可加载的镜像二进制文件，必须将此数据与基于 PL 的配置数据、启动加载程序和其它二进制文件相组合。Vitis™ 封装器会执行该函数，将来自 `libadf.a` 的信息与 Vitis 连接器生成的 XSA 文件相结合。

这需要使用 Vitis 封装命令 (`v++ --package`)，如《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的 Vitis 编译器命令中所述。

对于 Versal ACAP，可编程器件镜像 (PDI) 文件用于启动硬件器件并对其进行编程。对于硬件仿真，`--package` 命令会添加 PDI、“EMULATION_DATA”节和 XSA 文件，并输出 XCLBIN 文件。对于硬件构建，封装进程会为 AI 引擎应用创建 XCLBIN 文件，其中包含 ELF 文件和 graph 配置数据对象 (CDO)。

在 Vitis IDE 中，封装进程会自动执行，工具会基于构建目标、平台和操作系统来创建所需的文件。但在命令行流程中，您必须以正确的选项来为作业指定 Vitis 封装命令 (`v++ --package`)。

为硬件封装系统

对于硬件和硬件仿真，`v++ --package` 命令会取 XSA 文件和 `libadf.a` 作为输入、生成脚本以启动硬件仿真 (`launch_hw_emu.sh`)，并写入所需的支持文件。命令行示例如下所示：

```
v++ --package --config package.cfg ./aie_graph/libadf.a \
./project.xsa -o aie_graph.xclbin
```

其中，`--config package.cfg` 选项可指定含下列选项的配置文件：

```
platform=xilinx_vck190_base_202220_1
target=hw_emu
save-temps=1

[package]
boot_mode=sd
out_dir=./emulation
```

```
enable_aie_debug=1
rootfs=<path_to_platform>/sw/versal/xilinx-versal-common-v2022.2/rootfs.ext4
image_format=ext4
kernel_image=<path_to_platform>/sw/versal/xilinx-versal-common-v2022.2/Image
sd_file=host.exe
```

下表对硬件和硬件仿真的选项进行了解释。

表 86：**硬件和硬件仿真选项**

命令行标志	硬件	硬件仿真	详细信息
platform	目标平台	目标平台	满足 AI 引擎流程要求的基础平台或定制平台。
target	hw	hw_emu	指定硬件仿真构建目标。指定 hw_emu 作为目标会导致生成多个文件，包括用于启动器件的 PDI 和仿真所需的文件。指定 hw 仅生成配置和启动硬件所需的 PDI 文件。
save-temps			使 Vitis 编译器保存在构建和封装进程中所创建的中间文件。
封装选项			
boot_mode[1]	sd	sd	指示器件是从 SD 卡启动还是从闪存中的 QSPI 镜像启动。可能的值包括：sd 或 qspi。
out-dir	<path>	<path>	指定将用于创建输出文件的目录。如不指定 out-dir，则这些文件会写入当前工作目录。
kernel_image	<path>/Image	<path>/Image	指定在链接命令中所指定的镜像文件。对于两种目标，此处的文件应相同。
rootfs	<path>/rootfs.cpio	<path>/rootfs.cpio	指定到 Root FS（根文件系统）文件的路径，此路径必须包含在链接命令中。对于两种目标，此文件应相同。
enable_aie_debug			为 AI 引擎内核生成调试功能特性。可在硬件构建和仿真构建中使用。
defer_aie_run			AI 引擎将由 PS 应用启用。如不设置，则改为在 PDI 加载期间，生成 CDO 命令以启用 AI 引擎。仅当 libadf.a 是输入文件且平台为 Versal 平台时才有效。
ps_elf	<file>,core	<file>,core	仅适用于裸机设计。自动对要运行的 PS 核进行编程。例如，host.elf 或 a72-0
domain	aiengine	aiengine	指定要运行的域。对于 AI 引擎设计，此项应始终设为 aiengine。
sd_file	<file>	<file>	复制将在裸机的 Cortex-A72 处理器上运行的主应用的 ELF，以及在 Linux 上运行所需的任何文件。XCLBIN 文件会被自动复制到 out-dir 或 sd_card 文件夹内。要将更多文件复制到 sd_card 文件夹，必须多次指定该选项。

注释：

1. xilinx_vck190_v202220_1 平台不支持 qspi 选项。配置为支持该选项的定制平台则有效。

下表显示了为硬件和硬件仿真执行构建时生成的 -out-dir 所定义的输出。

表 87：**输出表**

构建	输出

硬件	<pre>|-- BOOT.BIN
	|-- boot_image.bif
	|-- sd_card
	| |-- BOOT.BIN
	| |-- boot.scr
	| |-- aie_graph.xclbin
	| |-- host.exe
	| |-- Image
	|-- sd_card.img</pre>

硬件仿真	<pre>|-- BOOT_bh.bin //Boot header |-- BOOT.BIN //Boot File |-- boot_image.bif |-- launch_hw_emu.sh //Hardware emulation launch script |-- libadf //AIE emulation data folder | `-- cfg |      |-- aie.control.config.json |      |-- aie.partial.aiecompile_summary |      |-- aie.shim.solution.aiesol |      |-- aie.sim.config.txt |      `-- aie.xpe |-- plm.bin //PLM boot file |-- pmc_args.txt //PMC command argument specification file |-- pmc_cdo.bin //PMC boot file |-- qemu_args.txt //QEMU command argument specification file |-- sd_card | |-- BOOT.BIN | |-- boot.scr | |-- aie_graph.xclbin | |-- host.exe | |-- Image |-- sd_card.img `-- sim //Vivado simulation folder</pre>

对于硬件仿真，关键输出文件是用于启动仿真的 `launch_hw_emu.sh` 脚本。`sd_card.img` 镜像包含 `BOOT.BIN`（用于启动 Linux 的 U-Boot、PDI 启动数据等）、镜像（内核镜像）、XCLBIN 文件、用户应用 (`host.exe`) 和其它文件。例如，所有生成的文件都置于名为 `emulation` 的文件夹内。

要在 Linux 主机上使用 `sd_card.img` 文件，请使用 `dd` 命令将此镜像写入 SD 卡。如果以 Linux 为目标但搭配 `package.image_format=fat32` 使用，则请将 `sd_card` 文件夹复制到专为 FAT32 格式化的 SD 卡上。硬件仿真无需此操作。

提示：PS 主机应用包含在 `sd_card` 输出内，但它不会被整合到 `rootfs` 中。如果要将可执行镜像包含在 `rootfs` 中，则必须先重新构建 `rootfs`，然后运行 `v++ --package` 命令。

如果设计需编程到本地闪存，请确保使用 `--package.boot_mode qspi`。这样即可支持使用 `program_flash` 命令或使用 Vitis IDE 对器件或闪存进行编程，如 第 9 章：使用 Vitis IDE 中所述。

为软件封装系统

对于软件仿真 ps_on_x86 流程，无需任何额外的封装选项（`sd_card` 镜像、rootfs 等）。在封装步骤中无需执行 `sd_card` 或基于 Linux 的设置。

表 88：**软件仿真封装选项**

命令行标志	软件仿真 PS on x86	软件仿真 QEMU	详细信息
platform	目标平台	目标平台	满足 AI 引擎流程要求的基础平台或定制平台。
target	sw_emu	sw_emu	指定软件仿真构建目标
save-temps			使 Vitis 编译器保存在构建和封装进程中所创建的中间文件
boot_mode	非必需	sd	指示器件是从 SD 卡启动还是从闪存中的 QSPI 镜像启动。 值可设为：sd 或 qspi。
out-dir	<path>	<path>	指定将用于创建输出文件的目录。如不指定 out-dir，则这些文件会写入当前 工作目录。
Kernel_image	非必需	<path>/Image	指定在链接命令中所指定的镜像文件。 对于两种目标，此处的文件应相同。
rootfs	非必需	<path>/rootfs.cpio	指定到 Root FS（根文件系统）文件的路径，此路径必须包含在链接命令中。 对于两种目标，此文件应相同。
defer_aie_run			AI 引擎将由 PS 应用启用。如不设置，则改为在 PDI 加载期间，生成 CDO 命令以启用 AI 引擎。 仅当 libadf 是输入文件且平台为 Versal 平台时才有效。
sd_file	非必需	<file>	复制将在裸机的 Cortex-A72 处理器上运行的主应用的 ELF，以及在 Linux 上运行所需的任何文件。 XCLBIN 文件会被自动复制到 out-dir 或 sd_card 文件夹内。 要将更多文件复制到 sd_card 文件夹，必须多次指定该选项 。

表 89：**软件仿真输出表**

构建	输出

PS on X86	在 out_dir/sd_dir 中不生成任何输出文件。
	当前目录包含以下文件：`xclbin` 和主机可执行文件。
	`|-- aie_graph.xclbin`
	`| |-- host.exe`

表 89：**软件仿真输出表** (续)

构建	输出

QEMU	<pre>|-- BOOT_bh.bin //Boot header
	|-- BOOT.BIN //Boot File
	|-- boot_image.bif
	|-- launch_sw_emu.sh //Hardware emulation launch script
	|-- libadf
	|-- sd_card
	| |-- BOOT.BIN
	| |-- boot.scr
	| |-- aie_graph.xclbin
	| |-- host.exe
	| |-- Image
	|-- sd_card.img</pre>

构建裸机系统

构建裸机系统在先前所述的标准应用流程基础上需执行多个附加步骤。所需具体步骤如下所述。

1. 构建裸机平台。

   构建裸机应用要求平台内具有裸机域。基础平台 `xilinx_vck190_base_202220_1` 并不具有裸机域，因此您必须创建含裸机域的平台。从 `v++` 链接进程起（如 系统链接 中所述），您必须创建定制平台，因为 PS 应用需要适用于设计中的 PL 内核的驱动程序。

   使用以下命令，以链接进程期间生成的 XSA 创建新平台：

   ```
   generate-platform.sh -name vck190_baremetal -hw <filename>.xsa \
                              -domain psv_cortexa72_0:standalone
   ```

   其中：

   - `-name vck190_baremetal`：指定将创建的平台的名称。此平台将根据指定名称来创建。在此示例中，它将写入：`./vck190_baremetal/export/vck190_baremetal`
   - `-hw <filename>.xsa`：指定 `v++ --link` 命令期间生成的输入 XSA 文件的名称。<filename> 将与针对 `.xclbin` 输出指定的文件名相同。
   - `-domain psv_cortexa72_0:standalone`：指定处理器域和操作系统以供应用于新平台。

   您可在自己的 `$PLATFORM_REPO_PATHS` 环境变量内添加此文件的位置来给自己的平台存储库添加新平台。举例来说，这样此平台即可供 Vitis IDE 访问，或者您可以在命令行中通过直接引用平台名称而不是整个路径的方式来指定平台。

   注释：生成的平台将仅用于在整个流程中构建裸机 PS 应用，而不可作他用。

2. 编译并链接 PS 应用。

   要为裸机流程构建 PS 应用，请使用先前步骤中生成的平台。您需要 PS 应用 (`main.cpp`) 和裸机 AI 引擎控制文件 (`aie_control.cpp`)，此文件是由 `aiecompiler` 命令创建的，可在 `./Work/ps/c_rts` 文件夹下找到。

使用以下命令编译 `main.cpp` 文件：

```
aarch64-none-elf-gcc -I.. -I. -I../src \
-I./vck190_baremetal/export/vck190_baremetal/sw/vck190_baremetal/
standalone_domain/bspinclude/include \
-g -c -std=c++17 -o main.o main.cpp
```

注释：您必须为生成的平台包含 BSP include 文件，这些文件位于：`./vck190_baremetal/export/vck190_baremetal/sw/vck190_baremetal/standalone_domain/bspinclude/include`

使用以下命令编译 `aie_control.cpp` 文件：

```
aarch64-none-elf-gcc -I.. -I. -I../src \
-I./vck190_baremetal/export/vck190_baremetal/sw/vck190_baremetal/
standalone_domain/bspinclude/include \
-g -c -std=c++17 -o aie_control.o ../Work/ps/c_rts/aie_control.cpp
```

使用两个已编译的对象文件来链接 PS 应用：

```
aarch64-none-elf-gcc main.o aie_control.o -g -mcpu=cortex-a72 -Wl,-T -
Wl,./lscript.ld \

-L./vck190_baremetal/export/vck190_baremetal/sw/vck190_baremetal/
standalone_domain/bsplib/lib  \
-ladf_api -Wl,--start-group,-lxil,-lgcc,-lc,-lstdc++,--end-group -o
main.elf
```

注释：链接期间还需要 BSP `libxil.a`，它位于：`./vck190_baremetal/export/vck190_baremetal/standalone_domain/bsplib/lib`。此处假设在平台管理控制器 (PMC) 启动期间，已启用 AI 引擎。

3. 系统封装

最后，您必须运行封装进程生成最终可启动镜像 (PDI)，以供在裸机平台上运行设计。此命令会生成 SD 卡内容，以供启动器件和运行应用。如需了解更多信息，请参阅 封装。这需要使用 `v++ --package` 命令，如下所示：

```
v++ -p -t hw \
    -f xilinx_vck190_base_202220_1 \
    libadf.a project.xsa \
    --package.out_dir ./sd_card \
    --package.domain aiengine \
    --package.defer_aie_run \
    --package.boot_mode sd \
    --package.ps_elf main.elf,a72-0 \
    -o aie_graph.xclbin
```

提示：对于 PS 核上运行的裸机 ELF 文件，还应将 `package.ps_elf` 选项添加到 `--package` 命令。

是否使用 `--package.defer_aie_run` 与 AI 引擎 graph 的运行方式有关。如果在镜像启动时加载并启动应用，则无需这些选项。如果主机应用启动并控制 graph，那么在编译和封装系统时则需要使用这些选项，如 系统部署 中所述。

`./sd_card` 文件夹由 `--out_dir` 选项来指定，其中包含为硬件构建生成的下列文件：

```
|-- BOOT.BIN    //BOOT.BIN file containing PDI and the application ELF
|-- boot_image.bif      //bootgen input file used to create BOOT.BIN
`-- sd_card              //SD card folder
    |-- aie_graph.xclbin     //xclbin output file (not used)
    `-- BOOT.BIN         //BOOT.BIN file containing PDI and the
application ELF
```

请将 `sd_card` 文件夹的内容复制到 SD 卡上以便为系统创建启动器件。

鉴于您已构建了裸机系统，现在您可对其进行运行或调试。

在硬件中运行系统

系统运行取决于构建目标。运行硬件仿真构建的进程不同于运行硬件构建。

对于硬件构建，封装进程会将所生成的 `sd_card` 文件夹内容复制到实际的 SD 卡上。此 SD 卡会成为系统的启动器件。您可按设计的方式启动自己的系统和应用。要在运行硬件时捕获事件追踪数据，请参阅 硬件中的事件追踪。要调试运行硬件，请参阅 第 10 章：调试 AI 引擎应用。

运行软件仿真

在典型 AI 引擎开发流程中，步骤如下。

1. 运行 x86 仿真器来验证 AI 引擎内核与 graph 的功能正确性
2. 对系统进行软件仿真，以验证完整系统的功能正确性
3. 运行 AI 引擎仿真器来验证 AI 引擎内核与 graph 满足应用需求
4. 对整个系统进行硬件仿真
5. 在硬件上执行测试和调试

通常，软件仿真是系统构建和测试的第一步，并采用您创建的定制主机代码而不是仿真器的测试主机代码来完成此功能性进程。

利用 AI 引擎执行系统软件仿真适用于：

- 以有限的已知数据集来检查初始系统行为
- 使用 GDB 对 PS、PL 和 ADF graph 进行功能性集成和调试
- 使用 Python 或 C++ 通过外部流量生成器进行系统测试
- 利用 RTL 内核的 C 语言模型来运行系统
- 通过 `Work/options` 中的 `x86.options` 文件来应用 AI 引擎仿真选项

重要提示！软件仿真要求所有 PL 内核都基于 HLS，或者包含 C/C++ 模型。

注释：

- 编写主机代码以供在软件仿真内执行时，请确保所有缓冲器对象 (`xrtBO`) 都与 `xrtBOSync` API 调用保持同步。
- 要多次运行自由运行的 PL 内核，可在 while (1) 循环内运行。

要为软件仿真构建工程，请确认 `aiecompiler` 的目标选项为 `-target=x86sim`。`v++` 链接和 `v++` 封装命名设为 `-target=sw_emu`。

利用 x86 模型为 PS Arm 处理器运行软件仿真

此 `v++ -p` 命令用于封装设计。除封装外，emconfigutil 命令还应用于生成器件、开发板和设备树详细信息 (`emconfig.json`)。

```
emconfigutil --platform $PLATFORM_REPO_PATHS/xilinx_vck190_base_202220_1/
xilinx_vck190_base_202220_1.xpfm --nd 1
```

随后，

```
cd ./sw
v++ -p -t sw_emu \
    --package.defer_aie_run \
    --platform $PLATFORM_REPO_PATHS/xilinx_vck190_base_202220_1/
xilinx_vck190_base_202220_1.xpfm \
    --package.sd_dir $PLATFORM_REPO_PATHS/sw/versal/xrt \
    --package.out_dir ./package.sw_emu \
    --package.ps_on_x86 \
    --package.sd_file ./emconfig.json \
    ../tutorial.xsa ../libadf.a
```

使用以下命令运行软件仿真

```
setenv XCL_EMULATION_MODE sw_emu
./host_ps_on_x86 a.xclbin
```

利用 QEMU 模型为 PS ARM 处理器运行软件仿真

`v++` 封装命令会在系统封装进程中生成 `launch_sw_emu.sh` 脚本。此脚本会为 AI 引擎应用启动 QEMU 仿真环境，以供测试和调试。软件仿真会为 PL 内核及 AI 引擎内核运行 x86 进程，并为 PS 主机应用运行 QEMU。

请使用以下命令来运行软件仿真：

```
./launch_sw_emu.sh
```

重要提示！对于 HLS 自由运行的内核，HLS 内核代码需更改为在软件仿真中无限运行。请参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的“自由运行的内核”。

要启动 QEMU，需搭配 `sd_card.img` 完成封装，但由于在本节中尚未高亮显示此 `sd_card.img`，因此只需保留如下注释即可：

注释：为了避免在完整 QEMU 上启动 PetaLinux 和启动 QEMU 的开销，您可使用 x86 GCC（代替 ARM-GCC）来编译相同的嵌入式应用。欲知详情，请参阅 为 x86 处理器编译嵌入式应用。

启动软件仿真时，您可以为运行 graph 应用的 `x86simulator` 仿真器指定选项。这些选项可从 `launch_sw_emu.sh` 脚本中使用 `-x86-sim-options` 来指定，如 复用 x86 仿真器选项 中所述。

当仿真完全启动并且出现 Linux 提示符后，请确保在 QEMU 环境中设置以下环境变量。

```
export XILINX_XRT=/usr
export LD_LIBRARY_PATH=/mnt/sd*1:/tmp:$LD_LIBRARY_PATH
export XCL_EMULATION_MODE=sw_emu
```

这样可确保主机应用正常工作。现在即可在 Linux 提示符处执行应用了。

运行硬件仿真

要为硬件仿真构建工程，请确认 `V++` 链接命令的目标选项是 `target=hw_emu`。下一步，`v++ --package` 命令会在系统封装进程中生成 `launch_hw_emu.sh` 脚本。此脚本会为 AI 引擎应用启动仿真环境，以供测试和调试。硬件仿真会为 graph 应用运行 AI 引擎仿真器、为 PL 内核运行 Vivado 逻辑仿真器，并为 PS 主机应用运行 QEMU。

以下命令可用于从命令行启动硬件仿真。

```
./launch_hw_emu.sh --graphic-xsim
```

注释：`--graphic-xsim` 是可选开关，用于启动 Vivado 逻辑仿真器窗口，以便您在其中指定要查看设计中的哪些信号。它不包含内部 AI 引擎信号。您必须在此处单击窗口中的“Run All”（全部运行）按钮才能继续执行。

`launch_hw_emu.sh` 脚本会在系统模式下启动 QEMU、加载并运行 AI 引擎应用，并在 Vivado 仿真器内运行 PL 内核。如果仿真流程成功完成，那么仿真结束时，您应看到如下结果：

```
[LAUNCH_EMULATOR] INFO: 09:44:09 : PS-QEMU exited
[LAUNCH_EMULATOR] INFO: 09:44:09 : PMU/PMC-QEMU exited
[LAUNCH_EMULATOR] INFO: 09:44:09 : Simulation exited
pmu_path /scratch/aie_test1/hw_emu_pmu.log
pl-sim_dir /scratch/aie_test1/sim/behav_waveform/xsim
Please refer PS /simulate logs at /scratch/aie_test1 for more details.
DONE!
INFO: Emulation ran successfully
```

启动硬件仿真时，您可以为运行 graph 应用的 AI 引擎仿真器指定选项。这些选项可从 `launch_hw_emu.sh` 脚本中使用 `-aie-sim-options` 来指定，如 复用 AI 引擎仿真器选项 中所述。

当仿真完全启动并且出现 Linux 提示符后，请确保在 QEMU 环境中设置以下环境变量。

```
export XILINX_XRT=/usr
export LD_LIBRARY_PATH=/mnt/sd*1:
export XCL_EMULATION_MODE=hw_emu
```

这样可确保主机应用正常工作。请注意，此操作同样必须在硬件上运行时完成。

生成流量用于软硬件仿真

概述

本节旨在描述如何使用 AXI Traffic Generator 在所有仿真模式下对 AI 引擎阵列提供输入并从中捕获输出。在 AI 引擎仿真器中，输入数据激励是使用 PLIO 对象提供的，此对象用于指定包含数据的文本文件：

```
input_plio plin = input_plio::create("DataIn", adf::plio_32_bits, "data/
input.txt");
```

虽然这样可使您的首次仿真快速就位，但此方法的主要限制在于，如果您要更改输入文件名用于其它仿真，就必须重新编译整个应用。因此可通过如下方法来避免文件名规范并依靠外部流量生成器在 PLIO 上生成数据流量：

```
input_plio plin = input_plio::create("DataIn", adf::plio_32_bits);
```

对于软硬件仿真，存在一项等效的功能特性可用于对此 PLIO 和 AXI4-Stream 接口的行为进行仿真。为此提供了 Python 和 C++ API 用于创建外部流量生成器，这些流量生成器将在任意仿真模式下无缝连接。

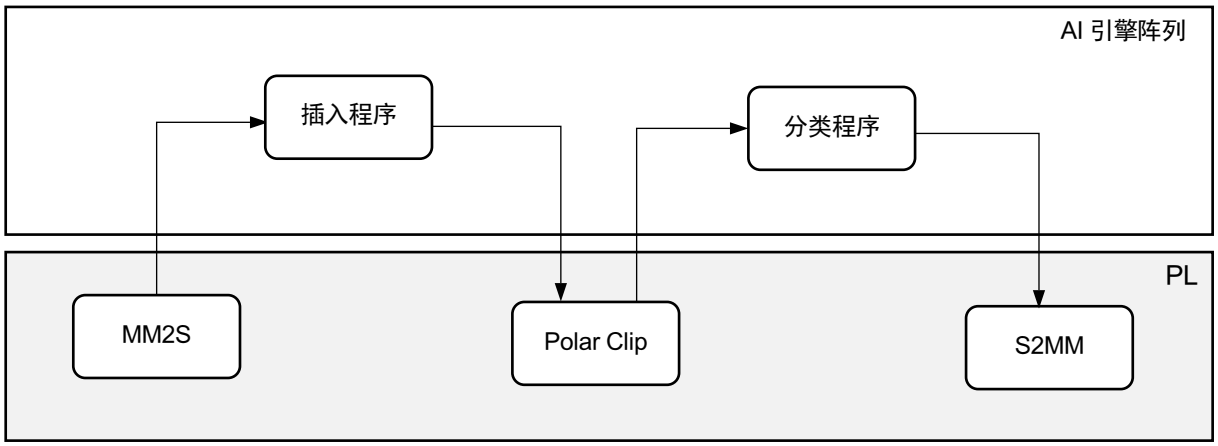
AI 引擎阵列的主要外部数据接口是 AXI4-Stream 接口。这些接口称为 PLIO，允许 AI 引擎检索数据、对数据进行操作并通过独立 AXI4-Stream 接口发回数据。AI 引擎的输入接口是 AXI4-Stream 使用者接口，输出则是 AXI4-Stream 生产者接口。为了在软硬件仿真期间与这些顶层接口进行交互，另外提供了补充性的 AXI4-Stream 模块。这些补充性模块称为 AXI Traffic Generator。

注释：PLIO 接口的宽度是一项重要的系统级别设计决策。此接口越宽，每个 PL 时钟周期内可发送的数据就越多。

标准用例

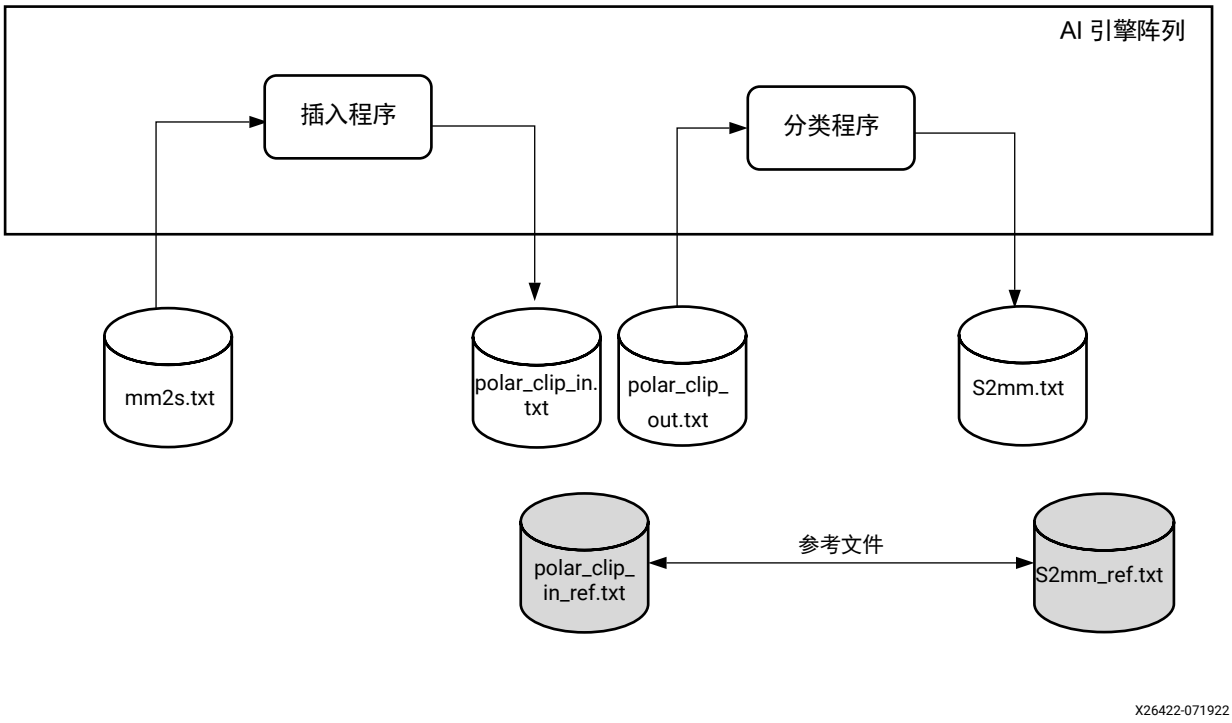
如果您开发 AI 引擎应用并且希望通过 simulation 仿真（x86sim 或 aiesim）或者 emulation 仿真（`sw_emu` 或 `hw_emu`）对其进行测试，那么您需要提供输入数据并收集输出数据，以便将这些数据与部分预定义的参考文件进行比较。此外，如果您的 AI 引擎 graph 与位于可编程逻辑（HLS C++ 或 RTL）中的内核交错，那么您还需要处理这些数据流中断。一般应用如下所示：

图 71：**一般 AI 引擎 + 可编程逻辑应用**



在第一步中，您将 AI 引擎阵列中不包含的所有连接都替换为文本文件以生成流量：

图 72：**初始仿真框架**



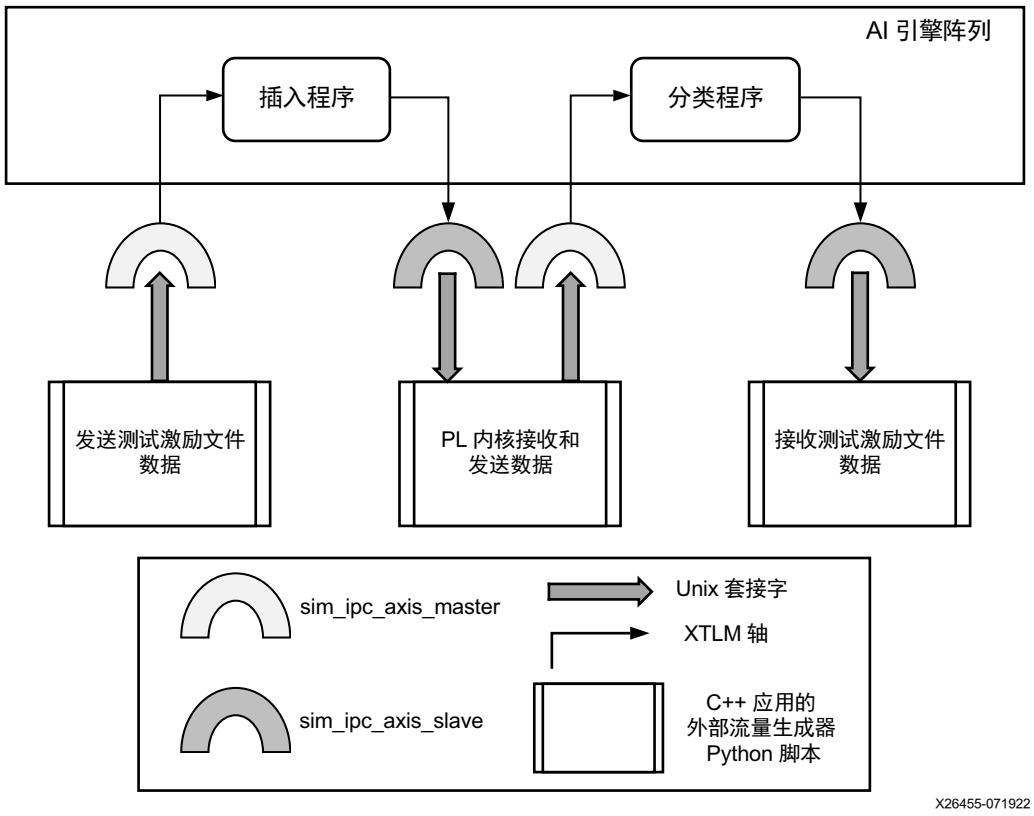
为了提升数据生成和验证的灵活性，您可以将必须通过流量生成器来独立管理的所有文本文件进行交换，从而在 PL 与 AI 引擎阵列之间通过连接到 Unix 套接字的 AXI4-Stream TLM 来达成动态仿真的通信。这些外部流量生成器的优势在于，它们无需修改即可直接在所有仿真框架内使用：

- x86 仿真
- AI 引擎仿真
- 软件仿真

· 硬件仿真

整体仿真框架如下：

图 73：**基于外部流量生成器的仿真**



AI 引擎 graph 修改

在 graph 中无需进行任何有关内核连接的更改。外部流量生成器的调整是在 graph 的 I/O 处完成的，只需更改这部分内容即可，方法是将其从寻常基于文本文件的格式更改为能避免声明文本文件名称的新编写方式即可：

```
plin = input_plio::create("DataIn1",adf::plio_32_bits);
clip_in = output_plio::create("clip_in",adf::plio_32_bits);
clip_out = input_plio::create("clip_out",adf::plio_32_bits);
plout = output_plio::create("DataOut1",adf::plio_32_bits);
```

输入/输出 plio 声明的第一个参数的重要性在于，它所表示的名称将在流量生成器一侧用于连接到正确的套接字。

一旦以 Python 或 C++ 编写完流量生成器之后，即可立即启动 x86 仿真和 AI 引擎仿真。流量生成器还能以 HDL 编写。

启动仿真包括将 `aiesimulator` 或 `x86simulator` 与外部流量生成器并行运行。

PL 内核更改

在思考 AI 引擎应用及其适用于系统仿真模式（软件和硬件）的环境时，需要对往来可编程逻辑一侧的数据传输进行建模。开始开发时，内核尚未准备就绪，无法在 sw_emu 或 hw_emu 框架内使用，因此无法创建赛灵思对象 (.xo) 文件以供在 Vitis 链接阶段中使用。我们必须在可编程逻辑内引入挂钩，以便将外部流量生成器连接到这些挂钩。赛灵思提供了一整套预编译的 .xo 文件，可用于满足此目的：

- $(XILINX_VITIS)/data/emulation/XO/sim_ipc_axis_slave_32.xo, $(XILINX_VITIS)/data/emulation/XO/sim_ipc_axis_master_32.xo
- $(XILINX_VITIS)/data/emulation/XO/sim_ipc_axis_slave_64.xo, $(XILINX_VITIS)/data/emulation/XO/sim_ipc_axis_master_64.xo
- $(XILINX_VITIS)/data/emulation/XO/sim_ipc_axis_slave_128.xo, $(XILINX_VITIS)/data/emulation/XO/sim_ipc_axis_master_128.xo

您应将一组正确的 .xo 文件复制到工程上的正确位置，以便在 Vitis 链接阶段内，将其用于您的配置文件中。

链接阶段

在 Vitis 链接阶段 (v++ -l)，先前定义的 .xo 文件将用于把相关计算单元连接到 AI 引擎 graph。system.cfg 配置文件的创建方式可保证计算单元的名称与您在 graph 中为 input_plio 和 output_plio 定义的名称相匹配：

```
[connectivity]
nk=sim_ipc_axis_master:2:DataIn1.clip_out
nk=sim_ipc_axis_slave:2:DataOut1.clip_in


sc=DataIn1.M00_AXIS:ai_engine_0.DataIn1
sc=ai_engine_0.clip_in:clip_in.S00_AXIS
sc=clip_out.M00_AXIS:ai_engine_0.clip_out
sc=ai_engine_0.DataOut1:DataOut1.S00_AXIS
```

凭借此命名措施，即可确保所有仿真都使用同一个流量生成器。

对于软件仿真 (sw_emu) 和硬件仿真 (hw_emu)，如果您熟悉 RTL 编码，即可采用 C++、Python 或 HDL 来编写外部流量生成器。

主机代码

主机代码的创建十分简单。由于其中并没有可编程逻辑内核，因此查找和运行 PL 内核的所有阶段以及为所有缓冲器对象分配存储器的部分都可跳过。剩余阶段仅包括：

- 打开器件
- 加载 xclbin 文件
- 寄存 XRT 以连接到设计
- 运行 AI 引擎 graph

编译主机代码后，即可对整个工程进行封装。运行仿真包括并行运行外部流量生成器和标准仿真启动。

AXI Traffic Generator

AXI Traffic Generator 是作为 XO 文件来提供的，这些文件需使用 Vitis 编译器 (v++) 链接至您的仿真平台。这些 XO 文件名为 sim_ipc_axis_master_XY.xo 和 sim_ipc_axis_slave_ZW.xo，其中 XY 和 ZW 对应于 PLIO 接口中的位数。例如，sim_ipc_axis_master_128.xo 提供位宽为 128 位的 AXI4-Stream 主数据总线。接口越宽，PL 就能以更低的时钟频率达成相同的吞吐量，并允许 AI 引擎阵列尽可能增大其存储器带宽。但每个 PLIO 接口拼块位宽均为 64 位，并且属于受限资源。以时钟速度两倍的速度来使用一个 64 位 PLIO 接口所提供的带宽与仅使用单个 PLIO 拼块时的 128 位 PLIO 所提供的带宽相同。这要求 PL 以时钟速度两倍的速度运行，并且最优选择因应用而异。

要将流量生成器与 Vitis 编译器搭配使用，需要两个步骤。首先，在 AI 引擎阵列上的 sim_ipc 模块与其对应的 AXI4-Stream 端口之间建立连接。这通常是在 system.cfg 文件中完成的。示例如下：

```
[connectivity]
nk=sim_ipc_axis_master:1:inst_sim_ipc_axis_master
nk=sim_ipc_axis_slave:1:inst_sim_ipc_axis_slave
stream_connect=sim_ipc_axis_master.M00_AXIS:ai_engine_0.DataIn
stream_connect=ai_engine_0.DataOut:sim_ipc_axis_slave.S00_AXIS
```

连接 sim_ipc_axis XO 文件的语法如下。

```
nk=sim_ipc_axis_master:<Number Of Masters>:<inst_name_1>.<inst_name_2>.<...>
nk=sim_ipc_axis_slave:<Number Of Slaves>:<inst_name_1>.<inst_name_2>.<...>
```

sim_ipc_axis_master/slave 用于指定 XO 文件的类型，实例名称应对于您的应用有意义。

下一步，将 XO 文件添加到 Vitis 链接命令中。请注意，sim_ipc XO 文件只能搭配目标 hw_emu 使用。

```
v++ -l --platform <platform.xpfm> sim_ipc_axis_master_128.xo
sim_ipc_axis_slave_128.xo libadf.a -target hw_emu --config system.cfg
```

如需了解有关如何将 XO 文件与 Vitis 编译器搭配使用的更多信息，请访问 https://github.com/Xilinx/Vitis-Tutorials/tree/master/AI_Engine_Development/Feature_Tutorials/05-AI-engine-versal-integration。

注释：要同时使用多个 AXI4-Stream 主接口，请按需将 system.cfg 文件中的 Number of Masters 字段从 1 更改为尽可能大的值（最大值为 8）。

以 Python 和 C++ 创建流量生成器

概述

同时（并行）启动仿真器和流量生成器 (TG) 即可使用外部流量生成器进行仿真。这些 TG 可以 Python 或 C++ 来编写，并且可使用这两种语言的多线程功能。

以 Python 编写流量生成器

以 Python 编写流量生成器需导入多个库：

```
# Mandatory
import os, sys

import multiprocessing as mp
import threading
import struct

from xilinx_xtlm import ipc_axis_master_util
from xilinx_xtlm import ipc_axis_slave_util
```

```
from xilinx_xtlm import xtlm_ipc

# Optionnal, just for ease of use
import numpy as np
import logging
```

进出 AI 引擎阵列的每个端口都应由作为独立进程启动的函数来处理。首先，应创建阻塞传输实用工具。阻塞传输实用工具将在处理端口的函数中使用：

```
mm2s_util = ipc_axis_master_util("DataIn1")
self.s2mm_util = ipc_axis_slave_util("DataOut1")
```

处理端口的函数（以下示例中的 `mm2s`）应作为独立进程来启动：

```
tx = mp.Process(target=mm2s)
tx.start()
```

函数结束时，应停止该进程：

```
tx.join()
```

这是一个阻塞函数，它会等待函数结束后再停止进程。

存在下列机制用于在父进程与子进程之间进行通信：`pipes`。父进程会声明管道，通信则是使用 `send` 和 `recv` 函数来操作的：

```
parent0, child0 = mp.Pipe()

child0.send(Tx_data)
Rx_data = parent0.recv()
```

如果端口是 AI 引擎到可编程逻辑的端口，那么必须首先从该端口读取数据：

```
self.s2mm_util = ipc_axis_slave_util("DataOut1")
```

`payload` 变量实际采用的是包含多个不同字段的结构：

- `data_length` 是数据的字节数。
- `data` 是数据本身。
- `tlast` 是 TLAST 标志，设为 true 或 false。

如果此端口是可编程逻辑到 AI 引擎的端口，那么必须首先创建包：

```
payload = xtlm_ipc.axi_stream_packet()
```

随后，设置不同字段的值，并使用 `b_transpor` 方法将其发送至 AI 引擎阵列：

```
mm2s_util.b_transport(payload)
```

通过流量生成器以 Python 来格式化数据

要对 AXI4-Stream 传输事务进行仿真，AXI Traffic Generator 需将有效载荷数据分割为相应大小的突发。要发送 PLIO 宽度为 32 位（4 字节）的 128 字节数据，就需要 128 字节/4 字节 = 32 项 AXI4-Stream 传输事务。字节阵列与 AXI 传输事务之间的转换可由 Python 来处理。

Python `struct` 库可提供相应机制以在 Python 数据类型与 C 语言数据类型之间进行转换。尤其是，`struct.pack` 和 `struct.unpack` 函数可根据格式字符串实参对字节阵列进行打包和解包。下表显示了常用 C 语言数据类型和 PLIO 宽度的格式字符串。

如需了解更多信息，请访问：https://docs.python.org/3/library/struct.html

表 90：**C 语言数据类型和 PLIO 宽度的格式字符串**

数据类型	PLIO 宽度	Python 代码片段
cfloat	PLIO32	不适用

	PLIO64	`rVec = np.real(data)`
		`iVec = np.imag(data)`
		`out2column = np.zeros((L,2)).astype(np.single)`
		`out2column.tobytes()`
		`formatString = "<"+str(len(byte_arry)//4)+"f"`

	PLIO128	

cint16	PLIO32	`rVec = np.real(data).astype(np.int16)`
		`iVec = np.imag(data).astype(np.int16)`
		`formatString = "<"+str(len(byte_arry)//2)+"h"`

	PLIO64	
	PLIO128	
int8	PLIO32	`intvec = np.real(data).astype(np.int8)` `formatString = "<"+str(len(byte_arry)//1)+"b"`
	PLIO64	
	PLIO128	
int32	PLIO32	`intvec = np.real(data).astype(np.int32)` `formatString = "<"+str(len(byte_arry)//4)+"i"`
	PLIO64	
	PLIO128	

以 C++ 编写流量生成器

使用 C++ 语言来实现外部流量生成器时，需要各种头文件才能使用某些库。Makefile 依赖关系如下所述：

```
# Libraries directories
PROTO_PATH=$(XILINX_VIVADO)/data/simmodels/xsim/2022.2/lnx64/6.2.0/ext/
protobuf/
IPC_XTLM= $(XILINX_VIVADO)/data/emulation/ip_utils/xtlm_ipc/
xtlm_ipc_v1_0/cpp/src/
IPC_XTLM_INC= $(XILINX_VIVADO)/data/emulation/ip_utils/xtlm_ipc/
xtlm_ipc_v1_0/cpp/inc/
LOCAL_IPC= $(IPC_XTLM)../

LD_LIBRARY_PATH:=$(XILINX_VIVADO)/data/simmodels/xsim/2022.2/
lnx64/6.2.0/ext/protobuf/:$(XILINX_VIVADO)/lib/lnx64.o/Default:$
(XILINX_VIVADO)/lib/lnx64.o/:$(LD_LIBRARY_PATH)

# Kernel directories
PLKERNELS_DIR := ../../pl_kernels
PLKERNELS := $(PLKERNELS_DIR)/polar_clip.cpp
PLHEADERS := $(PLKERNELS_DIR)/polar_clip.hpp $(PLKERNELS_DIR)/s2mm.hpp $
(PLKERNELS_DIR)/mm2s.hpp

# XTLM source files
IPC_SRC := $(LOCAL_IPC)/src/axis/*.cpp $(LOCAL_IPC)/src/common/*.cpp $
(LOCAL_IPC)/src/common/*.cc

# Compiler/linker flags
INC_FLAGS := -I$(LOCAL_IPC)/inc -I$(LOCAL_IPC)/inc/axis/ -I$(LOCAL_IPC)/inc/
```

```
common/ -I$(PROTO_PATH)/include/ -I$(PLKERNELS_DIR) -I$(XILINX_HLS)/include
LIB_FLAGS := -L$(PROTO_PATH)/ -lprotobuf -L$(XILINX_VIVADO)/lib/lnx64.o/ -
lrdizlib -L$(GCC)/../../lib64/ -lstdc++ -lpthread

# Compilation
compile: main.cpp $(PLHEADERS) $(PLKERNELS)
  $(GCC) -g main.cpp $(PLKERNELS) $(IPC_SRC) $(INC_FLAGS) $(LIB_FLAGS) -o
chain
```

适用于处理这些库的头文件包括：

```
# For the traffic generator
#include "xtlm_ipc.h"
#include <thread>
```

根据流量生成器是发射器还是接收器（实际可能两者兼具），套接字声明将不尽相同：

```
# Transmitter Traffic Generator
using b_init_socket =
xtlm_ipc::axis_initiator_socket_util<xtlm_ipc::BLOCKING>;

# Receiver Traffic Generator
using b_targ_socket = xtlm_ipc::axis_target_socket_util<xtlm_ipc::BLOCKING>;
```

在此示例中，类用于处理流量生成器的各项功能：

```
class mm2s
{
std::thread m_thread;
std::unique_ptr<b_init_socket> m_socket_ptr;
int count;

void sock_data_handler()
{
m_socket_ptr = std::make_unique<b_init_socket>(m_sock_name);
std::vector<char> data_to_send;

while (count<512)
  {
  // Create a data to send ot the AI Engine Arra (vector of bytes)
  data_to_send = ...;

  m_socket_ptr->transport(data_to_send,count%128==127?true:false); //
transport(data, tlast),   128 sample frame

  count++;
  }
}


protected :
// Name of the socket
const std::string m_sock_name;

public:
mm2s(const std::string sock_name) :
m_sock_name(sock_name), m_socket_ptr(nullptr),count(0)
{}

void run()
{
```

```
m_thread = std::thread(&mm2s::sock_data_handler, this);
}

// This function allows the user to check for the end of the transmission
int dataTransferred()
{
return(count);
}

// The destructor ends the thread
virtual ~mm2s()
{
std::cout << this->m_sock_name << " before join " << std::endl;
if(m_thread.joinable())
  m_thread.join();
std::cout << this->m_sock_name << " after join " << std::endl;
}
};
```

`main` 函数很简单，它仅用于启动流量生成器的各组件，同时在这些组件之间插入部分延迟，以使系统能够轻松完成初始化：

```
int main(int argc, char *argv[])
{

mm2s chain_1_mm2s("DataIn1");
polar_clip chain_1_pc ("clip_in", "clip_out");
s2mm_chain_1_s2mm("DataOut1");

using namespace std::chrono_literals;

chain_1_mm2s.run();
std::cout << "Started mm2s " << std::endl;
std::this_thread::sleep_for(500ms);

chain_1_pc.run();
std::cout << "Started polar_clip " << std::endl;
std::this_thread::sleep_for(400ms);

chain_1_s2mm.run();
std::cout << "Started s2mm " << std::endl;

# Waits for the end of the simulation (1024 samples received from S2MM
block)
while(chain_1_s2mm.dataTransferred()!=1024)
  {
  // Waits 2s before retesting
  std::this_thread::sleep_for(2s);
  }
return(0)
}
```

C++ 流量生成器的有趣之处在于，HLS 内核一旦创建完成，即可立即使用和测试，无需在 `.xo` 文件内进行综合。这样您无需重新创建 `.xclbin` 文件即可给自己的仿真添加更多真实性和灵活性。

使用 System Verilog/Verilog 创建流量生成器

您可将来自外部 System Verilog /Verilog 流量生成器和测试激励文件的流量驱动至 AI 引擎仿真器或 x86 仿真器。

在集成流量生成器模块前，请在 graph 中声明外部 PLIO。

```
pl_in0 = adf::input_plio::create("in_classifier",adf::plio_32_bits);
out0 = adf::output_plio::create("out_interpolator",adf::plio_32_bits);
```

要在 System Verilog/Verilog 流量生成器与 AI 引擎 graph 的外部 PLIO 之间建立连接，需要 xtlm_ipc SystemC 模块。

外部 AI 引擎封装文件将基于 ADF graph 中的外部 PLIO 声明来生成。请遵循以下步骤来为 AI 引擎生成此封装文件模块。

1. 以下命令可用于执行 ADF graph 编译，以生成 `scsim_config.json` 文件，此文件驻留在 `work/config/scsim_config.json` 目录中。此配置文件包含有关 graph 中声明的 PLIO 的信息。

   ```
   aiecompiler --aiearch=aie --platform=$(PLATFORM) -v -log-level=3 --pl-freq=500 -include=./aie --dataflow --output=graph.json aie/graph.cpp
   ```

2. 此配置文件作为实参传递给 `${XILINX_VITIS}/data/emulation/scripts/gen_aie_wrapper.py` 中可用的 Python 脚本，以生成基于 Verilog 的 AI 引擎封装文件模块。如需了解有关如何生成 AI 引擎封装文件模块的更多详细信息，请参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的外部 RTL 流量生成器和 AI 引擎仿真。

3. 生成 AI 引擎封装文件模块后，您需要在外部测试激励文件中将其例化，以在 System Verilog/Verilog 流量生成器与 AI 引擎 graph PLIO 之间建立连接。如需了解有关如何将封装文件模块集成到外部 RTL 测试激励文件中的详细信息，请参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 的外部 RTL 流量生成器和 AI 引擎仿真中的**在测试激励文件中例化 AI 引擎封装文件**。

建立连接后，即可与 aiesimulation 或 x86 仿真器并行启动 HDL 仿真。如需了解有关如何启动 HDL 仿真的详细信息，请参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的将 RTL 流量生成器与 AI 引擎仿真一起运行。

系统部署

Vitis 设计执行模型具有多个考量因素，这些因素会影响 AI 引擎 graph 加载到开发板、运行、复位和重新加载的操作。根据应用需求，您可以选择在开发板启动时加载 AI 引擎 graph，或者也可以使用 PS 主机应用来加载。此外，您还可以在 graph 完成加载后尽快运行或者也可以延后运行。您可以选择无限运行 graph，或者也可以按固定迭代次数或周期数来运行。

AI 引擎 graph 加载和运行

AI 引擎 graph 可在启动时立即加载并运行，或者也可由 PS 主机应用加载。此外，您还可以选择延后其运行，在 graph 使用 `graph.run() host` API XRT 调用完成加载后再运行。默认情况下，赛灵思平台管理控制器 (PMC) 会加载并运行 graph。但 `v++ --package.defer_aie_run` 选项将允许您延后运行 graph，直至 graph 使用 `graph.run()` API 调用完成加载后再运行。下表中列出了部署选项。

表 91：**部署 AI 引擎 graph**

主机控制	永久运行
指定 `v++ --package.defer_aie_run` 即可阻止 AI 引擎在启动时开始运行。	在 PDI 中将其启用，并允许 graph 永久运行。
使用 `graph.run()` 从 PS 程序启用 graph	

AI 引擎运行迭代

AI 引擎 graph 可运行指定迭代次数或者无限运行。默认情况下，graph 无限运行。您可使用 `graph.run(run_iterations)` 或 `graph.end(cycles)` 来将 graph 运行次数限制为特定迭代次数或特定周期数。请参阅《AI 引擎内核与 Graph 编程指南》(UG1079) 中的运行时 graph 控制 API。

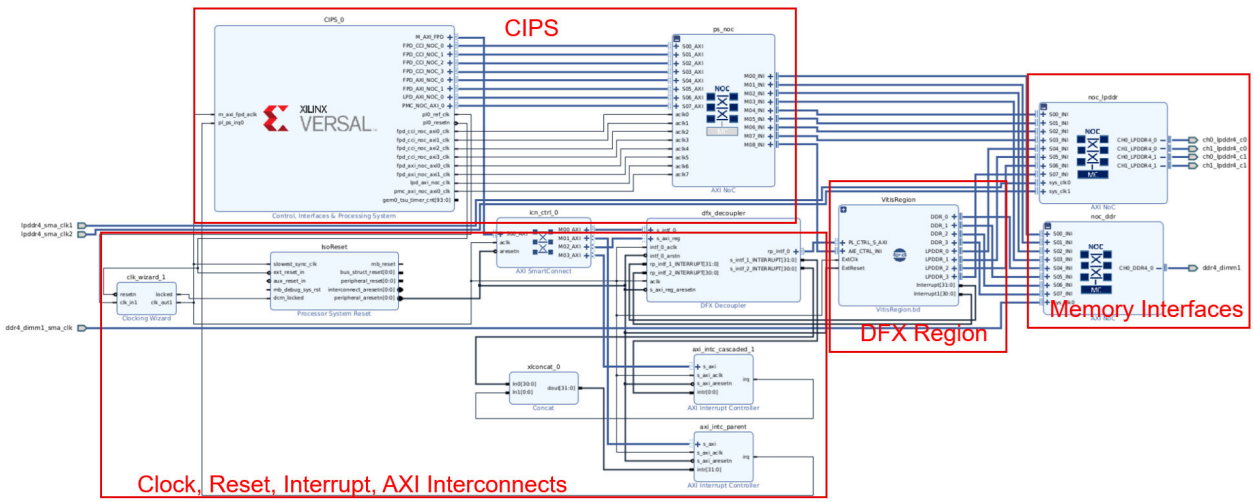
以 DFX 平台为目标

Vitis DFX 平台包含静态区域和 DFX 区域。

- 静态区域：系统启动时加载的硬化的块。此块在运行时不执行重新配置。
- DFX 区域：可重配置分区 (RP)，由 Vivado IP integrator 块设计容器 (BDC) 来实现。重配置模块 (RM) 在运行时可以重复加载。

赛灵思提供了基础 DFX 平台 `xilinx_vck190_base_dfx_202220_1`，它是单一 RP DFX 平台，换言之，它包含一个静态区域和唯一的 DFX 区域（或 RM）。RM 包含 AI 引擎阵列和 PL 内核。加载时，将同时加载整个 AI 引擎阵列和所有 PL 内核。下图显示了这些 DFX 平台的块设计 (BD)。

图 74：**基础 DFX 平台的块设计**



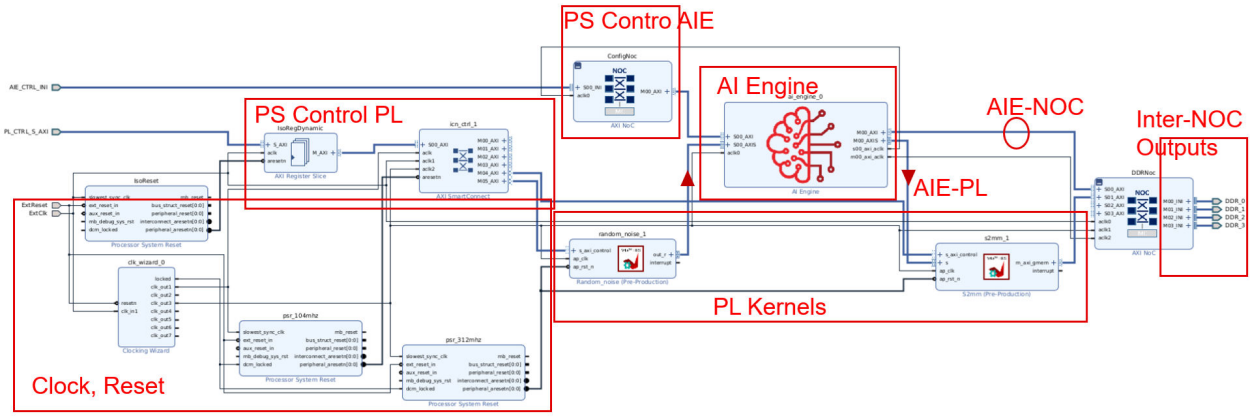
静态分区包含：

- CIPS 和 PS NoC
- 时钟、复位、中断和 AXI 互连

- NoC 接口和 DDR 存储器控制器

DFX 区域包含 AI 引擎阵列和 PL 内核，通过 Vitis 连接器 (`v++ -l`) 将其链接在一起。下图显示了由 Vitis 连接器生成的 DFX 区域示例。

图 75：**DFX 区域的块设计**



DFX 区域和静态区域接口包括：

- 时钟与复位
- NoC 间接口 (INI) 输出至常规存储器
- INI 输入源自 PS，用于控制 AI 引擎
- AXI 接口源自 PS，用于控制 PL 内核

DFX 区域中可包含：

- AI 引擎
- PL 内核
- AXI-NoC IP
- 时钟和复位模块
- AXI Interconnect 模块
- ILA
- FIFO、数据宽度转换器 (DWC) 和 CDC 模块

以 DFX 平台为目标时，在基础平台中可引用 `aiecompiler`、v++ 编译器 (`v++ -c`) 和 v++ 连接器 (`v++ -l`) 命令。DFX 平台中的 v++ 封装器 (`v++ -p`) 有别于基础平台中的封装器。在 DFX 平台的 `hw` 和 `hw_emu` 模式下，v++ 封装器也不同。

硬件部署流程

对于 hw 流程，DFX 平台的 v++ 封装器如下所示：

```
v++ -p -t hw -f xilinx_vck190_base_dfx_202220_1
  --package.defer_aie_run \
  -o rm.xclbin \
  ${XSA} \
  libadf.a

v++ -p -t hw -f xilinx_vck190_base_dfx_202220_1
  --package.rootfs ${ROOTFS}
  --package.kernel_image ${IMAGE} \
  --package.boot_mode=sd \
  --package.image_format=ext4 \
  --package.sd_dir data \
  --package.sd_file ${HOST_EXE} \
  --package.sd_file rm.xclbin
```

其中：

- 第一条 v++ -p 命令用于生成 xclbin 文件，其中包含将在运行时加载的 RM PDI。
- 第二条 v++ -p 命令用于将 RM xclbin 文件 (rm.xclbin) 封装到 SD 卡中，该卡可供主机程序读取。

启动 Linux 后，在命令提示符处输入以下命令：

```
cd /run/media/mmcblk0p1
./host.exe rm.xclbin
```

注释：如果 XRT 检测到已下载同一 XCLBIN，那么默认它将不会再次下载此 XCLBIN。而是改为仅从 xclbin 文件加载元数据。

为确保 XCLBIN 已下载，并且每次都清除器件，请设置 xrt.ini 以强制将 XCLBIN 下载至器件，如下所示：

1. 请将以下配置添加到 xrt.ini 文件中：

   ```
   [Runtime]
   force_program_xclbin=true
   ```

2. 将此文件置于从中运行主机程序的目录内。

这样即可确保 XRT 每次都能将 XCLBIN 下载至器件。

硬件仿真部署流程

对于 hw_emu 流程，v++ 封装器命令如下：

```
emconfigutil --platform xilinx_vck190_base_dfx_202220_1 --nd 1
v++ -p -t hw_emu -f xilinx_vck190_base_dfx_202220_1 \
  --package.defer_aie_run \
  --package.rootfs ${ROOTFS} \
  --package.kernel_image ${IMAGE} \
  --package.boot_mode=sd \
  --package.image_format=ext4 \
  --package.sd_dir data \
```

```
  --package.sd_file ${HOST_EXE} \
  --package.sd_file emconfig.json \
  -o rm.xclbin \
  ${XSA} \
  libadf.a
```

使用以下命令启动仿真：

```
./launch_hw_emu.sh
```

在 QEMU 中，可运行以下命令：

```
cd /run/media/mmcblk0p1
export XCL_EMULATION_MODE=hw_emu
./host.exe rm.xclbin rm.xclbin
```

注释：在 `hw_emu` 模式下，XCLBIN 只能下载一次。无法重新加载 RM。

AMD XILINX

第 9 章

使用 Vitis IDE

Vitis™ 工具流程（如 第 8 章：使用 Vitis 工具流程来集成应用 中所述）在 Vitis IDE 中同样可用。在以下章节中描述了构建系统工程（含 AI 引擎 graph、PL 内核及 PS 应用）中所涉及的不同步骤。

使用 Vitis IDE 前，必须首先设置开发环境，如 设置 Vitis 工具环境 中所述。

创建 AI 引擎 graph 工程和顶层系统工程

Vitis 集成设计环境 (IDE) 可用于构建、运行和调试 AI 引擎程序。本节提供了对应于前述命令行步骤的截屏。

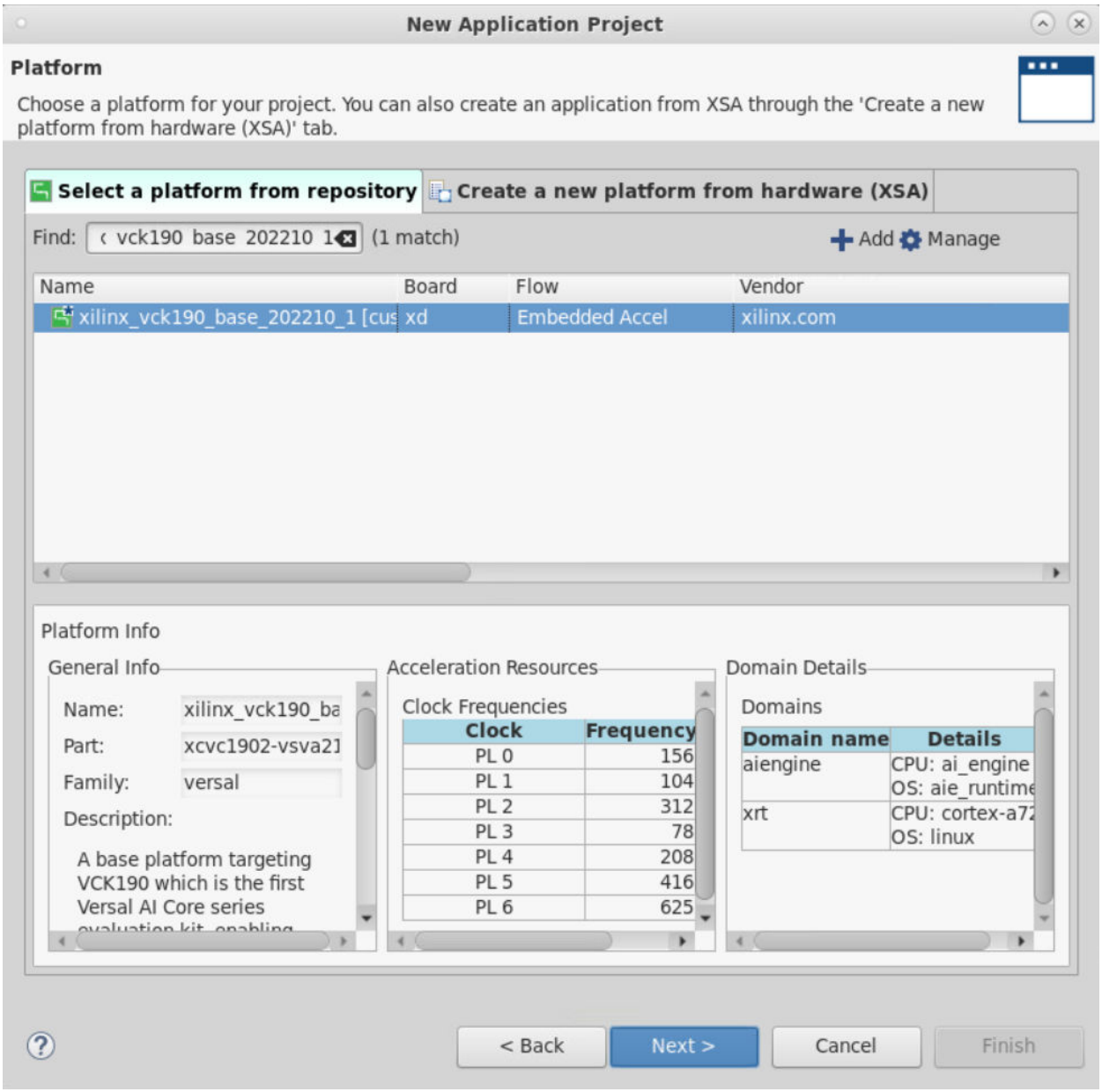
1. 从命令行输入 `vitis` 以启动 Vitis IDE。首次启动时，必须指定工作空间目录用于存储工程。也可以在启动该工具时指定工作空间：

   ```
   vitis -workspace ./myWorkspace
   ```

2. 选择“File” → “New” → “Application Project”（文件 > 新建 > 应用工程），创建新的 AI 引擎工程。

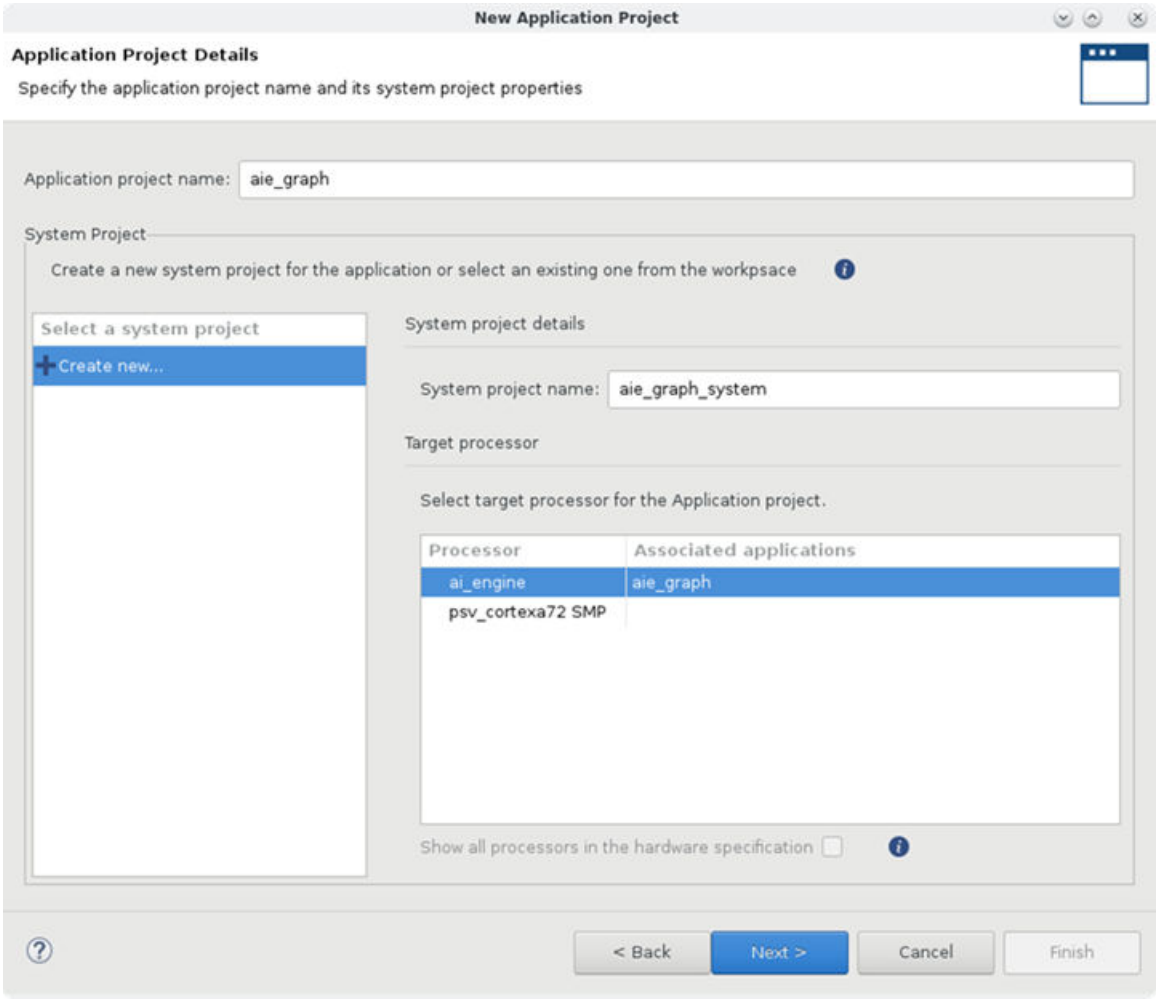
   提示：如果这是您首次启动 Vitis IDE，则会打开“Welcome”（欢迎）屏幕。在此情况下，请单击“Create Application Project”（创建应用工程）。

   这样会打开“New Application Project”Wizard（新建应用工程向导），并显示简介页面，其中描述了用于创建新工程的流程。选择“Next”（下一步）打开“Platform”（平台）选择页面，如下图所示。



选择 `xilinx_vck190_base_202220_1` 平台，然后单击“Next”（下一步）。

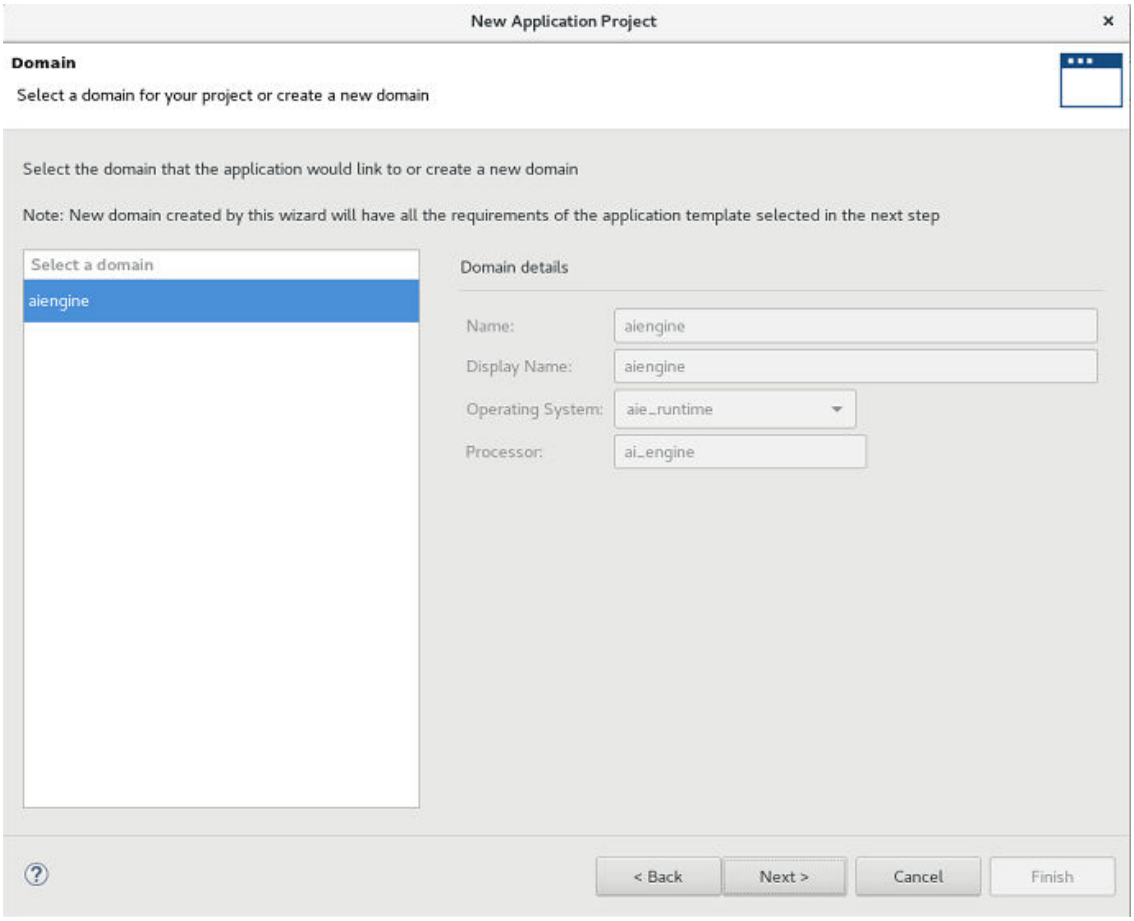
3. 这样会打开“Application Project Details”（应用工程详细信息）页面，如下图所示。



您可在此处指定“Application project name”（应用工程名称）。在“System project”（系统工程）下，您可选择现有系统工程，以便将您自己的应用工程（如果存在）添加到其中，或者也可交由 Vitis IDE 创建新的系统工程。创建新的系统工程时，会基于您指定的应用工程名称自动生成系统工程名称。但您也可以在“Project name”（工程名称）字段中输入新的名称。

下一步，指定要与您的新工程关联的“Processor”（处理器）。在此示例中，当前创建的是 AI 引擎 graph 工程。选择 `ai_engine` 的“Processor”，然后单击“Next”。

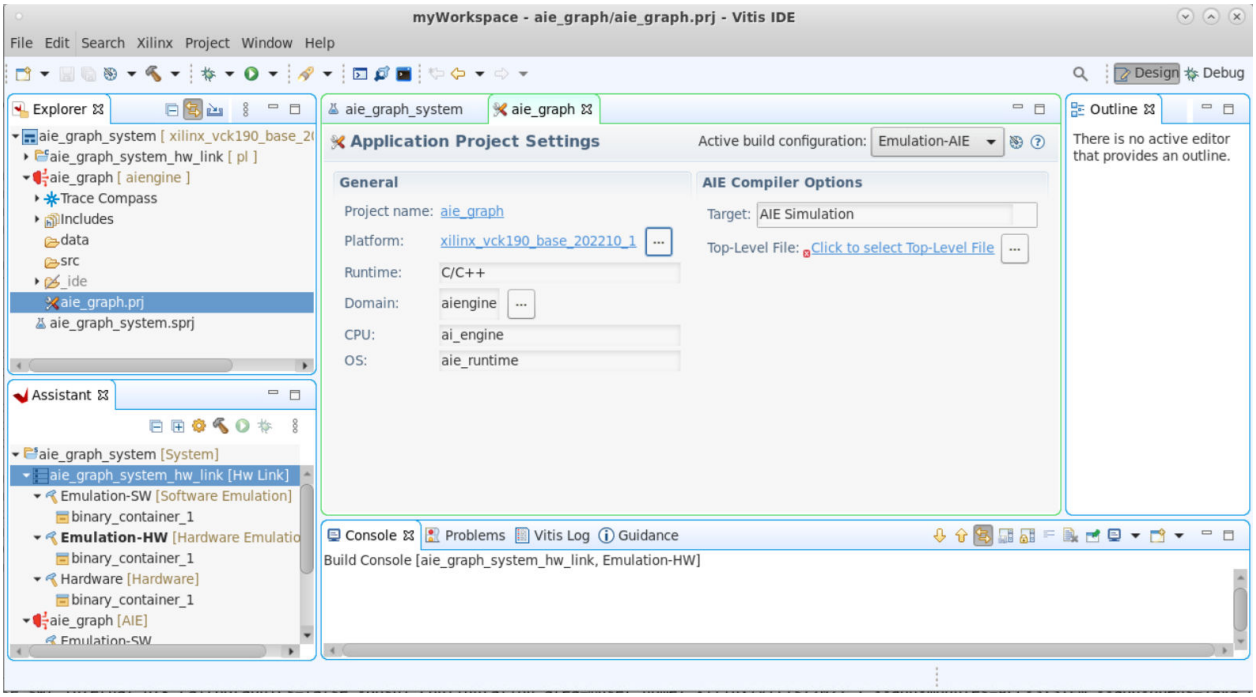
4. 这样会打开“Domain”（域）页面，如下图所示。



在“Domain”（域）选择页面上，您可为新的应用工程指定处理器域。在此示例中，由于已指定 `aiengine` 的“Processor”，因此仅列出一个域并且已预先将其选中。单击“Next”（下一步）以继续。

5. 这样会打开“Templates”（模板）页面，如 Vitis 工具模板示例 中所述。

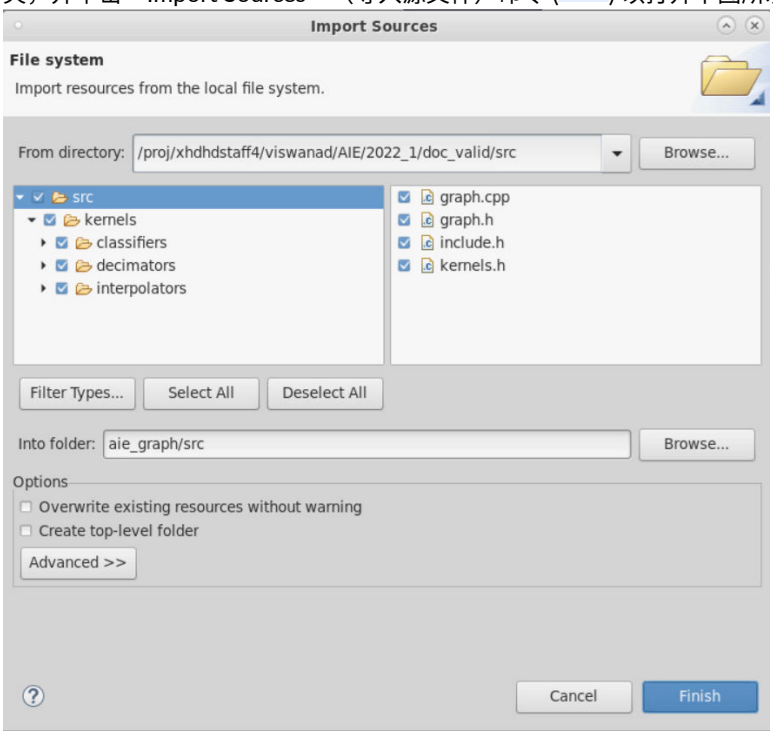
   在此示例中，当前创建的是新定制工程。选择“Empty Application”（空应用）模板（默认），然后单击“Finish”（完成）以创建工程。这样即可创建新的 AI 引擎 graph 工程，Vitis IDE 会在“Design”（设计）透视图中打开此工程。



导入源文件

在 IDE 中打开 AI 引擎 graph 工程后，您即可导入所需的源文件并配置自己的工程。

1. 要为 AI 引擎工程导入源文件，请在“Explorer”（资源管理器）视图中选中工程。展开文件夹，选中 `src` 文件夹，并单击“Import Sources”（导入源文件）命令 ( ) 以打开下图所示对话框。



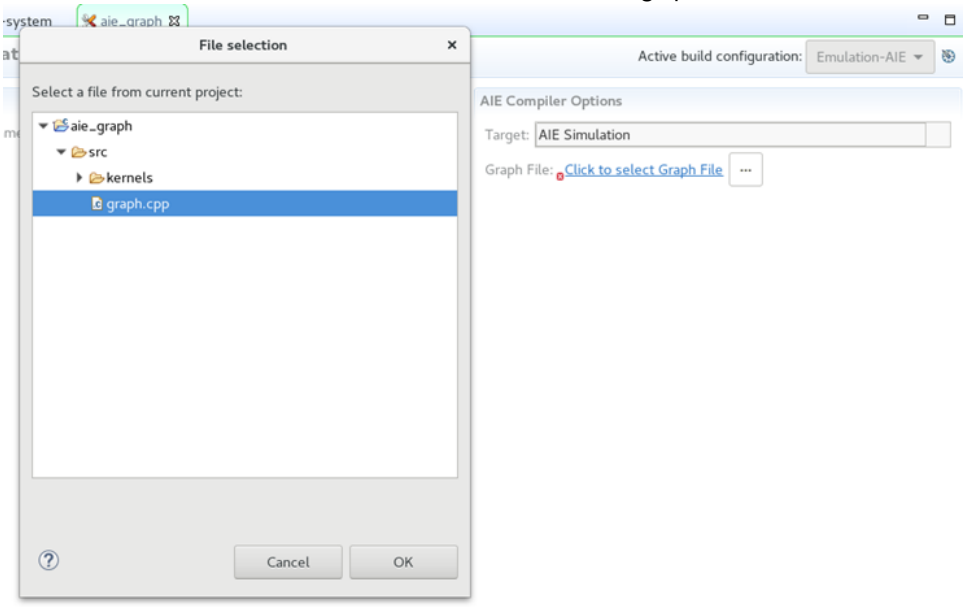
重要提示！此对话框的关键是确保“Into folder”（目标文件夹）字段指向 graph 工程的 `src` 文件夹，如 `aie_graph` 工程所示。

“Browse”（浏览）命令可用于导航到包含 AI 引擎 graph 源文件的文件夹。选中 graph 应用所需的 graph 和头文件，然后单击“Finish”（完成）。

2. 使用类似进程导入 graph 应用所需的数据文件。区别在于，这些文件并非导入 `src` 文件夹，而是导入 AI 引擎 graph 工程的 `data` 文件夹。

提示：数据文件可包含输入源文件，用于运行您的 graph 或黄金输出数据文件，以比较仿真结果。

3. 导入源文件和数据文件后，您需要定义 graph 的顶层。在“Project Editor”（工程编辑器）窗口右侧，单击“Click to select Graph File”（单击以选中 graph 文件）链接，以打开“File Selection”（文件选择）对话框，如下图所示。穿过应用工程层级导航到 `src` 文件夹，选择包含 graph 应用的 C/C++ 代码文件。



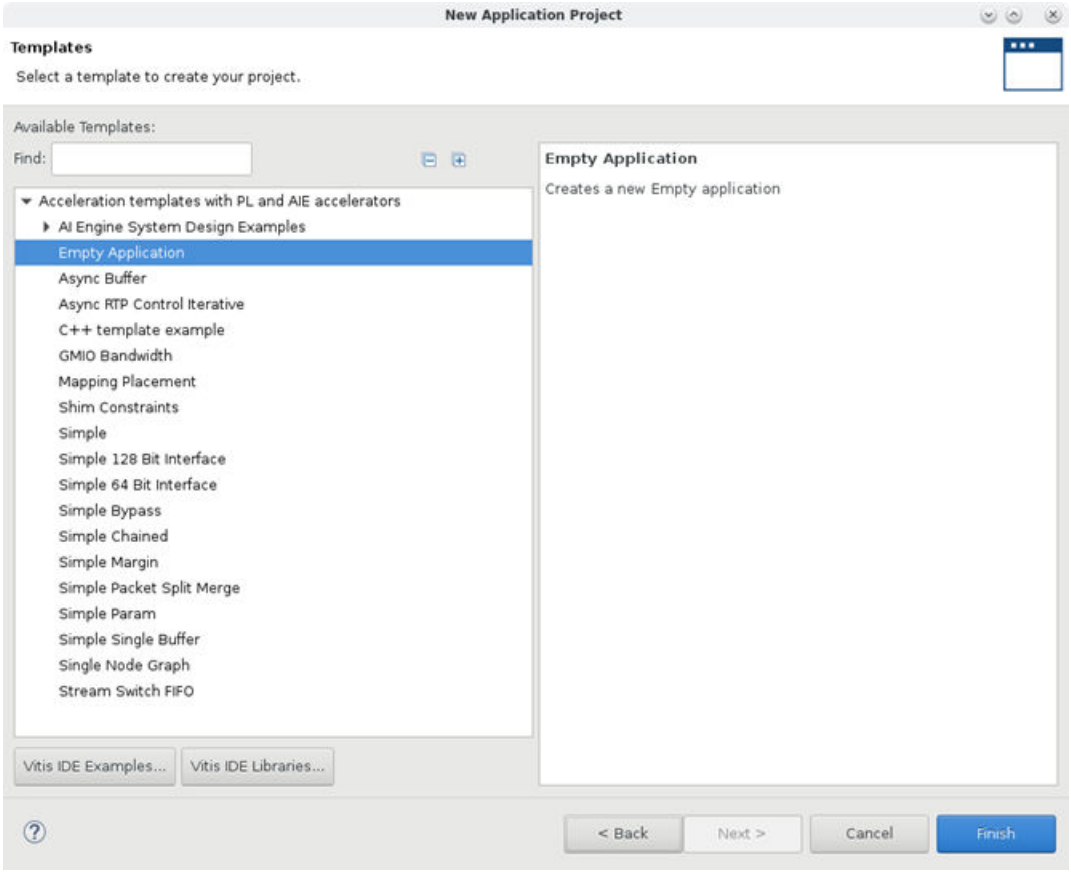
至此工程的构建准备已就绪。

提示：先前步骤适用于单内核编程和多内核编程。在 Vitis IDE 中提供了“Pipeline”（流水线）视图，以提供单内核调试（请参阅 单内核调试的流水线视图）。

Vitis 工具模板示例

Vitis IDE 中的“New Application Project”Wizard（新建应用工程向导）最后一页（如下图所示）显示了可用于您的设计的应用模板列表。选择模板会创建 AI 引擎 graph 应用样本，并导入必要的源代码，以便您构建并检验应用仿真设计的不同要素。

图 76：**AI 引擎应用模板**



模板工程展示了 AI 引擎编程的基本功能特性。您可学习这些模板、将其用作为您自己的工程的起点，或者混用搭配这些功能特性来创建您自己的复杂计算 graph。下表描述了其中部分的模板。

表 92：**应用模板示例**

模板名称	描述	更多信息
“AI Engine, PL and PS System Design”（AI 引擎、PL 和 PS 系统设计）	此设计演示的是在系统内将 AI 引擎阵列与可编程逻辑和处理器系统加以集成的方法。它会执行硬件协同仿真和硬件实现。	第 8 章：使用 Vitis 工具流程来集成应用。
“Async Buffer”（异步缓冲器）	此 graph 用于演示异步窗口 API。	《AI 引擎内核与 Graph 编程指南》(UG1079) 中的异步窗口访问。
“Async RTP Control Iterative”（异步 RTP 控制迭代）	此 graph 用于演示异步 RTP 更新和运行搭配指定测试迭代的简单使用方法。	《AI 引擎内核与 Graph 编程指南》(UG1079) 中的 graph 执行控制。
“C++ template example”（C++ 模板示例）	此示例演示了 C++ 模板数据类型和状态封装。	《AI 引擎内核与 Graph 编程指南》(UG1079) 中的 C++ 模板支持。
“GMIO Bandwidth”（GMIO 带宽）	此 graph 用于演示 GMIO 性能剖析。	《AI 引擎内核与 Graph 编程指南》(UG1079) 中的配置 input_gmio/output_gmio。
“Mapping Placement”（映射布局）	此模板 graph 带有可重定位的内核映射和内核位置约束。	《AI 引擎内核与 Graph 编程指南》(UG1079) 中的位置约束。

表 92：**应用模板示例** (续)

模板名称	描述	更多信息
“Shim Constraints” （Shim 约束）	此 graph 用于演示 AI 引擎到 PL 接口边界上的物理通道分配约束。	《AI 引擎内核与 Graph 编程指南》(UG1079) 中的 AI 引擎/可编程逻辑集成。
“Simple” （简单）	简单的 2 内核 graph，具有基于窗口的数据通信。	《AI 引擎内核与 Graph 编程指南》(UG1079) 中的基于窗口的访问。
“Simple 128 Bit Interface” （简单 128 位接口）	此 graph 用于演示 AI 引擎与 PL 之间的 128 位接口。	《AI 引擎内核与 Graph 编程指南》(UG1079) 中的配置 input_plio/output_plio。
“Simple 64 Bit Interface” （简单 64 位接口）	此 graph 用于演示 AI 引擎与 PL 之间的 64 位接口。	《AI 引擎内核与 Graph 编程指南》(UG1079) 中的配置 input_plio/output_plio。
“Simple Bypass” （简单旁路）	此 graph 用于演示如何使用内核旁路。	《AI 引擎内核与 Graph 编程指南》(UG1079) 中的内核旁路。
“Simple Margin” （简单裕度）	此 graph 用于演示如何使用窗口内的裕度（重叠窗口）。	《AI 引擎内核与 Graph 编程指南》(UG1079) 中的基于窗口的访问。
“Simple Packet Split Merge” （简单的包拆分合并）	此 graph 用于演示包串流数据的简单拆分与合并。	《AI 引擎内核与 Graph 编程指南》(UG1079) 中的显式包切换。
“Simple Param” （简单参数）	简单的 1 内核 graph，具有使用外部触发器的标量参数更新。	《AI 引擎内核与 Graph 编程指南》(UG1079) 中的指定运行时数据参数。
“Simple Single Buffer” （简单的单缓冲器）	此 graph 演示了连接上的单缓冲器约束。	《AI 引擎内核与 Graph 编程指南》(UG1079) 中的缓冲器分配控制。
“Single Node Graph” （单节点 graph）	这是简单的单节点 graph，其中具有演示窗口（单缓冲器和双缓冲器）、串流和 RTP 阵列连接。	单内核开发
“Stream Switch FIFO” （串流开关 FIFO）	此 graph 用于演示如何使用串流开关 FIFO 来避免再收敛的串流出现死锁。	《AI 引擎内核与 Graph 编程指南》(UG1079) 中的 FIFO 深度。

构建 AI 引擎 graph

在“Assistant” （助手）视图中查看工程。它起初包含三个工程：首先是定义应用工程时，由 Vitis IDE 创建的顶层系统工程，其次是应用工程本身，以及“hw_link”工程，该工程用于将 AI 引擎域和 PL 区域的域链接到单个器件二进制 (XCLBIN) 文件或平台文件 (XSA)。

此例中的应用工程即您的 AI 引擎 graph 应用。创建 graph 工程时，Vitis IDE 创建了顶层系统工程以及“hw-link”工程，欲知详情，请参阅 配置硬件链接工程。

展开 AI 引擎 graph 应用工程可以看到，其中包含三个构建目标：

- Emulation-SW（AI 引擎仿真）：功能仿真构建目标。由 AI 引擎编译器完成编译，并在“x86simulator”中运行。
- Emulation-AIE（AI 引擎仿真）：硬件仿真构建。在 AI 引擎编译器内完成编译，并在 AI 引擎 SystemC 仿真器 (`aiesimulator`) 中运行。
- 硬件：硬件构建，由 AI 引擎编译器完成编译，以供在实际器件中使用。

所有这些构建目标均由 AI 引擎编译器工具来构建，如 第 3 章：对 AI 引擎 Graph 应用进行编译 中所述，并由 AI 引擎仿真器或“x86simulator”来进行仿真，如 第 4 章：对 AI 引擎 graph 应用进行仿真 中所述。

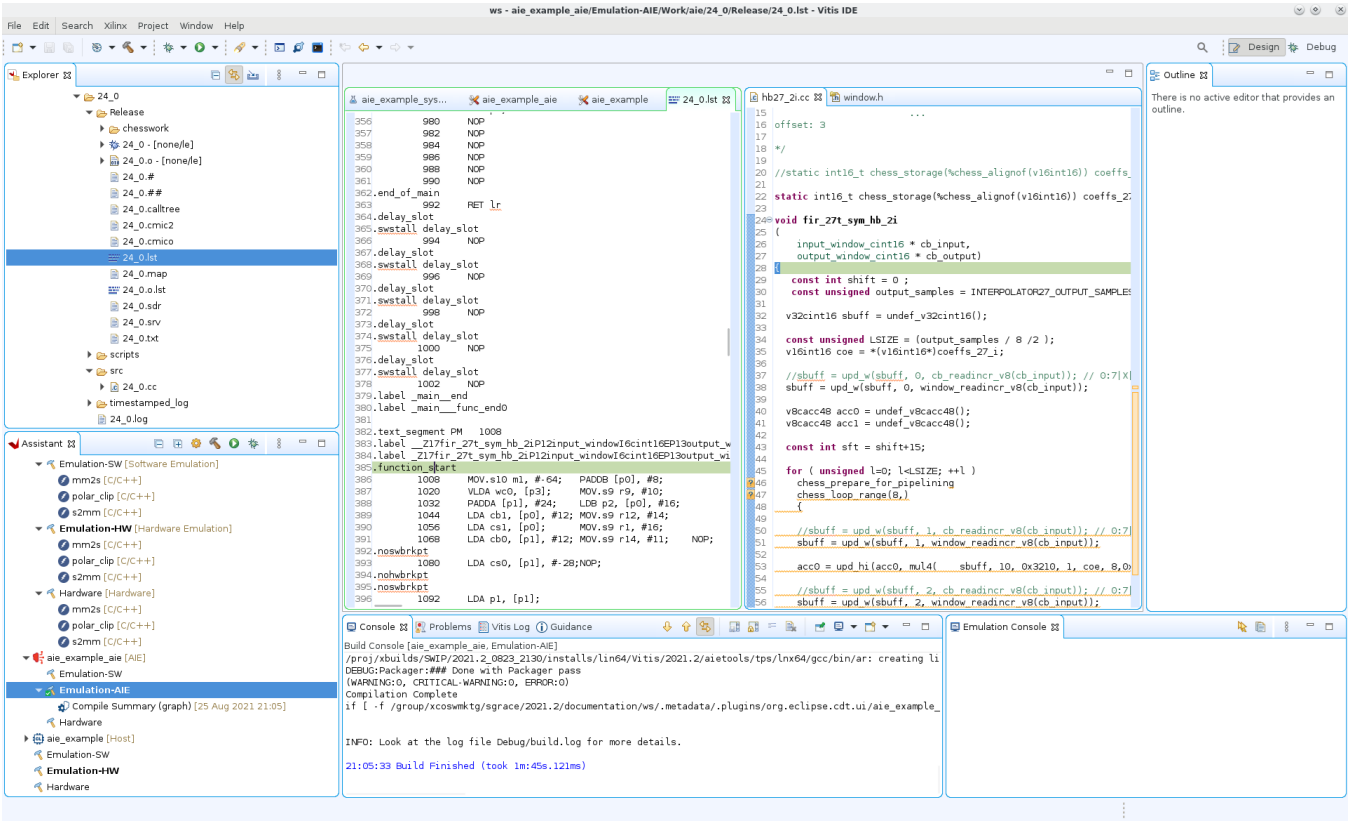
如果在“Assistant”视图中选中其中任一构建目标，并选择“Settings” （设置）命令 ( )，即可看到编译器不含任何构建设置。Vitis IDE 会按需为构建配置 AI 引擎编译器。

在“Assistant”视图中或者在“Project Editor”（工程编辑器）视图中激活目标并单击“Build”（构建）图标 () 来构建目标。

查看微码

构建完成后，右键单击“Explorer”（资源管理器）视图中的工程，然后选择“Open Disassembly View”（打开反汇编视图）即可查看 AI 引擎编译器生成的微码。在“Select Active Core”（选择活动的核）对话框中，选中该核（AI 引擎）。这样即可打开包含微码的 LST (`.lst`) 文件，所含微码对应于调度为在该 AI 引擎拼块上执行的内核代码。完成 `main end` 函数后，此内核代码会嵌入 LST 文件。您可选择微码的各部分并与内核源文件中对应的行进行交叉探测。这样您即可检验特定内核代码行所耗用的周期数，查看有哪些方面值得进一步最优化和提升效率。在下图中您可看到，LST 文件中的微码位于中间左侧，对应的内核代码文件位于右侧。单击任一视图即可在另一个视图内显示对应的代码。

图 77：**交叉探测视图**

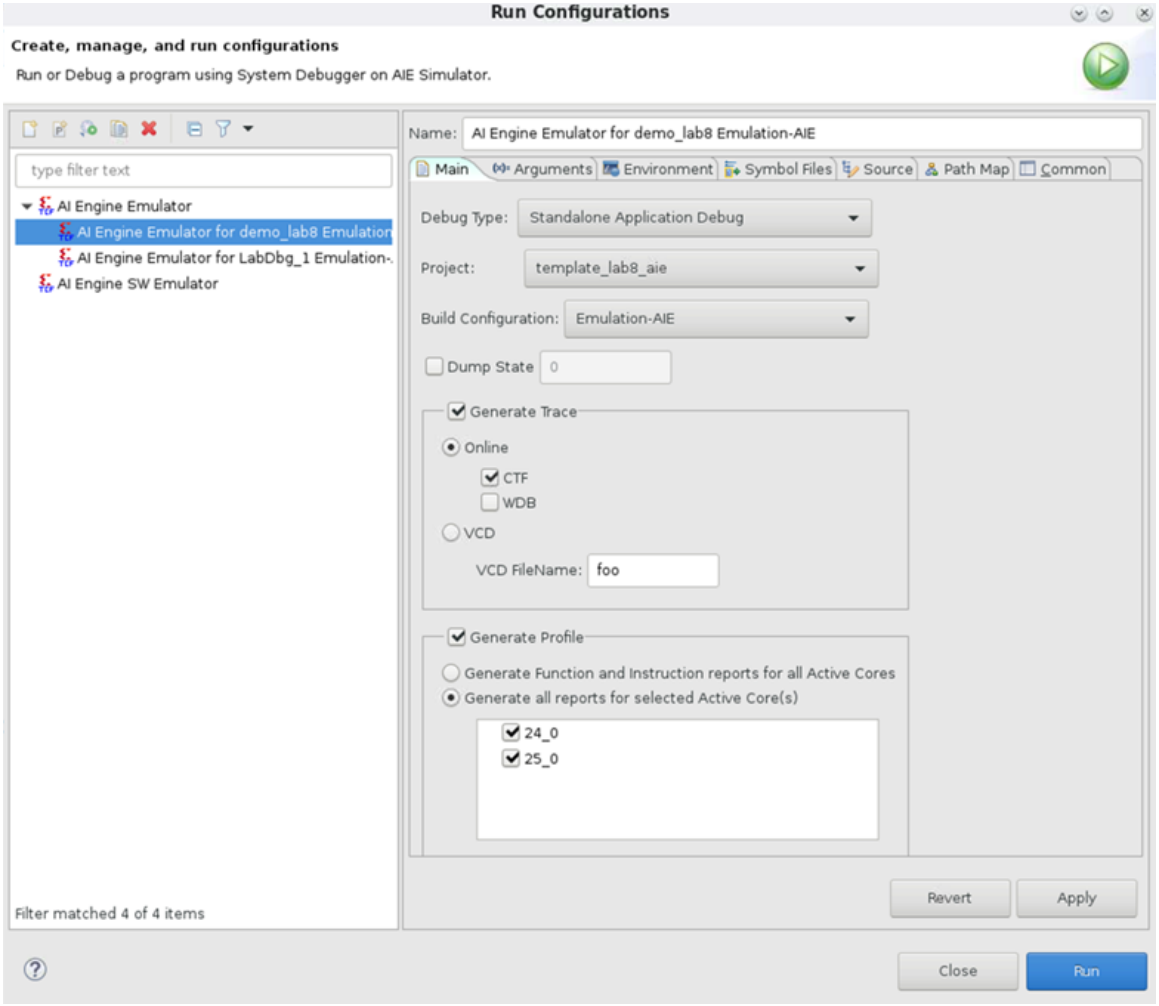


运行和分析 graph

成功完成构建后，即可在 Vitis IDE 中运行“Emulation-SW”（软件仿真）构建或“Emulation-AIE”（AI 引擎仿真）构建。您可检验编译器生成的各项报告并对自己的设计进行仿真。例如，对于硬件构建，您可复制 SD 卡输出，然后将其用于在硬件卡上启动和运行应用。

1. 在“Assistant”（助手）视图中，展开特定构建目标、右键单击“Compile Summary”(graph)，并选中“Open in Vitis Analyzer”（在 Vitis 分析器中打开）。这样即可打开“Compile Summary”（编译汇总）报告，如 在 Vitis 分析器中查看编译结果 中所述。
2. 在“Explorer”（资源管理器）视图中浏览您的工程并选择“Emulation AIE”→“Work”→“Reports”（AI 引擎仿真 > 工作 > 报告）以查找更多编译器生成的报告。

3. 要运行程序以进行硬件仿真，请在“Assistant”视图中单击“Run”（运行）按钮 ( ) 并选择“Run Configurations”（运行配置）。这样即可打开“Run Configurations”（运行配置）对话框以创建新的运行配置或者编辑现有的运行配置，如图所示。

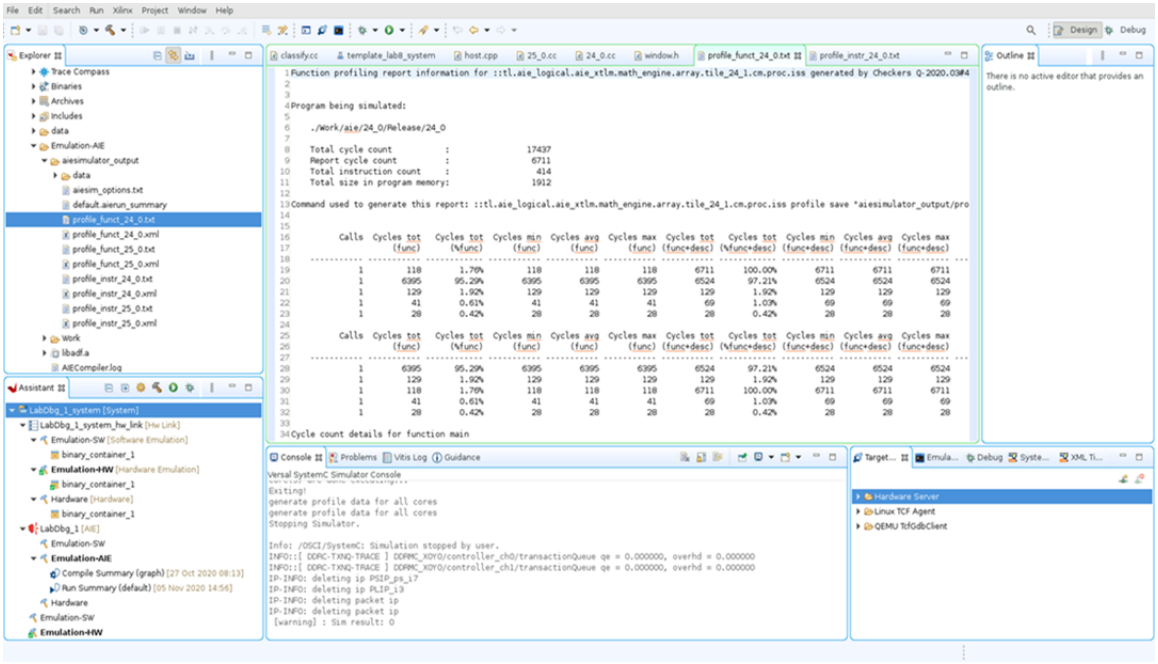


- 您可指定配置名称，这样即可创建多项配置，以供在设计流程中应用于不同场景或者不同构建目标。
- 您可启用“Generate Trace”（生成追踪），并在 AI 引擎仿真器中使用 `--dump-vcd` 为仿真构建启用事件追踪。如需了解更多详情，请参阅 第 5 章：仿真期间对 AI 引擎 graph 应用进行性能分析。
- 您可启用“Generate Profile”（生成剖析）以在 AI 引擎仿真器中指定 `--profile` 选项，并为所有 AI 引擎处理器或选定的拼块触发剖析。报告会在工程 `./Emulation-AIE/aiesimulator_output` 目录中生成。

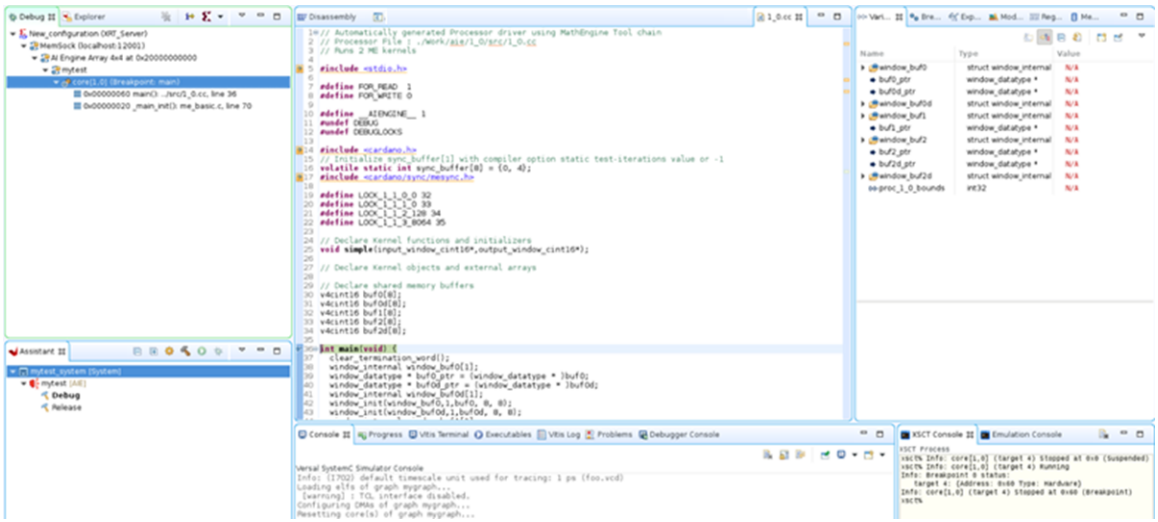
> **提示：**针对选定拼块启用该选项后即可生成时钟周期计数报告。

- 要添加其它 AI 引擎仿真器选项，请选中“Arguments”（实参）选项卡，像使用命令行一样输入选项。

准备好运行仿真后，选择“Apply”→“Run”（应用 > 运行）。



4. 要调试程序，请右键单击应用，然后选择“Debug As” → “Launch AIE Emulator”（调试方式 > 启动 AI 引擎仿真器）。这样即可在调试模式下启动仿真器，且 AI 引擎停止于其相应的 `main()` 处。您可以设置断点、单步执行、恢复执行以及检验寄存器、局部变量和存储器数据结构。如需了解更多信息，请参阅 从 Vitis IDE 进行硬件仿真调试。



单内核开发

为了在 AI 引擎上达成最高性能，单内核编程的主要目标是确保使用的矢量处理器可达成其理论最大值。算法矢量化至关重要，但管理矢量寄存器、存储器访问和软件流水打拍也同样不可或缺。由于矢量处理器能够在每个时钟周期内执行一项运算，因此程序员必须努力在当前运算期间加载用于下一项运算的数据。为 AI 引擎实现算法时，重要的是基于数据类型以及对这些数据类型执行的矢量内部函数来启动矢量化。根据数据类型，各内部函数能同时对两个或两个以上元素执行运算。如果内层循环具有顺序或循环进位依赖关系，那么可能可以展开外层循环，并行计算多个值。用户可凭借各种创造性方法来使用矢量内部函数解决问题。为 Versal® ACAP 实现算法时，重要的是了解 AI 引擎的优势所在，并了解有哪些方面在其它引擎（例如，标量引擎、自适应引擎和 DSP 引擎）中实现的效果更好。

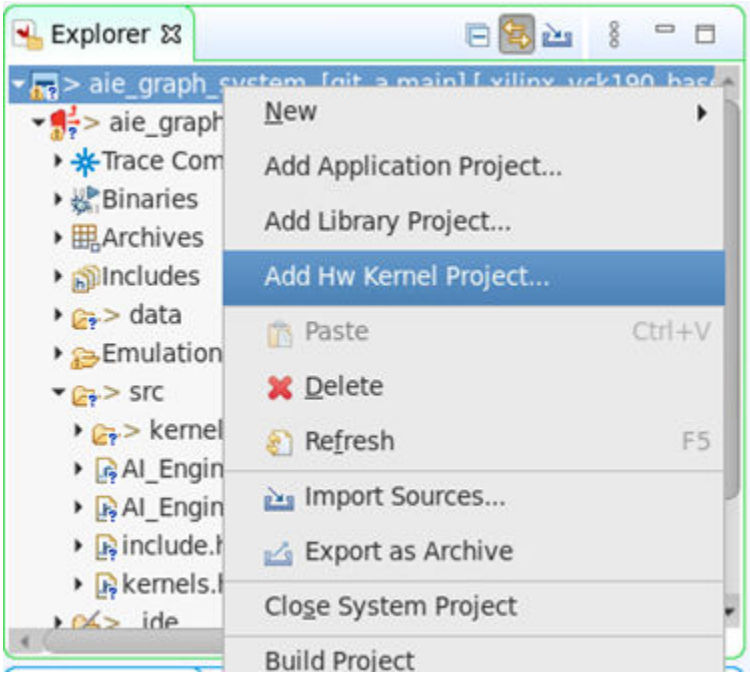
为了支持 AI 引擎单内核开发，Vitis IDE 除了提供传统处理器支持，还支持 AI 引擎内核开发。Vitis IDE 可提供单节点 graph 示例，此示例可用作为单内核开发的起点。Vitis IDE 具有调试视图，其中可显示寄存器、变量、可用断点、寄存器/存储器映射变量、内部/外部存储器内容，并且每个内核各有 1 条指令流水线（流水线视图）。

提示：完成仿真后，会生成时钟周期计数报告，并启用 `--profile` 选项。

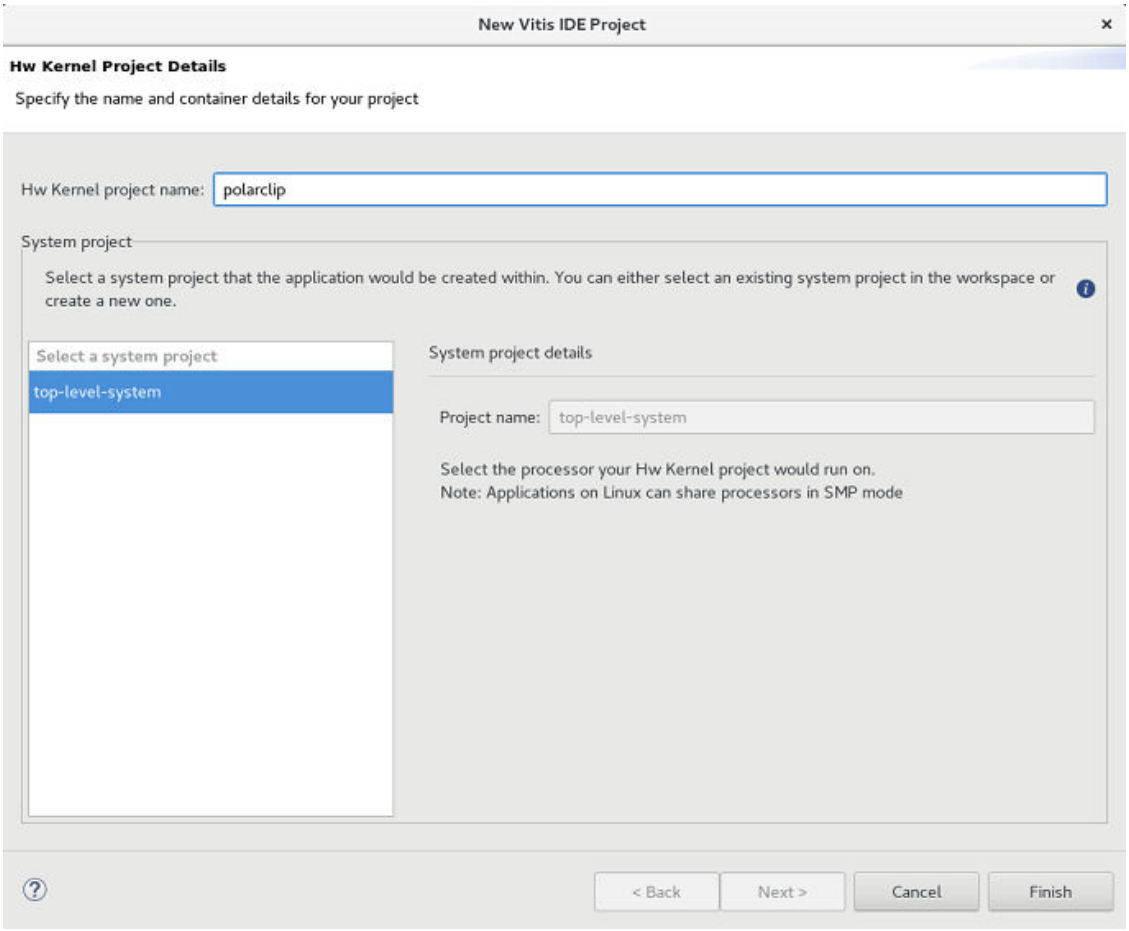
在系统中添加 PL 内核工程

创建 AI 引擎 graph 工程的同时也会创建顶层系统工程和“hw-link”工程，随后即可创建 PL 内核工程并将其添加到系统中。您需要创建 PL 应用工程，并使用以下进程将其添加到自己的系统工程内。

1. 在“Explorer”（资源管理器）视图中，选中顶层系统工程以创建新的 PL 工程并添加到系统工程中。右键单击系统工程，然后选择“Add Hw Kernel Project”（添加硬件内核工程）命令，如下图所示。

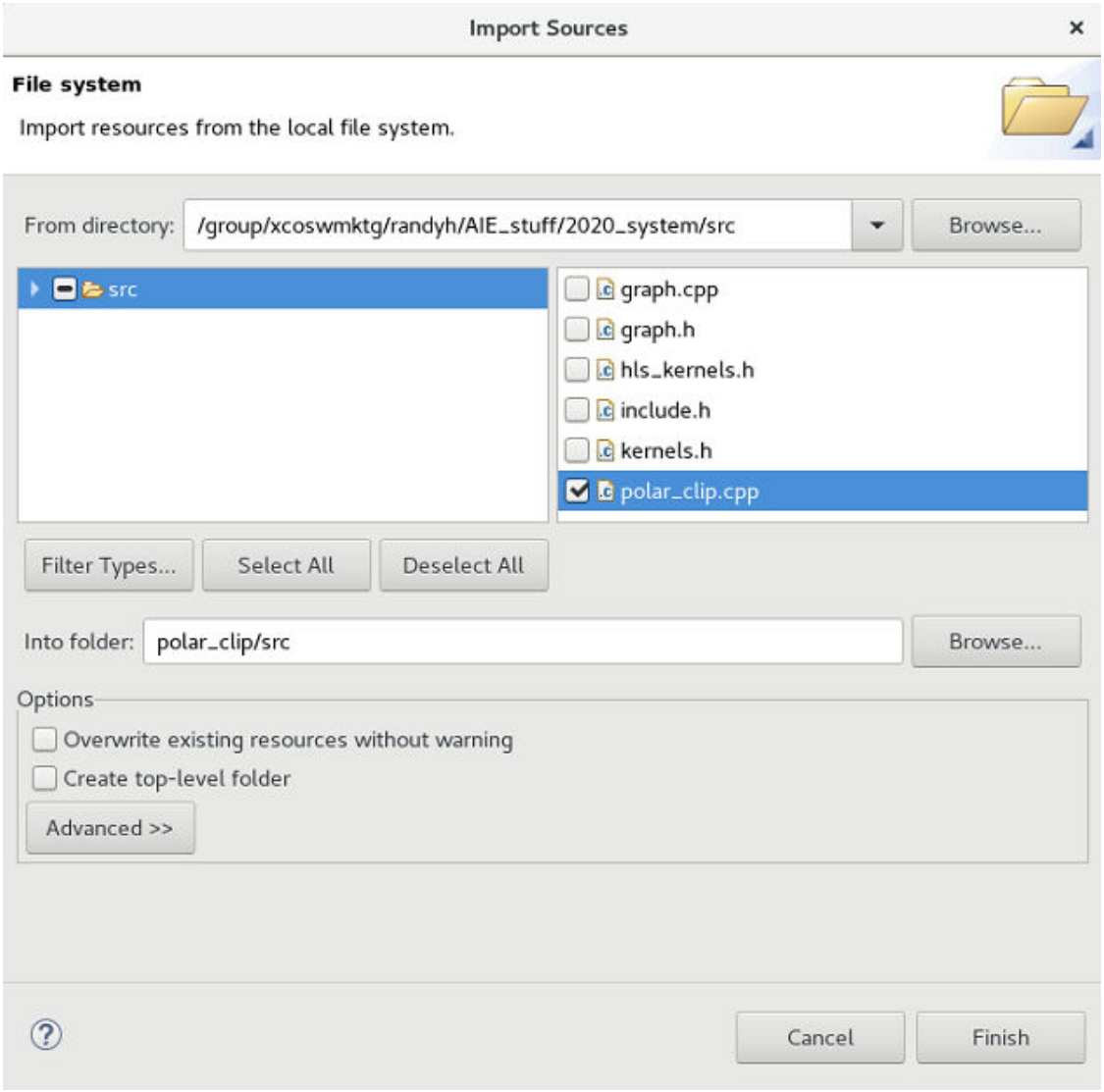


2. 这样即可显示“New Vitis IDE Project”Wizard（新建 Vitis IDE 工程向导）中的“Hw Kernel Project Details”（硬件内核工程详细信息）页面，如下图所示。



请确保将该工程添加到现有系统工程中，其中还包含您的 AI 引擎 graph 工程。指定“HW Kernel project name”（硬件内核工程名称）。单击“Finish”（完成）以继续。

3. 这样即可创建 PL 内核工程，并将其添加到顶层系统工程的层级内。下一步，您必须为内核添加源代码。在“Explorer”视图中，选择 PL 内核的 `src` 文件夹，然后单击“Import Sources”（导入源文件）命令 ( ) 以打开下图所示对话框。



浏览并选中 PL 内核必要的源文件。单击“Finish”（完成）以将源文件导入硬件内核工程。

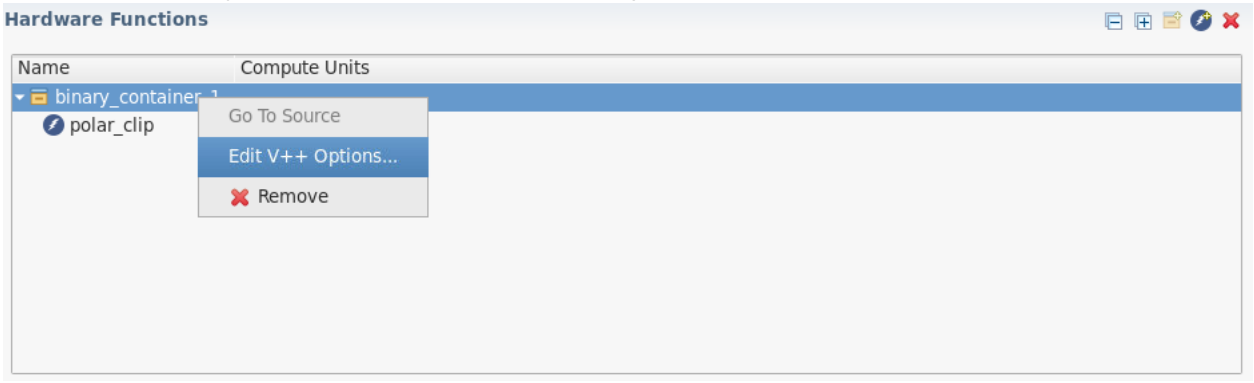
4. 将源文件添加到工程中后，必须定义要布局到 PL 区域中的硬件函数。在“Project Editor”（工程编辑器）窗口中，选择“Add Hardware Function”（添加硬件函数）命令 ( ) 并指定要在 PL 区域内实现的函数的名称。

创建完 PL 内核并定义硬件函数后，即可为“Emulation-SW”（软件仿真）、“Emulation-HW”（硬件仿真）或“Hardware”（硬件）目标构建内核工程。您可在 PL 内核工程内直接构建这些目标，或者也可以在构建顶层系统工程的过程中构建。顶层工程使用增量构建方法，可识别子工程的状态，并且仅重构需要更新的工程。

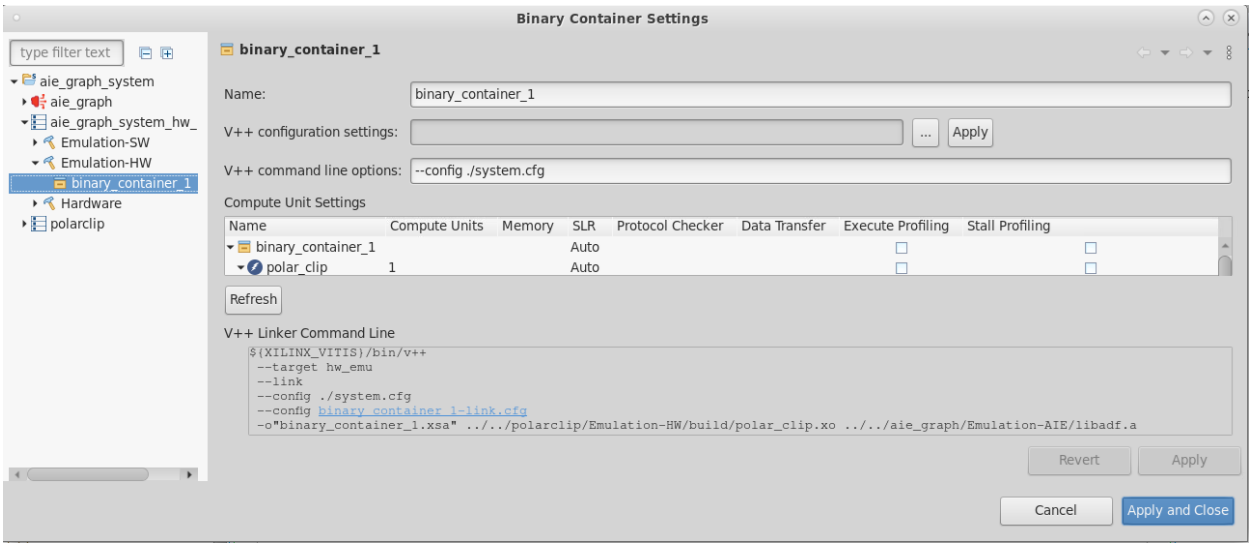
配置硬件链接工程

将各种域应用工程添加到顶层系统工程时，只需使用“hw_link”工程来定义 AI 引擎 graph 与 PL 内核之间的连接即可。在创建 AI 引擎 graph 工程期间，会自动生成此工程。

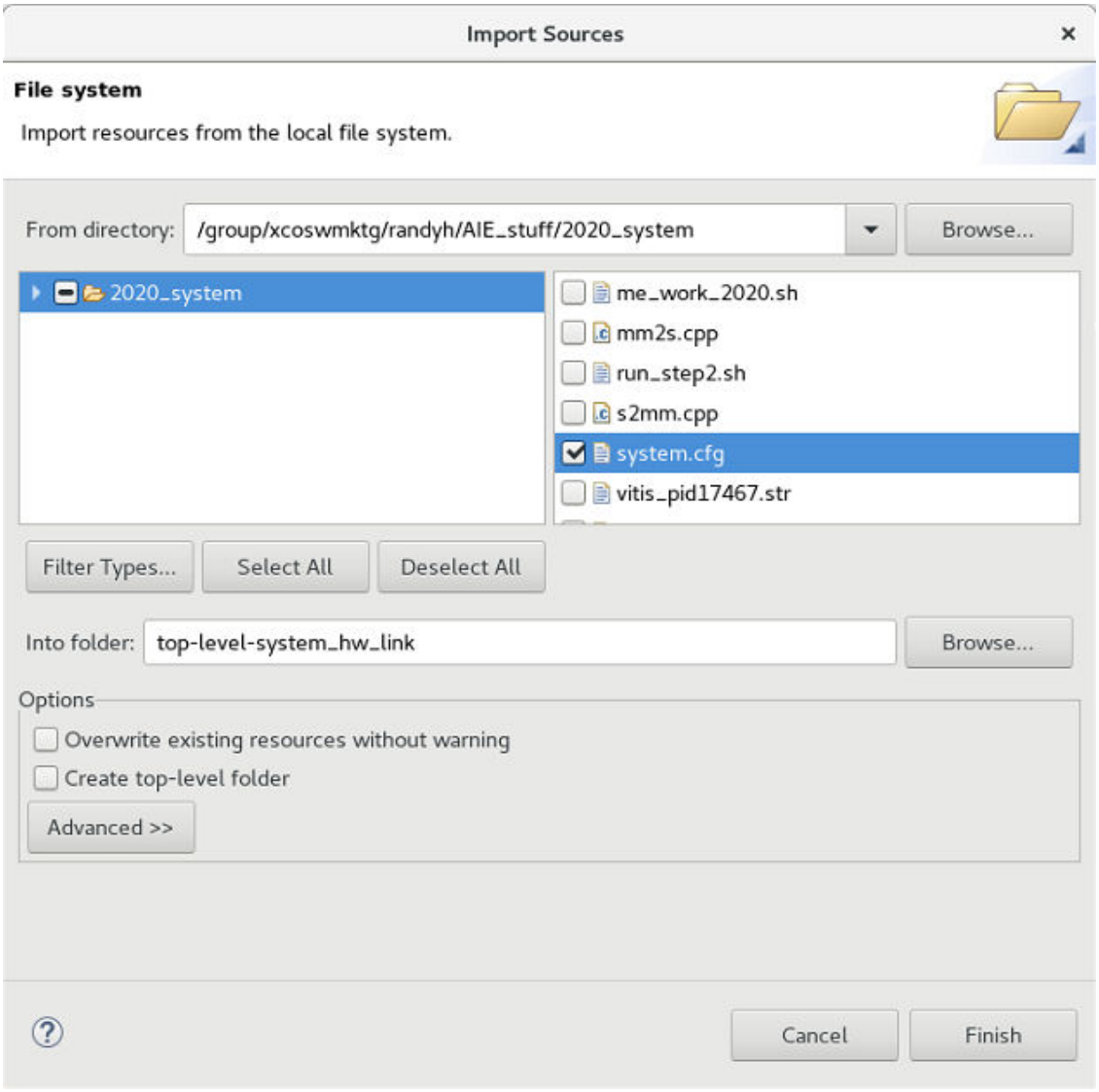
1. 在“Explorer”（资源管理器）视图中双击“hw_link”工程将其打开。右键单击“Hardware Functions”（硬件函数）窗口中的“binary_container”，然后选择“Edit V++ Options”（编辑 V++ 选项）命令。



2. 编辑“V++ Options”（V++ 选项）字段。添加 `config` 文件的正确路径至关重要，因为 `v++` 命令需要将此路径用于 Vitis IDE 工作空间。



3. 将指定的 `config` 文件导入 `hw_link` 工程文件夹。



提示：系统配置文件会被添加到 `hw_link` 工程文件夹内，而不是添加到 `src` 文件夹内。

如 系统链接 中所述，对于 AI 引擎 graph 应用，Vitis 编译器需要获取有关如何将 PL 内核连接到 graph 的指令。在 IDE 中已配置要例化的内核数量。但必须指定 PL 内核与 graph 之间的连接定义。

对于 Vitis IDE，配置文件示例如下所示：

```
[connectivity]
stream_connect=mm2s_1.s:ai_engine_0.DataIn1
stream_connect=ai_engine_0.clip_in:polar_clip_1.in_sample
stream_connect=polar_clip_1.out_sample:ai_engine_0.clip_out
stream_connect=ai_engine_0.DataOut1:s2mm_1.s
```

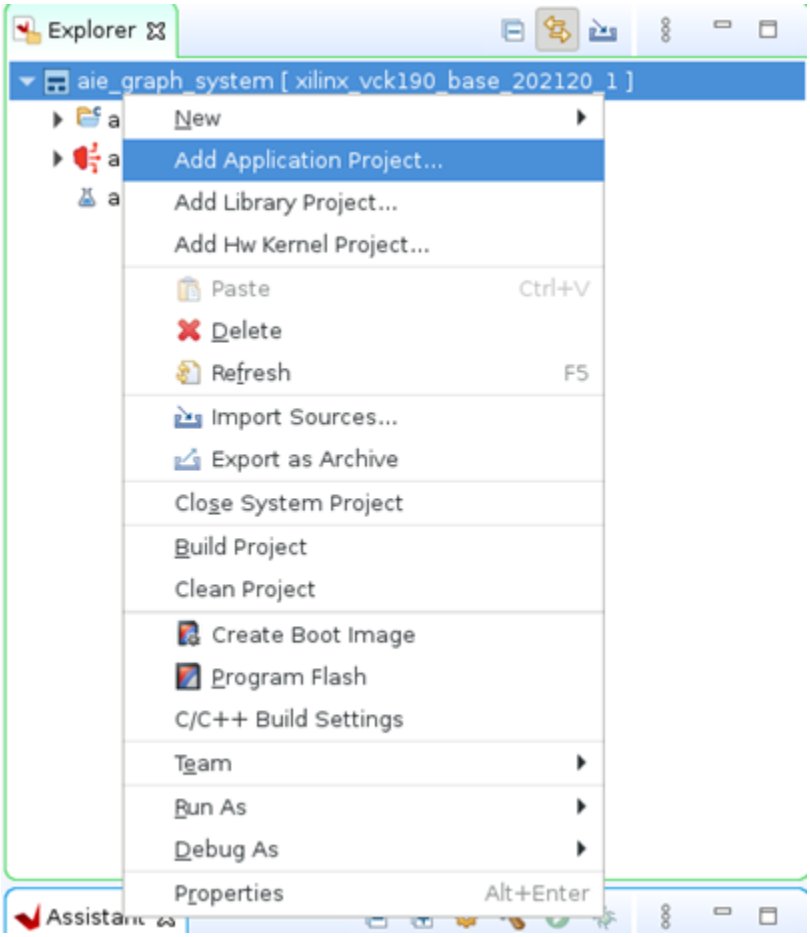
连接 `sc` 选项用于定义 AI 引擎 graph 的端口与 PL 内核的串流端口之间的连接。连接可定义为将某一个内核的串流输出连接到另一个内核的串流输入，或者连接到目标平台中实现的 IP 的串流输入端口。

在 Vitis IDE 与命令行流程之间，`system.cfg` 文件存在某些差异，如 系统链接 中所述。主要差异在于，IDE 提供 `config` 文件的连接 `nk` 选项，按内核例化指定数量的计算单元 (CU)，此数量在构建设置内指定。此外，IDE 为 CU 使用的命名约定形式为 `<kernel>_#`，其中 `#` 表示 CU 实例。如果仅有 1 个 CU 实例，那么它会包含 `_1` 扩展。这表示对于 Vitis IDE，`system.cfg` 不应指定 `nk` 选项，并且 `sc` 选项应使用来自 IDE 的 CU 实例名称。

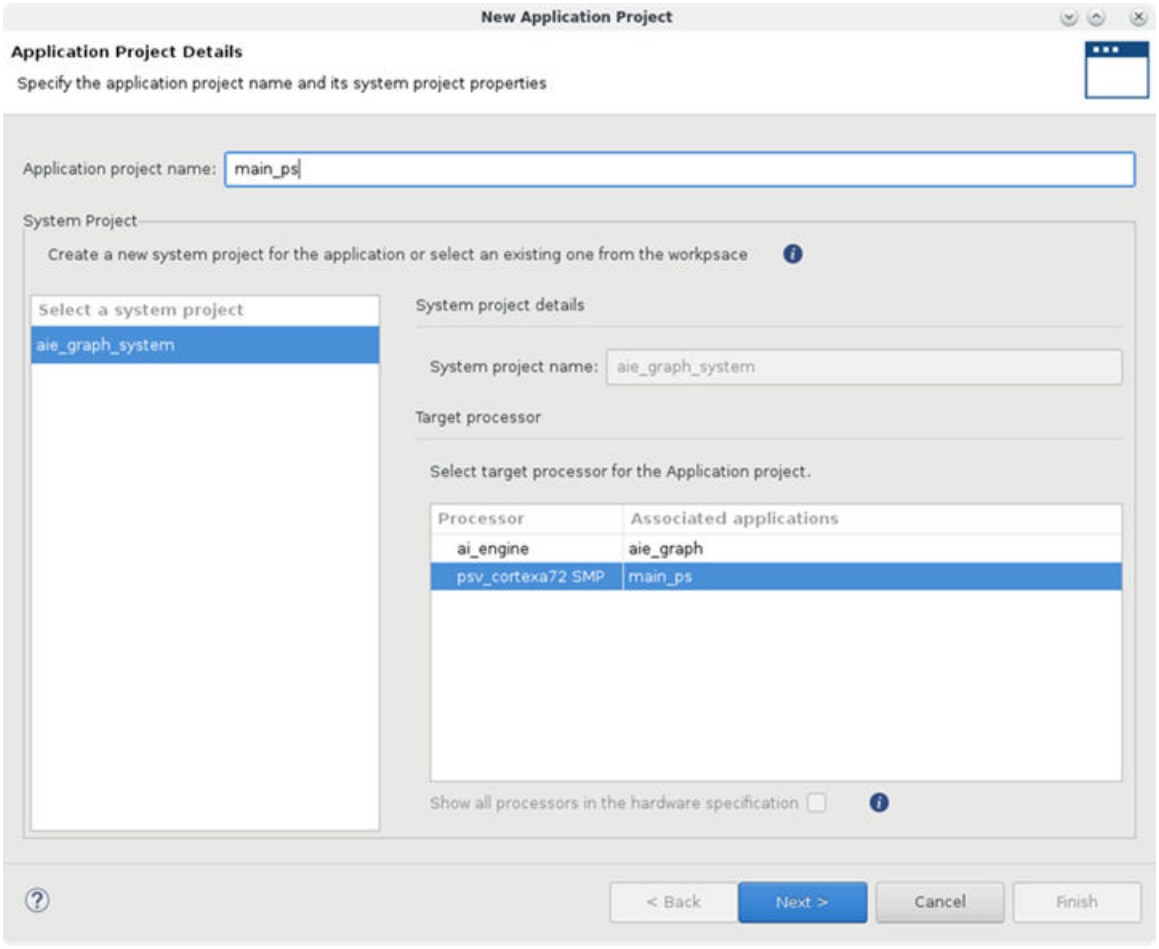
将 PS 应用添加至系统

您的顶层系统工程也支持在 Versal ACAP 的 PS 域中运行应用以便加载和运行 AI 引擎 graph 和 PL 内核。以下进程可用于创建 PS 应用工程并将其添加到您的系统工程内。

1. 在“Explorer”（资源管理器）视图中，右键单击系统工程并选中“Add Application Project”（添加应用工程）命令，如下图所示。



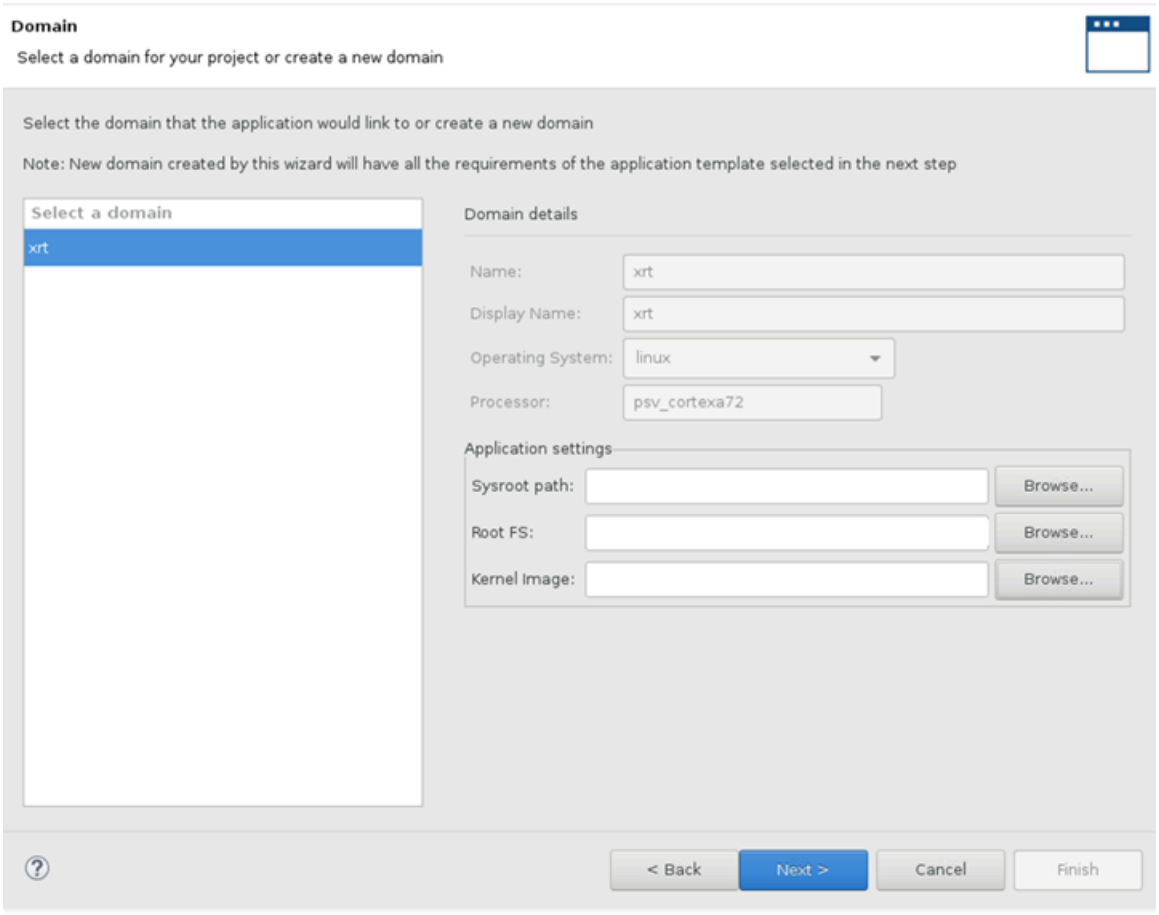
2. 这样即可显示“Application Project Details”（应用工程详细信息）页面，如下图所示。



请确保将该工程添加到现有系统工程中，其中还包含您的 AI 引擎 graph 工程。指定“Application project name”（应用工程名称）。

选择 Cortex®-A72 处理器核，然后单击“Next”（下一步）继续。

3. 这样即可显示工程向导的“Domain”（域）页面，如下所示。

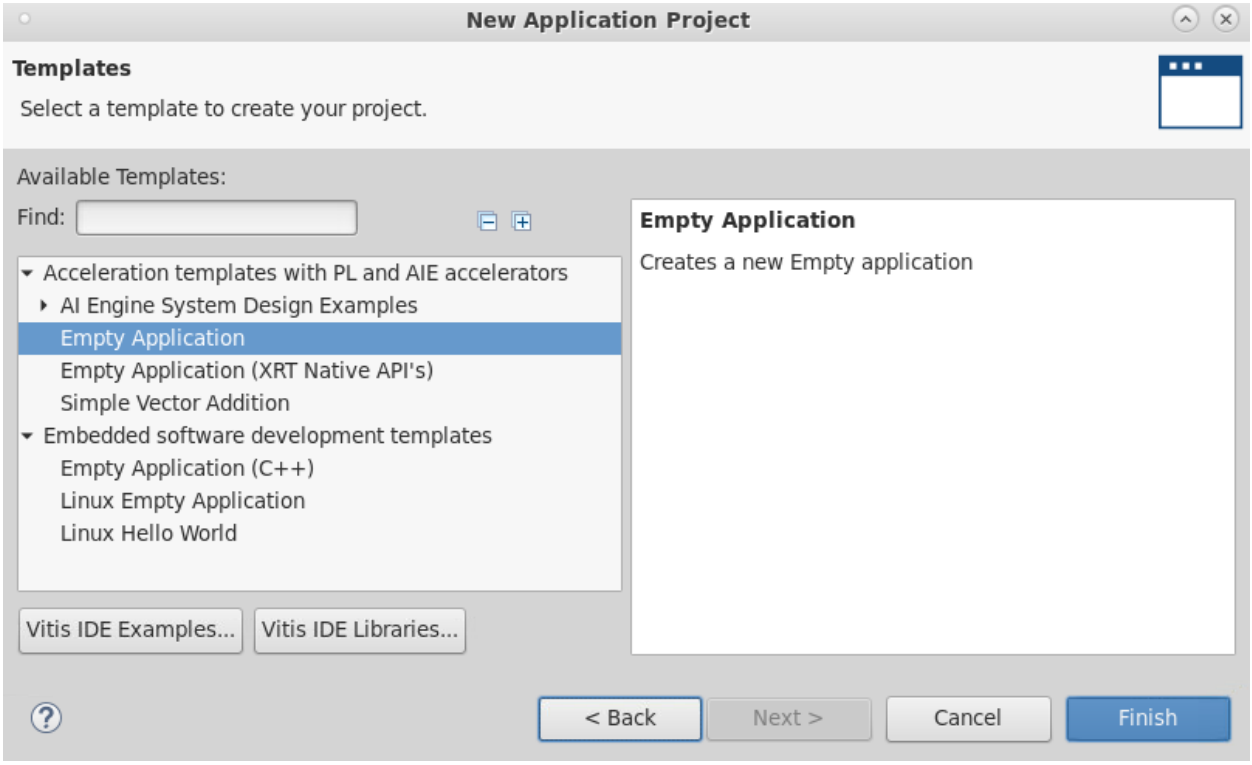


由于您已选中 Cortex®-A72 处理器核，因此 XRT 域是唯一可用的选项。这表示该域包含 Linux 操作系统和 XRT 库。您还必须指定嵌入式平台的以下三个要素：

- “Sysroot”（系统根目录）
- “Root FS”（根文件系统）
- “Kernel Image”（内核镜像）

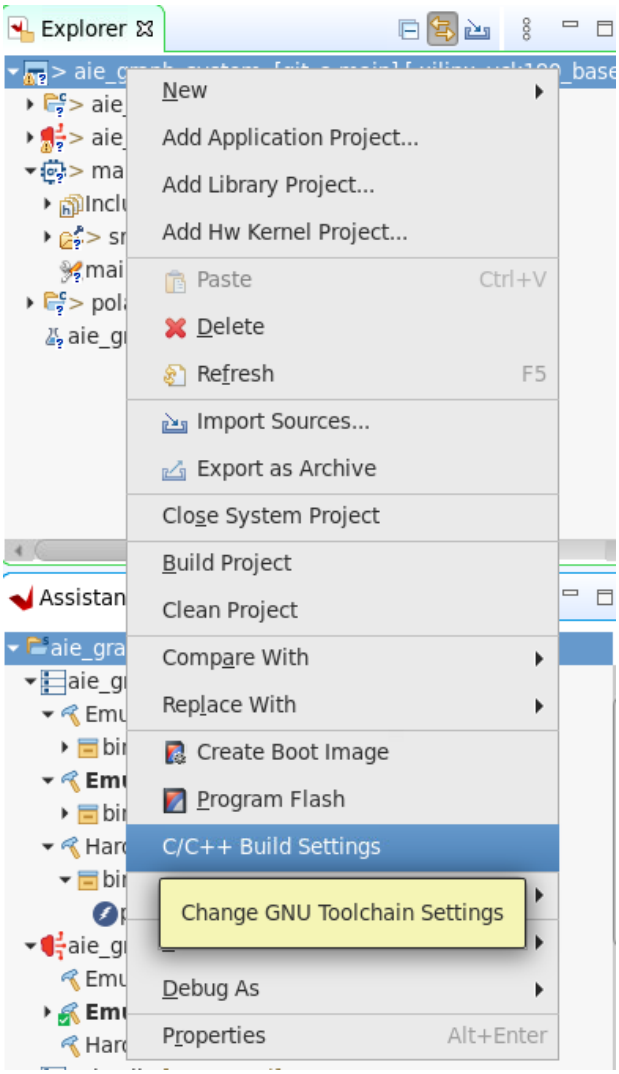
这些项是加载和启动操作系统与运行时驱动程序所必需的。这些文件可从“Embedded Platforms”（嵌入式平台）下载页面获取。指定这些文件后，单击“Next”继续。

4. 这样会为“PS Application”（PS 应用）工程打开“Templates”（模板）页面，如下图所示。



在此情况下，鉴于您正在创建新的 PS 工程，请选中“Empty Application”（空应用）模板（默认），然后单击“Finish”（完成）以创建工程。这样即可创建新的 PS 工程，并将其添加到您正在处理的顶层系统工程。

5. 为 PS 应用添加源代码。在“Explorer”（资源管理器）视图中，右键单击 PS 工程的 `src` 文件夹，然后单击“Import Sources”（导入源文件）命令 ( )。浏览并选中 PS 应用必要的源文件。单击“Finish”以导入所选源文件。
6. 右键单击 PS 工程并选中“C/C++ Build Settings”（C/C++ 构建设置），按需添加 include 目录、链接库、C++ 标准和构建配置。您可为 PS 工程配置工程属性。



创建 PS 应用工程以及 AI 引擎 graph 和 PL 内核后，您的顶层系统工程即将准备就绪，自上而下执行构建已近在眼前。

构建和运行系统

定义完顶层系统工程后，就表示 AI 引擎 graph 应用已添加、PL 内核已添加、PS 应用已添加、“HW-Link”（硬件链接）工程已配置，这样一切准备就绪，可以开始构建和运行系统了。

系统工程支持三种不同构建目标：“Emulation-SW”（软件仿真）、“Emulation-HW”（硬件仿真）和“Hardware”（硬件）。您可使用下列步骤来构建顶层系统工程。

1. 双击“Explorer”（资源管理器）视图中的 `<project>.sprj` 文件，在编辑器区域内打开系统工程。
2. 将“Project Editor”（工程编辑器）窗口中的“Active build configuration”（活动的构建配置）设置为“Emulation-SW”、“Emulation-HW”或“Hardware”，以选择特定构建目标。
3. 在“Assistant”（助手）视图中，选中顶层工程，然后单击“Settings”（设置）命令 ( ) 以显示“System Project Settings”（系统工程设置）对话框，并执行任何必要的更改，然后执行构建。

4. 单击工具栏菜单中的“Build”（构建）命令 ( )，以启动当前处于活动状态的构建配置。Vitis IDE 中的构建进程是递增进程，仅构建已更新的工程元素，上一次构建完成后需重新构建。您可以构建系统工程的单个元素，例如，AI 引擎 graph 或 PL 内核，该工具会识别是否需要重新构建这些元素。

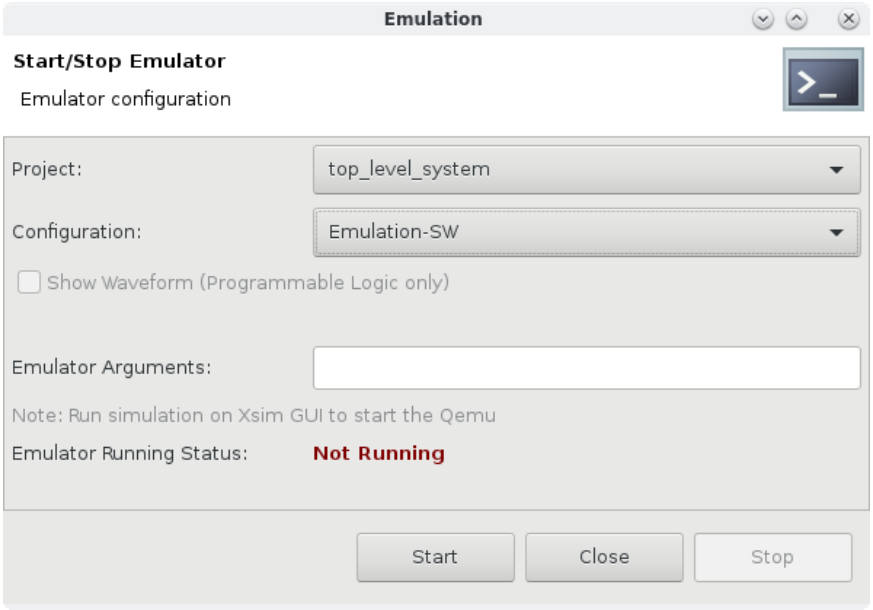
   注释：硬件的构建进程运行时间远超仿真构建的时间。因此在仿真构建中重要的是，先完成设计调试，而后再转至硬件构建。

5. 当构建完成后，请检验“Explorer”视图中的“Emulation-SW”、“Emulation-HW”或“Hardware”构建文件夹的内容。您可选中并展开构建目录的各文件夹。在输出层级中会显示 Vitis 编译器封装进程 (`v++ --package`) 的输出文件。构建进程会生成系统所需的仿真数据和启动文件，并将其写入 `sd_card` 文件夹。

   注释：在 Hardware 文件夹下会创建两个文件夹：`package` 和 `package_no_aie_debug`。`package` 文件夹内的 `sd_card.img` 文件用于硬件调试目的，而 `package_no_aie_debug` 文件夹内的 `sd_card.img` 文件则用于常规应用执行。

   重要提示！在 AI 引擎系统工程中，您可以调试和运行系统级工程，或者也可以调试并运行 AI 引擎工程。除部分顶层系统工程外，您无法调试和运行 PS 或 PL 工程。

6. 对于软件仿真构建，请选择“Xilinx” → “Start/Stop Emulator”（赛灵思 > 启动/停止仿真器）命令以启动 QEMU 仿真环境。

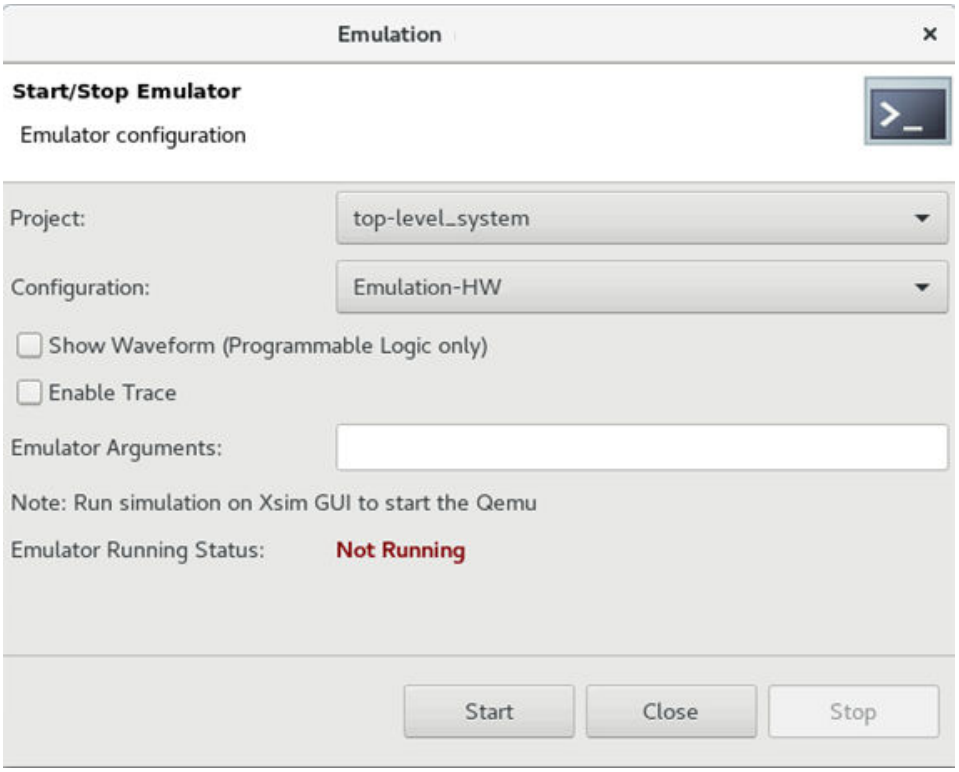


   这样即可启动该仿真器，然后等待至 QEMU 内 Linux 完成启动为止。“Emulation”（仿真）控制台会显示 QEMU 启动和 Linux 启动进程的转录文本。当进度对话框关闭并且 `qemu%` 提示为空白时，即表示此进程已完成。您可以检查转录文本以获取该进程的详细信息。

   启动软件仿真时，您可以为运行 graph 应用的 AI 引擎仿真器指定选项，如 复用 x86 仿真器选项 中所述。在前图中所示的“Emulator Arguments”（仿真器实参）字段中可以使用以下命令指定这些选项。

```
-x86-sim-options ${FULL_PATH}/x86sim.options
```

7. 对于硬件仿真构建，请选择“Xilinx” → “Start/Stop Emulator”（赛灵思 > 启动/停止仿真器）命令以启动 QEMU 仿真环境。



这样即可启动该仿真器，然后等待至 QEMU 内 Linux 完成启动为止。“Emulation”（仿真）控制台会显示 QEMU 启动和 Linux 启动进程的转录文本。当进度对话框关闭并且 `qemu%` 提示为空白时，即表示此进程已完成。您可以检查转录文本以获取该进程的详细信息。

启动硬件仿真时，您可以为运行 graph 应用的 AI 引擎仿真器指定选项，如 复用 AI 引擎仿真器选项 中所述。在前图中所示的“Emulator Arguments”（仿真器实参）字段中可以使用以下命令指定这些选项。

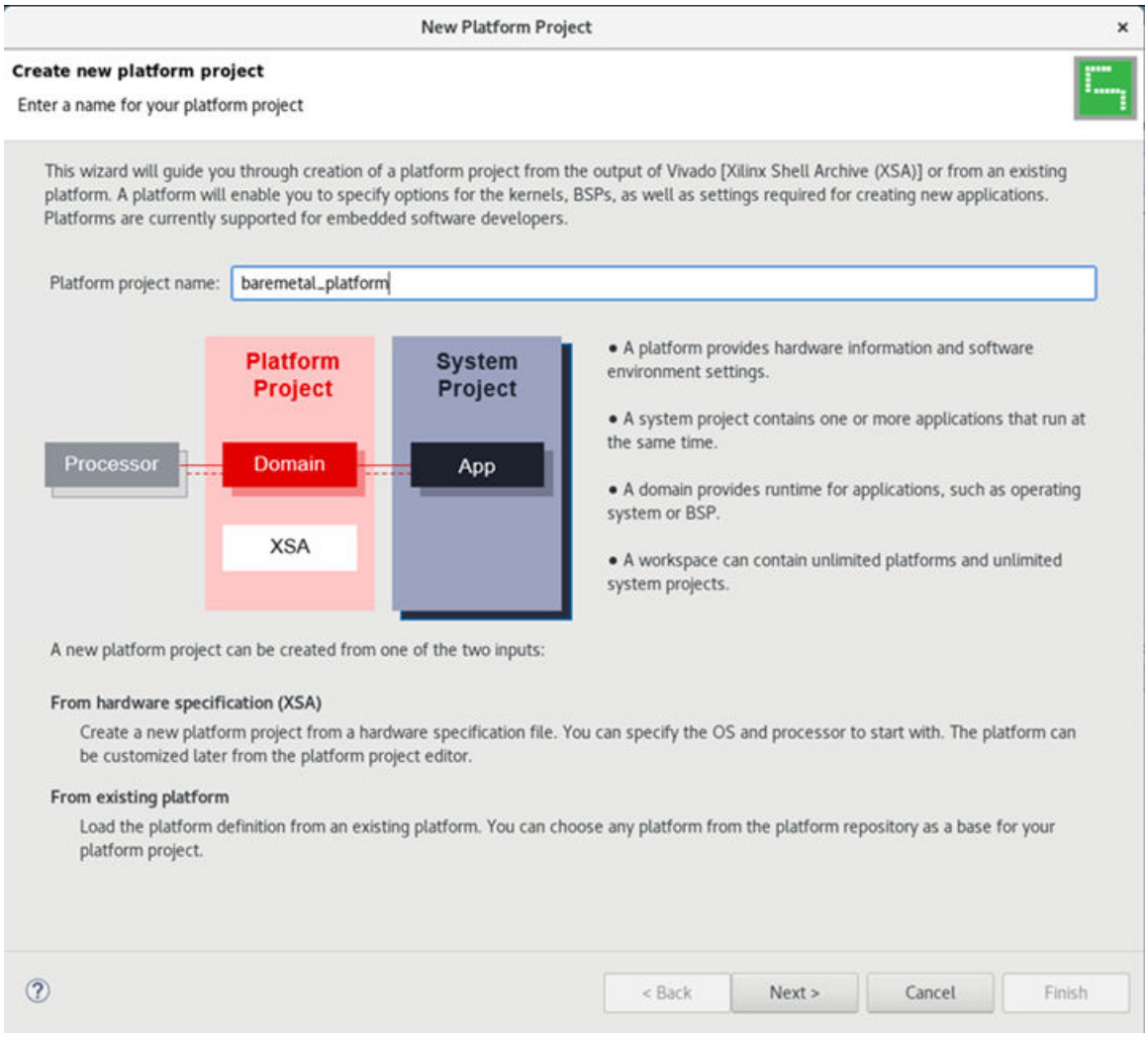
```
-aie-sim-options ${FULL_PATH}/aiesim_options.txt
```

8. 在“Run Configurations”（运行配置）对话框中，选中“Run”（运行）以继续。

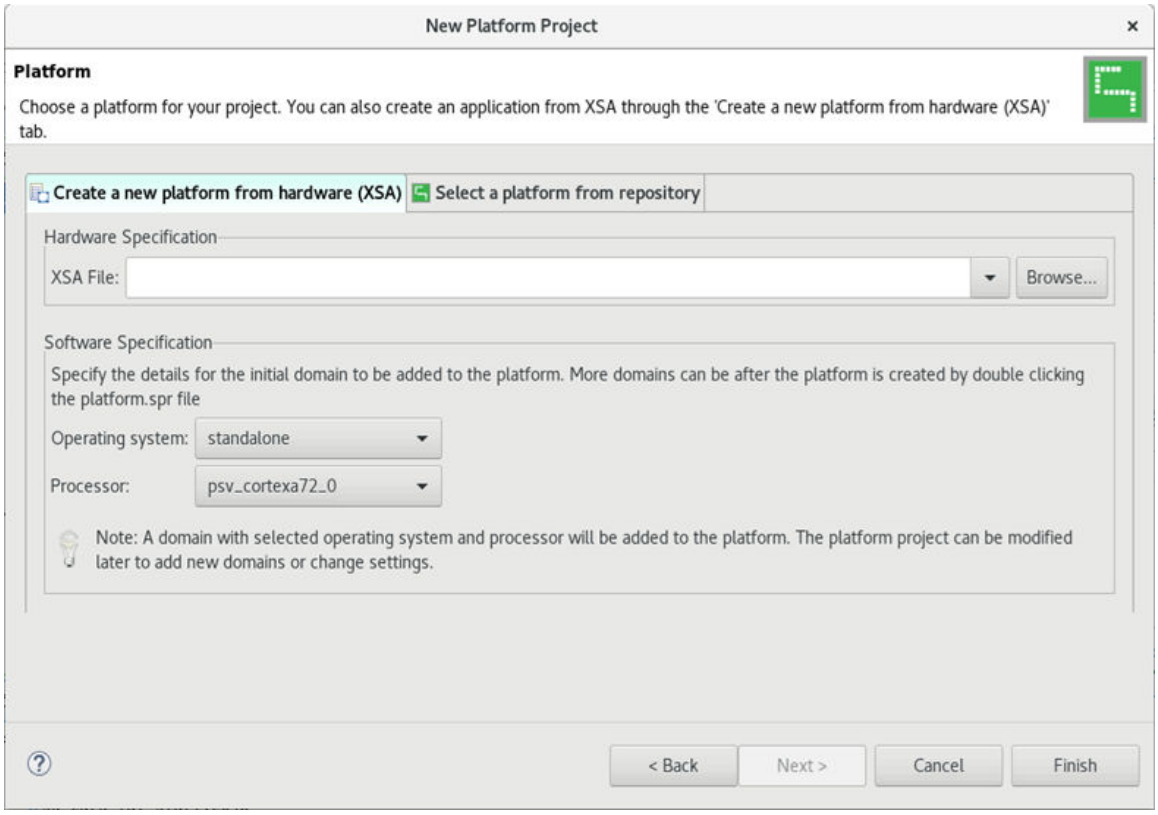
在 Vitis IDE 中构建裸机 AI 引擎

构建裸机系统有别于先前所述的标准应用流程。裸机系统需在 配置硬件链接工程 步骤中对进程添加分支，并指定此分支发生于该步骤之后，随后遵循新的进程进行操作。所需具体步骤如下所述。

1. 创建裸机平台。构建裸机应用需要具有裸机域的平台。由于 `xilinx_vck190_base_202220_1` 基础平台不含裸机域，因此您必须使用固定 XSA 文件创建含裸机域的定制平台。
   a. 选择 Vitis IDE 中的“File” → “New” → “Platform Project”（文件 > 新建 > 平台工程）命令。这样即可打开“New Platform Project”Wizard（新建平台工程向导），如下所示。



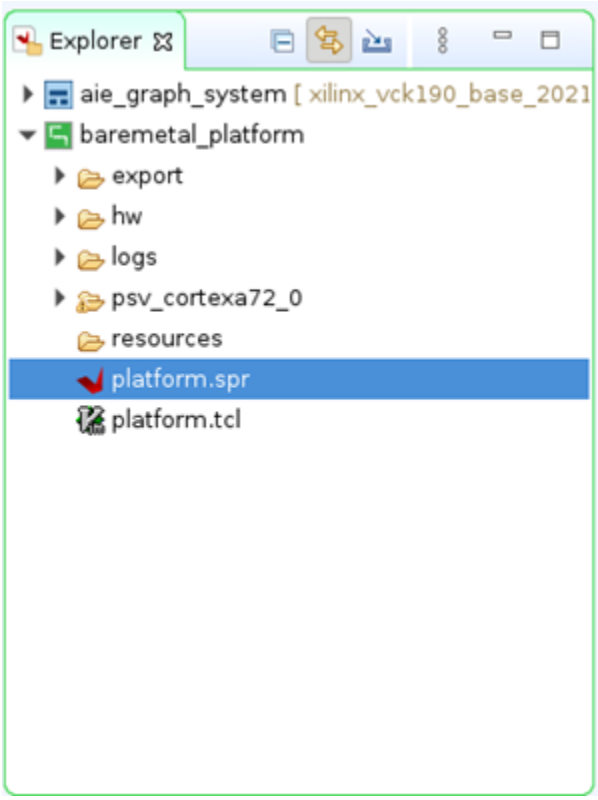
b. 指定“Platform project name”（平台工程名称），然后单击“Next”（下一步）继续。这样即可显示此向导的“Platform”（平台）页面，以便您在其中指定 XSA 以创建新平台。使用 `binary_container_1.xsa`。



c. 选择 XSA 后，Vitis IDE 会读取该文件、判定由 XSA 所定义的域的“Operating system”（操作系统）和“Processor”（处理器），并在该对话框中进行填充。单击“Finish”（完成）以创建平台工程。

> **提示：**根据为平台所选的固定 XSA，裸机平台对硬件仿真或硬件构建有效。

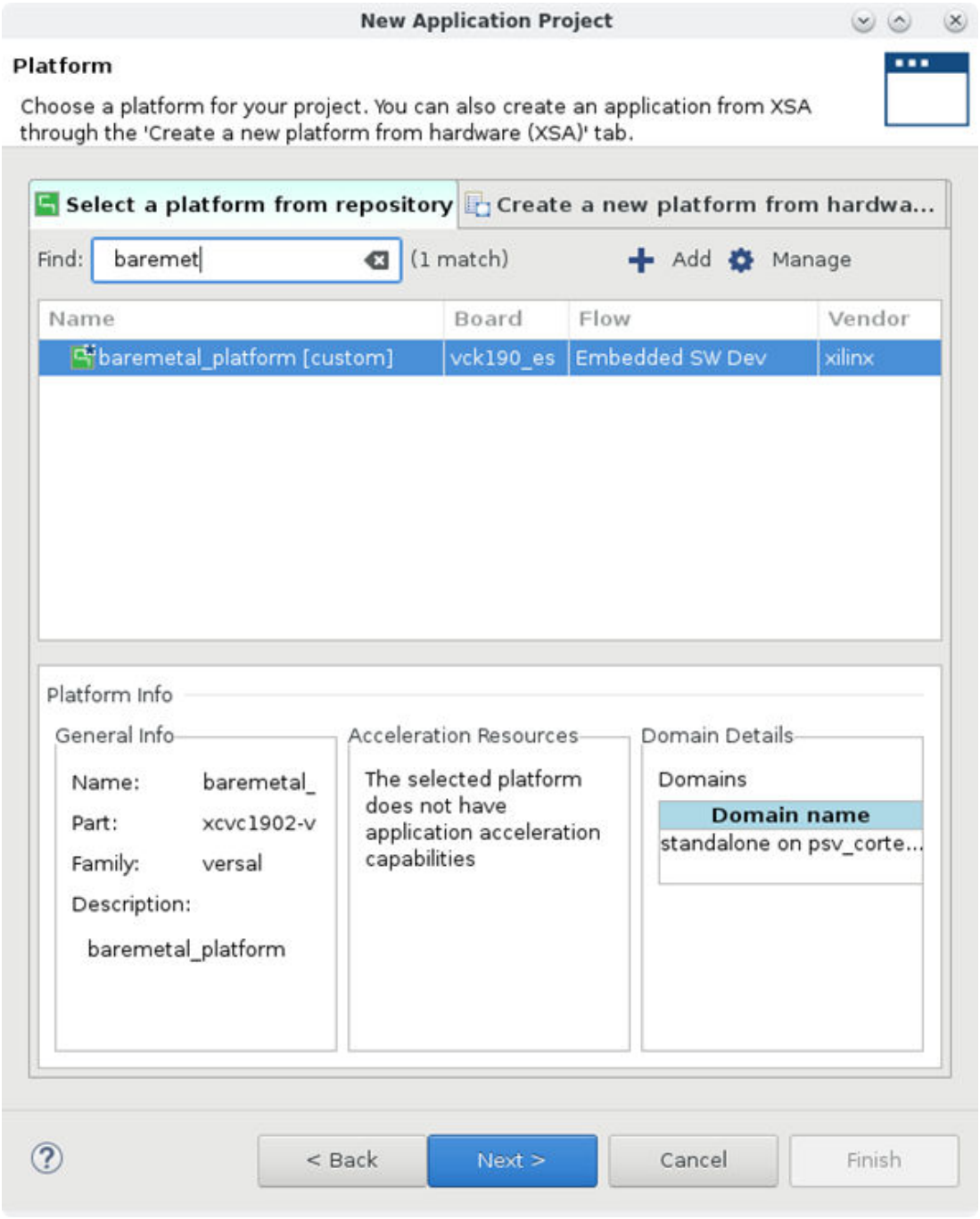
d. 单击“Build”（构建）命令以构建平台。在该工程的导出文件夹中会写入已完成的平台的副本，并在该工程的“Explorer”（资源管理器）视图内显示此副本。如下图所示，导出的平台包含 `platform.xpfm` 元数据文件以及平台的不同元素的 `./hw` 和 `./sw` 文件夹。



Vitis IDE 会将新平台自动添加到您的平台存储库中，使其可用于新工程。您也可在自己的 `$PLATFORM_REPO_PATHS` 环境变量中添加文件位置。这样此平台即可供 Vitis IDE 访问，或者您可以在命令行中通过引用平台名称而不是整个路径的方式来指定平台。

重要提示！生成的平台将仅用于在流程中构建裸机 PS 应用，而不可作他用。

2. 创建新的 PS 应用工程。
   a. 选择 Vitis IDE 中的“File” → “New” → “Application Project”（文件 > 新建 > 应用工程）命令。这样即可打开“New Application Project”Wizard（新建应用工程向导）。
   b. 单击“Next”（下一步）跳过第一页，并显示“Platform”（平台）页面，如下所示。
   c. 选中您在上一步中创建的“baremetal_platform”，然后单击“Next”（下一步）继续。

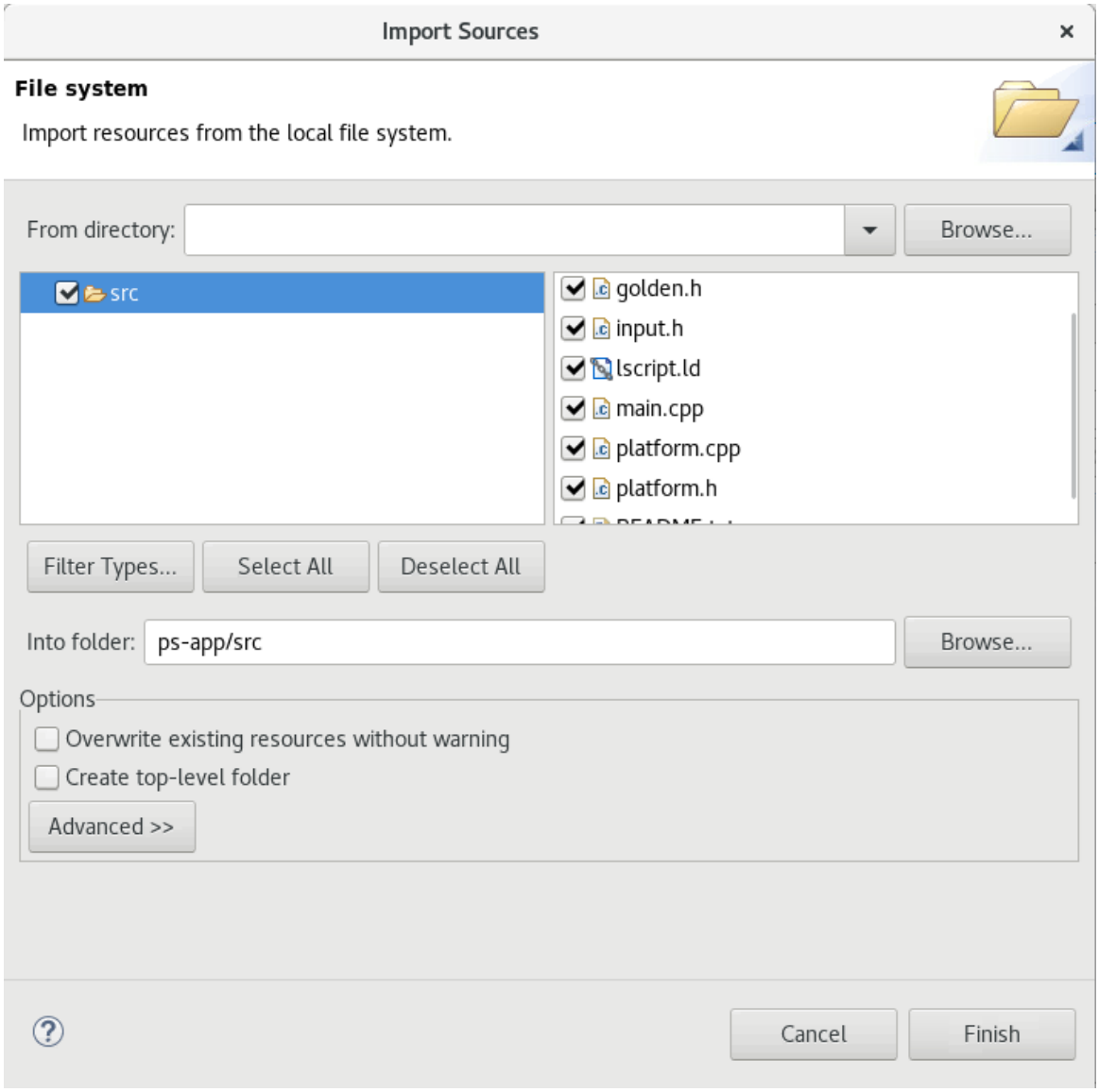


d. 提供“Application project name”（应用工程名称），然后单击“Next”（下一步）。

e. 复查“Domain”（域）页面，然后单击“Next”（下一步）继续。

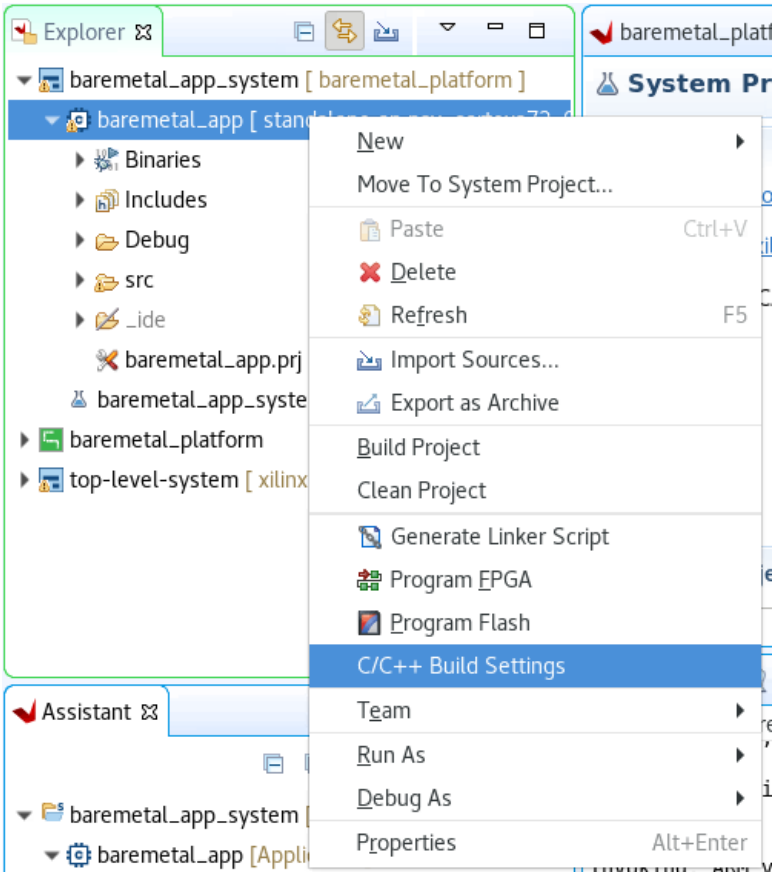
f. 在“Templates”（模板）页面上，选中“Empty Application”（空应用），然后单击“Finish”（完成）以创建工程。这样会在 Vitis IDE 中打开此工程。

g. 为 PS 应用 `main.cpp`、`platform.cpp` 和专为裸机工程编写的关联文件添加源代码。在“Explorer”（资源管理器）视图中选中该工程、展开文件夹、右键单击 `src` 文件夹，然后单击“Import Sources”（导入源文件）命令 ( ) 以打开下图所示对话框。选择要添加的文件，然后单击“Finish”（完成）。

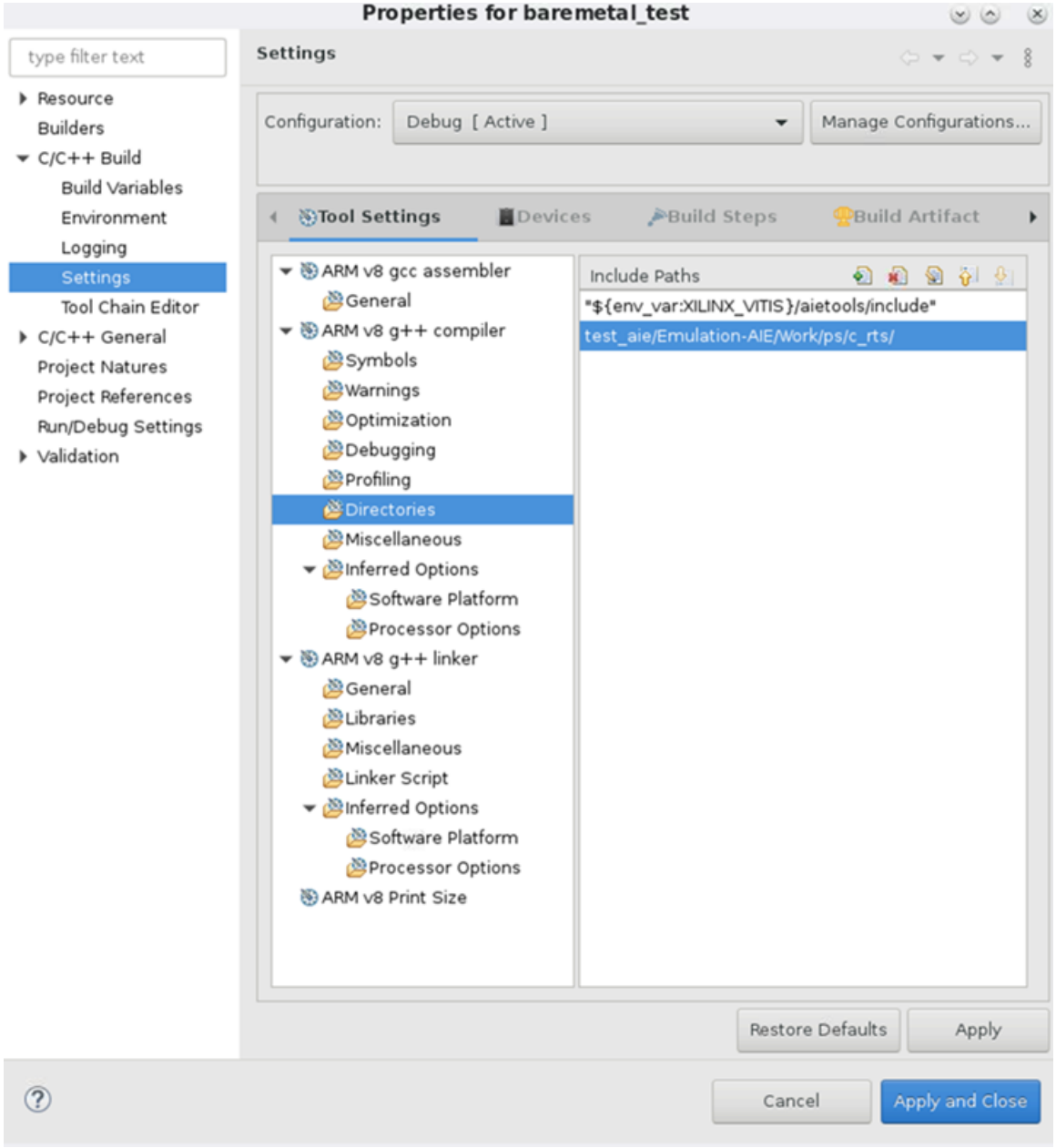


h. 您还必须添加裸机 AI 引擎控制文件 (`aie_control.cpp`)，此文件是由 `aiecompiler` 命令创建的，可在 `./Work/ps/c_rts` 文件夹下找到。再次选中 `src` 文件夹，然后单击“Import Sources”图标 ( ) 以打开对话框并向工程添加 `aie_control.cpp` 文件。

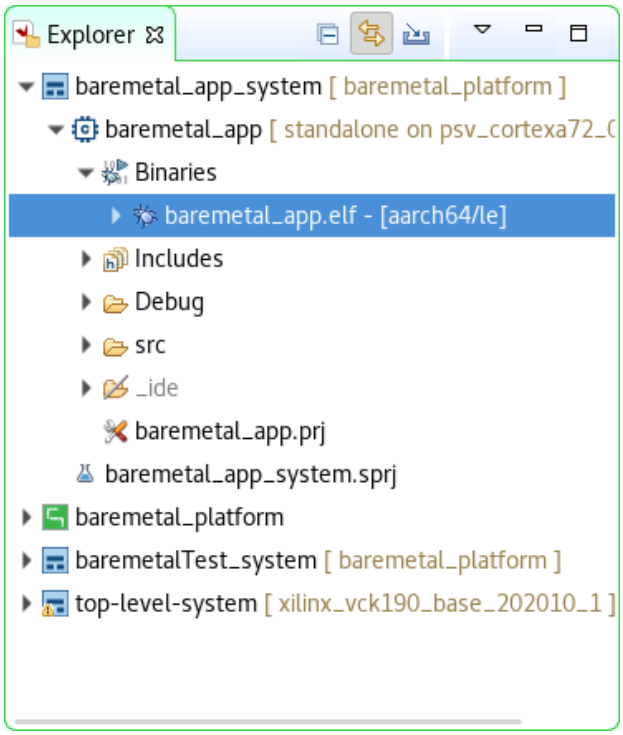
i. 最后，必须将其添加到工程的“Include”（包含）路径中。右键单击裸机应用工程，然后选中“C/C++ Build Settings”（C/C++ 构建设置），如下所示。



该选项会打开“Build Settings”（构建设置）对话框，如下图所示。如图所示，选中“Directories”（目录）选项，然后选中“Add”（添加）命令 () 以添加新的 include 路径。您将需要为自己的原始 AI 引擎 graph 应用添加对应源文件的条目。此处应指定工程的 `src` 文件夹，如 创建 AI 引擎 graph 工程和顶层系统工程 中所述。它应指向包含 AI 引擎 graph 源文件的文件夹。单击“Apply and Close”（应用并关闭）完成 include 路径的定义。



更新“Include”路径后，请选中“Build”（构建）( ) 图标以构建工程。构建完成后，您应可看到对应自己的裸机应用的 ELF 文件。



3. 封装系统。

   a. 为 PS 应用生成 ELF 文件后，准备即已完成，您可开始构建系统级别工程，并为裸机平台封装该系统。您必须运行封装进程来生成最终可启动镜像 (PDI)，并写入 SD 卡内容用于启动器件和运行应用。如需了解更多信息，请参阅 封装。

   b. 在您设置 AI 引擎 graph 工程时所创建的系统工程（按 创建 AI 引擎 graph 工程和顶层系统工程 中所述）中，为“Packaging options”（封装选项）字段添加以下内容：

      ```
      --package.ps_elf ../../baremetal_app/Debug/baremetal_app.elf,a72-0
      ```

      注释：要同时调试 AI 引擎 graph 和裸机 PS 应用，请勿添加先前示例中指定的 `--package` 选项。如果仅需调试 AI 引擎 graph，则需添加所述的选项。

   c. 现在，请构建 ( ) 顶层系统工程。

      这样即可添加您在裸机应用工程中创建的 ELF 文件、将其分配给处理器核并构建系统工程。如需了解更多信息，请参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的 Vitis 编译器命令中的 `--package` 选项。

      注释：在 Linux 系统中，您已向系统工程添加了 PS 应用，它会作为系统的一部分用于构建和调试。您在此处构建 PS 应用作为独立的裸机工程的一部分，并将其作为封装进程的启动文件添加到您的顶层系统中。

鉴于您已构建裸机系统，现在您可继续运行或调试应用。

重要提示！在 Vitis IDE 中，您无法为裸机工程调试硬件仿真构建。

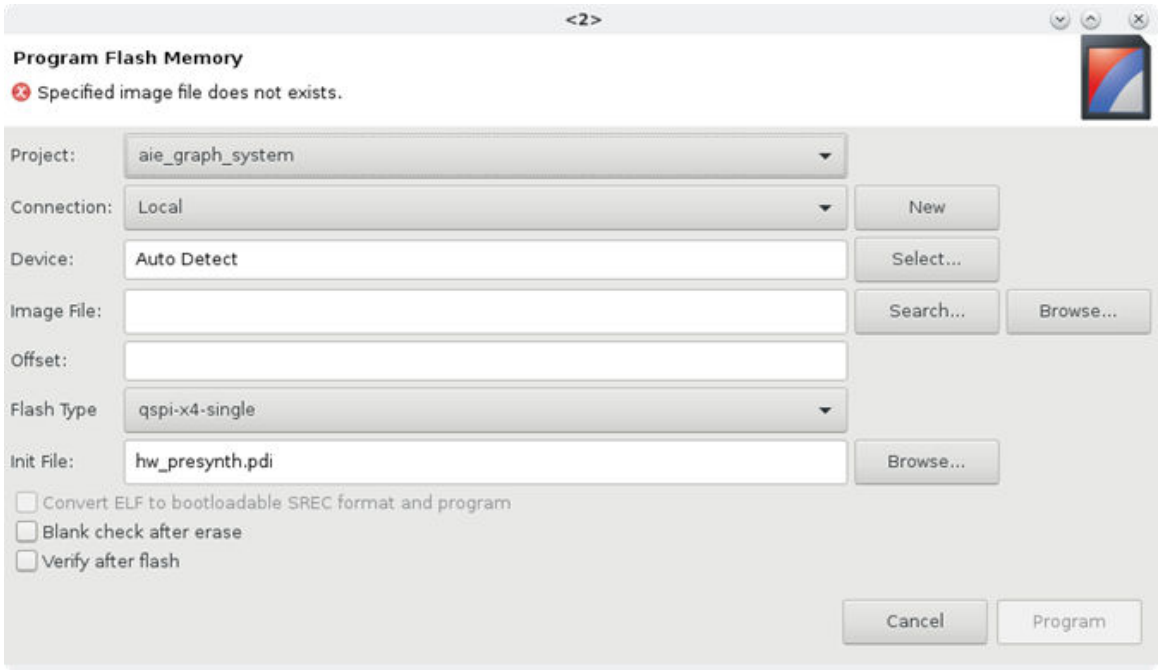
器件和闪存编程

Vitis IDE 具有使用 JTAG 对开发板/器件进行编程的功能特性，您无需将 `sd_card` 复制到实际 SD 卡上以启动开发板。

闪存编程

成功完成硬件设计构建后，请右键单击“System Project”（系统工程），然后选择“Program Flash”。这样应可看到如下所示对话框：

图 78：**“Program Flash Memory”对话框**



在此屏幕中，指定“Image File”（镜像文件，通常为 `BOOT.BIN`）、任意偏移（如果闪存需要偏移）以及“Init File”（启动文件）。“Init File”是来自您的目标平台的部分 PDI。为确保正确完成闪存编程，请选中“Verify after flash”（烧写后验证）复选框以确认它是否已正确完成编程。如果闪存已包含现有数据，那么可以选中“Blank check after erase”（擦除后空白检查）复选框来执行更安全的擦除，确保它被完全擦除后再执行编程。

请确保“System Project Settings”（系统工程设置）窗口中的“Packaging options”（封装选项）已设置 `--package.boot_mode=qspi`。

器件编程

如果无需闪存编程，只想通过 JTAG 进行器件编程，可在“Xilinx”（赛灵思）菜单下选中“Program Device”（器件编程）选项。

AMD XILINX

第 10 章

调试 AI 引擎应用

您可调试 AI 引擎 graph 应用，在独立模式下仅运行和调试 AI 引擎。或者，您可调试系统工程（包括顶层 PS 应用）和 AI 引擎 graph。在此框架内，您也可以调试从命令行构建的应用或者 Vitis™ IDE 中构建的系统工程。您可对 Linux 操作系统或裸机系统中运行的应用进行调试。最后，您可调试硬件仿真构建以供您执行应用仿真或者调试硬件上运行的实际应用。在以下主题中对所有这些配置都进行了解释。

注释：建议遵循以下步骤对 AI 引擎应用进行调试。

1. 使用 x86 仿真来执行单内核与多内核调试。它支持断点和使用 GNU 调试器来进行单步步进。
2. 使用 AI 引擎仿真来验证时序，检查是否能填入程序存储器和可用硬件存储器空间内的栈/堆大小。
3. 执行软件仿真来进行系统级功能验证（包括主机代码）以及用于 PL 代码的任意 C 语言模型。
4. 执行硬件仿真来进行完整的集成测试，包括 PL、PS 和 AI 引擎域（如适用）。
5. 建议使用编译器最优化选项 `--xlopt = 0`，因为更高的编译器最优化将降低调试可视性。

如果适用代码更改，请重复步骤 1 到 4。

从 Vitis IDE 启动调试

如 第 9 章：使用 Vitis IDE 中所述，您可构建系统级别工程，以整合 AI 引擎域中运行的内核、PL 域以及 PS 域中运行的应用。这些系统级别工程可在 Vitis IDE 内一起进行调试，或者您可聚焦特定平台来调试 AI 引擎 graph 应用。本节探讨了如何从 Vitis IDE 运行“Debug Environment”（调试环境）以进行硬件仿真和硬件构建。

使用 printf 执行事件追踪

最简单的追踪形式是在代码中使用格式化 `printf()` 语句打印调试消息。对中间值、地址等执行视觉检验有助于您了解程序执行的进程。使用 `printf()` 时，除了标准 C/C++ include 文件 (`stdio.h`) 外，无需使用其它 include 文件。您可以将 `printf()` 语句添加到自己的代码中，以供在仿真期间或者在硬件仿真期间进行处理，并在硬件构建时移除这些语句或者将其注释掉。

将 `printf` 语句添加到 AI 引擎内核代码中将增加 AI 引擎程序编译后的大小。请确保内核代码编译后的大小不超过每个 AI 引擎处理器存储器限制 (16 KB)。

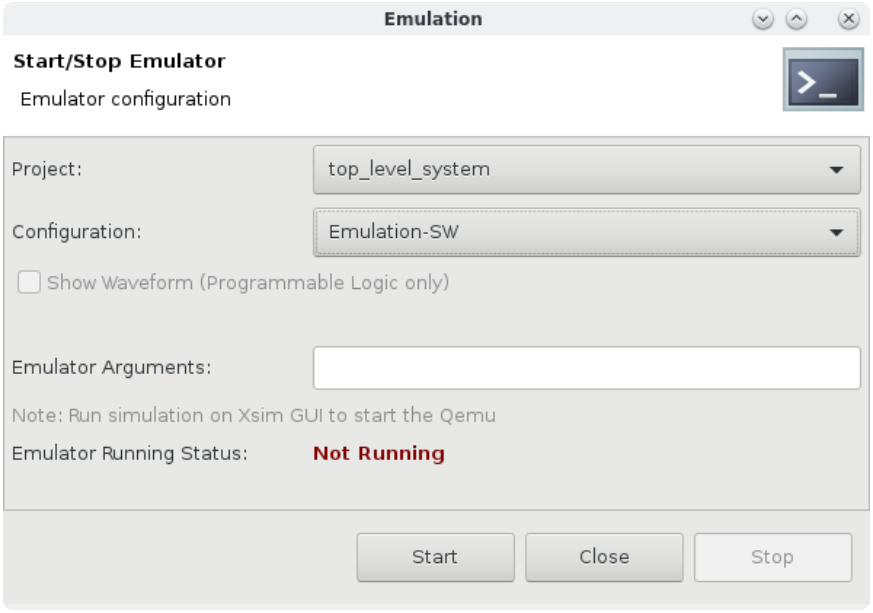
重要提示！您必须使用 `aiesimulator --profile` 命令在仿真器运行期间启用 `printf()` 执行。如果未指定 `--profile`，则忽略 `printf()` 函数。

对于该功能，使用独立驱动程序和二进制文件即可允许主仿真器尽可能保持高速。使用调试仿真器驱动程序可在输出目录下为每个拼块生成剖析报告，以提供详细的周期级别内核执行统计数据。此外，使用 `--profile` 选项可生成 `run_summary` 文件，并将其写入 `./aiesimulator_output` 文件夹，此文件可按 在 Vitis 分析器中查看运行汇总 中所述方式来查看。

从 Vitis IDE 进行软件仿真调试

在软件仿真中运行应用之前，您需要使用“Emulation-SW”（软件仿真）构建目标来构建系统工程。要在“Emulation-SW”构建目标中运行和调试应用，必须使用以下步骤。

1. 选择“Xilinx” → “Start/Stop Emulator”（赛灵思 > 启动/停止仿真器）启动 QEMU 仿真环境。



   这样即可启动仿真器，然后等待至 QEMU 内 Linux 完成启动为止。“Emulation”（仿真）控制台会显示 QEMU 启动和 Linux 启动进程的转录文本。当进度对话框关闭并且“Emulation Console”（仿真控制台）窗口显示空白 `qemu%` 时，即表示此进程已完成。您可以检查转录文本以获取该进程的详细信息。

   启动软件仿真时，您可以为运行 graph 应用的 AI 引擎仿真器指定选项。如 复用 x86 仿真器选项 中所述，在前图中所示的“Emulator Arguments”（仿真器实参）字段中可以使用以下命令指定这些选项。

```
-x86-sim-options ${FULL_PATH}/x86sim.options
```

   注释： `${FULL_PATH}` 是指向 `x86sim.options` 文件位置的完整路径。

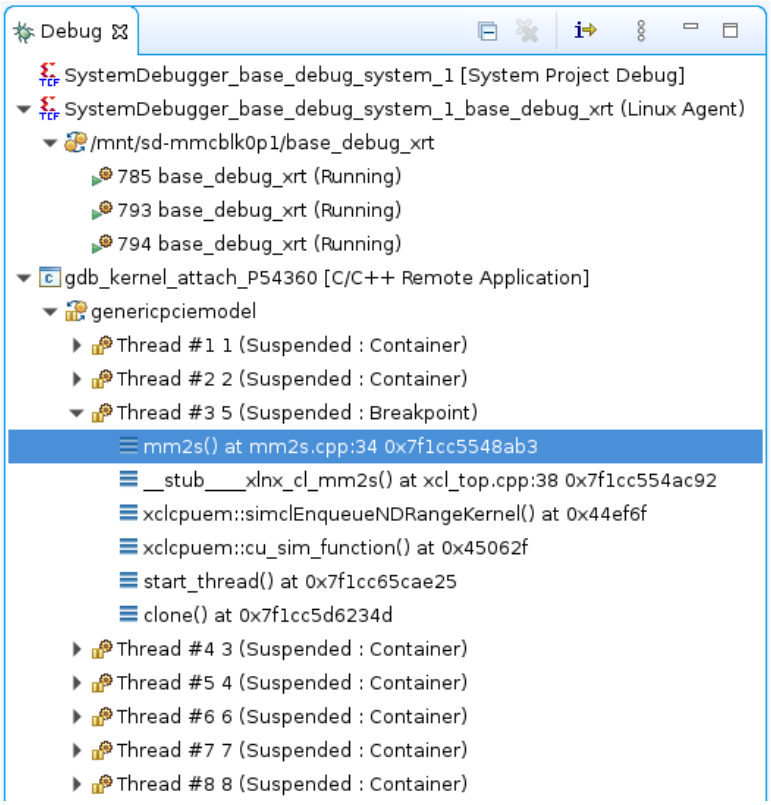
   重要提示！打开主机应用源文件。右键单击 `int main()` 一侧以设置断点，因为 IDE 不会在 `main()` 处自动设置断点。

2. 右键单击顶层系统工程，然后选择“Debug As” → “Launch SW Emulator”（调试方式 > 启动软件仿真器）命令。这样会打开“Debug Configurations”（调试配置）对话框。
3. 单击“Debug”（调试）以继续。

   这样即可在 Vitis IDE 中打开“Debug”透视图，并连接至 QEMU 中其各自的线程上运行的 PS 应用和 AI 引擎 graph。

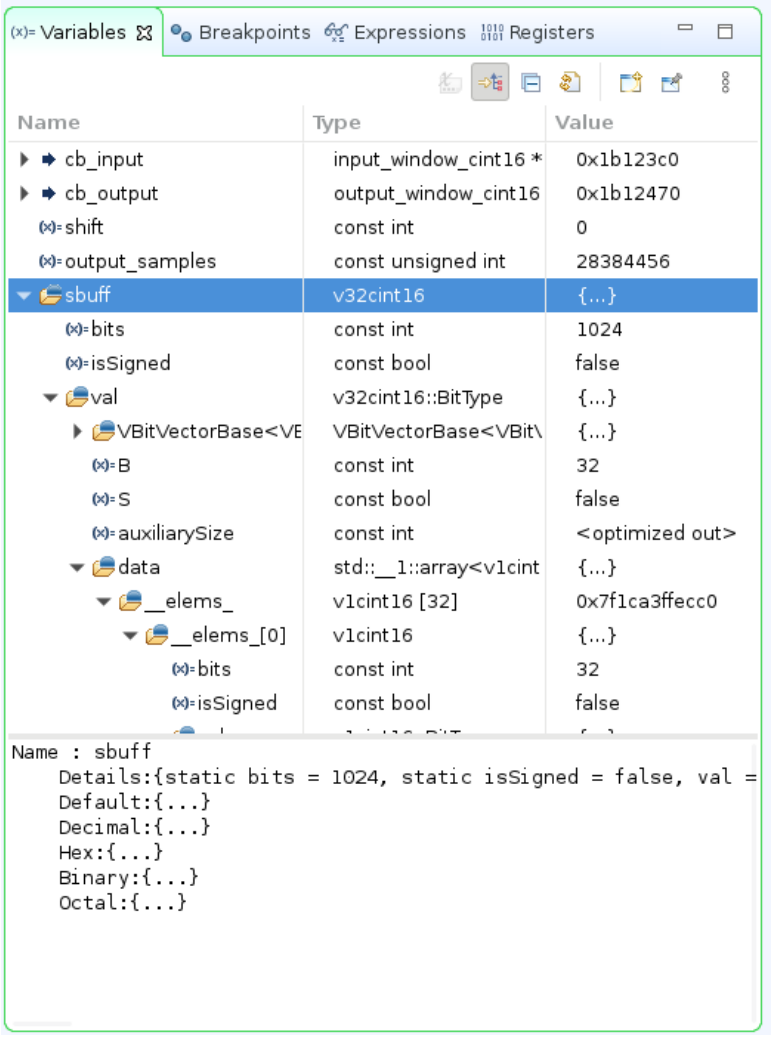
   注释： 如需了解调试期间使用的视图的更多详细信息，请参阅 用于软件仿真调试的 Vitis IDE 布局。

4. 单击“Resume”（恢复）按钮 转至下一个断点。观察“Debug”视图可以看到各种派生的线程。

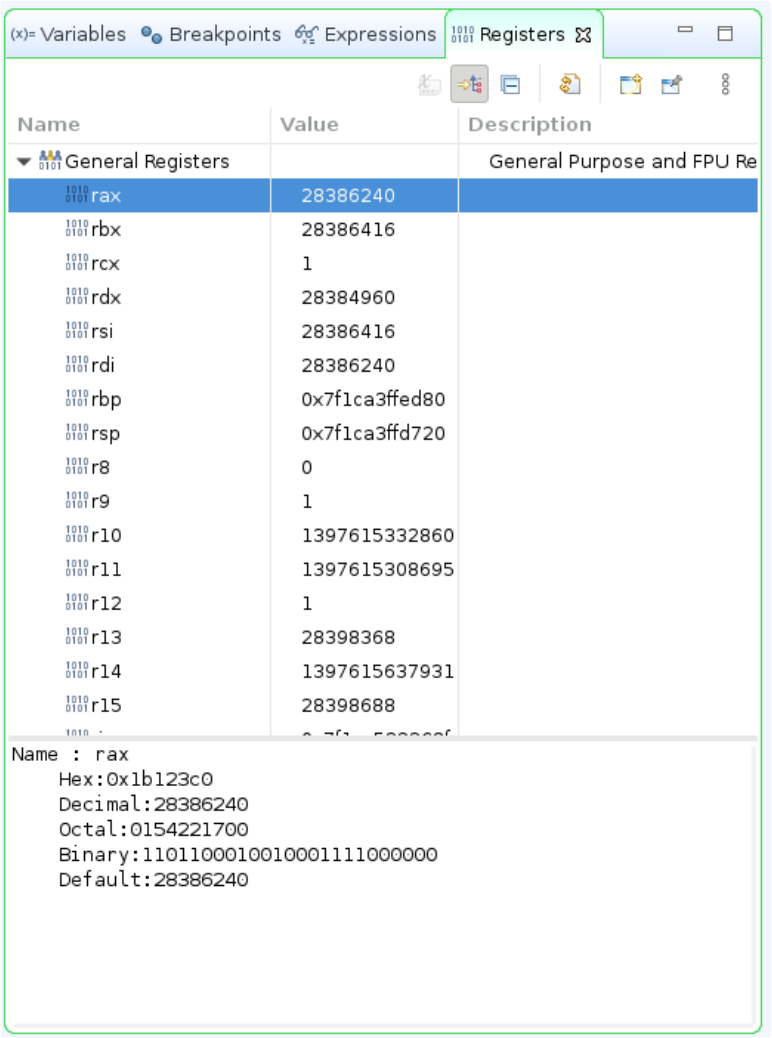


注释： “Debug”视图还会显示已命中断点的线程，并带有信息（暂挂：断点）。

5. 请复查“Variables”（变量）视图。此视图显示所有变量和对象以及 AI 引擎内核（如下图所示）。例如，下图显示了展开的“sbuff”，其中显示“v32cint16”数据类型已格式化。



6. 单击“Registers”（寄存器）视图。在 AI 引擎内核中触发断点后，此视图会显示所有相关触发器。

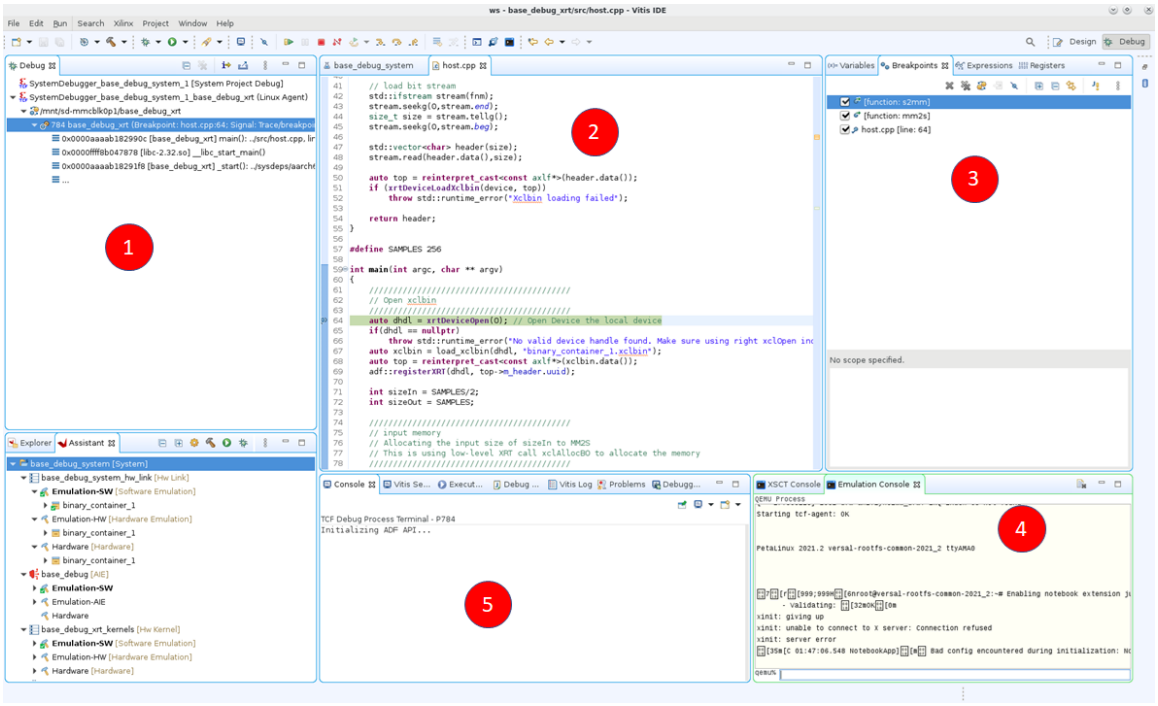


注释：如果在主机应用源中触发断点，那么它会显示 PS 寄存器。

用于软件仿真调试的 Vitis IDE 布局

本节描述了 Vitis IDE 用于对“Emulation-SW”（软件仿真）进行有效调试的方方面面。

下图显示了“Debug”（调试）透视图中的 Vitis IDE 典型布局。



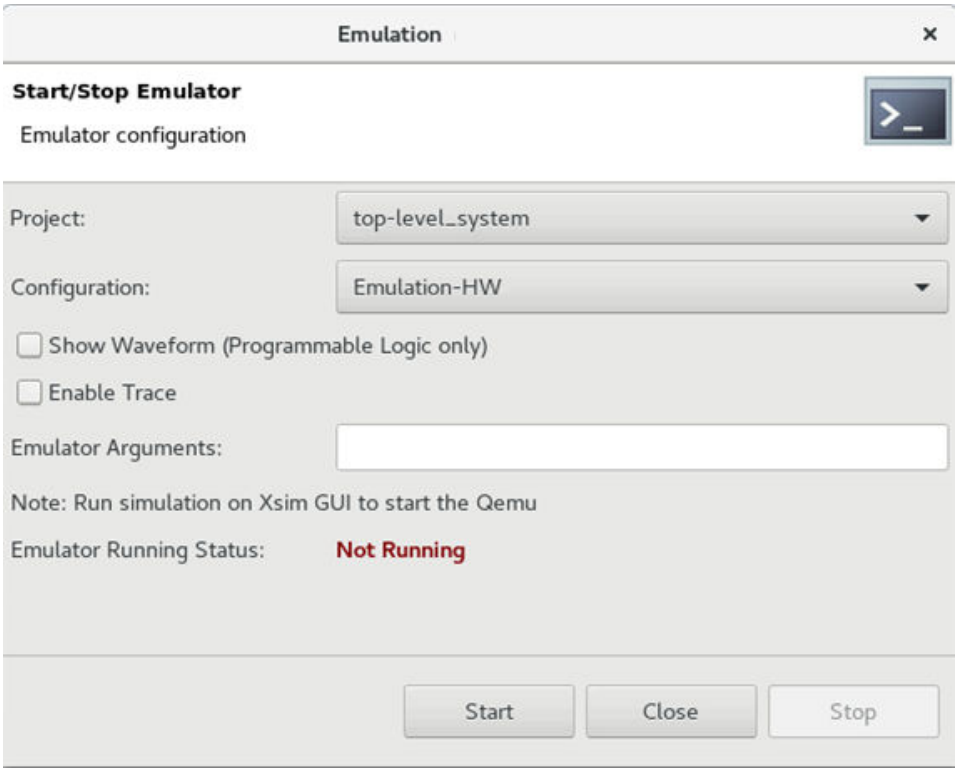
各节按编号如下所述：

1. “Debug”视图显示了与运行 PL、AI 引擎和 PS 应用（作为 PS 主机应用、PL 内核及 AI 引擎内核）关联的线程列表。
2. 代码窗口。触发源文件中的断点时，该窗口会打开此文件。
3. “Variables”（变量）、“Breakpoints”（断点）、“Expressions”（表达式）和“Registers”（寄存器）视图。
   - “Variables”视图显示了在触发的断点透视图中使用的所有可用变量。例如，如果断点位于主机应用源文件中，那么它将显示与此代码关联的所有对象和变量。
   - “Breakpoints”视图显示了当前正在调试的设计的所有断点（已启用或已禁用）。
   - “Expressions”允许编写特定表达式，用于在代码执行时触发暂停。例如，检查 for 循环变量何时到达某个值，到达该值时即中断。
   - “Registers”视图显示了在主机应用源代码中触发断点时的 Cortex®-A72 寄存器，在 AI 引擎内核中触发断点时，此视图会显示 AI 引擎。
4. “Emulation Console”（仿真控制台）视图和“XSCT Console”（XSCT 控制台）视图。
   - “Emulation Console”可提供 QEMU 的转录文本，并允许您运行 Linux 命令。
   - “XSCT Console”允许您在触发断点时查看程序存储器和代码片段。
5. “Console”（控制台）视图显示了程序执行输出结果。

从 Vitis IDE 进行硬件仿真调试

要在“Emulation-HW”（硬件仿真）构建目标中运行和调试应用，必须使用以下步骤：

1. 选择“Xilinx” → “Start/Stop Emulator”（赛灵思 > 启动/停止仿真器）命令启动 QEMU 仿真环境命令。



这样即可启动该仿真器，然后等待至 QEMU 内 Linux 完成启动为止。“Emulation”（仿真）控制台会显示 QEMU 启动和 Linux 启动进程的转录文本。当进度对话框关闭并且 `qemu%` 提示为空白时，即表示此进程已完成。您可以检查转录文本以获取该进程的详细信息。

启动硬件仿真时，您可以为运行 graph 应用的 AI 引擎仿真器指定选项，如 复用 AI 引擎仿真器选项 中所述。您可在“Emulator Arguments”（仿真器实参）字段中通过指定以下命令来指定这些选项，如前图所示：

```
-aie-sim-options ../aiesim_options.txt
```

2. 右键单击顶层系统工程，然后选择“Debug As”→“Launch HW Emulator”（调试方式 > 启动软件仿真器）命令。这样会打开“Debug Configurations”（调试配置）对话框。
3. 按“Debug”以继续。

   这样即可在 Vitis IDE 中打开“Debug”透视图，并连接至 QEMU 中其各自的核上运行的 PS 应用和 AI 引擎 graph。此应用会在所有 ELF 文件的 `main()` 函数处自动中断。

此时，您即可在仿真环境中执行所有调试活动，如单步进入、单步跳过、查看变量或植入断点。如需了解更多信息，请参阅 使用调试环境。

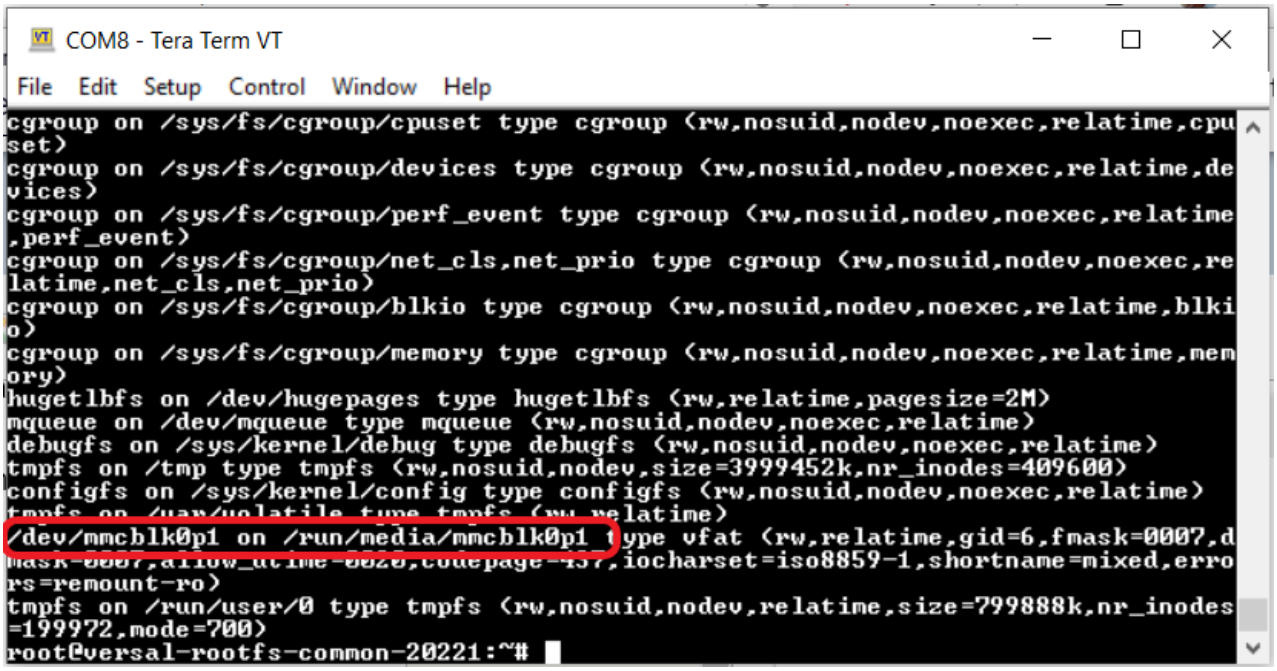
从 Vitis IDE 进行硬件调试

构建完顶层系统工程后，必须使用以下步骤在“Hardware”（硬件）构建目标中调试应用。

1. 将 `<project>/Hardware/package/sd_card.img` 烧写到物理 SD 卡上。这样即可为您的目标平台创建一个可启动的介质。
2. 将此 SD 卡插入 VCK190 评估套件的读卡器。
3. 将卡的启动模式设置更改为 SD 启动模式，然后给开发板上电。

4. VCK190 启动后，在命令提示符处输入 `mount` 命令以获取装载点列表。如下图所示，`mount` 命令显示了系统的装载信息。

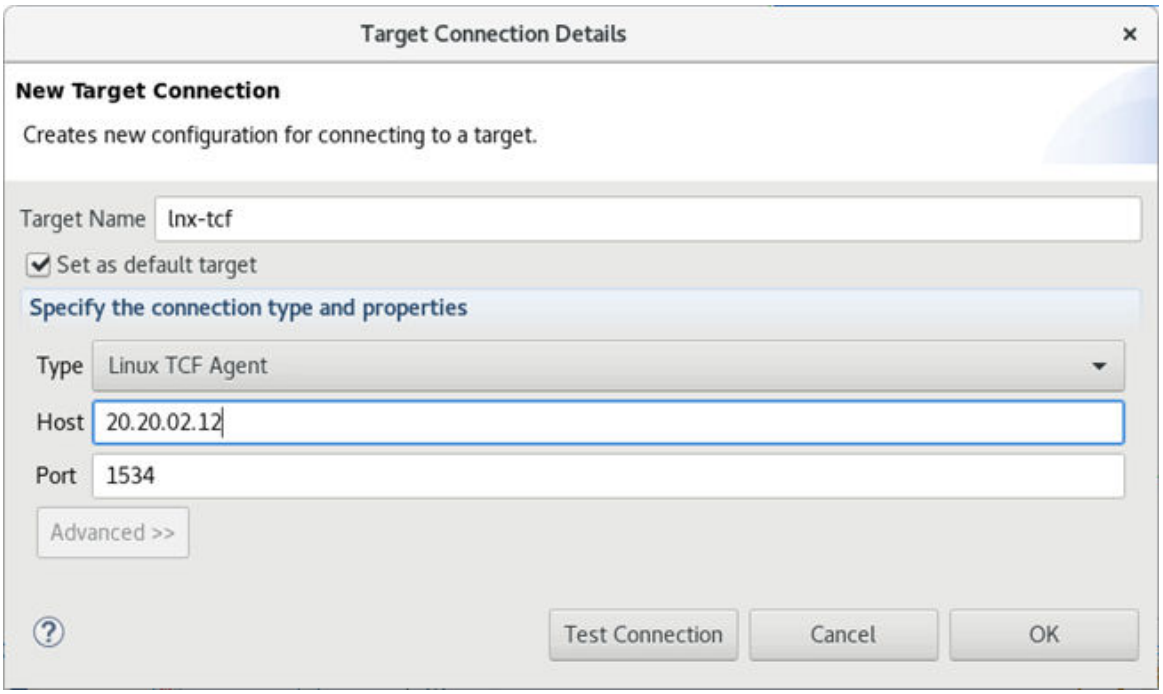
> **提示：**根据 `mount` 命令结果，请务必捕获下一步中 `cd` 命令以及后续命令的适当路径。



5. 执行下列命令：

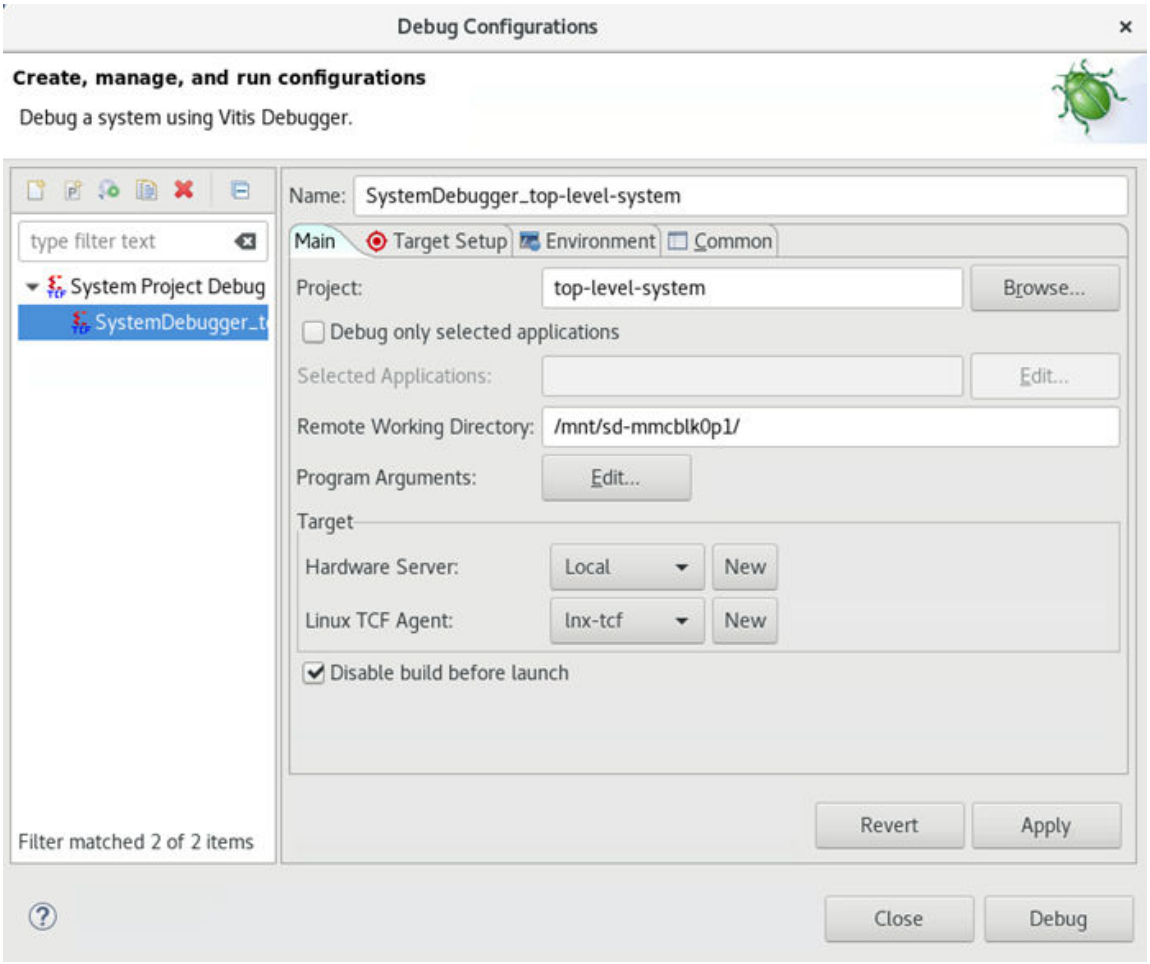
```
cd /run/media/mmcblk0p1
```

6. 运行 `ifconfig` 以获取目标卡的 IP 地址。IP 地址用于在 Vitis IDE 中设置 TCF 代理连接。此目标需连接到网络分配的 IP 地址。
7. 创建到远程加速器卡的目标连接。选择“Window” → “Show view” → “Xilinx” → “Target connections”（窗口 > 显示视图 > 赛灵思 > 目标连接）命令打开“Target Connections”视图。
8. 在“Target Connections”视图上，右键单击“Linux TCF Agent”（Linux TCF 代理）并选择“New Target”（新建目标）命令打开“New Target Connection”（新建目标连接）对话框。
9. 指定“Target Name”（目标名称）、启用“Set as default target”（设为默认目标）复选框，然后指定先前步骤中获取的加速器卡的“Host”（主机）IP 地址。



10. 单击“OK”即可关闭此对话框并继续操作。
11. 右键单击顶层系统工程，然后选择“Debug As”→“Debug Configurations”（调试方式 > 调试配置）命令。

    这样即可打开“Debug Configurations”对话框以供您设置工具。对于“Hardware”构建，您将需要创建两个 launch（启动）配置：一种配置用于顶层系统工程，另一种配置则用于 PS 应用。
12. 在“Debug Configurations”对话框中，选中“New Launch Configuration”（新建启动配置）( ) 命令以打开“Debug Configurations”对话框，如下所示。



请务必在对话框中设置以下字段，如前图所示。

- “Remote Working Directory”（远程工作目录）：指定来自先前步骤中确定的加速器卡的远程装载位置。
- “Linux TCF Agent”（Linux TCF 代理）：选择您使用指定 IP 地址为加速器卡构建的新代理。
- “Disable build before launch”（禁用启动前构建）：需要该选项的原因是因为，如果不使用该选项，工具将尝试在运行应用前构建系统。

13. 选择“Apply”（应用）以应用更改，然后选择“Debug”（调试）以启动此进程。

    这样即可在 Vitis IDE 中打开“Debug”透视图，并连接至 QEMU 中其各自的核上运行的 PS 应用和 AI 引擎 graph。此应用会在所有 ELF 文件的 `main()` 函数处自动中断。

此时，您即可在仿真环境中执行如下所有调试活动：单步进入、单步跳过、查看变量或应用断点。如需了解更多信息，请参阅 使用调试环境。

从 Vitis IDE 进行裸机调试

按 在 Vitis IDE 中构建裸机 AI 引擎 中所述完成裸机系统工程构建后，您必须使用以下步骤来调试 AI 引擎 graph 和“Hardware”（硬件）构建目标上的裸机 PS 应用。

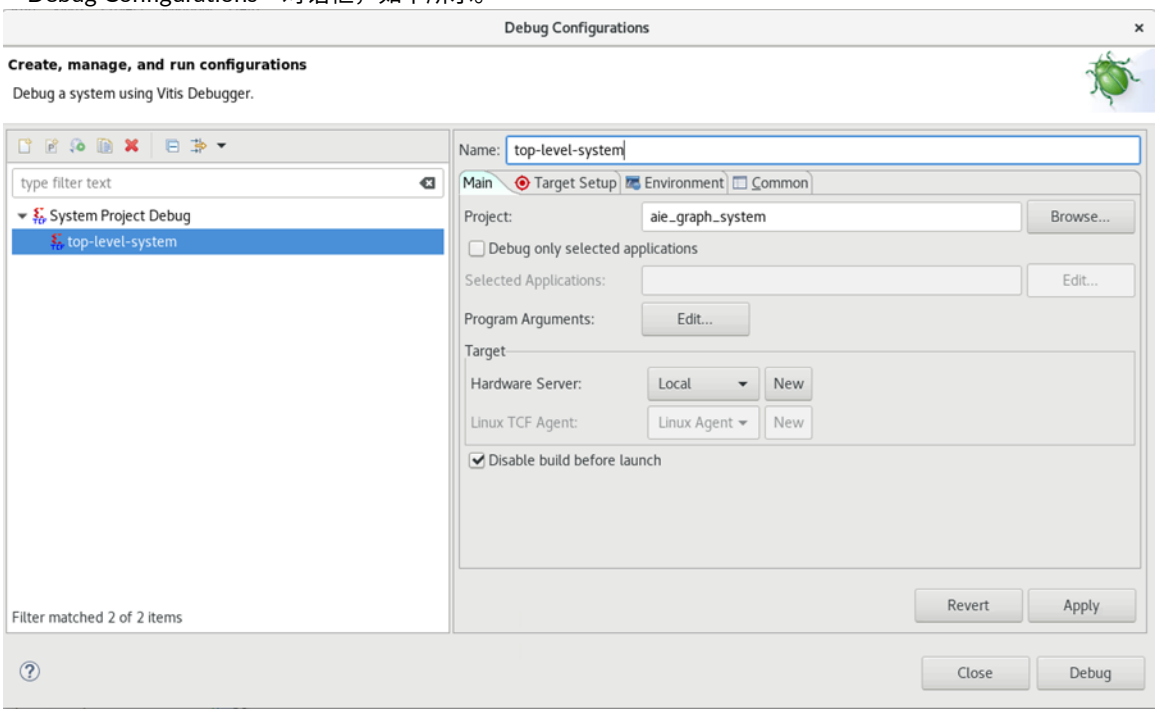
此进程比在 Vitis IDE 中调试 Linux 应用略复杂。后者中 PS 主机应用是作为顶层系统工程的一部分来完成构建的。此处的 PS 应用则是作为独立工程来构建的，并且必须在启动调试环境时单独包含在调试配置内。欲知详情，请参阅下列步骤。

1. 右键单击顶层系统工程，然后选择“Debug As”→“Debug Configurations”（调试方式 > 调试配置）命令。

   这样即可打开“Debug Configurations”对话框以供您设置工具。

   重要提示！对于“Hardware”构建，您将需要创建两个 Debug（调试）配置：一种配置用于顶层系统工程，另一种配置则用于裸机 PS 应用。

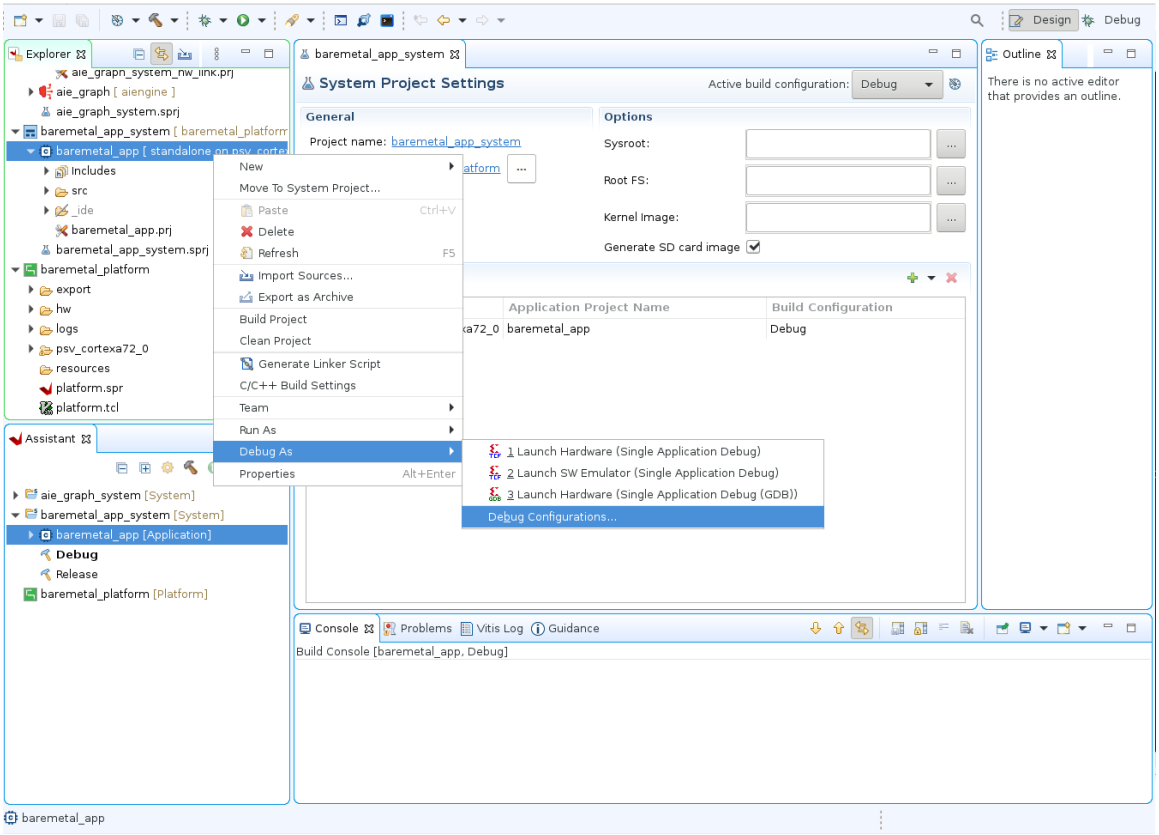
2. 在“Debug Configurations”对话框中，选中“New Launch Configuration”（新建启动配置）( ) 命令以打开“Debug Configurations”对话框，如下所示。



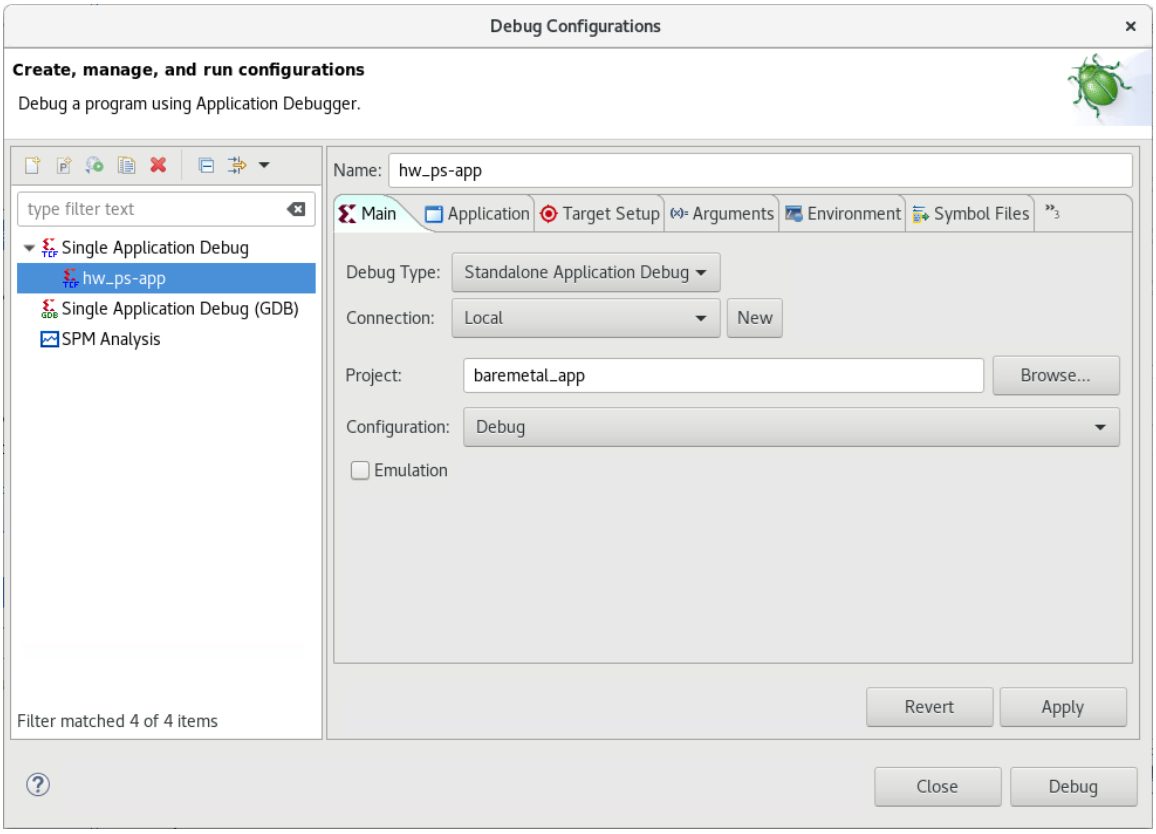
   请注意“Debug Configurations”对话框上的下列字段：

   - “Project”（工程）：反映顶层系统工程的名称，其中包括 AI 引擎 graph 应用、PL 内核以及 HW-Link（硬件链接）工程。
   - “Hardware Server”（硬件服务器）：指定到开发板的本地连接。对于远程连接的开发板，可指定不同配置。
   - “Linux TCF Agent”（Linux TCF 代理）：对于裸机系统禁用此项。
   - “Disable build before launch”（禁用启动前构建）：启用此项可以防止该工具在启动应用前构建系统。

3. 选中“Apply”（应用）以保存并应用您的更改，选中“Close”（关闭）即可关闭对话框。
4. 右键单击“Explorer”（资源管理器）视图中的“baremetal_app”工程，然后选中“Debug As”→“Debug Configurations”，如下所示。

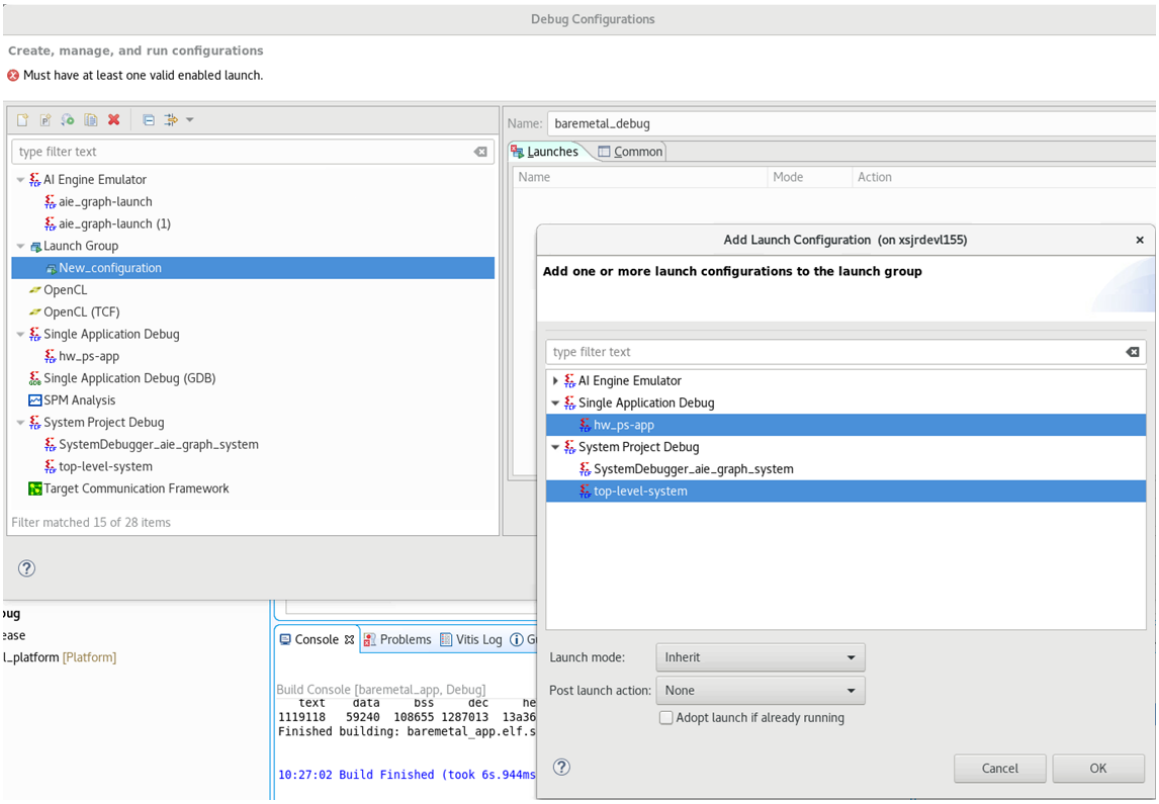


5. 这样会打开“Single Application Debug Configurations”（单应用调试配置）对话框，如下图所示。



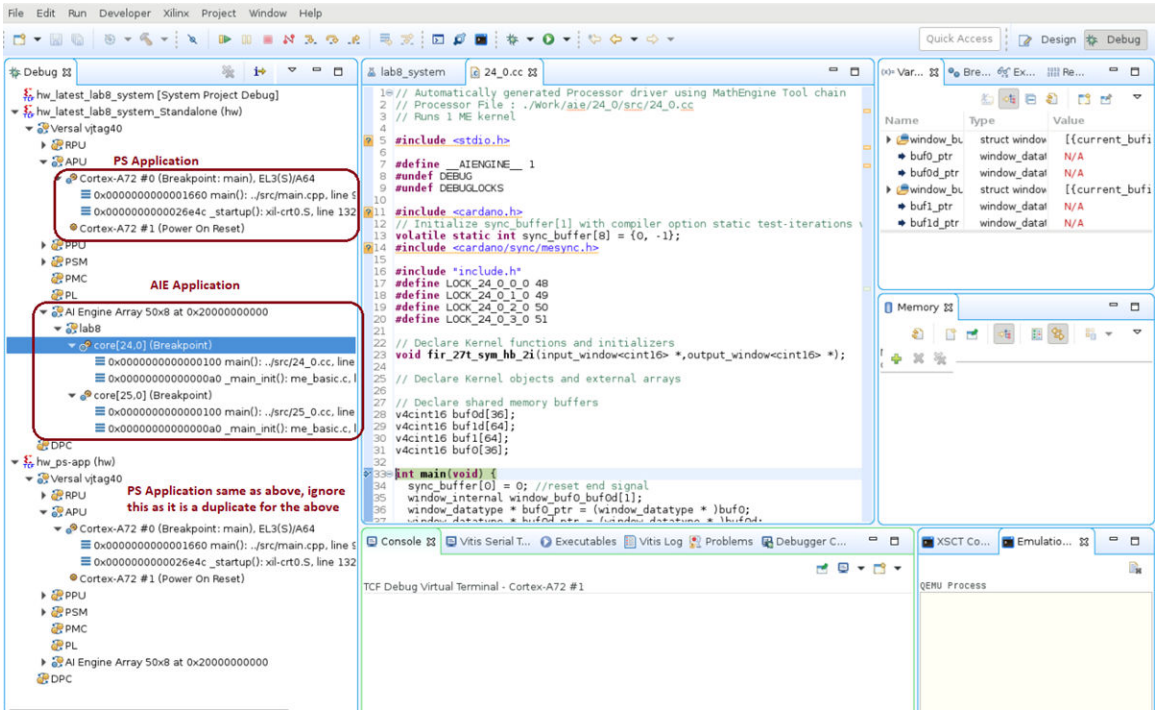
6. 指定“Name”（名称）以识别要应用到 PS 应用的配置。对于本地硬件，“Connection”（连接）会自动完成设置。
7. 切换为选中“Debug Configurations”对话框的“Target Setup”（目标设置）选项卡，取消选中对话框中的“Reset entire system”（复位整个系统）和“Program Device”（器件编程）复选框。
8. 单击“Apply”（应用）以继续。
9. 您还需要创建启动组，其中包含您刚创建的两种调试配置。启动组允许您将多种配置作为一个分组一起启动。使用主工具栏菜单，选中“Debug” → “Debug Configurations”以打开“Debug Configurations”对话框，然后选中“Launch Group”（启动组），如下所示。

重要提示！您必须使用主工具栏菜单“Debug”命令，因为它提供了完整的“Debug Configuration”选项集合，而不只是“Explorer”或“Assistant”视图中的精简选项，您的选择会限制后两个视图中的选项。



单击“Launch Group”，然后为新的组提供“Name”。单击“Add”以将顶层工程和 PS 应用调试配置添加到前图中所示的分组中。单击“OK”（确定）以创建启动组。

10. 对于添加的两个调试配置，都会显示“Launch Group”（启动组）。单击“Debug”以启动“Debug”透视图。
11. 由于您当前正在单一目标连接上同时启动顶层工程和 PS 调试配置，因此 Vitis IDE 将显示“Launch Config Conflict”（启动配置冲突）消息。在冲突消息上单击“No”（否）以继续。
12. 这样会打开“Debug”透视图，如下图所示。由于存在两个调试配置，您将看到两个 PS 应用实例。您可以忽略 PS 应用的第二个实例。



此时，您即可在仿真环境中执行下列所有调试活动：单步进入、单步跳过、查看变量或应用断点。如需了解更多信息，请参阅 使用调试环境。

从命令行启动调试

从命令行构建的调试工程可谓是特别的挑战，因为必须将系统的各种要素（编译后的 graph 应用 (`libadf.a`)、器件二进制文件 (XCLBIN) 和顶层应用 (`host.cpp`)）集合在一起并呈现为单个系统。在 Vitis IDE 中这是通过系统级工程来完成的，此工程是您首次创建 graph 工程时创建的。在命令行流程中，此服务是由 Vitis 编译器封装进程来提供的。如 封装 中所述，`v++ --package` 命令会将系统各要素集合在一起，为硬件仿真或硬件构建创建启动容器。

此外，您必须单独管理调试环境的不同要素，以运行 QEMU 仿真环境或者运行 `xrt_server` 以连接至硬件并启动 Vitis IDE 以运行调试进程。在后续主题中详细描述了命令行调试的步骤和要求。

从命令行运行软件仿真

完成“Software Emulation”（软件仿真）构建后，您可使用以下步骤来调试系统。此进程会启动新终端，以允许使用 `gdb` 命令并显示文件用于单步执行代码。

1. 使用 `launch_sw_emu.sh` 脚本启动 QEMU 仿真器环境，此脚本是在 `v++ --package` 进程期间生成的。
2. 使用专用命令行选项 `-kernel-dbg` 并将其设置为 `true`。
3. 指定内核、PL 或 AI 引擎内核。

1. 要含调试的启动仿真环境，请从构建目录中使用以下命令。

```
./emulation/launch_sw_emu.sh -kernel-dbg true
```

其中：

- `./emulation` 是封装进程的输出目录。
- `-kernel-dbg true` 将设置仿真器，以在执行应用时运行 `gdb`。

2. 出现 `qemu%` 提示时，请在 QEMU shell 中运行以下命令。

```
export LD_LIBRARY_PATH=/mnt/sd*1:/tmp:$LD_LIBRARY_PATH
export XCL_EMULATION_MODE=sw_emu
export XILINX_XRT=/usr
```

3. 运行 PS 应用。例如，`./host.exe a.xclbin`。

   这将开始运行主机应用，并在独立终端内启动 `gdb`。您可在此处执行所有调试活动，例如，断点、下一步以及继续执行 PL 内核与 AI 引擎内核。

   注释：要获取 `gdb` 的文本用户界面，请选中“Ctrl + X”和“Ctrl + A”。

从命令行进行硬件仿真调试

当“Hardware Emulation”（硬件仿真）构建完成（包括 AI 引擎 graph、PL 区域内核以及 PS 应用）后，您即可使用以下步骤来调试系统设计。此进程利用 Vitis IDE 从命令行启动调试环境。

- 使用 `launch_hw_emu.sh` 脚本启动 QEMU 仿真器环境，该脚本是在 `--package` 进程期间生成的。
- 在独立调试模式下使用 `vitis -debug` 选项启动 Vitis IDE。
- 配置调试环境以连接至系统内的 PS 和 AI 引擎域。

1. 对于 AI 引擎平台，系统仿真所需的文件由 `--package` 命令来定义，包括仿真脚本。要启动仿真环境，请从构建目录中使用以下命令：

```
./emulation/launch_hw_emu.sh -pid-file emulation.pid -no-reboot \
-add-env ENABLE_RDWR_DEBUG=true -add-env RDWR_DEBUG_PORT=10100 -forward-
port 1440 1534
```

   其中：

   - `./emulation` 是封装进程的输出目录，如 封装 中所述，其中还包含 `launch_hw_emu.sh` 脚本。
   - `-add-env RDWR_DEBUG_PORT=${aie_mem_sock_port}` 定义了用于与 AI 引擎域进行通信的端口。在前述示例中，此端口为 10100。
   - `-forward-port ${linux_tcf_agent_port} 1534` 定义了 Linux TCF 代理的端口。在前述示例中，此端口默认为 1440。

   提示：任何空闲端口均可用于以上命令模板中的 `aie_mem_sock_port` 和 `linux_tcf_agent_port`。但这些端口都是必需端口，用于分别启用 AI 引擎应用和 Linux 应用调试。

   此命令可启动仿真器，然后等待至 QEMU 内 Linux 完成启动为止。QEMU shell 会显示 QEMU 启动和 Linux 启动进程的转录文本。当显示 `qemu%` 提示时，即表示此进程已完成。至此您已准备就绪，可以继续下一步。

2. 在第二个终端窗口中，使用以下命令启动 XRT 服务器应用：

```
xrt_server -I300 -S -s tcp::4352
```

   其中：

   - `-I300` 定义了空闲超时，如果经此超时后无响应，服务器就会退出。
   - `-S` 以 JSON 格式指定打印服务器属性为“stdout”。

- `-s tcp::${xrt_server_port}` 定义了监听协议和端口的代理。在前述示例中，此端口为 4352，但可采用任意空闲端口。

3. 创建名为 `aie_app_debug.tcl` 的 Tcl 脚本以设置 AI 引擎调试环境：

```
#Set up the required environment
# The aie_mem_socket and xrt_server ports must match what was specified
in earlier commands.
set aie_work_dir "<AIE_Project>/Work"
set aie_mem_sock_port "10100"
set xrt_server_port "4352"
set app_name "aie_graph"

#Echo the environment setup
puts "Vitis install: $XILINX_VITIS"
puts "Application: $app_name, Work Directory: $aie_work_dir"
puts "XRT Server Port: $xrt_server_port, AIE Port: $aie_mem_sock_port"

#Set up AIE Debug environment
set source_tcl_cmd "source $XILINX_VITIS/scripts/vitis/util/
aie_debug_init.tcl"
puts "$source_tcl_cmd"
eval $source_tcl_cmd

##run the command to connect and display debug targets
set aie_debug_cmd "init_aie_debug -work-dir $aie_work_dir -url
tcp::$xrt_server_port \
-memsock-url localhost:$aie_mem_sock_port -sim-type memserver -name
$app_name -full-program"
puts "$aie_debug_cmd"
eval $aie_debug_cmd
```

注释：此脚本需设置“$XILINX_VITIS”环境变量。

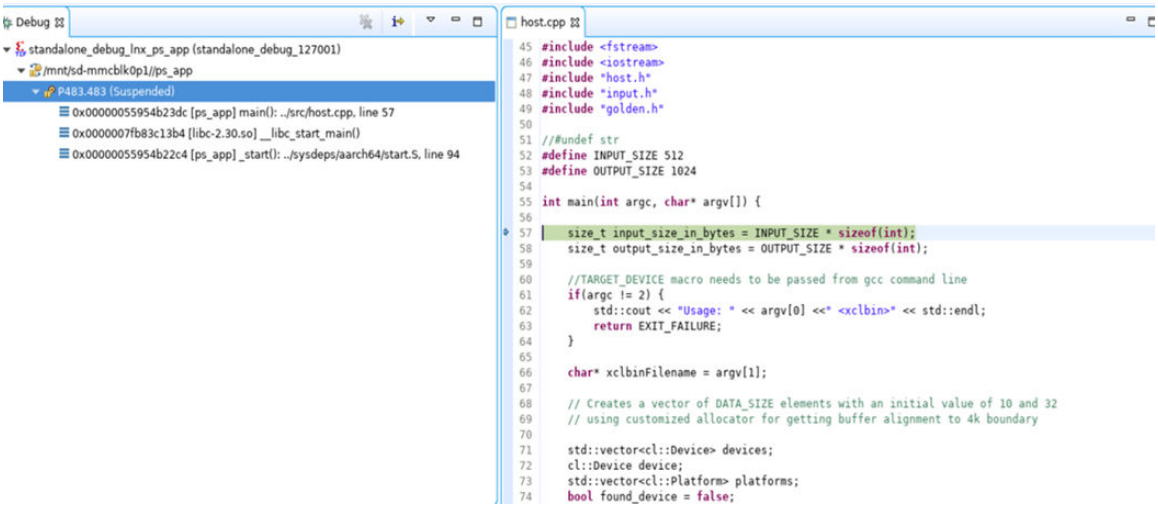
4. 当 QEMU 环境和“xrt_server”均正常启动并运行后，您可在第三个终端窗口中以独立调试模式启动 Vitis IDE：

```
vitis -debug -flow embedded_accel -target hw_emu -exe ./ps_app \
-program-args ${xcl_bin_dir}/binary_container_1.xclbin -port 1440
```

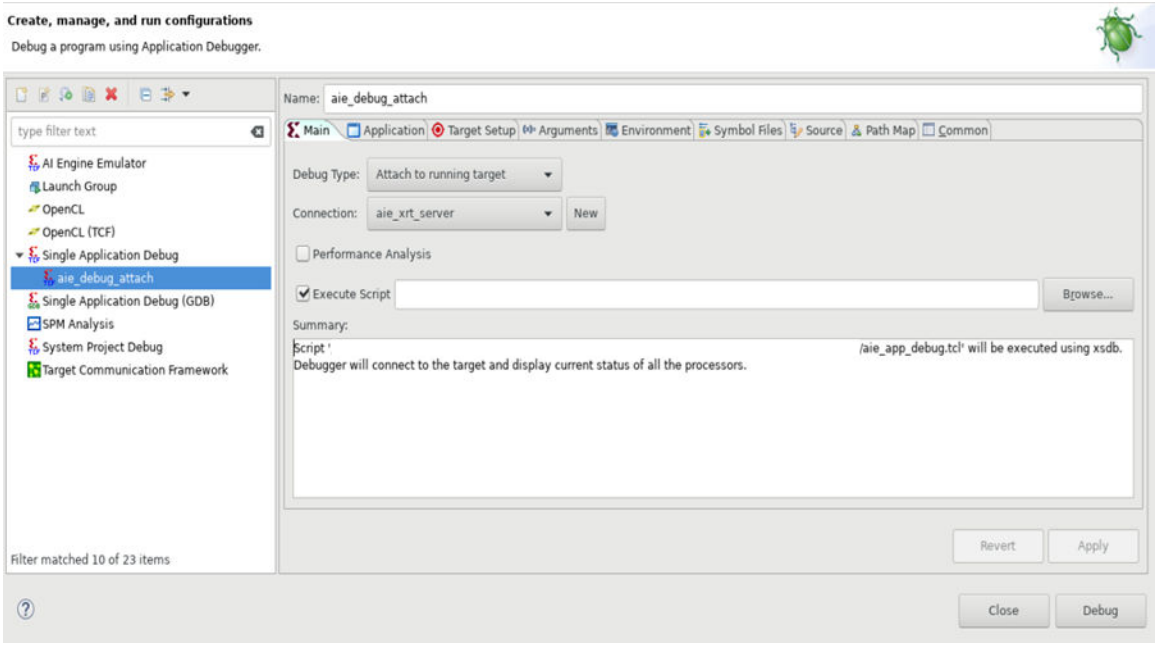
其中：

- `vitis -debug`：以独立调试模式启动 Vitis IDE。
- `-flow embedded_accel`：指定嵌入式处理器应用加速流程。
- `-target hw_emu`：指示要调试的目标构建。
- `-exe ./ps_app`：指示要运行和调试的 PS 应用。
- `-program-args ${xcl_bin_dir}/binary_container_1.xclbin`：表示要作为实参加载到可执行文件中的 XCLBIN 文件的位置。
- `-port 1440`：指定 `${linux_tcf_agent_port}`，如前所述。

这样即可打开 Vitis IDE 并显示“Debug”（调试）透视图，同时加载 PS 应用的调试配置。

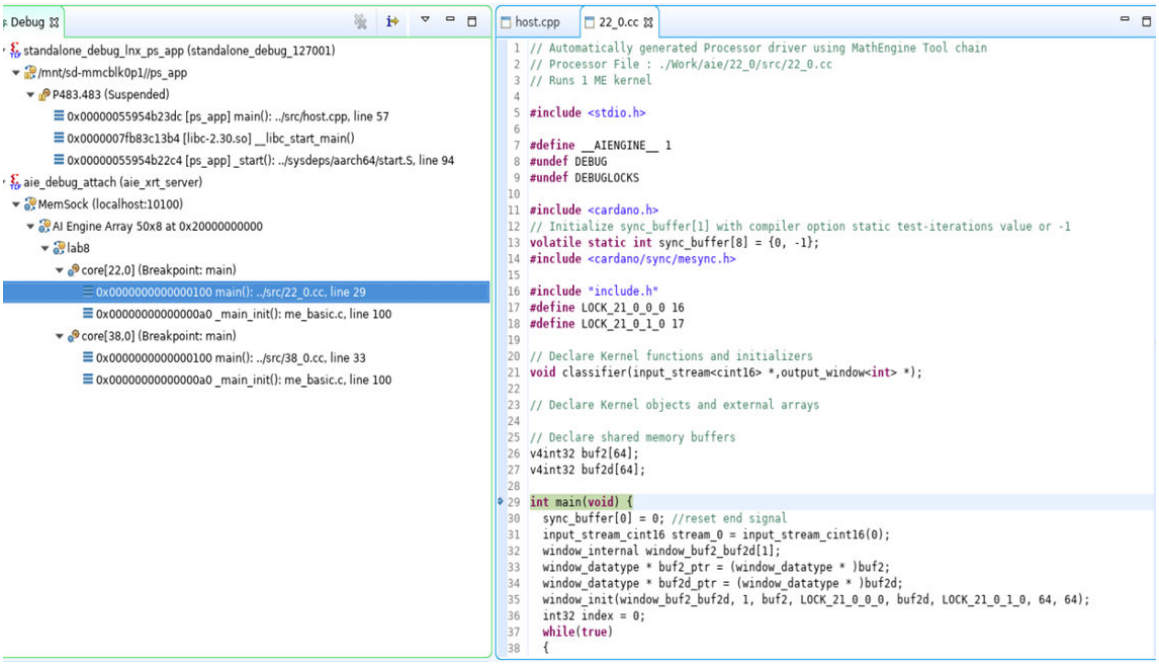


5. 在 Vitis IDE 的“Debug”透视图中，创建新的目标连接，类型为“Hardware Server”（硬件服务器），名为 `aie_xrt_server`。指定 `localhost` 作为主机，并指定 `xrt_server_port` 作为端口（前述示例中为 4352）。
6. 创建新的“Debug”（调试）配置，类型为“Single Application Debug”（单应用调试），如下所示。



   - “Debug Type”（调试类型）：`Attach to running target`
   - “Connection”（连接）：`aie_xrt_server`
   - “Execute Script”（执行脚本）：指定到步骤 4 中定义的 `aie_app_debug.tcl` 的路径。
7. 按“Debug”以继续。

   这样即可连接到 QEMU 中的 PS 应用及其相应的核上运行的 AI 引擎 graph。此应用会在所有 ELF 文件的 `main()` 函数处自动中断。



此时，您即可在仿真环境中执行所有调试活动，如单步进入、单步跳过、查看变量或植入断点。如需了解更多信息，请参阅 使用调试环境。

仅调试 AI 引擎 graph

要仅调试 AI 引擎域而不调试 PS，请遵循 从命令行进行硬件仿真调试 中的步骤 1 到 4 进行操作。随后，无需执行步骤 5，改为继续执行下列步骤：

1. 当 QEMU 环境和“xrt_server”均正常启动并运行后，您可在第三个终端窗口中以独立调试模式启动 Vitis IDE：

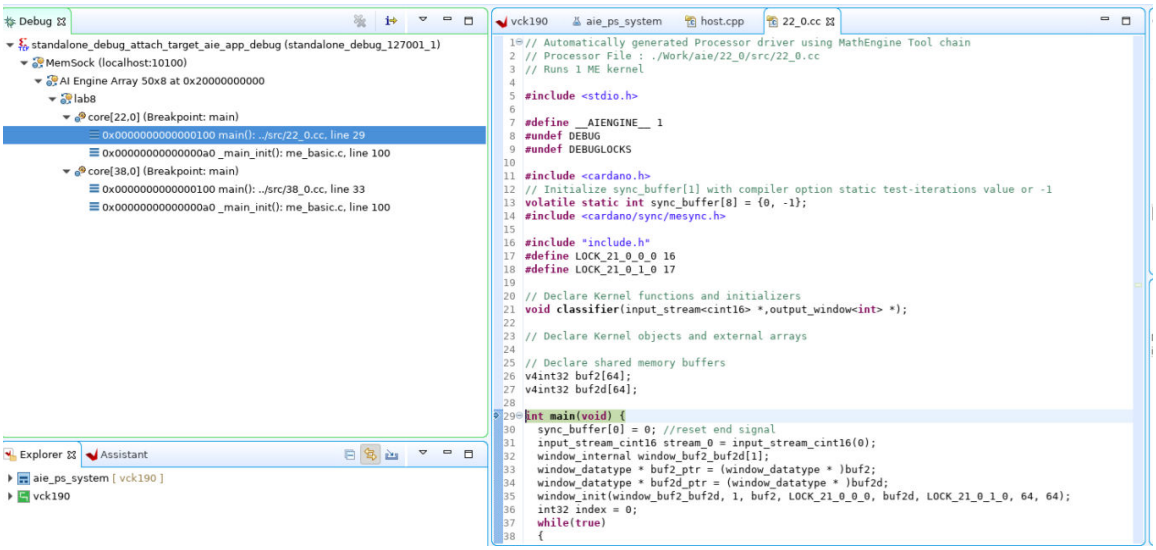
   ```
   vitis -debug -flow embedded -os baremetal -port 4352 \
   -launch-script <aie_project>/aie_app_debug.tcl
   ```

   其中：

   - `vitis -debug`：以独立调试模式启动 Vitis IDE。
   - `-flow embedded`：指定 AI 引擎处理器的嵌入式处理器流程。
   - `-os baremetal`：适用于 AI 引擎域的裸机操作系统。
   - `-port 4352`：按步骤 2 中所述指定“xrt_server”端口。
   - `-launch_script <aie_project>/aie_app_debug.tcl`：指定来自步骤 3 的 Tcl 脚本，该脚本用于设置 AI 引擎调试环境。

   这样即可打开 Vitis IDE 并显示“Debug”（调试）透视图，同时加载 AI 引擎应用的调试配置。

图 79：**AI 引擎调试配置**



从命令行进行硬件调试

当“Hardware”（硬件）构建完成（包括 AI 引擎 graph、PL 区域内核以及 PS 应用）后，您即可使用以下步骤来调试系统设计。此进程利用 Vitis IDE 从命令行启动调试环境。

- 在独立调试模式下使用 `vitis -debug` 选项启动 Vitis IDE。
- 使用硬件服务器 (`hw_server`) 命令连接到开发板。
- 为 AI 引擎域配置调试环境。

1. 将 `<project>/Hardware/package/sd_card.img` 烧写到物理 SD 卡上。这样即可为您的目标平台创建一个可启动的介质。
2. 将此 SD 卡插入 VCK190 评估套件的读卡器。
3. 将卡的启动模式设置更改为 SD 启动模式，然后给开发板上电。
4. 运行 `ifconfig` 以获取目标卡的 IP 地址。IP 地址用于在 Vitis IDE 中设置硬件服务器连接。此目标需连接到网络分配的 IP 地址。
5. 在终端窗口中，启动硬件服务器以连接到开发板。
6. 创建名为 `aie_app_debug.tcl` 的 Tcl 脚本以设置 AI 引擎调试环境：

```
#Set up the required environment
# The aie_mem_socket and xrt_server ports must match what was specified
in earlier commands.
set aie_work_dir "<AIE_Project>/Work"
set hw_server_host "gandalf"
set app_name "aie_graph"

#Printing the information
puts "Install: $$XILINX_VITIS"
puts "Application: $app_name, Work Directory: $aie_work_dir"
puts "Hardware Server: $hw_server_host"


set source_tcl_cmd "source ${vitis_install}/scripts/vitis/util/
```

```
aie_debug_init.tcl"
puts "$source_tcl_cmd"
eval $source_tcl_cmd

##run the command to connect and display debug targets
set aie_debug_cmd "init_aie_debug -work-dir $aie_work_dir -url
tcp:$hw_server_host:3121 -jtag -name $app_name"
puts "$aie_debug_cmd"
eval $aie_debug_cmd
```

注释：此脚本需设置“$XILINX_VITIS”环境变量。

7. 当开发板系统和“hw_server”正常启动并运行后，在第二个终端窗口内以独立模式启动 Vitis IDE：

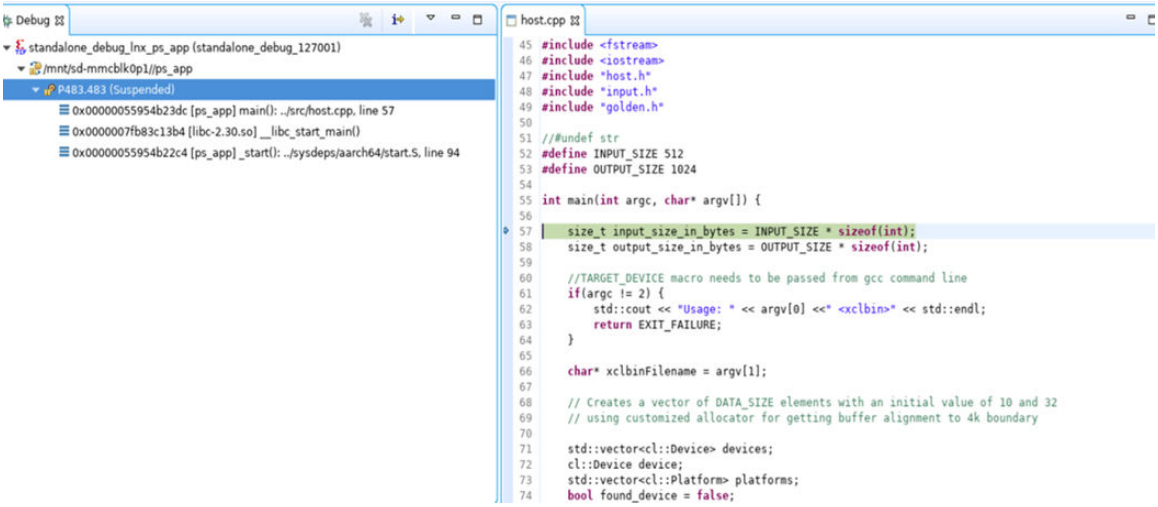
```
vitis -debug -flow embedded_accel -target hw -exe ./ps_app \
-program-args ${xcl_bin_dir}/binary_container_1.xclbin -host $
{linux_tcf_agent_host} \
-port 1534
```

其中：

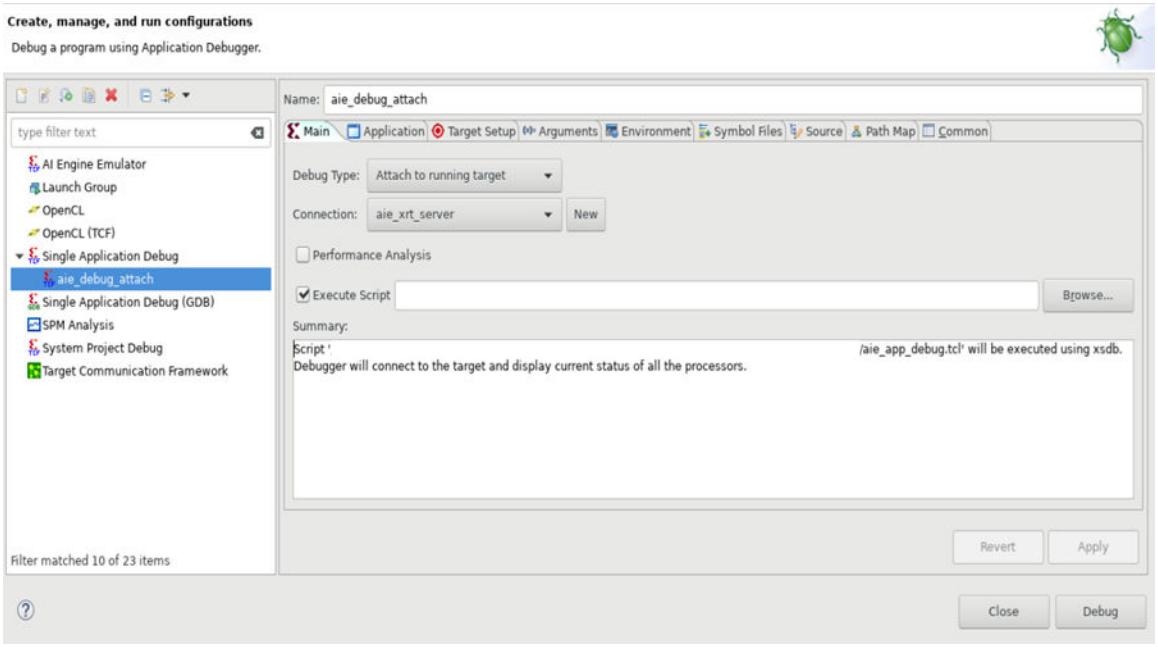
- `vitis -debug`：以独立调试模式启动 Vitis IDE。
- `-flow embedded_accel`：指定嵌入式处理器应用加速流程。
- `-target hw`：定义要调试的硬件构建。
- `-exe ./ps_app`：指示要运行和调试的 PS 应用。
- `-program-args ${xcl_bin_dir}/binary_container_1.xclbin`：表示要作为实参加载到可执行文件中的 XCLBIN 文件的位置。
- `-host ${linux_tcf_agent_host}`：指定从开发板上运行的 Linux 获取的主机名或 IP 地址：`${linux_tcf_agent_host}`。
- `-port 1534`：指定运行 Linux TCF 代理的端口：`${linux_tcf_agent_port}`。在此例中，端口为 1534。

这样即可打开 Vitis IDE 并显示“Debug”（调试）透视图，同时加载 PS 应用的调试配置。

图 80：**PS 调试配置**



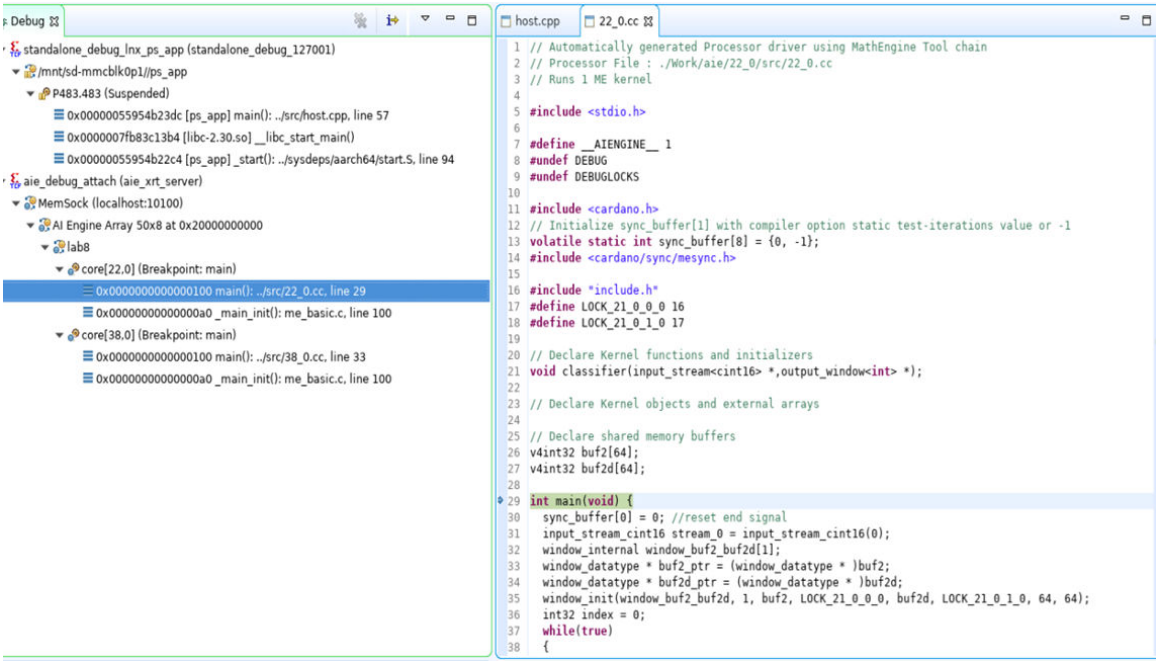
8. 创建新的调试配置，类型为“Single Application Debug”（单应用调试），如下所示。



- “Debug Type”（调试类型）：“Attach to running target”（连接到运行中的目标）
- “Connection”（连接）：步骤 5 中定义的 `hw_server`。
- “Execute Script”（执行脚本）：指定到步骤 6 中定义的 `aie_app_debug.tcl` 的路径。

9. 单击“Debug”（调试）以继续。

   这样即可连接到 QEMU 中的 PS 应用及其相应的核上运行的 AI 引擎 graph。此应用会在所有 ELF 文件的 `main()` 函数处自动中断。



此时，您即可在调试环境中执行如下所有调试活动：单步进入、单步跳过、查看变量或植入断点。如需了解更多信息，请参阅 使用调试环境。

仅调试 AI 引擎硬件

要仅调试 AI 引擎域而不调试 PS，请遵循 从命令行进行硬件调试 中的步骤 1 到 6 进行操作。随后，无需执行步骤 7，改为继续执行下列步骤：

1. 当开发板和“hw_server”正常启动并运行后，在单独的终端窗口中以独立调试模式启动 Vitis IDE：

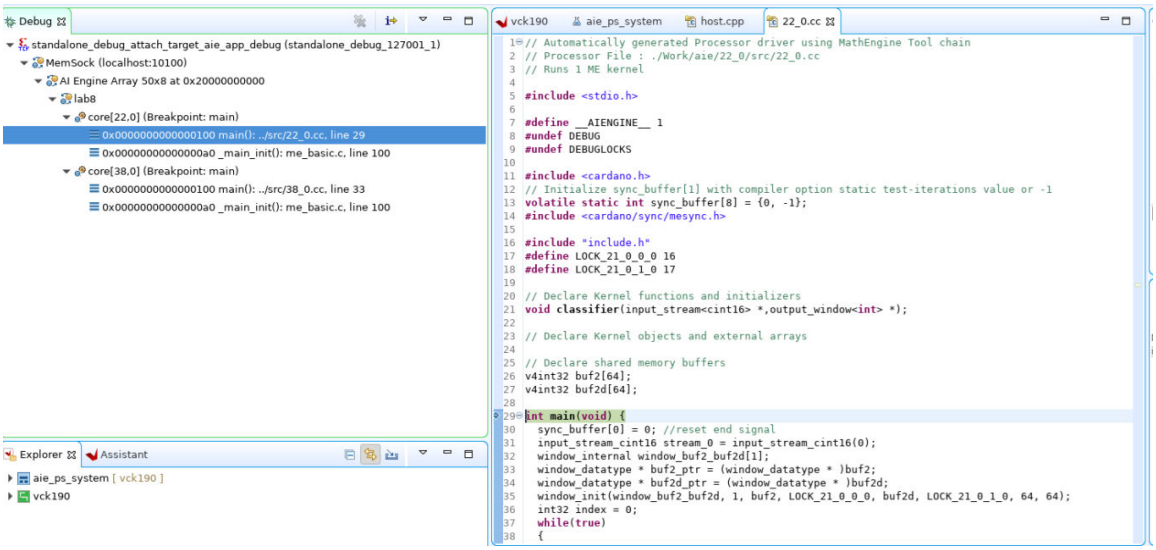
```
vitis -debug -flow embedded -os baremetal -host gandalf \
-launch-script <aie_project>/aie_app_debug.tcl
```

其中：

- `vitis -debug`：以独立调试模式启动 Vitis IDE。
- `-flow embedded`：指定 AI 引擎处理器的嵌入式处理器流程。
- `-os baremetal`：适用于 AI 引擎域的裸机操作系统。
- `-host gandalf`：指定“hw_server”的主机名（请参阅 从命令行进行硬件调试 的步骤 5）。
- `-launch_script <aie_project>/aie_app_debug.tcl`：指定来自 从命令行进行硬件调试 的步骤 6 的 Tcl 脚本，该脚本用于设置 AI 引擎调试环境。

这样即可打开 Vitis IDE 并显示“Debug”（调试）透视图，同时加载 AI 引擎应用的调试配置。

图 81：**AI 引擎调试配置**



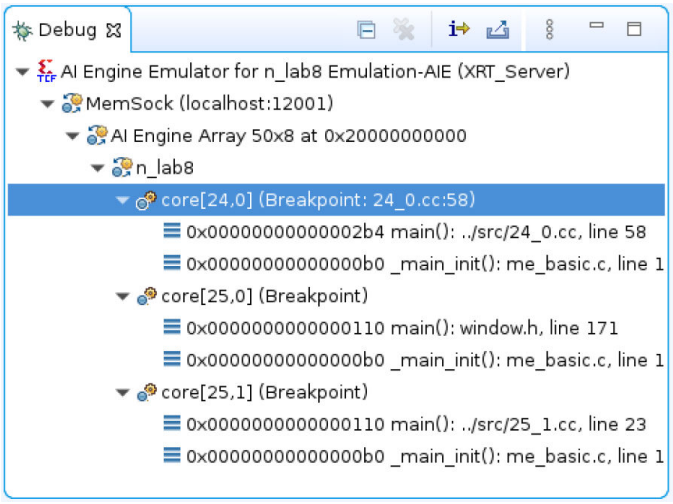
使用调试环境

Vitis IDE 调试环境具有诸多基于 GUI 的传统调试环境内常见的功能特性，例如 GDB。您可在代码中添加断点、单步跳过或单步进入代码、循环或函数的特定行、检验变量状态并将其强制设为特定值。这些只是 Vitis IDE 调试环境中的一小部分功能特性。

启动“Debug”（调试）透视图后，您将看到其中显示的多个窗口或视图，例如，显示在右上角的“Debug”视图，如下图所示。调试进程期间，有多个窗口会显示调试状态，包括位于断点处的代码、单步跳过状态、断点视图、变量视图、寄存器视图、反汇编视图和流水线视图（仅限单内核）。

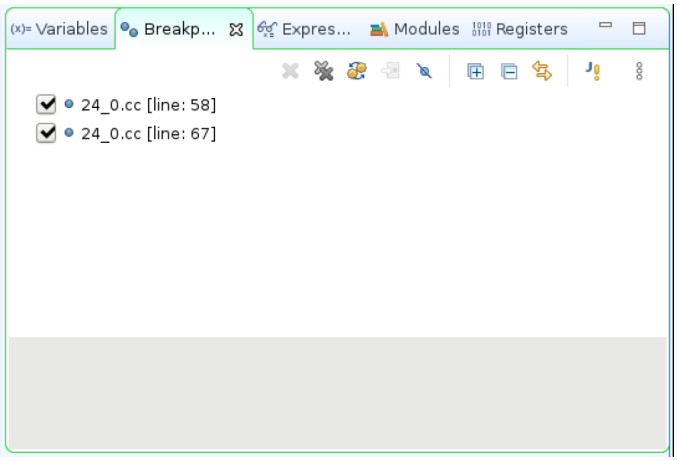
“Debug”视图会显示所调试的核的状态。它会显示调试器停止的位置（包括哪个文件和文件的哪一行源代码），及其采取的行动（断点、单步跳过等），如下图所示。

图 82：**调试视图**



下图显示了“Breakpoints”（断点）信息，包含当前设置断点。打勾的方块表示此断点已启用。单击复选标记即可将其清除并在调试期间禁用断点。这样您即可管理断点，而无需移除断点或者将其重新添加回代码中。

图 83：**断点视图**



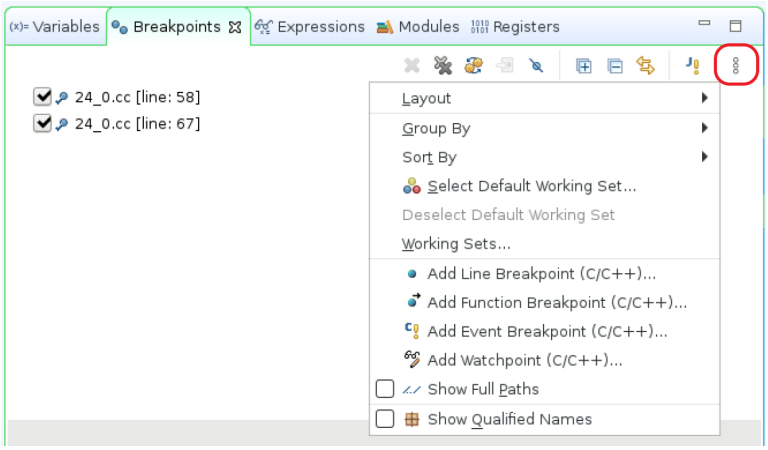
重要提示！对 AI 引擎仿真器或者协同仿真进行调试时，每个 AI 引擎拼块都支持 4 个断点。TCF 框架默认会在 AI 引擎内核 `main()` 处停止。连接到 `while` 语句的断点会耗用两个断点资源。变通方法是将断点附加到 `while` 循环内部。这样仅耗用一个断点。

观察点

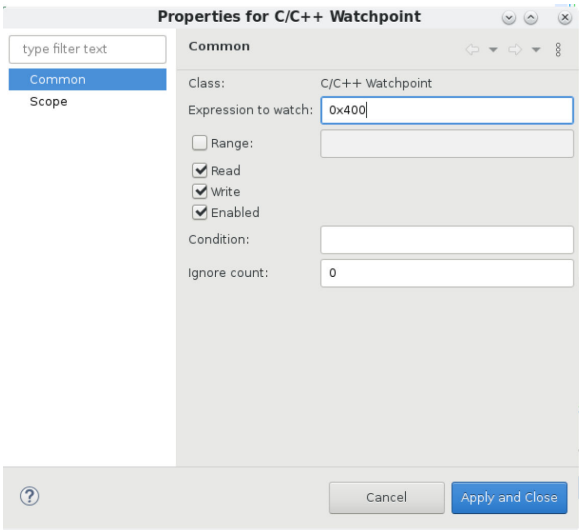
观察点属于一种断点类型。当某处地址上的值发生更改时，观察点可用于停止核的执行。这在判断值发生更改的位置时很有用。例如，可能意外覆盖某个变量值，导致程序流程中断。观察点有助于检测此类情况。

AI 引擎架构支持读/写观察点。这表示根据您的配置，在读取访问和/或写入访问时触发观察点。每个 AI 引擎 tile（拼块）都支持每个存储体各含 2 个观察点。由于每个 AI 引擎核（不包括边界处的拼块）都有权访问相邻拼块内的存储体，因此每个核最多可使用 8 个观察点。但由于共享存储体，因此每个核的可用观察点数量取决于存储体内已用的观察点数量。例如，如果某个核已使用了来自其 4 个相邻存储体的全部 8 个观察点，那么共享这 4 个存储体的其它核则无法使用来自共享的存储体的观察点。调试器会保留从每个存储体分配的观察点的记录，只要当前拼块内无可用观察点，就会抛出错误。

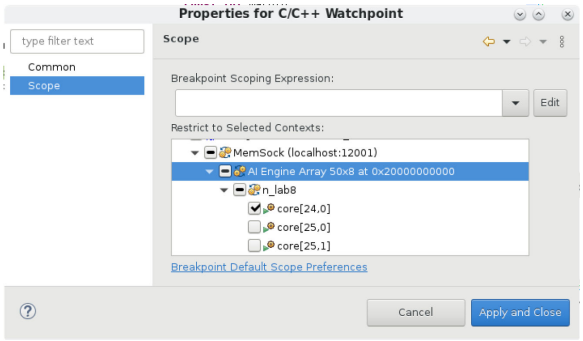
1. 要在 Vitis IDE 中添加观察点，请单击“Breakpoints”（断点）视图右上角的三个点，然后选择“Add Watchpoint (C/C++)”（添加观察点 (C/C++)）。



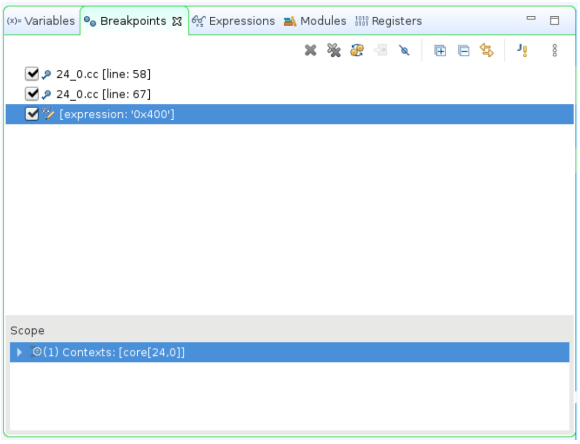
2. 在“Watchpoint Properties”（观察点属性）对话框中，输入感兴趣的存储器地址或变量名。选择读/写模式。如果取消选中“Enabled”（启用），则无论调试会话期间采用读取模式还是写入模式，都不会启用观察点。这样无需移除观察点配置即可将其保留，并在必要时重新添加即可。



3. 默认情况下，观察点添加到所有调试核中。要将观察点应用于特定的核，请单击“Watchpoint Properties”观察点属性窗口左侧窗格的“Scope”（作用域），然后仅选中观察点适用的特定核即可。



4. 在“Breakpoints”（断点）视图中验证观察点。



触发观察点

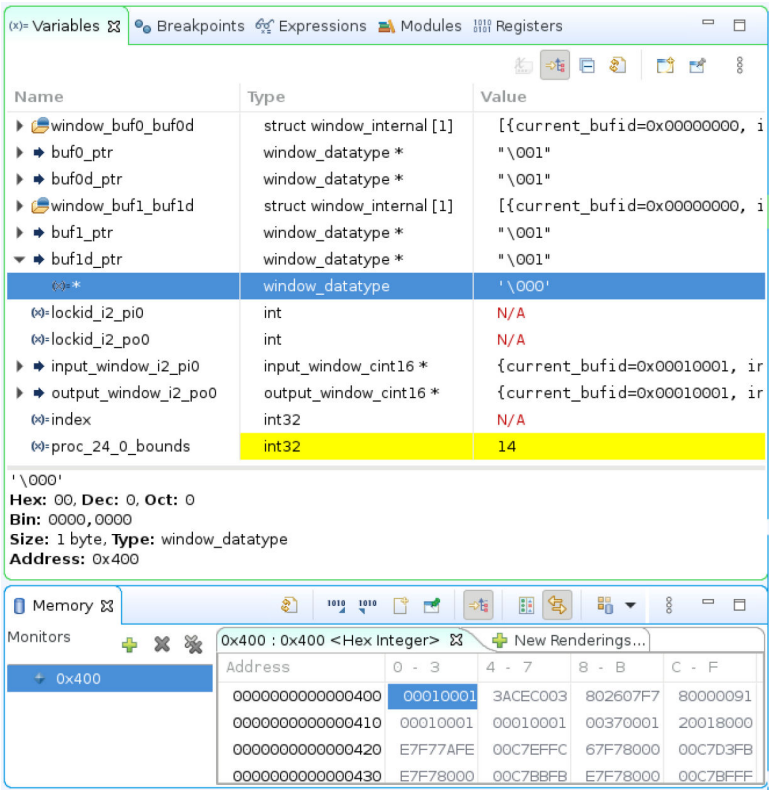
根据观察点的配置方式，在读取访问和/或写入访问时会触发观察点。已触发的观察点会导致核在访问相应地址的指令处停止。调试器会检测到该核已因观察点而停止，并发出相应报告。

重要提示！

- 观察点仅适用于硬件。不支持 AI 引擎 SystemC 模型。
- 存储体为共享。如有某个核使用了来自某个 bank 的两个观察点，那么就无法在该 bank 内再为任何其它核添加观察点。
- 对于归入未使用的拼块/存储体的存储器地址，不得设置观察点。将观察点设置到未使用的拼块可能导致 AXI 错误。已用的 AI 引擎和存储器拼块可在 `Work/aie/active_cores.json` 中找到。
- 对于位于完整 16 字节对齐地址范围内的存储器访问，可触发观察点。
- 调试器使用两条广播通道来处理观察点事件。在调试期间启用观察点时，请确保这两条广播通道的使用中不存在任何冲突。

“Variables”（变量）视图显示了内核变量值。根据变量类型，单击变量可显示其类型、值和变量地址。对于阵列/结构变量，单击变量的箭头即可展开此阵列的阵列/结构内容。

图 84：**“Variables”视图**

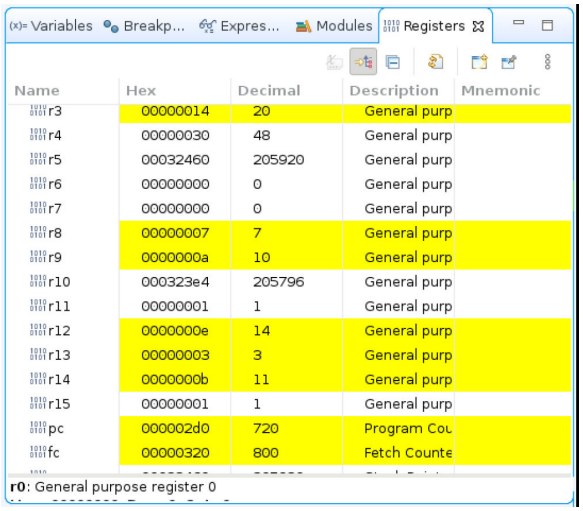


单击变量即可获取地址信息，然后单击“Memory”（存储器）视图中的“+”并输入变量的存储器地址。该变量的值会随其它存储器内容一并显示。单击变量的“value”（值）字段、输入新的值，然后按 Enter 键即可更改该变量的值。

要指定“Memory”（存储器）窗口中显示的数据的格式，请单击“New Renderings”（新的呈现方式）选项卡，指定要呈现的数据格式，然后单击“Add Rendering(s)”（添加呈现方式）。

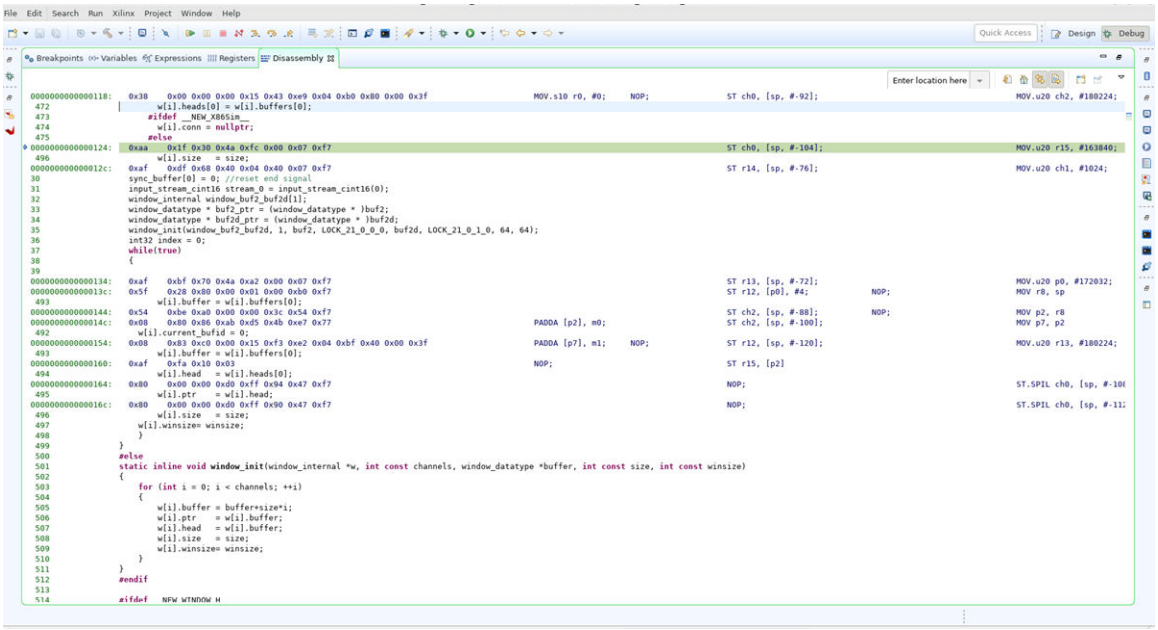
提示：要导出“Memory”窗口的内容，请从右键单击菜单中选中“Copy to Clipboard”（复制到剪贴板）命令，以将内容复制粘贴到文本编辑器并保存到文件。

图 85：**“Registers”视图**



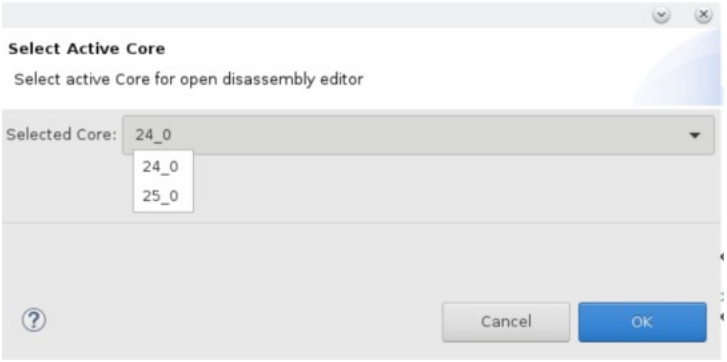
在“Registers”（寄存器）视图中，值会在调试进程中更新并在 Vitis IDE 中高亮显示。

图 86：**“Disassembly”视图**



“Disassembly”（反汇编）视图可显示机器代码和汇编代码。C/C++ 源码也嵌入在各行之间以供源代码引用。在“Explorer”（资源管理器）视图中，选中 AI 引擎工程，然后右键单击并选中“Open Disassembly View”（打开反汇编视图）。这将显示如下窗口，您可在其中选择要查看其反汇编代码的核。

图 87：**Select Core**



注释：在“Disassembly”（反汇编）视图中，默认跳过 NOP 指令。如需调试器逐步执行 NOP 指令，请单击下图中圈出的“Instruction Stepping Mode”（指令单步执行模式）图标。

图 88：**单步执行 NOP 指令**

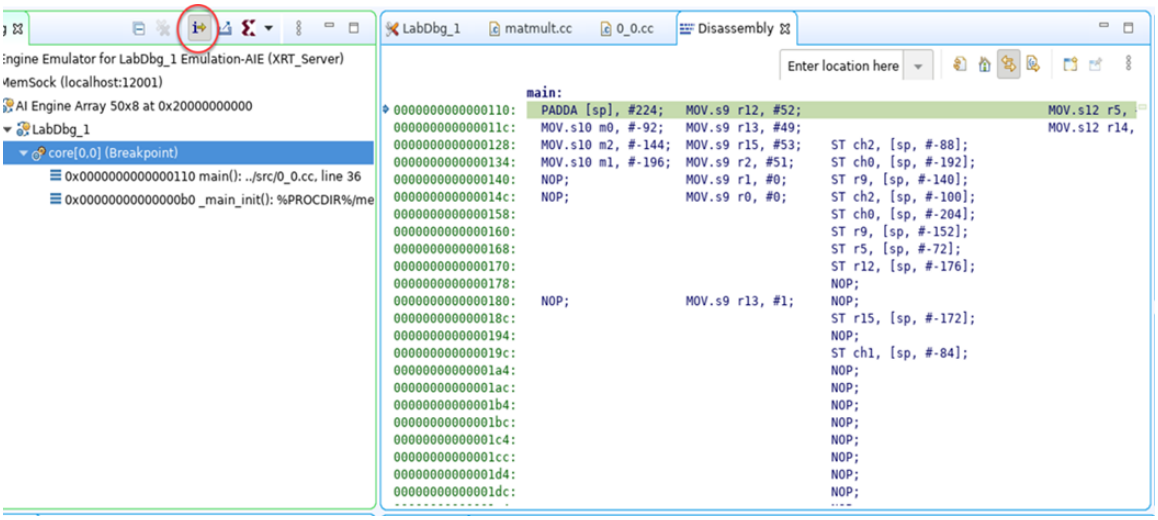
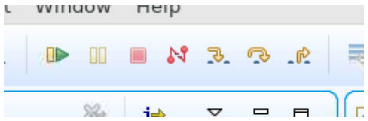


图 89：**调试控制**



调试控制命令允许您对源代码进行单步跳入/跳出/返回。其它命令允许您在 Vitis IDE 中恢复、停止、终止或断开连接调试器。

Vitis IDE 支持多核调试和多域（PS 和 AI 引擎）调试。根据应用，可能有数百个内核同时进行调试。对每个核以及每个域中的所有核都加以精细控制至关重要。从 Vitis IDE 中选择一个核，然后单击“Resume”（恢复）图标即可仅恢复执行该核的调试。选择 AI 引擎域中所有核的上一层，然后单击“Resume”图标即可恢复所有 AI 引擎核的执行。恢复 graph 中所有内核的执行则取决于诸多条件，例如，每个内核是否有数据可用于避免停滞，或者各内核断点的设置。您还必须确保未进行调试的内核均可自由运行。

图 90：**选择恢复多个 AI 引擎调试**

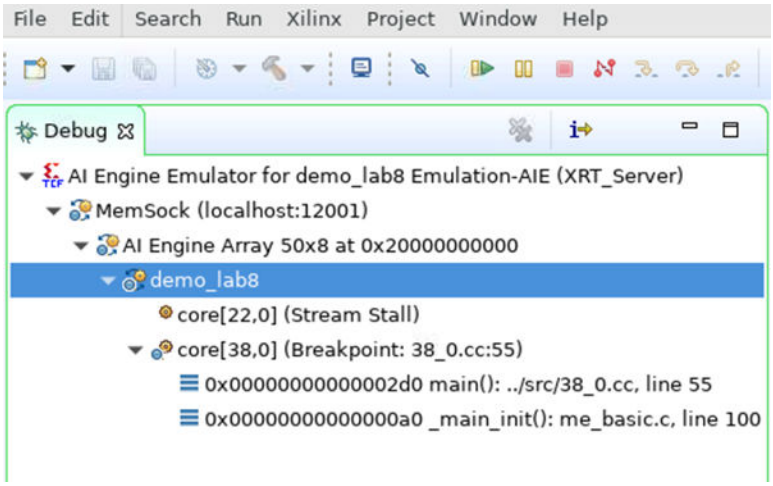
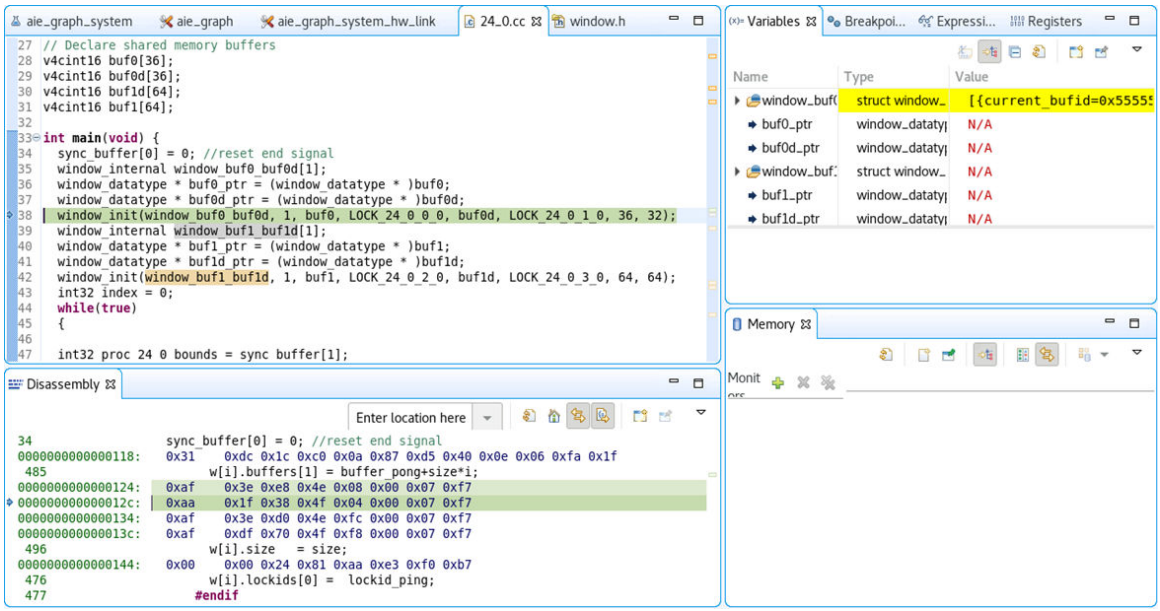


图 91：**“Debug”透视图**

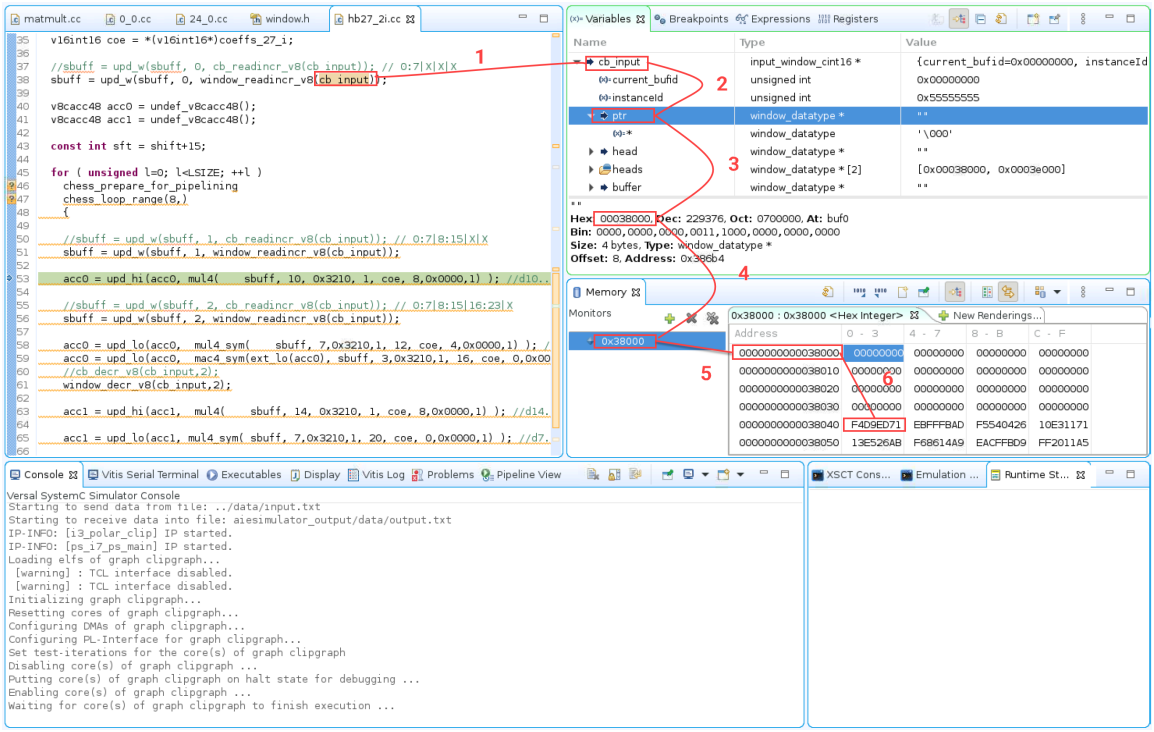


注释：对于卷绕内核实现的每个核，AI 引擎编译器都会为其自动创建 `main()` 函数。“Debug”（调试）视图可显示完整的源代码（包括已插入的部分）。

从窗口接口查看数据

如《AI 引擎内核与 Graph 编程指南》(UG1079) 中的窗口和串流数据 API 所述，数据流通过访问窗口或串流接口进出 AI 引擎内核。在调试期间，您可能想要在数据传递经过内核时，查看这些数据访问窗口的值。下图中，您可以追踪一系列对象，并通过处理这些对象来检验变量内容。

图 92：**数据访问链**



1. 在图中步骤 1 所示代码示例中，您可看到变量 `cb_input` 属于 `input_window_cint16` 类型的指针。这显示了输入窗口的声明，此输入窗口承载有复数整数，其中实数部分位宽为 16 位，虚数部分为 16 位。
2. 转至“Variables”（变量）视图中的 `cb_input` 变量。这是数据访问窗口的指针表示法，该数据访问窗口中保存有内核的输入数据。但内核函数仅对指针进行操作，这些指针指向作为实参传递给内核函数的窗口数据结构。输入窗口数据结构是由名为 `ptr` 的指针定义的。此 `ptr` 是由输入数据窗口 API 定义的，如《AI 引擎内核与 Graph 编程指南》(UG1079) 的内核窗口运算中所述。
3. 检验变量 `ptr` 的内容。它是值为 0x00038000 的地址。
4. 在“Memory”（存储器）窗口中，单击 + 号输入地址 0x38000，如步骤 4 中所示。
5. “Memory”窗口会显示地址 0x38000 处的内容。
6. 这是由 `cb_input` 变量定义的数据访问窗口中所包含的数据。前述示例的裕度大小为 64 字节，实际数据起始位置为 0x38000 加上 0x40 = 0x38040。您可检验数据内容、以特定数据格式来显示数据，或者将数据复制到剪贴板并将其导出至独立文件，如 使用调试环境 中所述。

   请注意十六进制值，例如前图中的“Memory”窗口中显示的 F4D9ED71，此类值表示 `cint16` 类型的复数，其中包含一个 16 位有符号整数实数和一个 16 位有符号虚数，可能的值如下（值可从输入文件读取）：

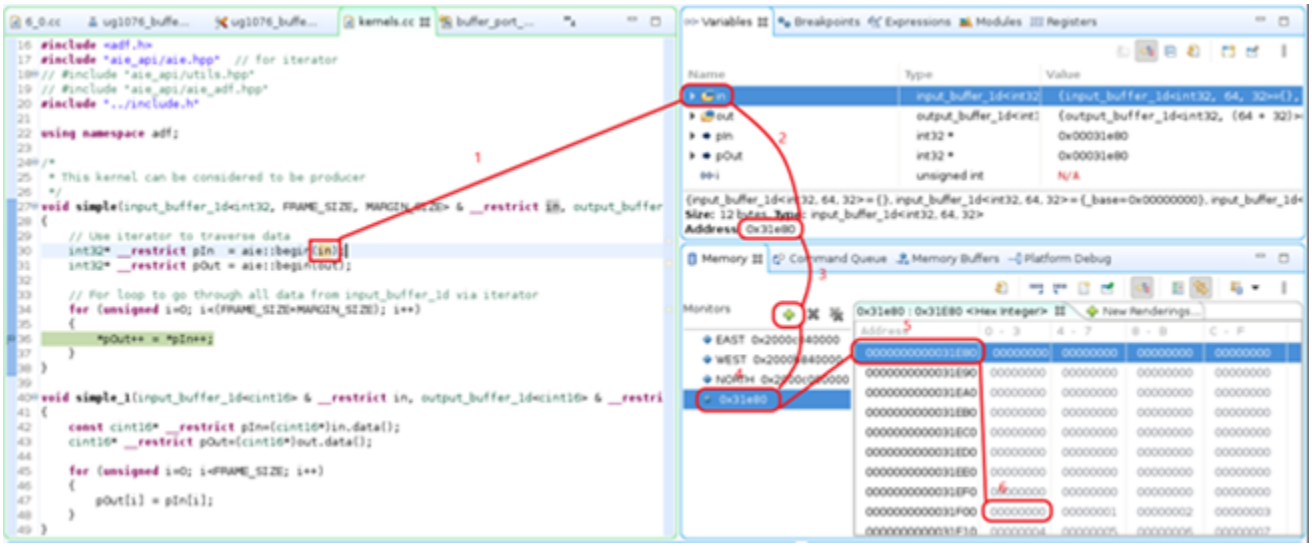
```
-4751 -2855
-1107 -5121
1062 -2732
4465 4323
9899 5093
5289 -2426
-1063 -5425
...
```

提示：如果与数据访问窗口的连接中已指定裕度，那么裕度数据包含在“Memory”窗口中。所分配的存储器总量等于窗口大小加上裕度大小。

查看来自缓冲器端口接口的数据

调试期间，您想要在数据传递经过内核时查看数据值。在下图中，您可以追踪各对象并观察存储在 Vitis IDE 中指定存储器位置的数据。

图 93：**Vitis IDE 中的追踪**



1. 在以上代码示例中（步骤 1）中，`in` 变量是指向 `input_buffer_1d`（类型为 `int32`）的指针。
2. 转至“Variables”（变量）视图中的 `in` 变量（步骤 2）。这是数据访问缓冲器端口的指针表示法，该端口中保存有内核的输入数据。但内核函数仅对指针进行操作，这些指针指向作为实参传递给内核函数的窗口数据结构。数据缓冲器端口用于保存数据。检验变量 `in` 的地址。它位于如下地址：`0x31e80`。
3. 在“Memory”（存储器）窗口中，单击“+”号（步骤 3）输入地址 0x31e80，如步骤 4 中所示。
4. “Memory”窗口会显示地址 0x31e80 处的内容（步骤 5）。这是由 `in` 变量定义的数据访问缓冲器端口中所包含的数据。
5. 此示例具有 32 个元素作为裕度大小，每个元素类型均为 int32，因此实际数据起始于 0x31e80 + 0x80 = 0x31f00。您可检验数据内容、以特定数据格式来显示数据，或者将数据复制到剪贴板并将其导出至独立文件，如使用调试环境 中所述。

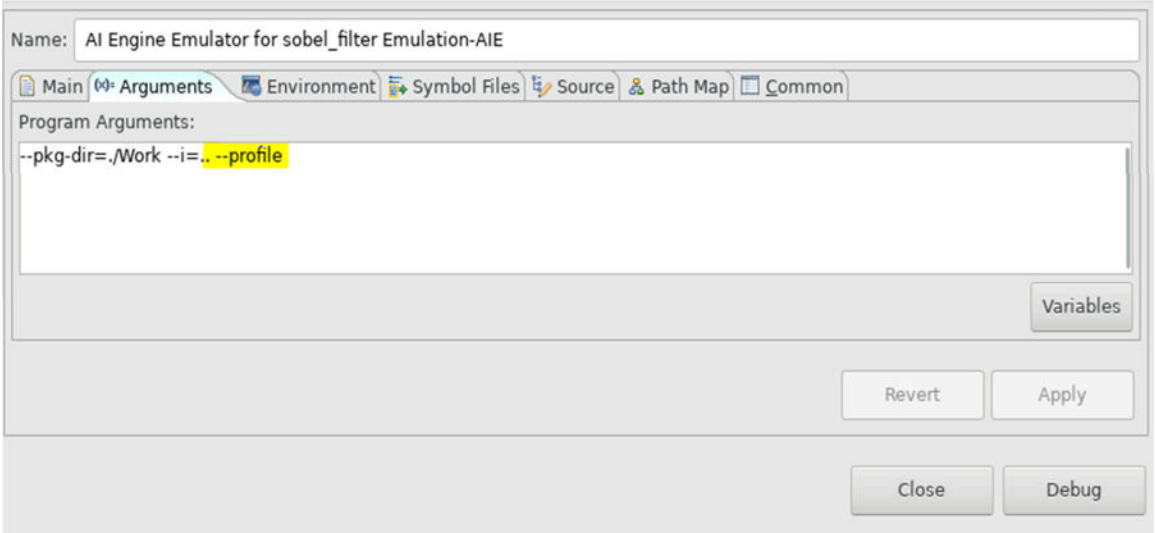
查看来自串流接口的数据

数据流通过访问窗口或串流接口进出 AI 引擎内核。在调试期间，您可能想要在数据传递经过内核时，查看这些数据访问窗口的值。对于数据窗口，Vitis IDE 调试环境提供了各种方法用于查看和访问数据，如 从窗口接口查看数据 中所述。对于串流接口连接，建议在代码中添加 `printf()` 语句，以便您检验流经内核的数据。

重要提示！在代码中添加 `printf()` 语句会制止编译器最优化，导致内核可执行程序更大，可能无法填入 AI 引擎处理器的可用存储器。

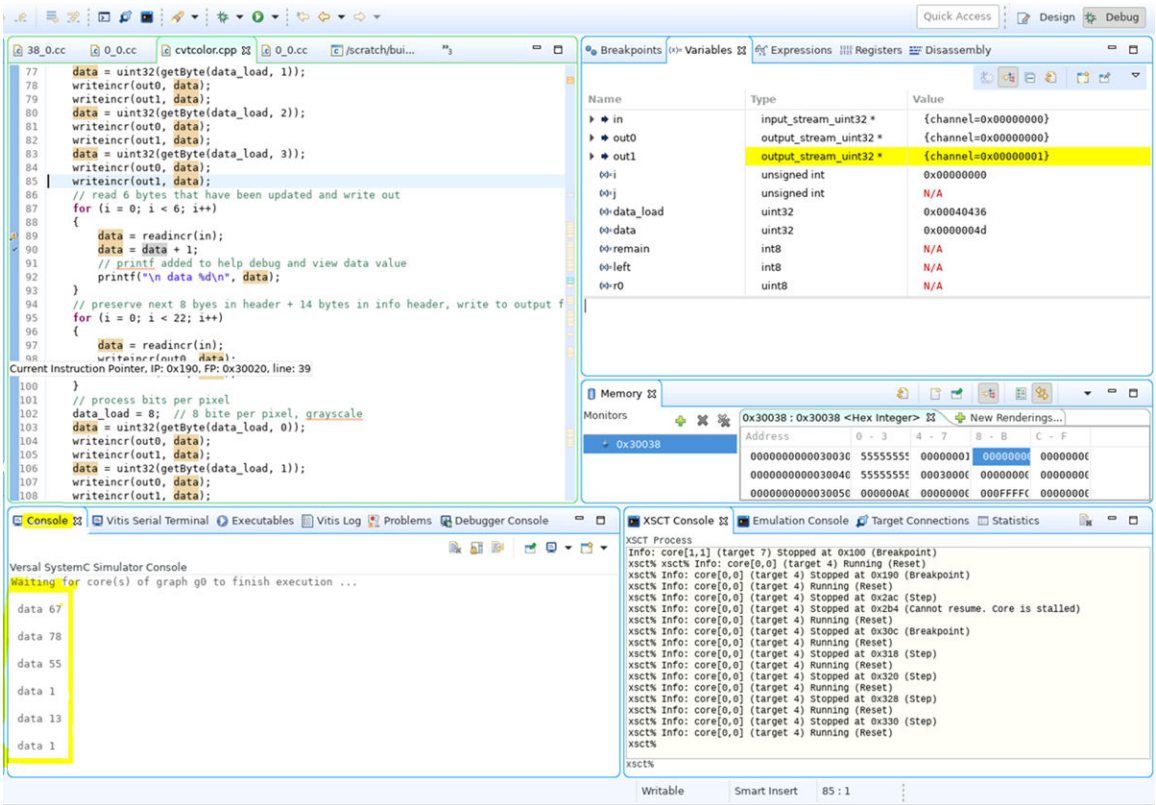
在内核代码中添加 `printf()` 语句时，还必须在 Vitis IDE 的“Run Configurations”（运行配置）或“Debug Configurations”（调试配置）中添加 `--profile` 选项。在“Debug Configuration”的“Arguments”（实参）选项卡中添加 `--profile` 以及任何其它已指定的选项，如下图所示。

图 94：**调试配置实参**



在源代码中添加 `printf()` 语句即可在内核处理串流数据的同时生成串流数据输出。下图显示了控制台窗口中此类输出的示例。由此即可通过串流接口捕获和调试数据流。

图 95：**打印来自串流接口的数据**



单内核调试的流水线视图

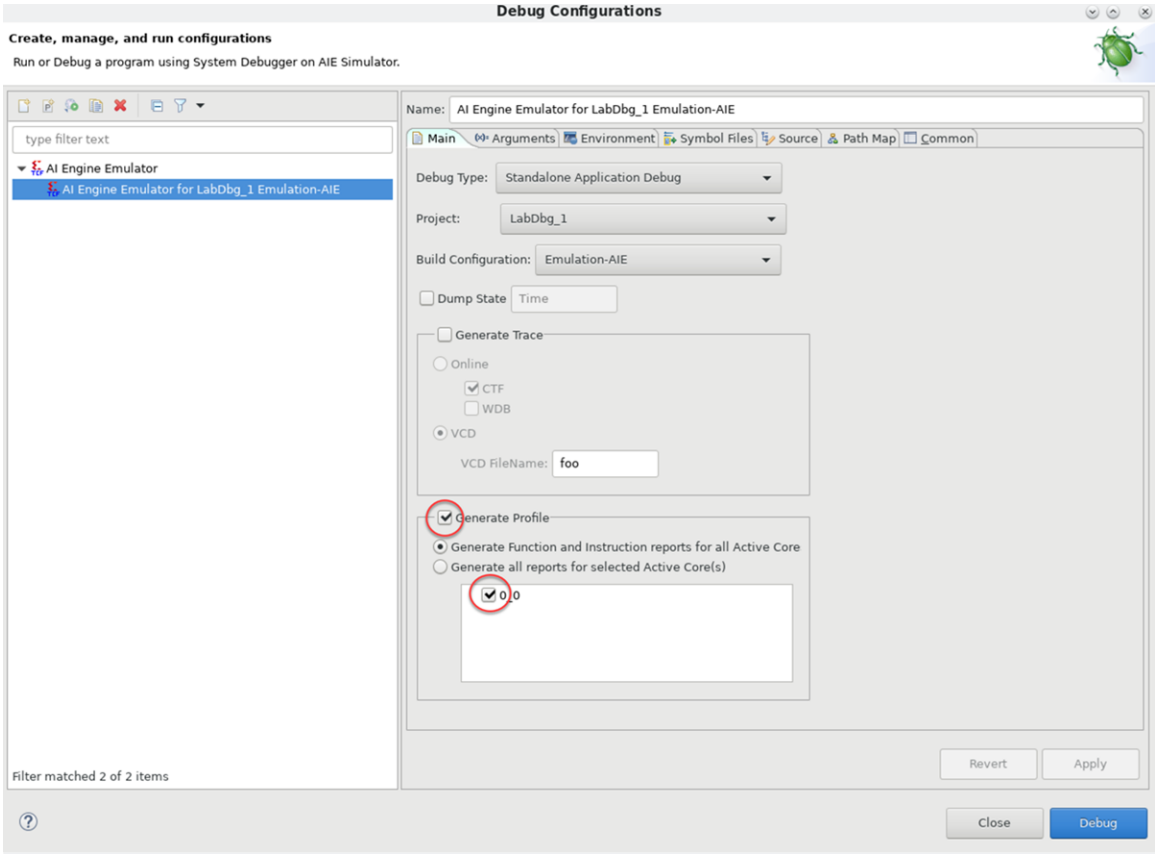
Vitis IDE 中 AI 引擎的“Pipeline”（流水线）视图允许您将特定时钟周期内执行的指令与“Disassembly”（反汇编）视图中的标签加以关联。底层 AI 引擎流水线在调试模式下使用流水线视图来公开。Vitis IDE 仅支持包含单个内核的 graph 的流水线视图。

要在含单个内核的 graph 上启用“Pipeline”视图，请在成功构建工程后，从工程调试配置中选中“Generate Profile”（生成剖析）。

重要提示！

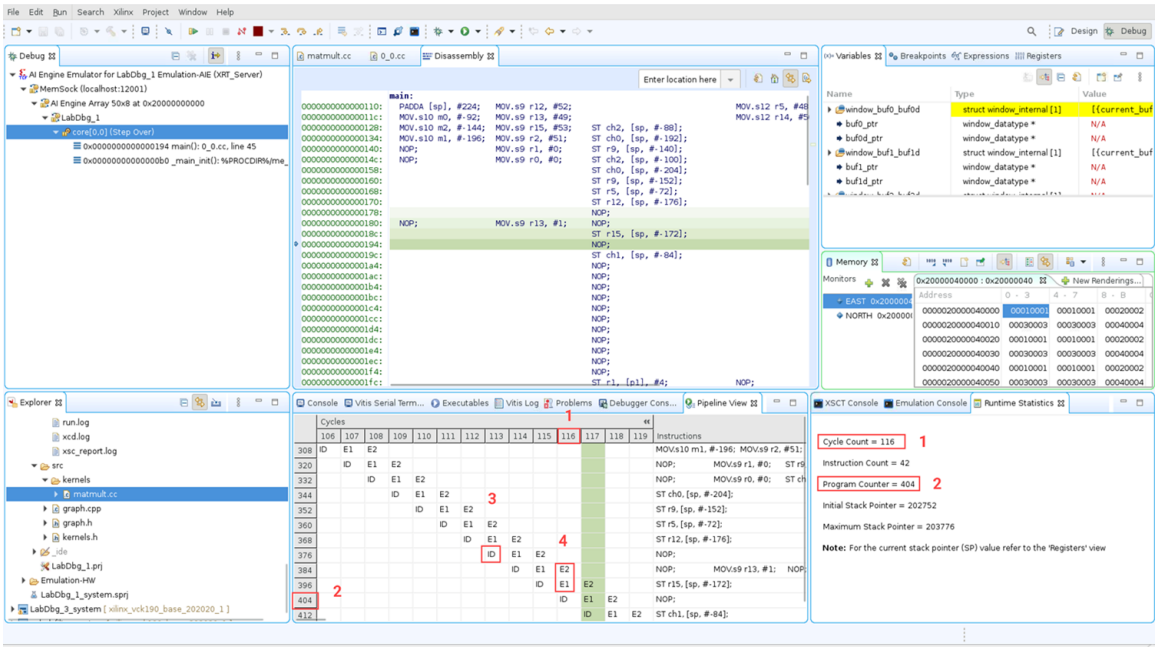
- 如果您的 graph 具有多个 tile（拼块），则此视图不予显示；但支持单个拼块内包含多个内核。
- 仅限 AI 引擎仿真器才提供流水线视图支持。

图 96： **“Debug Configuration”对话框**



单击“Debug”（调试）开始调试应用。请注意，“Pipeline”视图会自动显示在“Debugger Console”（调试器控制台）窗口中。

图 97：**流水线视图**



前图中高亮显示了流水线视图中内核的运行时统计数据。

1. AI 引擎内核周期计数
2. 程序计数器
3. ID = 指令解码
4. E1-E7 为 AI 引擎执行阶段。标量单元中几乎所有运算都调度为在流水线的 E1 阶段中运行（非线性运算除外）。矢量单元调度则从 ID 阶段贯穿至 E6 阶段。地址生成单元 (AGU) 贯穿两个流水线阶段。在流水线的 E2 阶段，地址已就绪。对于加载单元，在 E7 阶段，来自存储器模块的 AI 引擎中的数据将变为可用。对于存储单元，根据指令类型，在流水线的 E5 或 E6 阶段，数据将从 AI 引擎发出至存储器模块。

AMD XILINX

第 11 章

映射器/布线器方法论

设计收敛

本节描述了在映射器（布局）或布线器步骤中，处理 AI 引擎编译器中的失败操作的过程。

未找到映射解决方案

映射失败通常有两种模式。失败发生于预检查阶段，或者可能在实际映射阶段中发生。

预检查失败具有显式错误消息，以指出失败的准确原因，如以下示例所示。

```
ERROR: [aiecompiler 47-772] Inst g.kernel_a is in conflicting pblocks:(0,0)
(5,5) and (20,0) (25,5).
```

您可追踪此类错误，找到设计元件或约束。

映射阶段失败通常会显示如下错误消息。

```
ERROR: [aiecompiler 47-51] AIE Mapper failed to find a legal solution.
Please try to relax constraints and/or try alternate strategies like
disableFloorplanning.
```

在此情况下，请使用下列步骤缩小失败范围（前提是原因与设计相关），或者如果失败原因与工具相关，则请帮助工具找到解决方案。

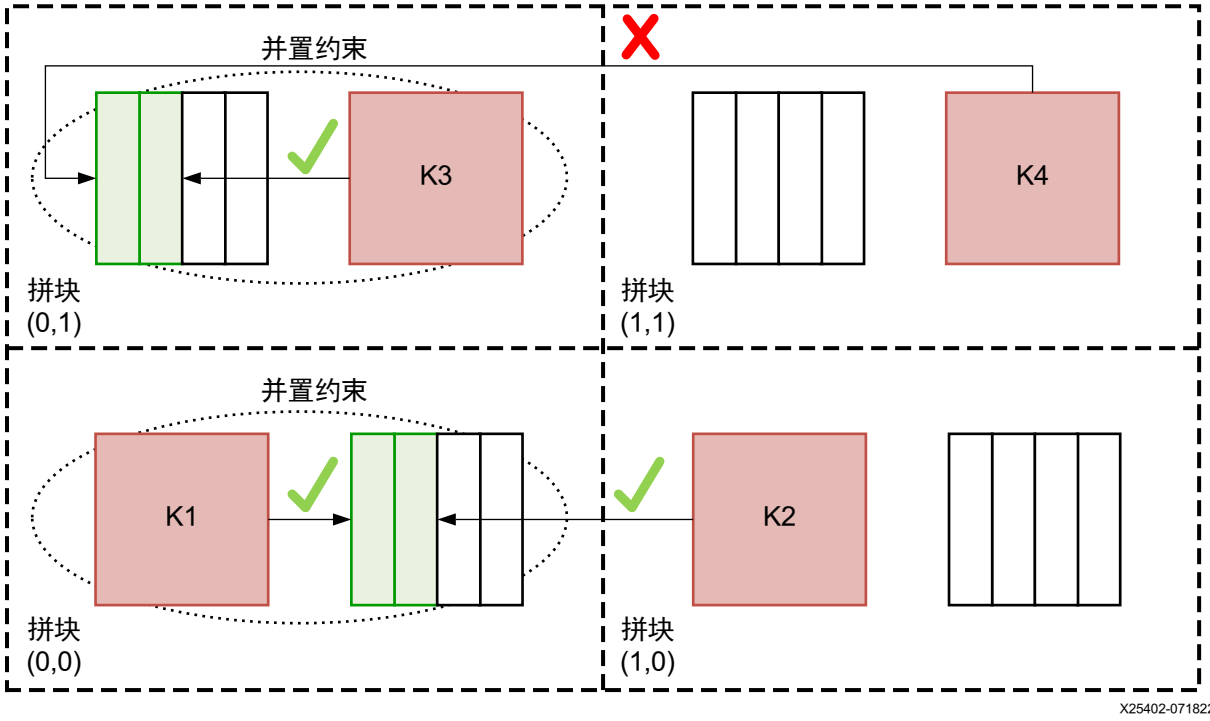
检查用户定义的约束

错误定义用户约束可能导致映射器故障。映射器预检器将捕获其中部分故障。但在预检阶段并不能捕获所有故障。在此类情况下，请检查确认是否存在下列状况。

1. 如果 graph 中有大量绝对位置约束或并置约束，请检查这些约束，确认并未向映射器提供冲突性指令。AI 引擎阵列固有的棋盘式性质可能造成此类问题，如下图所示。

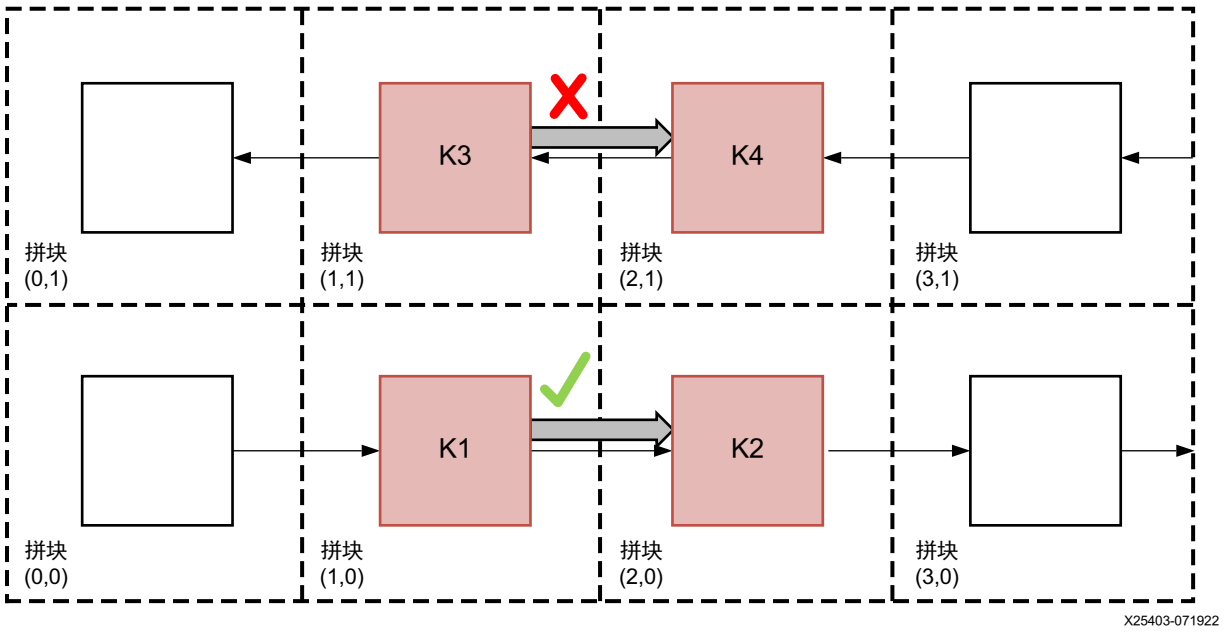
   在此图中，红色内核具有绝对位置约束，这些内核之间的窗口缓冲器（绿色）与第一个内核存在并置约束。这将导致映射器故障。

图 98：**存在冲突的绝对位置约束/并置约束**



2. 如果在级联链中包含内核的绝对位置约束，请检查确认这些约束与架构是否兼容。在 AI 引擎架构中，每一行的级联方向都会发生更改。如有如下图所示绝对约束和级联，将导致映射器故障。

图 99：**存在冲突的级联方向**



3. 在某些情况下，约束到特定拼块的缓冲器大小可能超过拼块的存储器容量（AI 引擎架构中为 32 KB）。这将导致映射器故障。

4. AI 引擎架构中的每个拼块都具有 2 条输入 DMA 通道和 2 条输出 DMA 通道。如果缓冲器的约束方式导致特定拼块所需的 DMA 通道数量超过该数量，那么就会导致映射器故障。

为超高存储器密度设计减小窗口缓冲器

判定设计的窗口大小的主要考量因素之一是数据加载所需的周期数与内核所需的计算周期数保持平衡。这样有助于将乒乓缓冲器数据加载与内核计算相结合进行流水打拍。对于超高存储器密度设计，有必要减小窗口大小同时仍能保持内核计算的平衡，其合理性在于，增大窗口大小可能导致映射器故障。

下表显示了两个含 16 位数据的矩阵的矩阵乘法所需的周期数。示例 1 和示例 2 的矩阵大小不同，但其计算和数据加载均保持平衡。请注意，判定数据加载时间时，仅以矩阵 A 和 B 中的较大者为准，而内核计算时间则是同时根据这两个矩阵的大小来判定的。此处显示示例 1 的窗口大小小于示例 2，但计算和数据加载时间相平衡并且可流水打拍。

表 93：**矩阵乘法示例**

	矩阵 A 大小	**矩阵 B 大小**	**乘法运算数量 (MultOps)**	**计算周期数** 32 次运算/周期	**数据加载周期数** 32 位/周期
示例 1	16x64	64x16	16384	512 (16384/32)	512 (64x16x16/32)
示例 2	16x64	64x32	32768	1024 (32768/32)	1024 (64x32x16/32)

向映射器提供用户指南

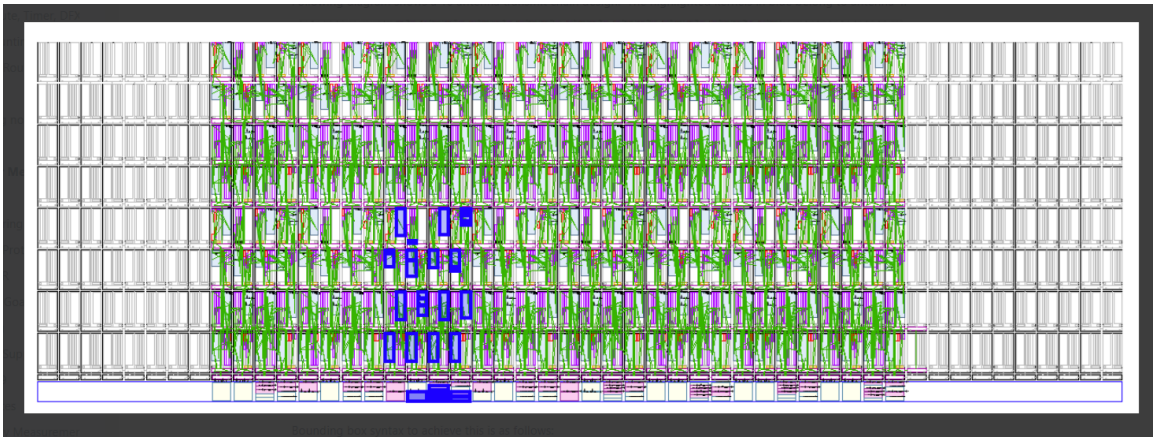
在某些情况下，可能由于工具限制而导致发生映射器错误。在此类情况下，向工具提供指南可能很有用。

1. 使用命令行开关关闭自动 AI 引擎编译器布局规划。

```
--Xmapper=disableFloorplanning
```

2. 在 graph 中使用边界框约束或者在约束文件中使用 `areaGroup` 约束来创建您自己的布局规划。如果设计中有多个非连续 graph，那么这尤其有效。其中每个分离的 graph 都能约束到阵列中的特定区域。此技巧最大程度减少了不同 graph 缓冲器之间的干扰，故而不仅有助于设计收敛，也有助于改善性能。下图显示了使用 16 天线发射链的设计示例。蓝色高亮内核属于天线 4。

图 100：**天线 4 中的内核**



用于达成此结果的约束文件语法如下所示。

```
{
  "GlobalConstraints": {
    "areaGroup": {
      "name": "ant4_cores",
      "nodeGroup": ["tx_chain4.*"],
      "tileGroup": ["(16,0):(19,3)"]
    }
  }
}
```

用于达成此结果的边界框语法如下所示。

```
location<graph>(tx_chain4) = bounding_box(16,0,19,3);
```

3. 按需添加并置约束或绝对位置约束。如果此项指南无法完成设计收敛，您可尝试添加并置约束或绝对位置约束。您可在预计将映射到同一个拼块的内核与缓冲器或系统存储器之间添加并置约束，如以下示例所示。

```
location<buffer>(kernel_1.out[0]) = location<kernel>(kernel_1);
location<stack>(kernel_1) = location<kernel>(kernel_1);
```

绝对位置约束也可以添加到某些关键内核或缓冲器中，以充当锚点并指引映射器完成其它组件的布局，如以下示例所示。

```
location<kernel>(kernel_1) = tile(20, 0);
location<buffer>(kernel_1.in[0]) =

        { address(19, 0, 0x0),
          address(19, 0, 0x2000) }; // double buffer needs two locations
```

未找到布线解决方案

在本节中，您将了解如何检查布线器拥塞，以查看映射器是否已将布线器置于不可能的处境以及如何修复问题。并且您可以查看包切换禁用是否影响拥塞。

首先检查用户定义的约束，其中包含 FIFO 深度约束和面积分组约束，然后检查映射结果。

FIFO 深度约束

器件上存在受限的开关和 DMA FIFO。判定 `fifo_depth` 约束时，重要的是考虑将为某个面积指定的 FIFO 量。这包括需要考量具有 `fifo_depth` 约束的信号线是否同时具有面积分组约束。在此情况下，请确保在指定面积内可满足所有 `fifo_depth` 约束。

如果开关 FIFO 存在高争用，请考虑转为 DMA FIFO。您使用以下约束指定 DMA FIFO 类型，无需更改 `fifo_depth`。

```
location<fifo>(net1) = { dma_fifo()}
```

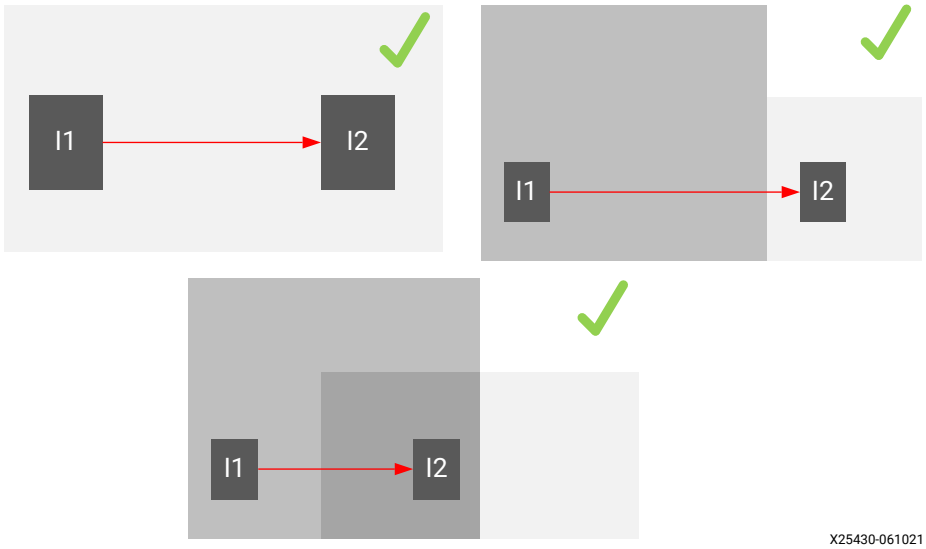
FIFO 位置约束的应用需审慎考量，如以下示例所示。

```
location<fifo>(net2) = { dma_fifo(aie_tile, 15, 0, 0x3100, 32) };
location<fifo>(net3) = { ss_fifo(shim_tile, 16, 0, 0), dma_fifo(aie_tile,
17, 0, 0x3100, 48)}
```

面积分组约束

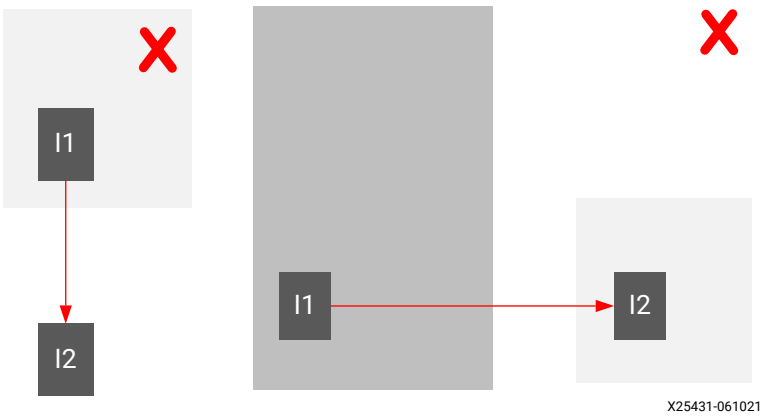
有时，仅当定义面积分组约束时，才考虑对象布局。这可能导致布线无法形成其所有连接。下图显示了允许布线形成连接的各种面积分组约束。在上述全部三种情况下，布线从不需要交由已定义的面积分组自行完成其布线。

图 101：**已定义的面积分组中的布线**



相反，布线和面积分组约束中存在两个常见错误。下图左侧显示了对象缺失面积分组错误。第二种情况下是，所有对象都包含在独立面积分组内，但两个分组不相邻。在此情况下，布线器无法在不发生面积分组约束违例的情况下完成其信号线的布线，且布线器将无法找到合规的解决方案。

图 102：**布线错误**

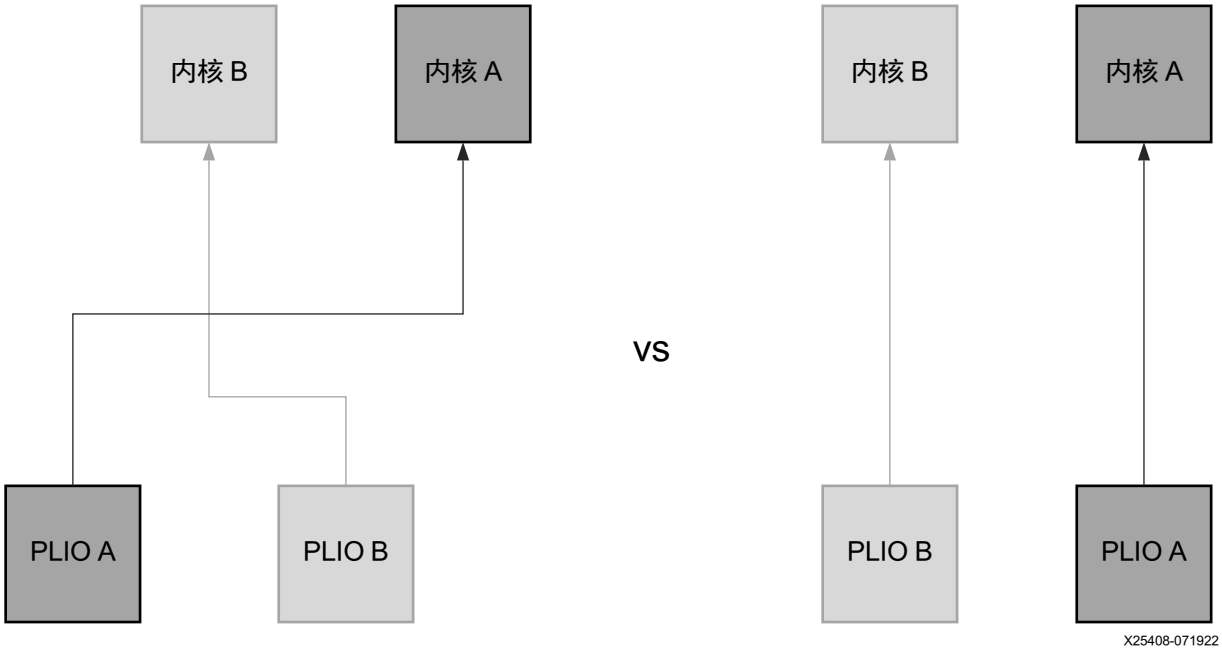


映射结果检查

检查 PLIO 布局，查看它是否导致布线拥塞。布线器上最常见的拥塞面积是接口拼块区域。现有 PLIO 通道数量多于接口通道连接到 AI 引擎阵列所使用的通道数量。

1. 如果 PLIO 当前处于约束状态，请确保 AI 引擎阵列中有足够资源可用于处理已锁定的 PLIO。
2. 如果可能，请将 PLIO 布局限制在阴影区域附近，以减少布线资源的拥塞。

3. 检查 PLIO 布局/约束，验证未发生如下图所示的交汇。



改善设计性能

存储器或串流停滞以及 graph 输出之间的偏差过大都会对设计吞吐量造成负面影响。这些情况可通过查看 Vitis™ 分析器工具中的仿真器输出来识别。以下章节探讨了上述每个类遵循的技巧。

存储器停滞

映射器的用途是尽可能防止缓冲器冲突。它还具有不同的缓冲器最优化级别，通过尝试膨胀或增大缓冲器大小来防止冲突。这些缓冲器最优化级别范围为 0（默认值）到 9。调用方式为通过 `Xmapper` 选项 `--Xmapper=BufferOptLevel<level num>` 来调用。对于最高缓冲器最优化级别 (9)，无法在相同 bank 内布局两个缓冲器。但随着缓冲器最优化级别的提升，映射器可能无法找到解决方案，并且将出错退出，明确这一点至关重要。因此，如果发现大量存储器停滞，那么第一个选项是循环运行 `BufferOptLevel` 选项，查看随着 `bufferOptLevels` 提高，存储器停滞数量是否会减少。

另一个可能性是，您可以显式告知映射器不得将两个缓冲器置于同一个 bank 内。如果仿真分析表明，由于缓冲器 `kernel_0.in[0]` 与 `kernel_1.out[0]` 之间存在 bank 冲突，引发存储器停滞并导致吞吐量严重降级，那么可以向映射器提供指令，指示不得将这些缓冲器置于同一个 bank 内，如下所示。

```
not_equal(location<buffer>(kernel_0.in[0]),
location<buffer>(kernel_1.out[0]));
```

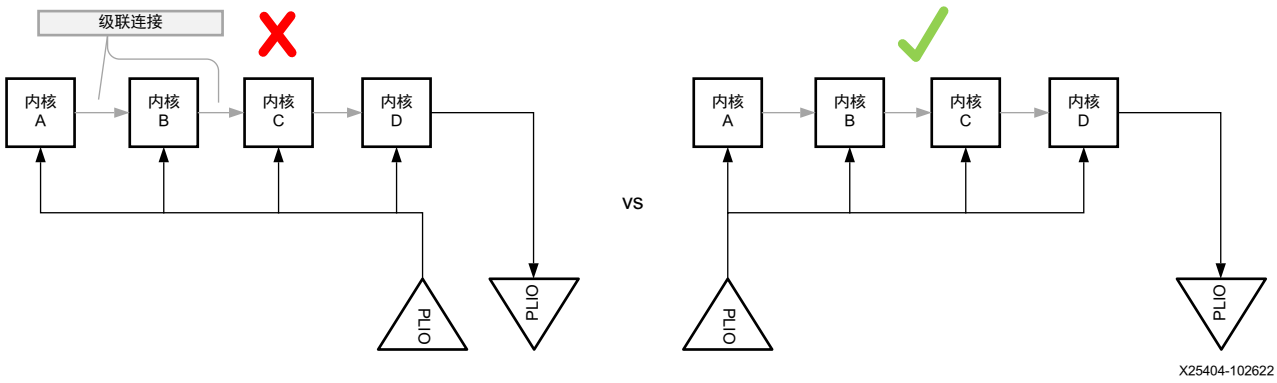
如果在设计中使用 DMA FIFO，并且将其布局在与其它缓冲器相同的 bank 内，然后 `Xrouter` 选项 `DMAFIFOsInFreeBankOnly` 可以强制布线器将这些 FIFO 布局在空闲 bank 内。这样即可消除与 DMA FIFO 之间的存储器冲突。如果无法保留完整的空闲 bank 以供 DMA FIFO 使用，那么即可搭配已知的外部存储缓冲器来使用位置约束。在此情况下，重要的是已知哪些缓冲器与 DMA FIFO 冲突时可能导致停滞。约束如下所示。

```
location<fifo>(net2) = { dma_fifo(aie_tile, 15, 0, 0x3100, 32) };
```

串流停滞

级联链的驱动程序应尽可能布局在靠近级联链头部的位置。这样可以减少这些冗长的时延路径的布线时延，并且可减少对于开销巨大的大型开关/DMA FIFO 的需求。

图 103：**级联链**



1. 请确保已为需缓冲的信号线指定 `fifo_depth` 约束。
2. 尽可能减少串流，首选窗口。

如果已指定 `fifo_depth`，您可检查 `AIECompiler.log` 日志中的 FIFO 报告部分。

相同 graph 输出之间存在较大偏差

如果设计对相同 graph 进行多次例化，请考虑在映射器中使用盖戳和重复流程。在此流程中，您向映射器提供的输入是，在您为每个 graph 提供的面积分组中，所有 graph 都应按相同方式进行映射。这样不仅可以简化映射器问题，还能显著降低不同 graph 输出之间的偏差。这对于含有多条天线的无线通信设计尤为重要。使用盖戳和重复流程的步骤如下。

1. 为每个 graph 定义一个面积分组。如下示例所示，在 `aiecst` 文件中添加约束。

```
"GlobalConstraints": {
  "areaGroup": {
    "name": "ant0_cores",
    "nodeGroup": ["tx_chain0.*"],
    "tileGroup": ["(0,0):(3,3)"]
  },
  "areaGroup": {
    "name": "ant1_cores",
    "nodeGroup": ["tx_chain1.*"],
    "tileGroup": ["(4,0):(7,3)"]
  }
}
```

或者在 graph 中添加约束：

```
location<graph>(tx_chain0) = bounding_box(0,0,3,3);
location<graph>(tx_chain1) = bounding_box(4,0,7,3);
```

2. 在 `aiecst` 文件内定义盖戳和重复约束。

```
{
  "GlobalConstraints": {
    "isomorphicGraphGroup": {
      "name": "isoGroup",
      "referenceGraph": "tx_chain0",
      "stampedGraphs": ["tx_chain1"]
    }
  }
}
```

请注意，对于指定为参考 graph 的 graph，同样可以为其提供额外约束，例如，并置约束或绝对位置约束。这些约束都会按相应的偏差被自动应用于其它 graph。

注释：欲知详情，请参阅《AI 引擎内核与 Graph 编程指南》(UG1079) 中的映射约束。

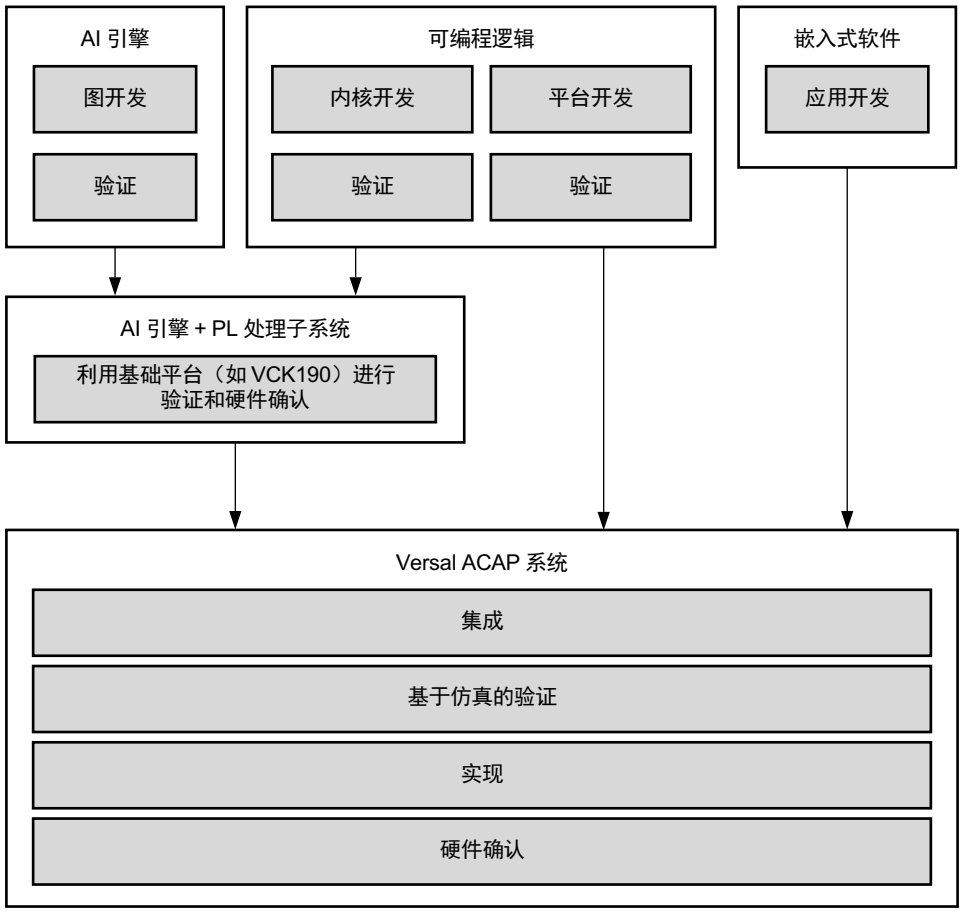
第 12 章

AI 引擎硬件剖析和调试方法论

Versal 系统剖析和确认方法论概述

下图显示了基于 Vitis 环境设计流程的开发方法论的高层次表示法。

图 104：**Vitis 环境设计流程的开发方法论**



Vitis 环境开发方法论反应了 Versal ACAP 系统的异构性质，此类系统通常是由 PS、PL 和 AI 引擎功能组成的。您可使用 Vitis 工具来独立开发并验证这些组件，并逐渐将其加以集成以构成最终系统。

Vitis 环境设计流程是迭代性进程，可能多次循环执行每个步骤，并通过后续迭代来向自适应系统添加更多层级或元件。各团队可以快速迭代早期步骤，将更多时间用于后续步骤，以便提供更详细的性能数据。

最佳实践

Vitis 环境设计方法论的基础是迭代方法和并行开发。因此，赛灵思强烈建议采用如下最佳实践：

- 并行开发自适应子系统和定制平台。

  对系统进行精确分区，即上述两个要素可各自单独开发和验证，从而节省时间和精力。

- 单独调试并验证 AI 引擎 graph 和每个 PL 内核，然后再进行集成。

  采用此方法可以尽可能提升在集成阶段快速融合的可能性。已知所有组件都正确无误的前提下，集成问题的调试难度大大降低。

- 使用标准赛灵思平台（例如，vck190）来集成并验证由 AI 引擎 graph 和 PL 内核组成的自适应子系统，然后再将目标瞄准定制平台。

  赛灵思平台都经过预验证，可立即部署到硬件上。通过使用标准赛灵思平台，AI 引擎 graph 和 PL 内核的开发者即可使用仿真或硬件开发板来验证自适应子系统，同时可以避免定制平台的不确定性和复杂性。

- 确保在流程每个阶段都能满足性能目标。

  在硬件中运行完整系统与在隔离环境中对个别组件进行仿真相比，性能结果并无明显改善。因此，有必要在流程中尽早对任何性能问题进行完整检查和调试。在组件级别确保满足性能目标难度远低于在包含所有组件间交互的复杂系统环境内满足性能目标。

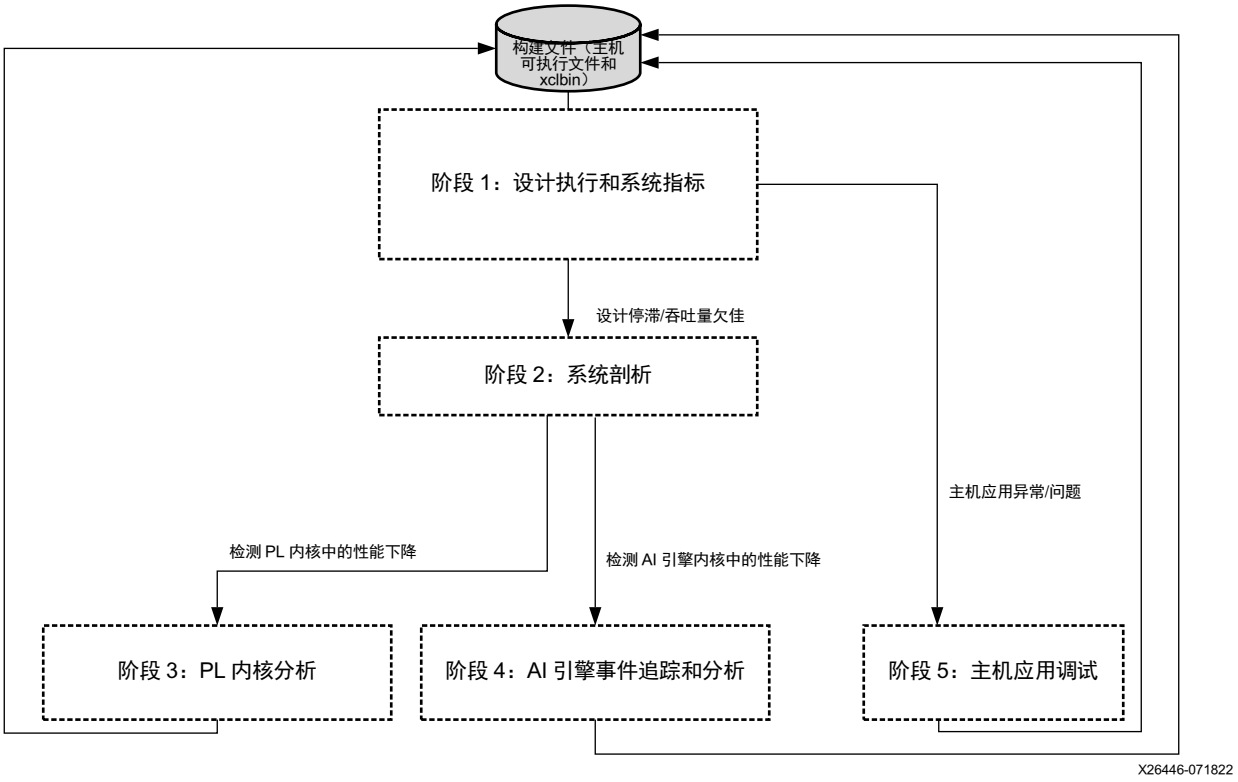
硬件剖析和调试方法论

在 Versal® AI 引擎器件上运行的设计能以 AI 引擎、PL 和 Arm® 主机作为设计目标。为了确保以此类多域器件为目标的设计能够正常运作并满足设计性能规格，赛灵思建议在硬件中采用五阶段式剖析和调试方法论。

这五个阶段如下所示：

1. 设计执行与系统指标
2. 系统剖析
3. PL 内核分析
4. AI 引擎事件追踪和分析
5. 主机应用调试

图 105：**剖析和调试方法论的五个阶段**



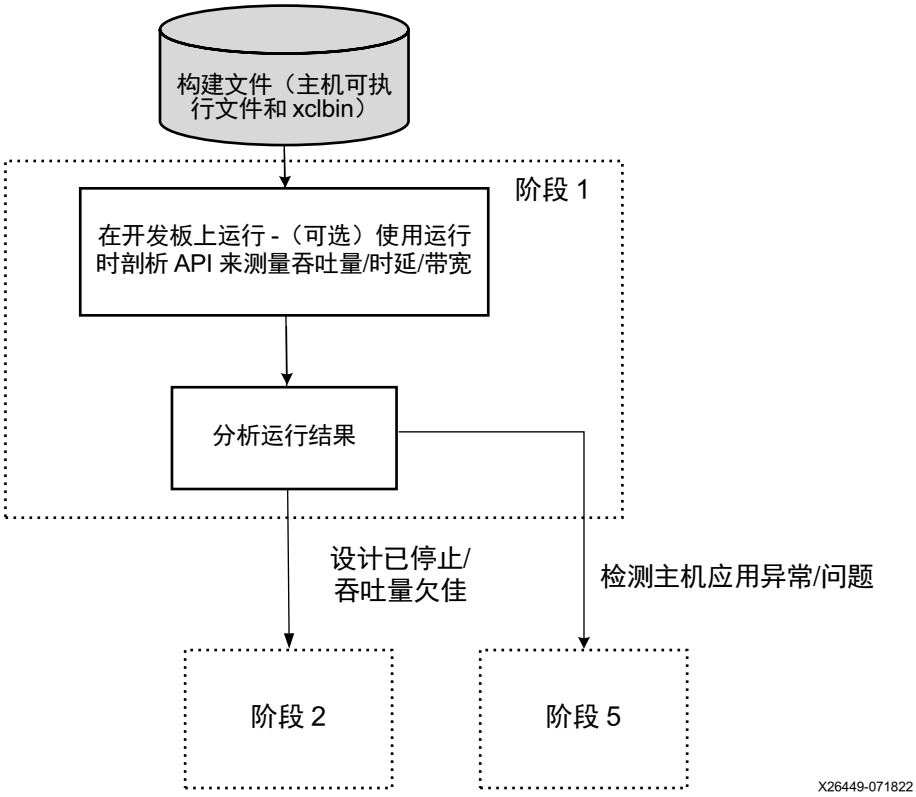
每个阶段的目标以及可用的工具和技巧如下所述。

阶段 1：设计执行与系统指标

第一阶段的目标是判定设计功能是否正确以及是否能在硬件中成功运行。您也可以分析结果并继续执行下一阶段，进行进一步调试和分析。

下图显示了此阶段中可用的任务和技巧。

图 106：**设计执行与系统指标**



此阶段可帮助您判定设计和主机应用能否在硬件中成功运行。在此阶段中，您可在自己的主机应用内使用 API 对硬件上正在运行的设计进行剖析，并判定设计是否满足吞吐量、时延和带宽目标。此外，您也可以使用在硬件中运行设计时所生成的报告，对 AI 引擎停滞和死锁进行故障排除。

以下章节列出了此阶段可用的技巧。

硬件中的主机代码中的错误处理和报告

在 Linux 上，XRT 可提供错误报告 API。在主机应用中，这些 API 可用于捕获和报告这些错误，以查明问题的根本原因。如需了解有关 XRT 错误报告 API 的详细信息，请参阅 通过 XRT API 报告错误。要检验由 AI 引擎阵列报告的错误消息，请启用并检验 `dmesg` log 日志。如需了解有关 AI 引擎阵列专用的错误处理技巧的详细信息，请参阅 AI 引擎错误事件。

下一阶段：如果判定主机应用需要进一步调试，请继续执行阶段 5。

注释：对于 AI 引擎上标记的错误，错误处理表中提供了有关后续步骤的指导信息。

分析硬件中的设计停滞

在硬件中，如果在 Linux 上遇到设计停滞，请使用 Linux 上的 XRT Xbutil 实用工具来跟踪设计中的 AI 引擎和 PL 内核的状态。如需了解有关如何使用该实用工具来生成当前设计状态和在 Vitis 分析器中直观显示结果的更多信息，请参阅 分析硬件中的 AI 引擎状态。

您也可以在 XSDB 中手动生成报告并在 Vitis 分析器中直观显示结果。欲知详情，请参阅 生成 AI 引擎状态。在 图 43：Graph 视图中的停滞和状态 和 图 44：含锁定状态的缓冲器 中提供了 Vitis 分析器中的死锁的可视化示例。

下一阶段：如果您判定设计已停滞并且需要进一步获取详细信息才能查明停滞的根本原因，请继续执行阶段 2。

报告硬件中的设计吞吐量、时延和带宽

您也可以在主机应用中通过 API 来剖析 graph 输入和输出，以判定 AI 引擎 graph 吞吐量、时延和带宽。在主机中开始和停止剖析 API 时，请谨慎处理。如需了解有关在主机应用中如何使用 API 的详细信息，请参阅 用于 graph 输入和输出的事件剖析 API。

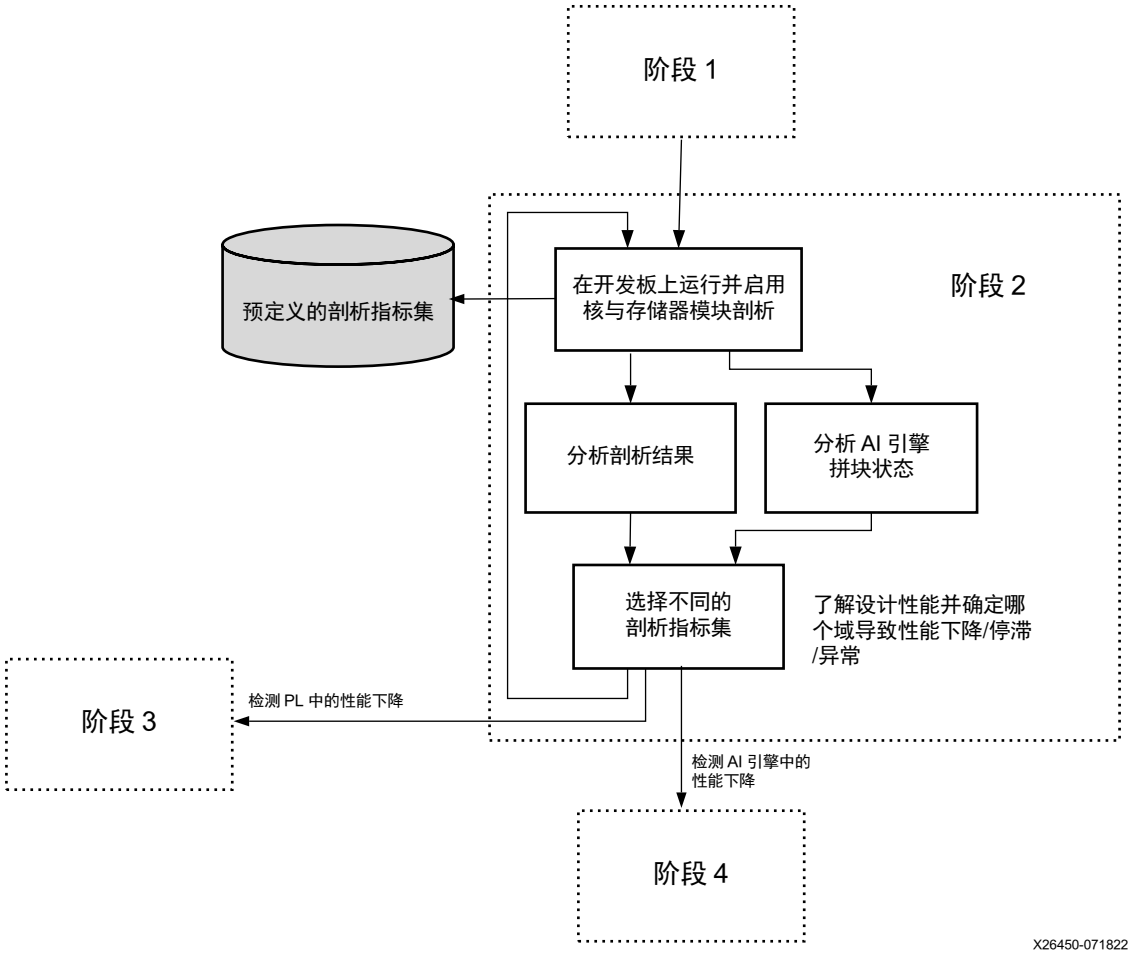
下一阶段：如果您判定设计的吞吐量或时延不达标，或者设计不满足性能目标，请继续执行阶段 2。在阶段 2 中，您可以确定可能造成性能下降的内核或 I/O。

阶段 2：系统剖析

此阶段的目标是剖析设计并判定是哪个域（AI 引擎、PL 或 NoC）导致吞吐量下降从而导致设计停滞。

下图显示了此阶段中可用的任务和技巧。

图 107：**系统剖析**



以下章节列出了此阶段可用的技巧。

剖析 AI 引擎核、接口和存储器模块

您可在 XRT 或 XSDB 流程中剖析 AI 引擎核、接口和存储器模块。这属于非侵入性功能特性，可在运行时使用 `XRT.ini` 文件或者在 XSDB 中运行脚本来启用。该功能特性使用 AI 引擎阵列中可用的性能计数器来收集剖析数据。收集的数据量和类型受到可用的性能计数器数量的限制。

表 94：**AI 引擎指标**

AI 引擎指标	
heat_map	用于剖析处于活动和停滞状态的矢量指令以及累积指令时间。这些指标表明 AI 引擎的效率，不仅包括代码效率（矢量指令）也包括与存储器和串流的交互。
stalls	用于剖析存储器、串流、锁定和级联停滞数。这些指标允许对 `heat_map` 指标集检测到的停滞的原因进行更深入的分析。
execution	用于剖析矢量的加载和存储指令时间。您可利用这些指标来判定内核代码的效率。
floating-point	用于剖析浮点异常。如果您当前使用浮点算法，那么这些指标会高亮显示代码中发生的异常。
aie_trace	用于剖析 AI 引擎和存储器模块追踪码字计数和停滞计数。使用时间追踪功能特性时，该指标十分适合用于判定追踪串流中是否存在拥塞。
write_bandwidths	用于剖析串流写入、级联写入和停滞时间。该指标可表明串流和级联输出的效率。如果存在大量停滞，则表明 graph 中的下一个内核无法以足够快的速度来耗用数据，这可能影响设计吞吐量。
read_bandwidth	用于剖析串流读取、级联读取和停滞时间。该指标可表明串流和级联输入的效率。如果存在大量停滞，则表明 graph 中的上一个内核无法以足够快的速度来提供数据，这可能影响设计吞吐量。

表 95：**存储器模块指标**

存储器模块指标	
conflicts	用于剖析存储器冲突和存储器错误。当两个存储器区块驻留在相同存储体内，并且供相同 AI 引擎（使用两个读取端口）访问或者供两个不同 AI 引擎访问时，就会发生存储器冲突。可能的解决方案是将这些存储器的位置约束到不同 bank。为了获取有关哪个 bank 导致这些冲突的更多详细信息，应对来自仿真（AI 引擎仿真）的事件进行分析，或者应在硬件中执行事件追踪。
dma_locks	用于剖析两个 DMA 上的锁定活动。4 条 DMA 通道（2xS2MM 和 2xMM2S）均由缓冲器描述符 (BD) 来驱动。“Cumulative DMA Activity”是由于所有通道上存在已停滞的锁定获取事件而导致耗费的时间的计数。所有这些 DMA 事件将帮助您了解穿过器件的部分连接速度低于期望速度的原因。
dma_stalls_s2mm	用于剖析 s2mm 通道上由于锁定获取冲突而导致的 DMA 停滞。如有 s2mm DMA 发生停滞，则表明访问目标存储器时存在冲突。原因可能是因为有另一个 s2mm 或 mm2s DMA 正在访问同一个 bank，或者内核正在执行存储器访问，导致锁定获取冲突。
dma_stalls_mm2s	用于剖析 mm2s 通道上由于锁定获取冲突而导致的 DMA 停滞。如有 mm2s DMA 发生停滞，则表明访问源存储器时存在冲突。原因可能是因为有另一个 s2mm 或 mm2s DMA 正在访问同一个 bank，或者内核正在执行存储器访问，导致锁定获取冲突。
write_bandwidths	用于剖析 s2mm DMA 所使用的带宽。这允许您评估是否能达到带宽目标。
read_bandwidths	用于剖析 mm2s DMA 所使用的带宽。这允许您评估是否能达到带宽目标。

表 96：**接口拼块指标**

接口拼块指标	
input_bandwidths	用于剖析除停滞和空闲时间之外的输入 PLIO 通道带宽。如果输入带宽过低，原因可能是停滞率过高，这也就意味着 AI 引擎阵列未能以正确的速率耗用样本。请继续执行 AI 引擎事件追踪（阶段 4）。原因也可能是空闲率过高，这表明设计的 PL 侧无法按正确的速率生成样本。继续执行 PL 内核分析（阶段 3）。

表 96：**接口拼块指标** (续)

接口拼块指标	
output_bandwidths	用于剖析除停滞和空闲时间之外的输出 PLIO 通道带宽。如果输出带宽过低，原因可能是空闲率过高，这也就意味着 AI 引擎阵列未能以正确的速率生产样本。请继续执行 AI 引擎事件追踪（阶段 4）。原因也可能是停滞率过高，这表明设计的 PL 侧无法按正确的速率耗用样本。继续执行 PL 内核分析（阶段 3）。
packets	用于剖析输入包和输出包的数量

您可多次运行设计，并在每次运行之间以 `xrt.ini` 文件中的不同参数来重新启动开发板。Vitis 分析器允许您整合不同 `xrt.run.summary` 文件报告，以便您总览接口级别的各项带宽、停滞和空闲。

如需了解有关在硬件中启用剖析和解释结果的详细信息，请参阅 AI 引擎剖析。

剖析结果允许您快速精确识别设计性能下降中所涉及的 AI 引擎、输入串流或输出串流。

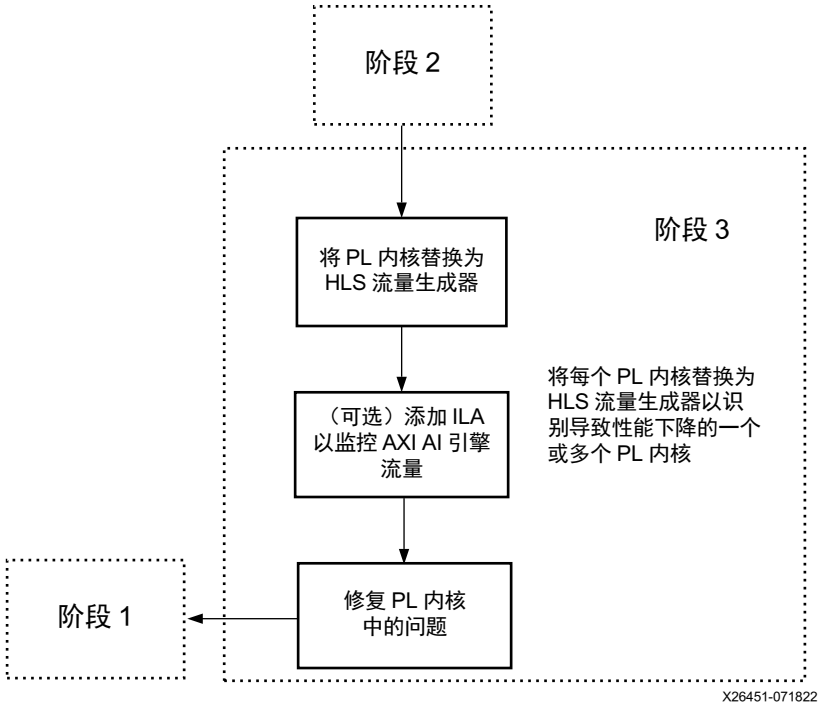
下一阶段：

- 如果您判定 PL 内核当前导致性能下降，请继续执行阶段 3。在阶段 3 中，您可以精确识别性能不达标的 PL 内核。
- 如果您判定 AI 引擎内核当前导致吞吐量下降，请继续执行阶段 4。

步骤 3：PL 内核分析

此阶段的目标是准确判定导致吞吐量下降的 PL 内核。

图 108：**PL 内核分析**



以下章节列出了此阶段可用的不同技巧。

使用 PL 剖析监控器进行剖析

您可使用 v++ 链接命令插入 PL 剖析监控器。这样您即可监控活动周期、停滞周期以及特定 PL-AI 引擎接口上传输的字节。在 AI 引擎中可将其搭配事件追踪一起启用以缩短构建时间。这样您即可识别导致性能下降的具体 PL 内核。如需了解有关添加 PL 剖析监控器的选项的更多信息，请参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的 --profile 选项。

替换 PL 内核

您可将怀疑导致性能下降的每个 PL 内核都替换为非节流 PL 内核。这样您即可判定 PL 内核是否导致性能下降。

插入 ILA 以监控特定 AXI 接口

您可插入一个或多个 ILA 以监控特定 PL AXI 接口，这样有助于准确识别发生吞吐量下降的位置和时间。它还将帮助您识别吞吐量下降发生的频率。如需了解有关使用 v++ 命令行插入 ILA 的选项的详细信息，请参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的启用内核以利用 Chipscope 进行调试。

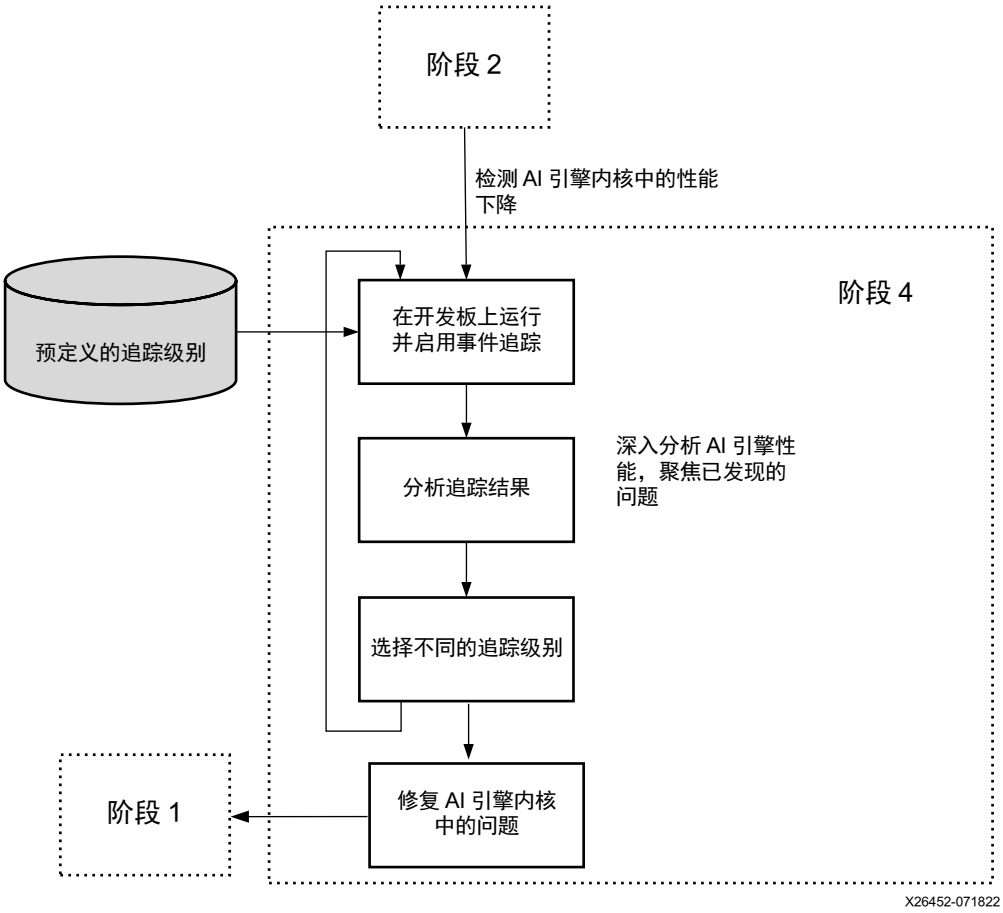
下一阶段：判定吞吐量下降的原因并修复问题后，继续执行阶段 1 以重新运行设计。

阶段 4：AI 引擎事件追踪和分析

此阶段的目标是判定导致设计性能下降或停滞或者导致死锁的 AI 引擎内核或 graph 构造。

下图显示了此阶段中可用的任务和技巧。

图 109：**AI 引擎事件追踪和分析**



以下章节列出了此设计阶段可用的不同调试技巧。

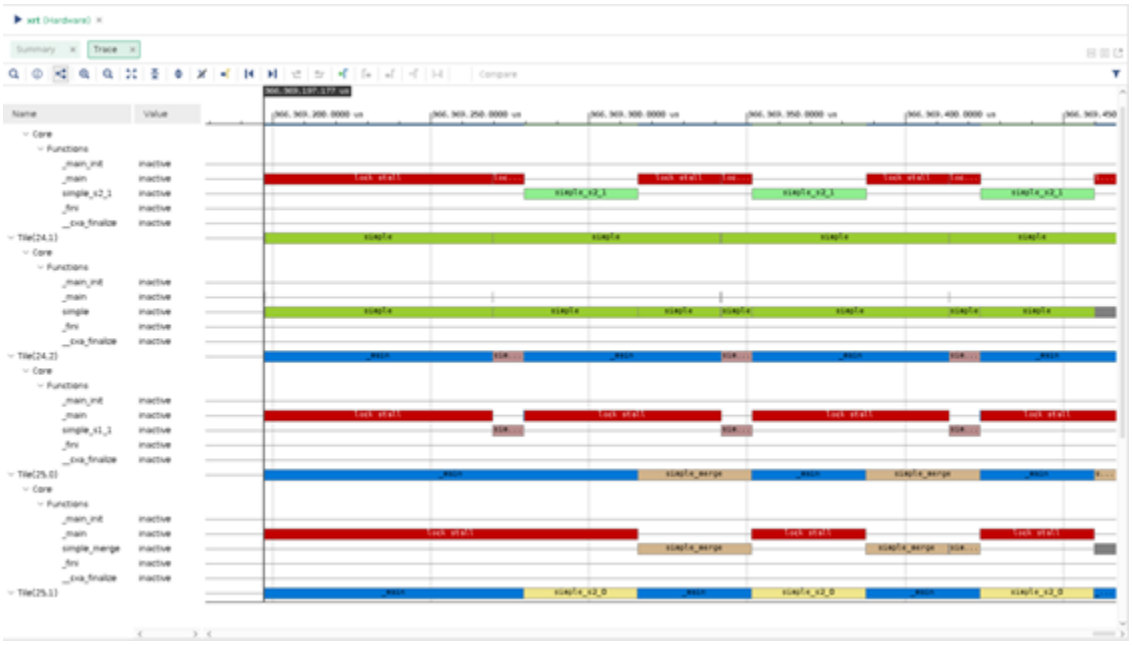
使用 AI 引擎事件追踪流程来运行和分析运行时追踪数据

AI 引擎 Event Trace（事件追踪）功能特性可在硬件中运行设计时，提供设计追踪数据的全方位视图。您可在此三阶段进程中：

1. 在启用事件追踪和其它事件追踪相关选项的前提下进行设计编译。
2. 在硬件中运行设计并收集事件追踪数据。
3. 在 Vitis™ 分析器内打开追踪汇总文件，提供以上收集的追踪数据的波形视图。

事件追踪数据允许您识别造成停滞、死锁或吞吐量下降的 AI 引擎内核，并且还可查看发生停滞/吞吐量下降状况之前的事件以及其它详细追踪信息。如需了解有关事件追踪功能特性的详细信息，请参阅 硬件中的事件追踪。

图 110：**事件追踪**



如需了解在硬件中运行事件追踪时遇到的特定技巧的详细解决方案，请参阅 在硬件中对事件追踪进行故障排除。该功能特性受到可供器件内的事件追踪使用的事件追踪计数器、串流、DDR 存储器和设计资源的限制。

剖析内核内性能

您还可以使用 `aie::tile::cycles()` API 来剖析特定内核内部的代码块。

要在硬件内获取该值，您可将该值写入存储器或者写入输出串流。写入输出串流的示例如下所示。随后即可在主机应用中检验此数据串流以读回剖析数据。

```
// get the current tile
aie::tile tile=aie::tile::current();
unsigned long long time=tile.cycles(); //cycle counter of [SS1] the AI
Engine tile
writeincr(out,time);
{//loop to be profiled
}
time=tile.cycles();//cycle counter of the AI Engine tile
writeincr(out,time);
```

这是极具侵入性的内核代码剖析方法。赛灵思建议您使用此方法搭配 AI 引擎仿真器来进行 graph 仿真。此外，仿真中的追踪和剖析数据也可用于此目的。

如需了解有关 `aie::tile::cycles()` API 的详细信息，请参阅《AI 引擎内核与 Graph 编程指南》(UG1079)。

Vitis IDE 调试器

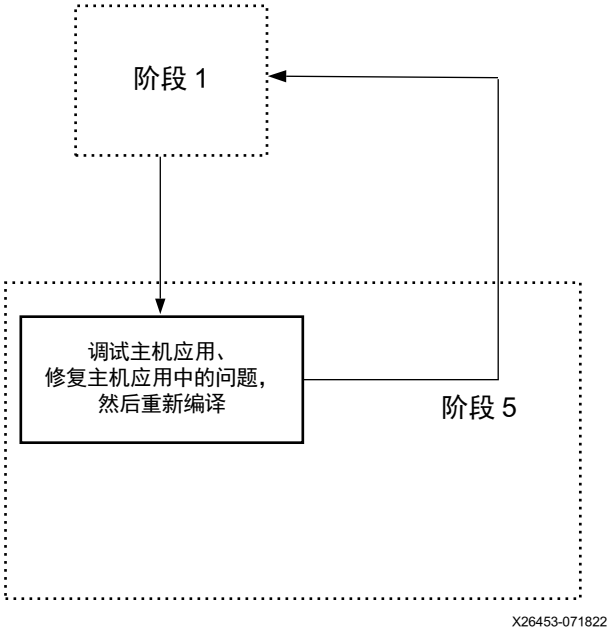
您也可以使用 Vitis IDE 调试器来调试内核源代码。如需了解有关 Vitis 调试器的详细信息，请参阅 第 10 章：调试 AI 引擎应用。

下一阶段：判定吞吐量下降的原因并修复问题后，继续执行阶段 1 以重新运行设计。

阶段 5：主机应用调试

此阶段的目标是调试主机应用并解决应用异常或崩溃（如果存在）。

图 111：**主机应用调试**



以下章节列出了此阶段可用的不同技巧。

Vitis IDE 调试器

您可使用 Vitis IDE 调试器来调试主机应用源。如需了解有关 Vitis 调试器的详细信息，请参阅 第 10 章：调试 AI 引擎应用。

Printf

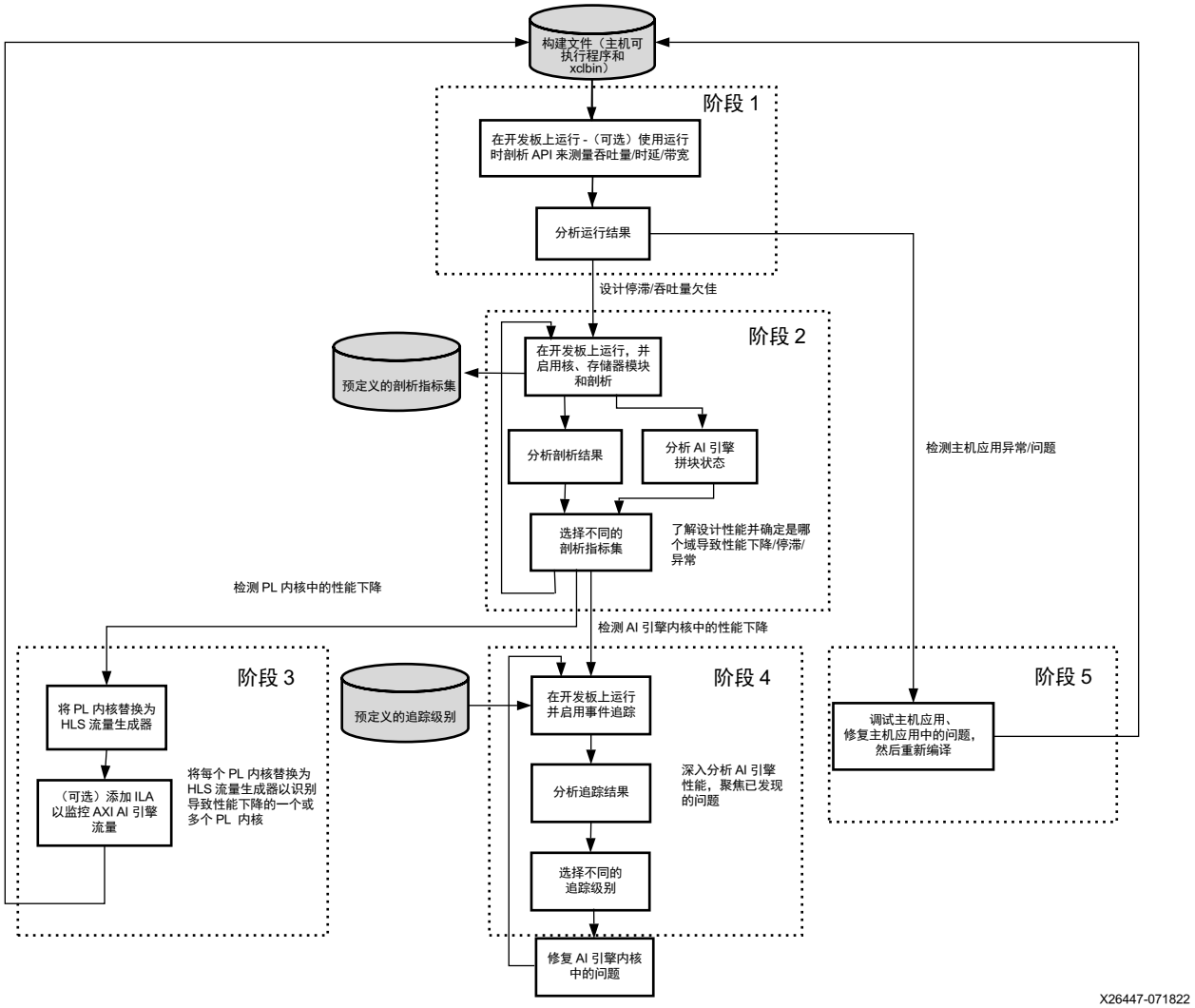
您也可以在主机代码中使用 `printf()` 语句来帮助调试主机应用。

下一阶段：判定失败原因并修复问题后，可以重新编译主机应用并继续执行步骤 1。

汇总

以下显示了完整的硬件剖析和调试方法论。

图 112：**硬件剖析和调试方法论**



AMD XILINX

附录 A

事件追踪参考

仿真事件追踪

AI 引擎仿真器中的事件可提供有关特定时刻的系统信息。与时间戳、类型和一系列数据值关联的事件被称为有效载荷。有效载荷的解读取决于事件类型。时间戳支持事件排序、计算因果关系和对一系列事件执行验证器的实现。

为便于事件建模，AI 引擎阵列的关键模块包括处理器核、DMA、锁定模块、存储器和 I/O 串流。每个模块均可视作为事件生成器/响应器。每个模块均可接收事件和响应事件。可生成新事件作为响应。事件基于事件生成器来分类。在事件定义中并不会显式提及时间戳。每起事件均以有效载荷来描述，有效载荷即多个与该事件关联的值。每个 AI 引擎、存储器、DMA 或锁定均可按二维索引 <col, row> 进行寻址，二维索引即 AI 引擎阵列中的列索引和行索引。下表显示了部分 AI 引擎事件。所有事件均包含时间戳（表中省略）。

表 97：**AI 引擎事件**

事件名称	值	描述
CORE_RESET	<col, row>	核 <col, row> 解复位，开始从 PC0 执行代码。
CORE_WAIT	<col, row>	核 <col, row> 正在等待锁定、串流输入或存储器响应。
CORE_READY	<col, row>	解决停滞问题后，核 <col, row> 已就绪。
ACQ_READ_LOCK_REQ ACQ_WRITE_LOCK_REQ	<col, row, lockid, dir, pc, info>	核 <col, row> 请求对其相邻的某一个存储体进行锁定。<pc> 对应当前程序计数器。<info> 是符号信息。
REL_LOCK_REQ	<col, row, lockid, dir>	核 <col, row> 释放相邻锁定模块中的锁定。
CORE_READ_REQ	<col, row, address, dir, port, bank, data, pcinfo, addrinfo>	核 <col, row> 请求对存储器地址执行读取。此地址为存储器本地的 15 位地址。搭配 <dir, address> 构成 17 位地址。<dir> 字段可指定 [E, W, N, S] 方向。<port> 用于指定加载端口 (A,B)，<bank> 则用于指定访问奇数或偶数存储体。在事件中会提供有关程序变量和 <pc> 值的符号信息。
CORE_WRITE_REQ	<col, row, address, dir, bank, data, size.pcinfo, addrinfo>	核 <col, row> 会在特定的 15 位地址写入数据。<dir> 可指定当前访问的存储器的方向。在事件中会提供有关 <pc> 值和数据符号的符号信息。
PC_CHANGE	<col, row, pc, info>	核 <col, row> 会显示 <pc> 值的更改。这对识别循环很有用。
DMA_S2MM_ACQ_LOCK	<col, row, ch>	DMA 请求锁定。<ch> 字段表示 DMA 通道编号。受支持的通道为 0 和 1。
DMA_S2MM_IDLE	<col, row, bd, ch>	DMA 处于空闲状态。<bd> 表示缓冲器描述符。
DMA_S2MM_START	<col, row, bd, ch, start>	在开始地址启动 DMA 传输
DMA_S2MM_DONE	<col, row, bd, ch>	在结束地址终止 DMA 传输
DMA_S2MM_LOCKSTALL	<col, row, bd, ch>	DMA 已停滞以获取锁定
DMA_S2MM_LOCKSTALL_RELEASE	<col, row, bd, ch>	DMA 获取锁定，并释放停滞
DMA_MM2S_ACQ_LOCK	<col, row, ch>	DMA <col, row> 请求锁定

表 97：**AI 引擎事件** (续)

事件名称	值	描述
DMA_MM2S_IDLE	<col, row, ch>	DMA <col, row> 处于空闲状态
DMA_MM2S_START	<col, row, bd, ch, start>	在 <col, row> 的开始地址处启动 DMA 传输，<bd> 表示缓冲器描述符。
DMA_MM2S_DONE	<col, row, bd, ch>	在 <col, row> 的结束地址处终止 DMA 传输，<bd> 表示缓冲器描述符。
DMA_MM2S_LOCKSTALL	<col, row, bd, ch>	DMA 已停滞以获取锁定
DMA_MM2S_LOCKSTALL_RELEASE	<col, row, bd, ch>	DMA 获取锁定，并释放停滞
IO	<dir, id, data >	I/O 事件表示输入串流上存在新数据。<dir> [E, W, N, S] 和 <id> [0–15] 相结合即可表示物理串流，data 字段表示串流上出现的 32 位数据。
DATA_HEAD	<col, row, name, netid, pktid, idx>	核 <col, row> 在信号线 <netid> 上通过布线 <name> 来传输包切换数据报头。此包带有包 ID <pktid>。<idx> 保留。
DATA_START	<col, row, name,netid, pktid, idx>	数据有效载荷（包切换或电路切换）起始。所有字段都遵循 DATA_HEAD 字段的含义。
DATA_RESUME	<col, row, name,netid, pktid, idx>	数据有效载荷在停滞后恢复。所有字段都遵循 DATA_HEAD 字段的含义。
DATA_STALL	<col, row, name, netid, idx>	数据有效载荷在包有效载荷内停滞。所有字段都遵循 DATA_HEAD 字段的含义。
DATA_END	<col, row, name, netid, idx>	数据有效载荷（包切换或电路切换）结束。所有字段都遵循 DATA_HEAD 字段的含义。
PL_TO_SHIM, SHIM_TO_PL	<name, col, channelId, data0, data1, tlast>	数据在 PL 与 AI 引擎阵列接口 (SHIM) 之间移动。名称显示 PL 块名称。
PL2PL 和 PL2PL_E	<name, port, data0, data1, tlast, tkeep>	PL 端口活动起始和结束。名称和端口显示 PL 块名称/端口。
DM_READ_REQ	<col, row, portname>	存储器模块的某个端口上的读取请求事件。读取端口可以是核、DMA 或 AXI4 存储器映射接口读取请求端口。
DM_WRITE_REQ	<col, row, portname>	存储器模块的某个端口上的写入请求事件。写入端口可以是核、DMA 或 AXI4 存储器映射接口写入请求端口。
DM_BANK_CONFLICT	<col, row, bankid, bank>	存储体冲突事件会显示因向存储器模块的同一个存储体发送多个请求而导致的存储体冲突。

硬件事件追踪

下表包含硬件事件追踪期间生成的所有可能事件。通过观察 `hwanalyze` 所生成的 `ctf` 数据，描述提供了有关当前调用的事件的信息。

事件读取方式如以下示例所示：如果选中 `functions` 事件，则仅生成“FUN_CALL”和“FUN_RETURN”，并提供有关函数驻留在其中的拼块以及程序计数器 (PC) 的详细信息，以及事件当前是否正在调用该函数，或者该函数是否正被调用的信息。

表 98：**硬件事件追踪**

事件名称	值	描述
FUN_CALL	<col, row, PC, val, info>	PC：函数调用指令 PC Val：0 或 2 参考信息： 调用含 val 0 的函数名称 已调用含 val 2 的函数名称
FUN_RETURN	<col, row, PC, val, info>	PC：函数返回指令 PC Val：0 或 2 参考信息： 返回含 val 0 的函数名称 已返回含 val 2 的函数名称
MEM_STALL	<col, row, val, info>	Val：23（固定） 参考信息： 函数名
MEM_STALL_RELEASE	<col, row, val, info>	Val：2（固定） 参考信息： 函数名
STREAM_STALL	<col, row, val, info>	Val：24（固定） 参考信息： 函数名
STREAM_STALL_RELEASE	<col, row, val, info>	Val：2（固定） 参考信息： 函数名
CASCADE_STALL	<col, row, val, info>	Val：25（固定） 参考信息： 函数名
CASCADE_STALL_RELEASE	<col, row, val, info>	Val：2（固定） 参考信息： 函数名
LOCK_STALL	<col, row, val, info>	Val：26（固定） 参考信息： 函数名
LOCK_STALL_RELEASE	<col, row, val, info>	Val：2（固定） 参考信息： 函数名
DMA_S2MM_RUNNING	<col, row, ch, val>	Ch：通道 ID Val：0（活动结束） 1（活动开始）
DMA_MM2S_RUNNING	<col, row, ch, val>	Ch：通道 ID Val：0（活动结束） 1（活动开始）
OVERRUN	<col, row>	拼块 <col, row> 中发生的溢出

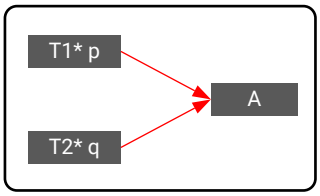
在 AI 引擎内核中使用 restrict 关键字

在 AI 引擎内核 C++ 代码中允许使用 restrict 关键字 (`__restrict`)。本附录着重介绍了赛灵思有关在 AI 引擎内核代码上下文中使用 restrict 关键字的建议。

指针混叠

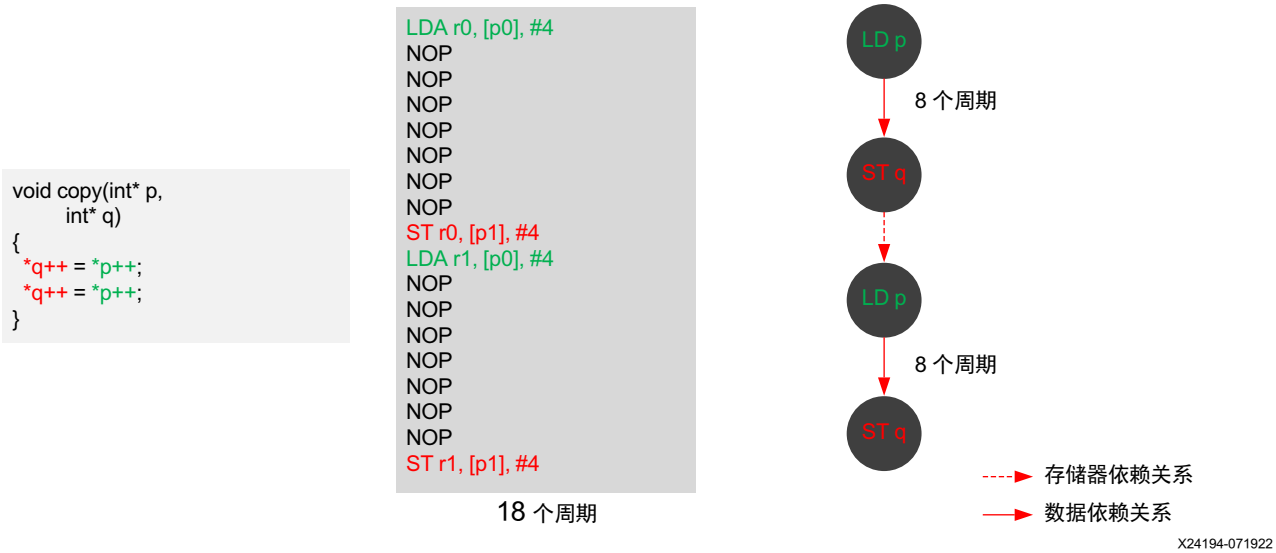
指针混叠表示可使用不同指针名称访问相同存储器位置的情况。在 C/C++ 中，严格混叠规则表示如果指针指向截然不同的类型，则假定这些指针不混叠。混叠会对程序执行顺序施加强约束。以下显示了 `p` 和 `q` 的混叠。

图 113：**指针混叠**



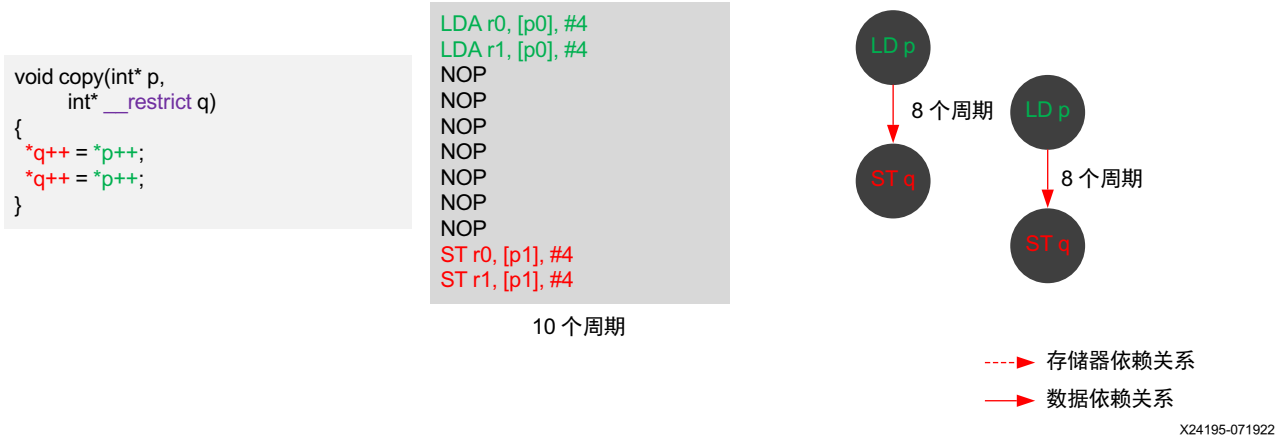
以下指针混叠示例中，指针 `p` 和 `q` 指向相同地址。中间列显示的是由编译器生成的汇编语言代码，右侧显示的是运算和时钟周期。

图 114：**混叠代码示例**



通过在此代码示例中添加 restrict 关键字，编译器即可最优化生成的汇编语言，以增加硬件中的运算并行度。以下示例显示了如何使用 restrict 关键字来防止混叠使用更少的时钟周期来完成同样的运算。

图 115：**使用 restrict 关键字来避免混叠**



存储器依赖关系

代码中的存储器依赖关系可能限制编译器尝试执行的最优化种类。例如，在以下代码中，`xyz` 与指针 `p` 和 `q` 可能无关。但在函数代码中，指针 `p` 和指针 `q` 指向同一个全局变量 `xyz`。编译器必须保证在上述情况下都能正确执行。鉴于这些不同种类的存储器依赖关系，编译器需采取保守操作并限制最优化。

图 116：**不相关的指针**

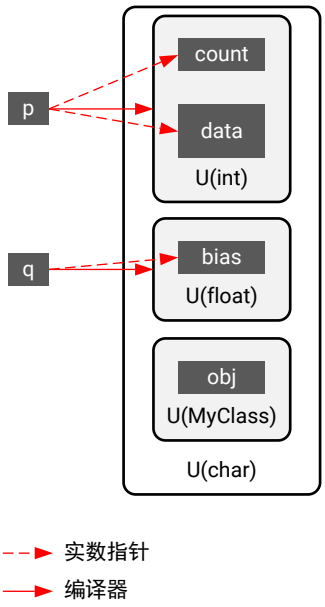
```
int xyz;
void bar(int *p, int *q)
{
   ...
   p=&xyz;
   ...
   q=&xyz;
   ...
}
```

X24245-071520

严格混叠规则

严格混叠规则规定，如果各指针指向截然不同的类型，则假定这些指针不会发生混叠，但 `char*` 和 `void*` 除外，这两者可混叠到任何其它数据类型。如下图所示，其中显示了对象全类 (universe) 和关联的指针。

图 117：**对象全类**




```
int count;
float bias;
int data[1000];
MyClass obj;
```

```
void foo(int* p, float q)
{
  // Graph at this time
  *p = *p + 1;
  *q = *q + 0.5;
}
```

```
void bar()
{
  foo(&count, = &bias);
  foo(data, &bias);
  count = count + 1;
  data[0] = data[0] + 0.5;
}
```

X24196-071922

- 指针与 universe 类型关联：U(T)：T 表示模板 (template)，在前图中显示了各种模板，包括 `int` 全类和 `float` 全类；此外每个设计还有一个 `MyClass` 全类。此外，默认还有一个包含所有全类的 `char` 全类。
- 全类不发生混叠：指针 `p` 只能指向 `int` 全类内的任意地址，而指针 `q` 则只能指向 `float` 全类内的任意地址。因此，指针 `p` 与指针 `q` 无法混叠。
- 衍生的指针指向原始全类：衍生自 restrict 指针的指针被视为 restrict 指针，并指向相同的受限存储器区域。请参阅衍生的指针。
- `char*` 全类包含所有全类：`char` 指针可指向所有全类中的任意变量。

对于两个相同类型的指针，如下所示，其中 `p` 和 `q` 均为 `int`，则编译器采用保守方式，并应用混叠，导致性能损失。

图 118：**性能损失**

```
void foo(int* p, int* q {
    *p = *p + 1;
    *q = *q + 2;
}
```

```
LDA r0, [p0]
NOP
NOP
NOP
NOP
NOP
NOP
ADD r0, r0, #1
ST r0, [p0]
LDA r1, [p1]
NOP
NOP
NOP
NOP
NOP
NOP
NOP
ADD r1, r1, #2
ST r1, [p1]
```

X24197-071420

对于两个不同类型的指针，如下所示，其中 `p` 是 `int`，`q` 是 `float`，编译器会应用严格混叠规则，如果存在混叠，则会发生未定义的行为。

图 119：**两个不同类型的指针**

```
void foo(int* p, float* q {
    *p = *p + 1;
    *q = abs(*q);
}
```

```
LDA r0, [p0]; LDB r1, [p1]
NOP
NOP
NOP
NOP
NOP
NOP
NOP
ADD r0, r0, #1
ST r0, [p0]; AND r1, r1, r2
LDA r1, [p1]
```

X24198-071420

restrict 关键字

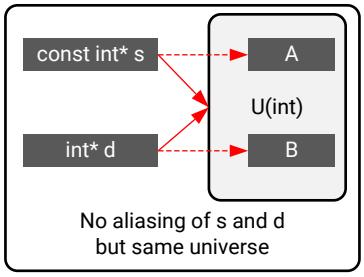
restrict 关键字主要在指针声明中用作为指针的类型限定符。它并不会添加任何新功能。它允许您将有关可能的最优化的信息告知编译器。将 `__restrict` 与指针搭配使用即可告知编译器，该指针是访问所指向的对象的唯一途径，并且该编译器无需执行任何额外检查。

注释：如果程序员使用 restrict 关键字并违反上述条件，则可能发生未定义的行为。

以下是指针（默认）不含混叠的另一个示例。

图 120：**无混叠示例**

```
void copy_block(int *d, const int *s) {
  for (int n = 0; n < 128; n++) {
    d[n] = s[n];
  }
}
```




```
MOV.s12 lc, #128
Loop_start:
  LDB r1, [p1]
  NOP
  NOP
  NOP
  NOP
  NOP
  NOP
  NOP
  ST r1, [p0] #4
```

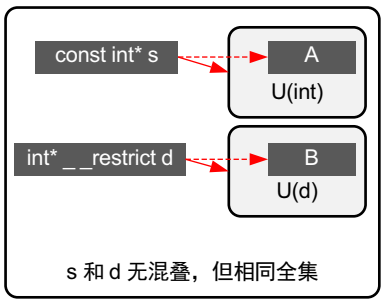
1153 个周期，9 个周期循环

X24199-071822

应用 restrict 关键字以改善性能。以下示例显示了与其它指针之间不存在存储器依赖关系。

图 121：**与其它指针之间不存在存储器依赖关系**

```
void copy_block(int * __restrict d, const int *s) {
  for (int n = 0; n < 128; n++) {
    d[n] = s[n];
  }
}
```




```
MOV.s12 lc, #120
LDA r1, [p1] #4
LDA r1, [p1] #4
LDA r1, [p1] #4
LDA r1, [p1] #4
LDA r1, [p1] #4
LDA r1, [p1] #4
LDA r1, [p1] #4
LDA r1, [p1] #4
loop_start:
  LDA r1, [p1] #4; ST r1, [p0] #4
ST r1, [p0] #4
ST r1, [p0] #4
ST r1, [p0] #4
ST r1, [p0] #4
ST r1, [p0] #4
ST r1, [p0] #4
ST r1, [p0] #4
ST r1, [p0] #4
```

137 个周期，1 个周期循环

X24200-071922

限制条件

C 语言标准可提供特定指针限定符 `__restrict`，此限定符旨在通过显式声明任意指针引用与所有其它变量之间的数据独立性，从而开展更激进的编译器最优化。例如：

```
int a; // global variable
void foo(int* __restrict p, int* q)
{
  for (...) { ... *p += a + *q; ...}
}
```

现在 `foo` 分析可在已知 `*p` 所表示的对象与 `*q` 和 `a` 并不相同的前提下继续操作。因此，`a` 和 `*q` 现在可在循环之前加载一次。

当前，编译器前端并不会消除对相同阵列进行不同访问之间所存在的任何歧义。因此更新阵列中的某个元素时，假定整个阵列均已更改该值。`__restrict` 限定符可用于覆盖此保守假定。如果要获取对同一阵列的多个独立指针，那么此限定符很有用。

```
void foo(int A[])
{
  int* __restrict rA = A; // force independent access
  for (int i = ...)
  rA[i] = ... A[i];
}
```

在此示例中，`__restrict` 限定符允许对循环进行软件流水打拍，在上一个阵列元素仍必须完成存储的同时，下一个阵列的元素可能已加载。为了尽可能扩大 `__restrict` 限定符的影响，默认情况下，编译器前端会在初始化器中插入 `chess_copy` 运算，编写方式如下：

```
int* __restrict rA = chess_copy(A);
```

为了使优化器中的两个指针之间保持有所差别（例如，无公共子表达式消除），这是必需的。对于 AI 引擎编译器前端，可通过 `-mllvm -chess-implicit-chess_copy=false` 选项来禁用此行为。因此，`chess_copy` 会创建两个指针，而 `__restrict` 则会告知编译器不考量通过这些指针的存储/加载之间的任何相互依赖关系。对于具有局部作用域的 `__restrict` 指针，仅在 `__restrict` 指针生存期内，相互独立性的假设才有效。

衍生自 `__restrict` 指针的指针（例如，`rA+1` 或者穿越指针内部调用）会保留此限制，即这些指针被视为指向相同的受限存储区域。

注释：如需了解有关 `chess_copy` 的详细信息，请参阅 AI 引擎专区内提供的“Chess 编译器用户手册”。

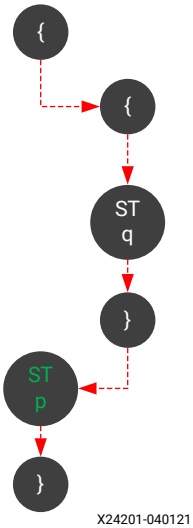
未定义的行为

使用 restrict 关键字可改善性能，如上一个主题所述。但如果关键字使用不当，则会出现问题。`__restrict` 子指针必须在不同于父指针的块级作用域内使用，例如，指针 `p` 和 `q`，如以下示例所示。

有效示例 1

图 122：**restrict 关键字的使用**

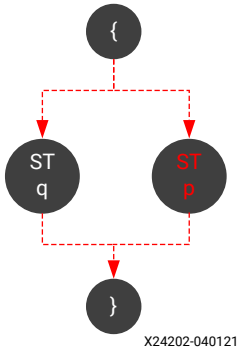
```
void foo(int *p)
{
  {
    int * __restrict q = p;
    *q = 3;
  }
  *p = 1; // The result is 1
}
```



在相同作用域内使用父指针可能中断 `__restrict` 约定，并生成未定义的行为，如以下示例中的指针 `p` 和 `q` 所示。

图 123：**未定义的行为**

```
void foo(int *p)
{
    int * __restrict q = p;
    *q = 3;
    *p = 1; // UNDEFINED!
}
```

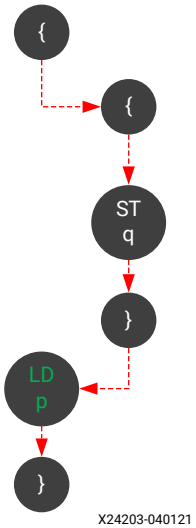


有效示例 2

这也可能在 `load` 操作期间发生，如下图中的绿色文本所示 (`return *p;`)。

图 124：**加载操作**

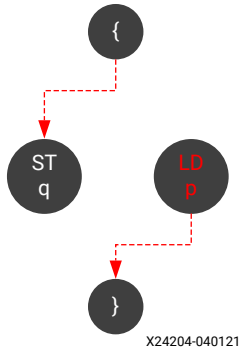
```
int bar(int *p)
{
  {
    int * __restrict q = p;
    *q = 3;
  }
  return *p; // Returns 3
}
```



当在相同作用域内使用 restrict 指针时，会发生未定义的行为，如以下示例中的指针 `p` 和 `q` 所示。

图 125：**相同作用域内的 restrict 指针**

```
int bar(int *p)
{
  int * __restrict q = p;
  *q = 3;
  return *p; // UNDEFINED!
}
```



内联函数的有效示例

以下代码显示了有效的内联函数调用，其中，指针 `p` 与指针 `q` 在不同作用域内使用。

图 126：**内联函数调用**

```
inline
void bar(int *q){
  *q = 3;
}

void foo(int *p)
{
 *p = 1;
 {
  int * __restrict q = p;
  bar(q);
 }
}
```

X24205-040121

当在相同作用域内使用 restrict 指针时，会发生未定义的行为，如以下示例中的指针 `p` 和 `q` 所示。

图 127：**相同作用域内的内联函数调用**

```
inline
void bar(int *q){
  *q = 3;
}

void foo(int *p)
{
 *p = 1;
  int * __restrict q = p;
  // UNDEFINED BEHAVIOR!
  bar(q);
}
```

X24206-040121

内联函数中 restrict 关键字的作用域

如果作用域内没有任何其它访问，那么声明 restrict 指针对于性能没有任何益处。

图 128：**不含性能益处的有效示例**

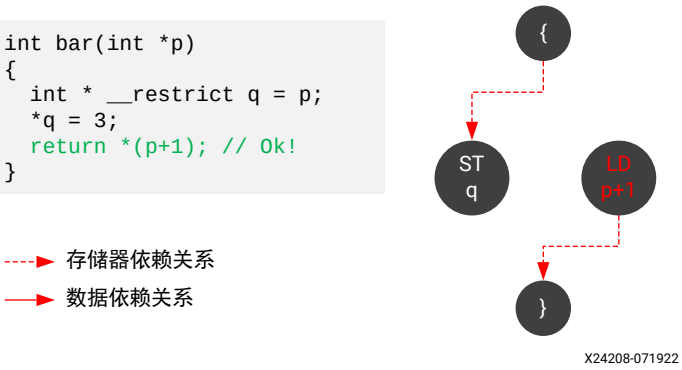
```
inline int read(int *p) {
  int * __restrict q = p;
  return *q;
}

inline void write(int *p, int d){
  int * __restrict q = p;
  *q = d;
}

int foo(int *p1, int *p2) {
  write(p1, 4);
  return read(p2);
}
```

X24207-040121

在特殊情况下，可执行无混叠访问，如以下示例所示。此处使用了父指针 `p`，但它指向不同位置，因此这是可接受的。

图 129：**特殊情况 - 无混叠访问**

```
int bar(int *p)
{
  int * __restrict q = p;
  *q = 3;
  return *(p+1); // Ok!
}
```



对读取/修改/写入循环使用 restrict 关键字的益处

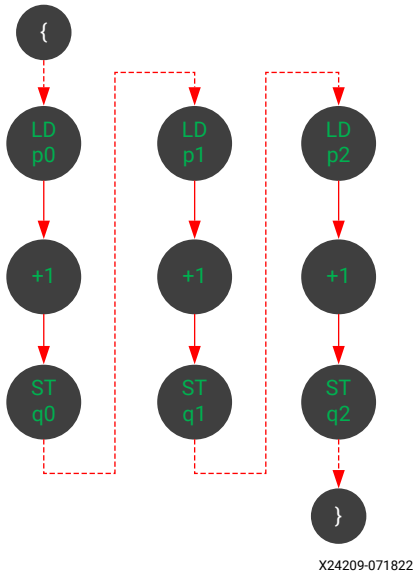
以下示例无需 restrict 关键字也有效，但性能欠佳。

图 130：**不含 restrict 关键字的示例**

```
void inc1(int* p)
{
  for (int n = 0; n < 128; n++)
  {
    p[n] = p[n] + 1;
  }
}
```

```
MOV.s12 lc, #128
loop_start:
  LDA r1, [p0], #4
NOP
NOP
NOP
NOP
NOP
NOP
NOP
ADD r0, r1, #1
ST r0, [p0], #4
```

10 个周期/迭代



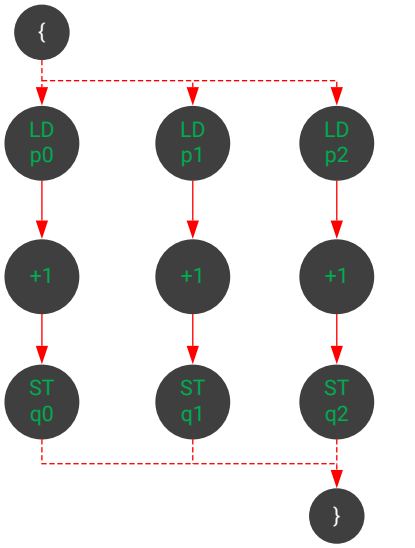
添加 restrict 关键字即可支持每次迭代访问不同位置，其中各迭代之间不存在混叠 (`__restrict`) 并且数据依赖性会保留迭代内部的混叠。增加并行度可以提升性能。

图 131：**添加 restrict 关键字**

```
void inc2(int* p){
  int * __restrict q = p;
  for (int n = 0; n < 128; n++){
    q[n] = p[n] + 1;
  }
}
```

```
loop_start:
  LDA r1, [p1], #4;
  ST r0, [p0], #4;
  ADD r0, r1, #1
```

1 个周期/迭代



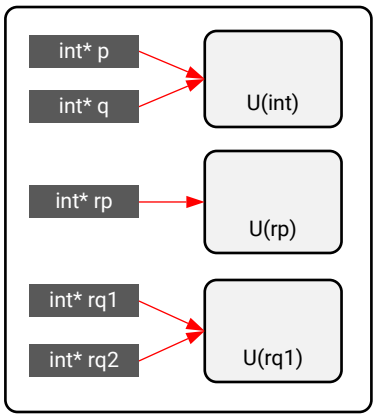
X24210-071922

衍生的指针

衍生自 restrict 指针的指针被视为 restrict 指针，并指向相同的受限存储器区域，如以下示例所示，其中，`rq2` 衍生自 `rq1`（定义为 restrict 指针），因此同样属于 restrict 指针，并指向相同的全类。

图 132：**指向相同受限存储器区域的指针**

```
void foo(int * p, int *q){
  int * __restrict rp = p;
  int * __restrict rq1 =
q;
  int * rq2 = rq1 + 3;
  ...
}
```

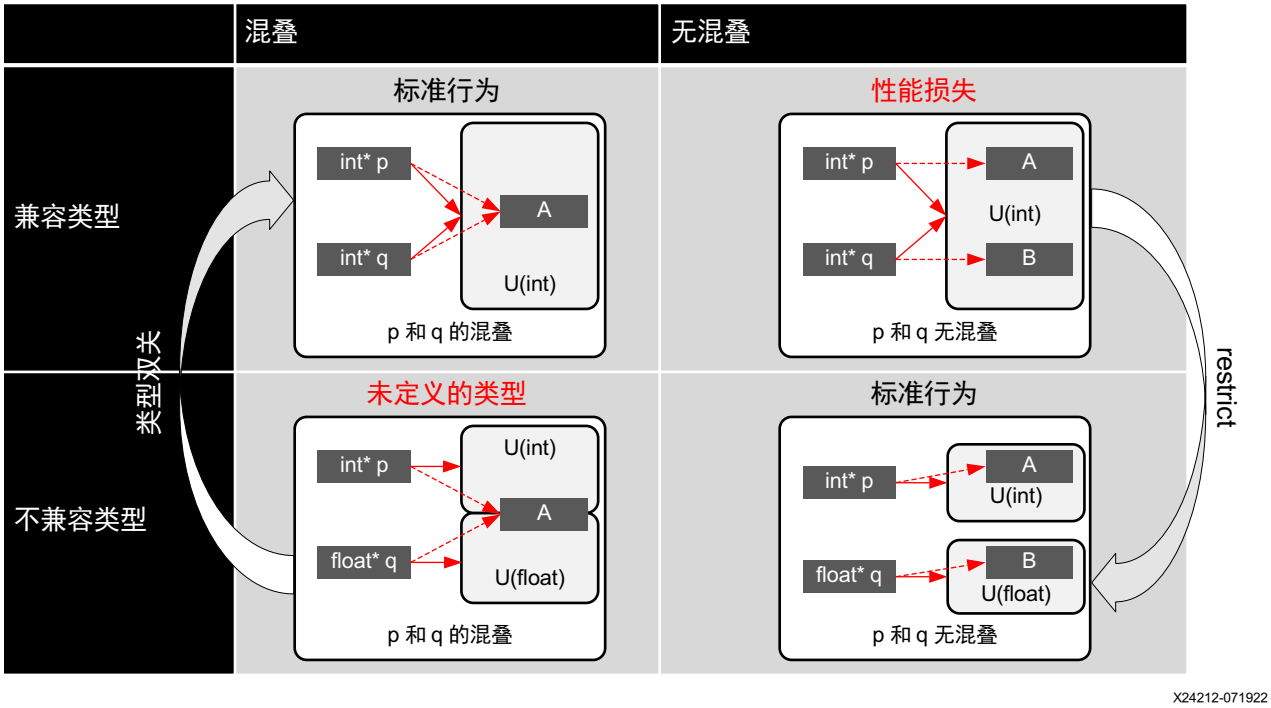


X24211-040121

汇总

在 AI 引擎内核编程中正确使用限制关键字 (`__restrict`) 可以提升代码性能并消除代码中未定义的行为。但请注意，如果将 restrict 指针分配到相同作用域，则可能导致设计中出现未定义的行为。

图 133：**restrict 关键字使用信息汇总**



窗口和串流 API 的非模板版本

重要提示！请注意，这些 API 仅适用于内部函数，不适用于 AI 引擎 API。

内核窗口运算

窗口数据类型

表 99：**受支持的窗口数据类型**

输入窗口类型	输出窗口类型
input_window_int8	output_window_int8
input_window_int16	output_window_int16
input_window_int32	output_window_int32
input_window_int64	output_window_int64
input_window_uint8	output_window_uint8
input_window_uint16	output_window_uint16
input_window_uint32	output_window_uint32
input_window_uint64	output_window_uint64
input_window_cint16	output_window_cint16
input_window_cint32	output_window_cint32
input_window_float	output_window_float
input_window_cfloat	output_window_cfloat

将当前读/写位置前移

在下表中，<input_window_type> 表示任意允许的输入窗口数据类型。同样，<output_window_type> 表示任意允许的输出窗口数据类型。

表 100：**允许的窗口类型**

用途	输入窗口类型	输出窗口类型
用于增加当前读/写位置，增加的值为底层窗口类型的计数次数。	void window_incr(<input_window_type> *w, int count);	void window_incr(<output_window_type> *w, int count);

表 100：**允许的窗口类型** (续)

用途	输入窗口类型	输出窗口类型
用于增加当前读/写位置，增加的值为底层窗口类型的计数次数的 4 倍。	`void window_incr_v4(<input_window_type> *w, int count);`	`void window_incr_v4(<output_window_type> *w, int count);`
用于增加当前读/写位置，增加的值为底层窗口类型的计数次数的 8 倍。	`void window_incr_v8(<input_window_type> *w, int count);`	`void window_incr_v8(<output_window_type> *w, int count);`
用于增加当前读/写位置，增加的值为底层窗口类型的计数次数的 16 倍。	`void window_incr_v16(<input_window_type> *w, int count);`	`void window_incr_v16(<output_window_type> *w, int count);`
用于增加当前读/写位置，增加的值为底层窗口类型的计数次数的 32 倍。	`void window_incr_v32(<input_window_type> *w, int count);`	`void window_incr_v32(<output_window_type> *w, int count);`
用于增加当前读/写位置，增加的值为底层窗口类型的计数次数的 64 倍。	`void window_incr_v64(<input_window_type> *w, int count);`	`void window_incr_v64(<output_window_type> *w, int count);`

将当前读/写位置后移

在下表中，`<input_window_type>` 表示任意允许的输入窗口数据类型。同样，`<output_window_type>` 表示任意允许的输出窗口数据类型。

表 101：**允许的窗口类型**

用途	输入窗口类型	输出窗口类型
用于减少当前读/写位置，减少的值为底层窗口类型的计数次数。	`void window_decr(<input_window_type> *w, int count);`	`void window_decr(<output_window_type> *w, int count);`
用于减少当前读/写位置，减少的值为底层窗口类型的计数次数的 4 倍。	`void window_decr_v4(<input_window_type> *w, int count);`	`void window_decr_v4(<output_window_type> *w, int count);`
用于减少当前读/写位置，减少的值为底层窗口类型的计数次数的 8 倍。	`void window_decr_v8(<input_window_type> *w, int count);`	`void window_decr_v8(<output_window_type> *w, int count);`
用于减少当前读/写位置，减少的值为底层窗口类型的计数次数的 16 倍。	`void window_decr_v16(<input_window_type> *w, int count);`	`void window_decr_v16(<output_window_type> *w, int count);`
用于减少当前读/写位置，减少的值为底层窗口类型的计数次数的 32 倍。	`void window_decr_v32(<input_window_type> *w, int count);`	`void window_decr_v32(<output_window_type> *w, int count);`
用于减少当前读/写位置，减少的值为底层窗口类型的计数次数的 64 倍。	`void window_decr_v64(<input_window_type> *w, int count);`	`void window_decr_v64(<output_window_type> *w, int count);`

从输入窗口读取数据

以下代码用于从特定类型的输入窗口中读取相同类型的标量值。不修改当前位置，同时提供了功能形式（返回值）和过程形式（修改参考实参）。

```
int8 window_read(input_window_int8 *w);
int16 window_read(input_window_int16 *w);
int32 window_read(input_window_int32 *w);
int64 window_read(input_window_int64 *w);
uint8 window_read(input_window_uint8 *w);
uint16 window_read(input_window_uint16 *w);
uint32 window_read(input_window_uint32 *w);
uint64 window_read(input_window_uint64 *w);
cint16 window_read(input_window_cint16 *w);
cint32 window_read(input_window_cint32 *w);
float window_read(input_window_float *w);
cfloat window_read(input_window_cfloat *w);

void window_read(input_window_int8 *w, int8 &v );
void window_read(input_window_int16 *w, int16 &v );
void window_read(input_window_int32 *w, int32 &v );
void window_read(input_window_int64 *w, int64 &v );
void window_read(input_window_uint8 *w, uint8 &v );
void window_read(input_window_uint16 *w, uint16 &v );
void window_read(input_window_uint32 *w, uint32 &v );
void window_read(input_window_uint64 *w, uint64 &v );
void window_read(input_window_cint16 *w, cint16 &v);
void window_read(input_window_cint32 *w, cint32 &v);
void window_read(input_window_float *w, float &v);
void window_read(input_window_cfloat *w, cfloat &v);
```

以下代码用于从特定类型的输入窗口中读取相同类型的 4 路矢量值。不修改当前位置，同时提供了功能形式（返回值）和过程形式（修改参考实参）。对于矢量运算，存储器数据路径位宽为 128 位或 256 位。

```
v4cint16 window_read_v4(input_window_cint16 *w);
v4int32 window_read_v4(input_window_int32 *w);
v4cint32 window_read_v4(input_window_cint32 *w);
v4int64 window_read_v4(input_window_int64 *w);
v4float window_read_v4(input_window_float *w);
v4cfloat window_read_v4(input_window_cfloat *w);

void window_read(input_window_cint16 *w, v4cint16 &v);
void window_read(input_window_int32 *w, v4int32 &v);
void window_read(input_window_cint32 *w, v4cint32 &v);
void window_read(input_window_int64 *w, v4int64 &v);
void window_read(input_window_float *w, v4float &v);
void window_read(input_window_cfloat *w, v4cfloat &v);
```

以下代码用于从特定类型的输入窗口中读取相同类型的 8 路矢量值。不修改当前位置，同时提供了功能形式（返回值）和过程形式（修改参考实参）。对于矢量运算，存储器数据路径位宽为 128 位或 256 位。

```
v8int16 window_read_v8(input_window_int16 *w);
v8cint16 window_read_v8(input_window_cint16 *w);
v8int32 window_read_v8(input_window_int32 *w);
v8float window_read_v8(input_window_float *w);

void window_read(input_window_int16 *w, v8int16 &v);
void window_read(input_window_cint16 *w, v8cint16 &v);
void window_read(input_window_int32 *w, v8int32 &v);
void window_read(input_window_float *w, v8float &v);
```

以下代码用于从特定类型的输入窗口中读取相同类型的 16 路矢量值。不修改当前位置，同时提供了功能形式（返回值）和过程形式（修改参考实参）。对于矢量运算，存储器数据路径位宽为 128 位或 256 位。

```
v16int8 window_read_v16(input_window_int8 *w);
v16uint8 window_read_v16(input_window_uint8 *w);
v16int16 window_read_v16(input_window_int16 *w);
v16cint16 window_read_v16(input_window_cint16 *w);
v16int32 window_read_v16(input_window_int32 *w);
v16cint32 window_read_v16(input_window_cint32 *w);
v16float window_read_v16(input_window_float *w);
v16cfloat window_read_v16(input_window_cfloat *w);

void window_read(input_window_int8 *w, v16int8 &v);
void window_read(input_window_uint8 *w, v16uint8 &v);
void window_read(input_window_int16 *w, v16int16 &v);
void window_read(input_window_cint16 *w, v16cint16 &v);
void window_read(input_window_int32 *w, v16int32 &v);
void window_read(input_window_cint32 *w, v16cint32 &v);
void window_read(input_window_float *w, v16float &v);
void window_read(input_window_cfloat *w, v16cfloat &v);
```

以下代码用于从特定类型的输入窗口中读取相同类型的 32 路矢量值。不修改当前位置，同时提供了功能形式（返回值）和过程形式（修改参考实参）。对于矢量运算，存储器数据路径位宽为 128 位或 256 位。

```
v32int8 window_read_v32(input_window_int8 *w);
v32uint8 window_read_v32(input_window_uint8 *w);
v32int16 window_read_v32(input_window_int16 *w);
v32cint16 window_read_v32(input_window_cint16 *w);
v32int32 window_read_v32(input_window_int32 *w);
v32float window_read_v32(input_window_float *w);

void window_read(input_window_int8 *w, v32int8 &v);
void window_read(input_window_uint8 *w, v32uint8 &v);
void window_read(input_window_int16 *w, v32int16 &v);
void window_read(input_window_cint16 *w, v32cint16 &v);
void window_read(input_window_int32 *w, v32int32 &v);
void window_read(input_window_float *w, v32float &v);
```

以下代码用于从特定类型的输入窗口中读取相同类型的 64 路矢量值。不修改当前位置，同时提供了功能形式（返回值）和过程形式（修改参考实参）。对于矢量运算，存储器数据路径位宽为 128 位或 256 位。

```
v64int8 window_read_v64(input_window_int8 *w);
v64uint8 window_read_v64(input_window_uint8 *w);
v64int16 window_read_v64(input_window_int16 *w);

void window_read(input_window_int8 *w, v64int8 &v);
void window_read(input_window_uint8 *w, v64uint8 &v);
void window_read(input_window_int16 *w, v64int16 &v);
```

读取并递增输入窗口

以下代码会从特定类型的输入窗口读取相同类型的标量值，并递增窗口当前位置，递增值为 1 乘以底层数据类型的大小。其中提供了功能形式（返回值）和过程形式（修改参考实参）。

```
int8 window_readincr(input_window_int8 *w);
int16 window_readincr(input_window_int16 *w);
int32 window_readincr(input_window_int32 *w);
int64 window_readincr(input_window_int64 *w);
uint8 window_readincr(input_window_uint8 *w);
uint16 window_readincr(input_window_uint16 *w);
uint32 window_readincr(input_window_uint32 *w);
uint64 window_readincr(input_window_uint64 *w);
cint16 window_readincr(input_window_cint16 *w);
cint32 window_readincr(input_window_cint32 *w);
float window_readincr(input_window_float *w);
cfloat window_readincr(input_window_cfloat *w);

void window_readincr(input_window_int8 *w, int8 &v );
void window_readincr(input_window_int16 *w, int16 &v );
void window_readincr(input_window_int32 *w, int32 &v );
void window_readincr(input_window_int64 *w, int64 &v );
void window_readincr(input_window_uint8 *w, uint8 &v );
void window_readincr(input_window_uint16 *w, uint16 &v );
void window_readincr(input_window_uint32 *w, uint32 &v );
void window_readincr(input_window_uint64 *w, uint64 &v );
void window_readincr(input_window_cint16 *w, cint16 &v);
void window_readincr(input_window_cint32 *w, cint32 &v);
void window_readincr(input_window_float *w, float &v );
void window_readincr(input_window_cfloat *w, cfloat &v);
```

以下代码会从特定类型的输入窗口读取相同类型的 4 路矢量值，并递增窗口当前位置，递增值为 4 乘以底层数据类型的大小。其中提供了功能形式（返回值）和过程形式（修改参考实参）。对于矢量运算，存储器数据路径位宽为 128 位或 256 位。

```
v4cint16 window_readincr_v4(input_window_cint16 *w);
v4int32 window_readincr_v4(input_window_int32 *w);
v4cint32 window_readincr_v4(input_window_cint32 *w);
v4int64 window_readincr_v4(input_window_int64 *w);
v4float window_readincr_v4(input_window_float *w);
v4cfloat window_readincr_v4(input_window_cfloat *w);

void window_readincr(input_window_cint16 *w, v4cint16 &v);
void window_readincr(input_window_int32 *w, v4int32 &v);
void window_readincr(input_window_cint32 *w, v4cint32 &v);
void window_readincr(input_window_int64 *w, v4int64 &v);
void window_readincr(input_window_float *w, v4float &v);
void window_readincr(input_window_cfloat *w, v4cfloat &v);
```

以下代码会从特定类型的输入窗口读取相同类型的 8 路矢量值，并递增窗口当前位置，递增值为 8 乘以底层数据类型的大小。其中提供了功能形式（返回值）和过程形式（修改参考实参）。对于矢量运算，存储器数据路径位宽为 128 位或 256 位。

```
v8int16 window_readincr_v8(input_window_int16 *w);
v8cint16 window_readincr_v8(input_window_cint16 *w);
v8int32 window_readincr_v8(input_window_int32 *w);
v8float window_readincr_v8(input_window_float *w);

void window_readincr(input_window_int16 *w, v8int16 &v);
void window_readincr(input_window_cint16 *w, v8cint16 &v);
void window_readincr(input_window_int32 *w, v8int32 &v);
void window_readincr(input_window_float *w, v8float &v);
```

以下代码会从特定类型的输入窗口读取相同类型的 16 路矢量值，并递增窗口当前位置，递增值为 16 乘以底层数据类型的大小。其中提供了功能形式（返回值）和过程形式（修改参考实参）。对于矢量运算，存储器数据路径位宽为 128 位或 256 位。

```
v16int8 window_readincr_v16(input_window_int8 *w);
v16uint8 window_readincr_v16(input_window_uint8 *w);
v16int16 window_readincr_v16(input_window_int16 *w);
v16cint16 window_readincr_v16(input_window_cint16 *w);
v16int32 window_readincr_v16(input_window_int32 *w);
v16cint32 window_readincr_v16(input_window_cint32 *w);
v16float window_readincr_v16(input_window_float *w);
v16cfloat window_readincr_v16(input_window_cfloat *w);

void window_readincr(input_window_int8 *w, v16int8 &v);
void window_readincr(input_window_uint8 *w, v16uint8 &v);
void window_readincr(input_window_int16 *w, v16int16 &v);
void window_readincr(input_window_cint16 *w, v16cint16 &v);
void window_readincr(input_window_int32 *w, v16int32 &v);
void window_readincr(input_window_cint32 *w, v16cint32 &v);
void window_readincr(input_window_float *w, v16float &v);
void window_readincr(input_window_cfloat *w, v16cfloat &v);
```

以下代码会从特定类型的输入窗口读取相同类型的 32 路矢量值，并递增窗口当前位置，递增值为 32 乘以底层数据类型的大小。其中提供了功能形式（返回值）和过程形式（修改参考实参）。对于矢量运算，存储器数据路径位宽为 128 位或 256 位。

```
v32int8 window_readincr_v32(input_window_int8 *w);
v32uint8 window_readincr_v32(input_window_uint8 *w);
v32int16 window_readincr_v32(input_window_int16 *w);
v32cint16 window_readincr_v32(input_window_cint16 *w);
v32int32 window_readincr_v32(input_window_int32 *w);
v32float window_readincr_v32(input_window_float *w);

void window_readincr(input_window_int8 *w, v32int8 &v);
void window_readincr(input_window_uint8 *w, v32uint8 &v);
void window_readincr(input_window_int16 *w, v32int16 &v);
void window_readincr(input_window_cint16 *w, v32cint16 &v);
void window_readincr(input_window_int32 *w, v32int32 &v);
void window_readincr(input_window_float *w, v32float &v);
```

以下代码会从特定类型的输入窗口读取相同类型的 64 路矢量值，并递增窗口当前位置，递增值为 64 乘以底层数据类型的大小。其中提供了功能形式（返回值）和过程形式（修改参考实参）。对于矢量运算，存储器数据路径位宽为 128 位或 256 位。

```
v64int8 window_readincr_v64(input_window_int8 *w);
v64uint8 window_readincr_v64(input_window_uint8 *w);
v64int16 window_readincr_v64(input_window_int16 *w);

void window_readincr(input_window_int8 *w, v64int8 &v);
void window_readincr(input_window_uint8 *w, v64uint8 &v);
void window_readincr(input_window_int16 *w, v64int16 &v);
```

读取并递减输入窗口

以下代码会从特定类型的输入窗口读取相同类型的标量值，并递减窗口当前位置，递减值为 1 乘以底层数据类型的大小。其中提供了功能形式（返回值）和过程形式（修改参考实参）。

```
int8 window_readdecr(input_window_int8 *w);
int16 window_readdecr(input_window_int16 *w);
int32 window_readdecr(input_window_int32 *w);
int64 window_readdecr(input_window_int64 *w);
uint8 window_readdecr(input_window_uint8 *w);
uint16 window_readdecr(input_window_uint16 *w);
uint32 window_readdecr(input_window_uint32 *w);
uint64 window_readdecr(input_window_uint64 *w);
cint16 window_readdecr(input_window_cint16 *w);
cint32 window_readdecr(input_window_cint32 *w);
float window_readdecr(input_window_float *w);
cfloat window_readdecr(input_window_cfloat *w);

void window_readdecr(input_window_int8 *w, int8 &v );
void window_readdecr(input_window_int16 *w, int16 &v );
void window_readdecr(input_window_int32 *w, int32 &v );
void window_readdecr(input_window_int64 *w, int64 &v );
void window_readdecr(input_window_uint8 *w, uint8 &v );
void window_readdecr(input_window_uint16 *w, uint16 &v );
void window_readdecr(input_window_uint32 *w, uint32 &v );
void window_readdecr(input_window_uint64 *w, uint64 &v );
void window_readdecr(input_window_cint16 *w, cint16 &v);
void window_readdecr(input_window_cint32 *w, cint32 &v);
void window_readdecr(input_window_float *w, float &v );
void window_readdecr(input_window_cfloat *w, cfloat &v);
```

以下代码会从特定类型的输入窗口读取相同类型的 4 路矢量值，并递减窗口当前位置，递减值为 4 乘以底层数据类型的大小。其中提供了功能形式（返回值）和过程形式（修改参考实参）。对于矢量运算，存储器数据路径位宽为 128 位或 256 位。

```
v4cint16 window_readdecr_v4(input_window_cint16 *w);
v4int32 window_readdecr_v4(input_window_int32 *w);
v4cint32 window_readdecr_v4(input_window_cint32 *w);
v4int64 window_readdecr_v4(input_window_int64 *w);
v4float window_readdecr_v4(input_window_float *w);
v4cfloat window_readdecr_v4(input_window_cfloat *w);

void window_readdecr(input_window_cint16 *w, v4cint16 &v);
```

```
void window_readdecr(input_window_int32 *w, v4int32 &v);
void window_readdecr(input_window_cint32 *w, v4cint32 &v);
void window_readdecr(input_window_int64 *w, v4int64 &v);void
window_readdecr(input_window_float *w, v4float &v);
void window_readdecr(input_window_cfloat *w, v4cfloat &v);
```

以下代码会从特定类型的输入窗口读取相同类型的 8 路矢量值，并递减窗口当前位置，递减值为 8 乘以底层数据类型的大小。其中提供了功能形式（返回值）和过程形式（修改参考实参）。对于矢量运算，存储器数据路径位宽为 128 位或 256 位。

```
v8int16 window_readdecr_v8(input_window_int16 *w);
v8cint16 window_readdecr_v8(input_window_cint16 *w);
v8int32 window_readdecr_v8(input_window_int32 *w);
v8float window_readdecr_v8(input_window_float *w);

void window_readdecr(input_window_int16 *w, v8int16 &v);
void window_readdecr(input_window_cint16 *w, v8cint16 &v);
void window_readdecr(input_window_int32 *w, v8int32 &v);
void window_readdecr(input_window_float *w, v8float &v);
```

以下代码会从特定类型的输入窗口读取相同类型的 16 路矢量值，并递减窗口当前位置，递减值为 16 乘以底层数据类型的大小。其中提供了功能形式（返回值）和过程形式（修改参考实参）。对于矢量运算，存储器数据路径位宽为 128 位或 256 位。

```
v16int8 window_readdecr_v16(input_window_int8 *w);
v16uint8 window_readdecr_v16(input_window_uint8 *w);
v16int16 window_readdecr_v16(input_window_int16 *w);
v16cint16 window_readdecr_v16(input_window_cint16 *w);
v16int32 window_readdecr_v16(input_window_int32 *w);
v16cint32 window_readdecr_v16(input_window_cint32 *w);
v16float window_readdecr_v16(input_window_float *w);
v16cfloat window_readdecr_v16(input_window_cfloat *w);

void window_readdecr(input_window_int8 *w, v16int8 &v);
void window_readdecr(input_window_uint8 *w, v16uint8 &v);
void window_readdecr(input_window_int16 *w, v16int16 &v);
void window_readdecr(input_window_cint16 *w, v16cint16 &v);
void window_readdecr(input_window_int32 *w, v16int32 &v);
void window_readdecr(input_window_cint32 *w, v16cint32 &v);
void window_readdecr(input_window_float *w, v16float &v);
void window_readdecr(input_window_cfloat *w, v16cfloat &v);
```

以下代码会从特定类型的输入窗口读取相同类型的 32 路矢量值，并递减窗口当前位置，递减值为 32 乘以底层数据类型的大小。其中提供了功能形式（返回值）和过程形式（修改参考实参）。对于矢量运算，存储器数据路径位宽为 128 位或 256 位。

```
v32int8 window_readdecr_v32(input_window_int8 *w);
v32uint8 window_readdecr_v32(input_window_uint8 *w);
v32int16 window_readdecr_v32(input_window_int16 *w);
v32cint16 window_readdecr_v32(input_window_cint16 *w);
v32int32 window_readdecr_v32(input_window_int32 *w);
v32float window_readdecr_v32(input_window_float *w);

void window_readdecr(input_window_int8 *w, v32int8 &v);
void window_readdecr(input_window_uint8 *w, v32uint8 &v);
void window_readdecr(input_window_int16 *w, v32int16 &v);
void window_readdecr(input_window_cint16 *w, v32cint16 &v);
void window_readdecr(input_window_int32 *w, v32int32 &v);
void window_readdecr(input_window_float *w, v32float &v);
```

以下代码会从特定类型的输入窗口读取相同类型的 64 路矢量值，并递减窗口当前位置，递减值为 64 乘以底层数据类型的大小。其中提供了功能形式（返回值）和过程形式（修改参考实参）。对于矢量运算，存储器数据路径位宽为 128 位或 256 位。

```
v64int8 window_readdecr_v64(input_window_int8 *w);
v64uint8 window_readdecr_v64(input_window_uint8 *w);
v64int16 window_readdecr_v64(input_window_int16 *w);

void window_readdecr(input_window_int8 *w, v64int8 &v);
void window_readdecr(input_window_uint8 *w, v64uint8 &v);
void window_readdecr(input_window_int16 *w, v64int16 &v);
```

将数据写入输出窗口

以下代码用于将特定类型的标量值写入相同类型的输出窗口。不修改当前位置。

```
void window_write(output_window_int8 *w, int8 v);
void window_write(output_window_uint8 *w, uint8 v);
void window_write(output_window_int16 *w, int16 v);
void window_write(output_window_uint16 *w, uint16 v);
void window_write(output_window_cint16 *w, cint16 v);
void window_write(output_window_int32 *w, int32 v );
void window_write(output_window_uint32 *w, uint32 v );
void window_write(output_window_cint32 *w, cint32 v);
void window_write(output_window_int64 *w, int64 v );
void window_write(output_window_uint64 *w, uint64 v );
void window_write(output_window_float *w, float v );
void window_write(output_window_cfloat *w, cfloat v);
```

以下代码用于将特定类型的 4 路矢量值写入相同类型的输出窗口。不修改当前位置。

```
void window_write(output_window_cint16 *w, v4cint16 v);
void window_write(output_window_int32 *w, v4int32 v );
void window_write(output_window_cint32 *w, v4cint32 v);
void window_write(output_window_int64 *w, v4int64 v );
void window_write(output_window_float *w, v4float v );
void window_write(output_window_cfloat *w, v4cfloat v);
```

以下代码用于将特定类型的 8 路矢量值写入相同类型的输出窗口。不修改当前位置。

```
void window_write(output_window_int16 *w, v8int16 v);
void window_write(output_window_cint16 *w, v8cint16 v);
void window_write(output_window_int32 *w, v8int32 v );
void window_write(output_window_float *w, v8float v );
```

以下代码用于将特定类型的 16 路矢量值写入相同类型的输出窗口。不修改当前位置。

```
void window_write(output_window_int8 *w, v16int8 v);
void window_write(output_window_uint8 *w, v16uint8 v);
void window_write(output_window_int16 *w, v16int16 v);
void window_write(output_window_cint16 *w, v16cint16 v);
void window_write(output_window_int32 *w, v16int32 v );
void window_write(output_window_cint32 *w, v16cint32 v);
void window_write(output_window_float *w, v16float v );
void window_write(output_window_cfloat *w, v16cfloat v);
```

以下代码用于将特定类型的 32 路矢量值写入相同类型的输出窗口。不修改当前位置。

```
void window_write(output_window_int8 *w, v32int8 v);
void window_write(output_window_uint8 *w, v32uint8 v);
void window_write(output_window_int16 *w, v32int16 v);
void window_write(output_window_cint16 *w, v32cint16 v);
void window_write(output_window_int32 *w, v32int32 v );
void window_write(output_window_float *w, v32float v );
```

以下代码用于将特定类型的 64 路矢量值写入相同类型的输出窗口。不修改当前位置。

```
void window_write(output_window_int8 *w, v64int8 v);
void window_write(output_window_uint8 *w, v64uint8 v);
void window_write(output_window_int16 *w, v64int16 v);
```

写入并递增输出窗口

以下代码会将特定类型的标量值写入相同类型的输出窗口，并基于该类型递增当前位置。

```
void window_writeincr (output_window_int8 *w, int8 v);
void window_writeincr (output_window_uint8 *w, uint8 v);
void window_writeincr (output_window_int16 *w, int16 v);
void window_writeincr (output_window_uint16 *w, uint16 v);
void window_writeincr (output_window_cint16 *w, cint16 v);
void window_writeincr (output_window_int32 *w, int32 v );
void window_writeincr (output_window_uint32 *w, uint32 v );
void window_writeincr (output_window_cint32 *w, cint32 v);
void window_writeincr (output_window_int64 *w, int64 v);
void window_writeincr (output_window_uint64 *w, uint64 v);void
window_writeincr (output_window_float *w, float v );
void window_writeincr (output_window_cfloat *w, cfloat v);
```

以下代码会将特定类型的 4 路矢量值写入相同类型的输出窗口，并递增当前位置，递增值为 4 乘以底层类型的大小。

```
void window_writeincr(output_window_cint16 *w, v4cint16 v);
void window_writeincr(output_window_int32 *w, v4int32 v );
void window_writeincr(output_window_cint32 *w, v4cint32 v);
void window_writeincr(output_window_int64 *w, v4int64 v );
void window_writeincr(output_window_float *w, v4float v );
void window_writeincr(output_window_cfloat *w, v4cfloat v);
```

以下代码会将特定类型的 8 路矢量值写入相同类型的输出窗口，并递增当前位置，递增值为 8 乘以底层类型的大小。

```
void window_writeincr(output_window_int16 *w, v8int16 v);
void window_writeincr(output_window_cint16 *w, v8cint16 v);
void window_writeincr(output_window_int32 *w, v8int32 v );
void window_writeincr(output_window_float *w, v8float v );
```

以下代码会将特定类型的 16 路矢量值写入相同类型的输出窗口，并递增当前位置，递增值为 16 乘以底层类型的大小。

```
void window_writeincr(output_window_int8 *w, v16int8 v);
void window_writeincr(output_window_uint8 *w, v16uint8 v);
void window_writeincr(output_window_int16 *w, v16int16 v);
void window_writeincr(output_window_cint16 *w, v16cint16 v);
void window_writeincr(output_window_int32 *w, v16int32 v );
void window_writeincr(output_window_cint32 *w, v16cint32 v);
void window_writeincr(output_window_float *w, v16float v );
void window_writeincr(output_window_cfloat *w, v16cfloat v);
```

以下代码会将特定类型的 32 路矢量值写入相同类型的输出窗口，并递增当前位置，递增值为 32 乘以底层类型的大小。

```
void window_writeincr(output_window_int8 *w, v32int8 v);
void window_writeincr(output_window_uint8 *w, v32uint8 v);
void window_writeincr(output_window_int16 *w, v32int16 v);
void window_writeincr(output_window_cint16 *w, v32cint16 v);
void window_writeincr(output_window_int32 *w, v32int32 v );
void window_writeincr(output_window_float *w, v32float v );
```

以下代码会将特定类型的 64 路矢量值写入相同类型的输出窗口，并递增当前位置，递增值为 64 乘以底层类型的大小。

```
void window_writeincr(output_window_int8 *w, v64int8 v);
void window_writeincr(output_window_uint8 *w, v64uint8 v);
void window_writeincr(output_window_int16 *w, v64int16 v);
```

内核串流运算

串流数据类型

表 102：**受支持的串流数据类型**

输入串流类型	输出串流类型
input_stream_int8	output_stream_int8
input_stream_int16	output_stream_int16
input_stream_int32	output_stream_int32
input_stream_int64	output_stream_int64
input_stream_uint8	output_stream_uint8
input_stream_uint16	output_stream_uint16
input_stream_uint32	output_stream_uint32
input_stream_uint64	output_stream_uint64
input_stream_cint16	output_stream_cint16
input_stream_cint32	output_stream_cint32
input_stream_acc48	output_stream_acc48
input_stream_cacc48	output_stream_cacc48

表 102：**受支持的串流数据类型** (续)

输入串流类型	输出串流类型
input_stream_acc80	output_stream_acc80
input_stream_cacc80	output_stream_cacc80
input_stream_accfloat	output_stream_accfloat
input_stream_caccfloat	output_stream_caccfloat
input_stream_float	output_stream_float
input_stream_cfloat	output_stream_cfloat

表中每一种数据类型均可作为标量或者以矢量组形式从 AI 引擎进行读取或写入。但基于 AI 引擎到可编程逻辑接口端口上或者穿过串流开关网络时所支持的总线数据宽度，对于有效分组存在某些限制。AI 引擎内核的有效组合是总计最高 32 位或 128 位的矢量捆绑。累加器数据类型仅用于指定相邻 AI 引擎之间的级联串流连接。其有效分组基于两个处理器之间的 384 位宽的级联通道。

注释：要使用这些数据类型，需要在内核源文件中使用 `#include <adf.h>`。

读取并递增输入串流

AI 引擎运算

以下运算会从给定输入串流读取数据，并在 AI 引擎上递增串流。由于在 AI 引擎上有 2 个输入串流端口，因此，物理端口分配由 AI 引擎编译器自动完成，并作为串流数据结构的一部分进行传递。数据值只能从串流中逐一读取或者作为矢量来读取。对于后者，除非所有值都存在，否则串流运算会停滞。数据基于底层单周期 32 位串流运算或者 4 周期 128 位宽串流运算来进行分组。级联连接会并行读取所有累加器值。

```
int32 readincr(input_stream_int32 *w);
uint32 readincr(input_stream_uint32 *w);
cint16 readincr(input_stream_cint16 *w);
float readincr(input_stream_float *w);
cfloat readincr(input_stream_cfloat *w);

v16int8 readincr_v16(input_stream_int8 *w);
v16uint8 readincr_v16(input_stream_uint8 *w);
v8int16 readincr_v8(input_stream_int16 *w);
v4cint16 readincr_v4(input_stream_cint16 *w);
v4int32 readincr_v4(input_stream_int32 *w);
v2cint32 readincr_v2(input_stream_cint32 *w);
v4float readincr_v4(input_stream_float *w);

v8acc48 readincr_v8(input_stream_acc48 *w);
v4cacc48 readincr_v4(input_stream_cacc48 *w);
v4acc80 readincr_v4(input_stream_acc80 * str);
v2cacc80 readincr_v2(input_stream_cacc80 * str);
v8float readincr_v8(input_stream_accfloat * str);
v4cfloat readincr_v4(input_stream_caccfloat * str);
```

写入并递增输出串流

AI 引擎运算

以下运算会向给定输出串流写入数据，并在 AI 引擎上递增串流。由于在 AI 引擎上有 2 个输出串流端口，因此，物理端口分配由 AI 引擎编译器自动完成，并作为串流数据结构的一部分进行传递。数据值可逐一写入输出串流或者作为矢量来写入输出串流。对于后者，仅当所有值都已写入后，串流运算才会停滞。数据基于底层单周期 32 位串流运算或者 4 周期 128 位宽串流运算来进行分组。级联连接会并行写入所有值。

```
void writeincr(output_stream_int32 *w, int32 v);
void writeincr(output_stream_uint32 *w, uint32 v);
void writeincr(output_stream_cint16 *w, cint16 v);
void writeincr(output_stream_float *w, float v);
void writeincr(output_stream_cfloat *w, cfloat v);

void writeincr_v16(output_stream_int8 *w, v16int8 v);
void writeincr_v16(output_stream_uint8 *w, v16uint8 v);
void writeincr_v8(output_stream_int16 *w, v8int16 v);
void writeincr_v4(output_stream_cint16 *w, v4cint16 v);
void writeincr_v4(output_stream_int32 *w, v4int32 v);
void writeincr_v2(output_stream_cint32 *w, v2cint32 v);
void writeincr_v4(output_stream_float *w, v4float v);

void writeincr_v8(output_stream_acc48 *w, v8acc48 v);
void writeincr_v4(output_stream_cacc48 *w, v4cacc48 v);
void writeincr_v4(output_stream_acc80* str, v4acc80 value);
void writeincr_v2(output_stream_cacc80* str, v2cacc80 value);
void writeincr_v8(output_stream_accfloat* str, v8float value);
void writeincr_v4(output_stream_caccfloat* str, v4cfloat value);
```

并行使用串流

对于串流输入和输出接口，如果性能受到串流接口的限制，那么可以并行使用双串流输入或双串流输出。要使用双并行串流，建议使用以下成对宏，其中 `idx1` 和 `idx2` 表示两条串流。将 `restrict` 关键字添加到串流端口中，对其进行最优化以便于并行处理。

```
READINCR(SS_rsrc1, idx1) and READINCR(SS_rsrc2, idx2)
READINCRW(WSS_rsrc1, idx1) and READINCRW(WSS_rsrc2, idx2)
WRITEINCR(MS_rsrc1, idx1, val) and WRITEINCR(MS_rsrc2, idx2, val)
WRITEINCRW(WMS_rsrc1, idx1, val) and WRITEINCRW(WMS_rsrc2, idx2, val)
```

以下代码示例显示了两条使用流水打拍的并行输入串流，间隔为 1。

```
void simple( input_stream_int32 * restrict data0, input_stream_int32 *
restrict data1,
    output_stream_int32 * restrict out) {
   for(int i=0; i<1024; i++)
   chess_prepare_for_pipelining
 {
     int32_t d = READINCR(SS_rsrc1, data0) ;
     int32_t e = READINCR(SS_rsrc2, data1) ;
     WRITEINCR(MS_rsrc1,out,d+e);
 }
}
```

AMD XILINX

附录 D

AI 引擎内核加密

AI 引擎源代码的加密方法论有助于 IP 开发者以更安全的方式向最终用户交付 IP 源代码。该功能特性并非以明文交付 AI 引擎源文件，并且并不会在编译期间公开任何内核源代码内容。

此功能特性基于 IP 作者的顾虑，以信任模型为目标。由 IP 作者判断其受保护的 IP 的查看和使用方式。此信任模型总结如下：源代码加密是 IP 作者的强制要求，IP 作者希望自己的 IP 受到保护。工具默认行为是在该工具完成其目标的同时，尽可能提供合理范围内最大限度的保护。

AI 引擎源代码加密功能特性可从专区网站获取：https://china.xilinx.com/member/ai_engine_encryption.html（需注册）。专区内提供了许可证和 Synopsys 密钥，这些都是对 AI 引擎源代码进行加密和解密所必需的。此外还提供了支持文档以供使用。

AMD XILINX

附录 E

附加资源与法律声明

赛灵思资源

如需获取答复记录、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。要打开 DocNav，请执行以下操作：

- 在 Vivado® IDE 中，单击“Help” → “Documentation and Tutorials”。
- 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它主题整理的文档链接，可供您用于了解关键概念以及常见问题解答。要访问设计中心，请执行以下操作：

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

注释：如需了解有关 DocNav 的更多信息，请参阅赛灵思网站上的 Documentation Navigotor 页面。

参考资料

以下技术文档是非常实用的补充资料，可配合本指南一起使用：

1. 《Versal ACAP AI 引擎架构手册》(AM009)
2. Versal ACAP 数据手册：
   - 《Versal 架构和产品数据手册概述》(DS950)
   - 《Versal AI Core 系列数据手册：DC 和 AC 开关特性》(DS957)
3. 《Vitis 统一软件平台文档》
4. 《Versal ACAP AI 引擎内部函数文档》(UG1078)

5. 《Versal ACAP AI 引擎寄存器参考资料》(AM015)
6. 《Versal ACAP 设计指南》(UG1273)
7. 《AI 引擎 API 用户指南》(UG1529)
8. 答复记录：
   - AI 引擎编译器 - Vitis 2020.2 和更高版本工具的通用指南和已知问题（答复记录 75790）
   - AI 引擎仿真器 - Vitis 2020.2 和更高版本工具的通用指南和已知问题（答复记录 75788）
   - 赛灵思 AI 引擎解决方案中心（答复记录 75837）

修订历史

下表列出了本文档的修订历史。

章节	修订综述
2022 年 10 月 5 日 2022.2 版	
文档标题	标题更改为《AI 引擎工具和流程用户指南》。
整个文档	将 graph 编程内容移入《AI 引擎内核与 Graph 编程指南》(UG1079)。
AI 引擎编译器选项	添加 DRC 选项。更新事件追踪选项。
仿真输入和输出数据串流	添加仿真选项文件详细信息。
启用第三方仿真器	将 v++ --link 配置更新为使用通用 <SIMULATOR DIRECTORY> 变量。
标量 RTP 数据分析	新增主题。
graph 时延剖析	为硬件流程和硬件仿真流程添加代码示例。
AI 引擎剖析	移除 `stream_put_get` 指标。
XSDB 流程	添加 XSDB 追踪选项。
限制	新增主题。
存储器模型	添加有关 `X86SIM_THREAD_LOCAL` 宏如何使全局读/写线程安全的说明。
接口拼块剖析	更新 `input_bandwidths` 和 `output_bandwidths` 指标。新增 `packets` 指标。
XSDB 流程	更新合添加新的 `aieprofile` 选项。
XRT 流程 和 XRT 流程	全文更新 xrt.ini 选项和示例。
在硬件中对事件追踪进行故障排除	更新在某些串流中丢弃追踪包时可用的故障诊断选项。
控制 AI 引擎 GMIO 传输	新增主题。
迭代 AI 引擎应用编译	新增主题。
查看来自缓冲器端口接口的数据	新增主题。
系统链接	在 Connectivity 节的选项中添加 `sp`
在硬件仿真中分析 AI 引擎状态	新增主题，用于描述 Vitis 分析器针对硬件仿真中的 AI 引擎剖析和状态的支持。
为 x86 处理器编译嵌入式应用	新增主题，用于描述针对软件仿真中的 PS on x86 流程的支持。

章节	修订综述
使用 System Verilog/Verilog 创建流量生成器	新增主题，用于描述如何使用 Verilog 或 System Verilog 模块/测试激励文件来驱动流量进出 AI 引擎仿真器中运行的 ADF graph。
附录 D: AI 引擎内核加密	新增附录，其中引用了“加密”Web 专区以提供更多详细信息。
2022 年 5 月 25 日 2022.1 版	
AI 引擎编译器选项	添加有关两个保留码字的注释：`aie` 和 `adf` 在 graph 编程中均为无效的名称空间标识符
在 Vitis 分析器中查看编译结果	更新 2022.1 版的 graph。
可编程逻辑 (PL)、信号线 和 拼块	更新 2022.1 版的示例详细信息以及 Vitis 分析器 GUI 截屏。
接口通道	添加新的 graph 接口通道详细信息。
用于 graph 输入和输出的事件剖析 API	添加有关信号线上的事件的详细信息。
使用 Vitis 分析器查看指南	添加本指南。
生成流量用于软硬件仿真	澄清除 C++ 和 Python 外，您也可在 HDL 中写入外部流量生成器。
2022 年 4 月 26 日 2022.1 版	
第 1 章：概述	更新 2022.1 版的 Vitis 核开发套件详细信息。
创建数据流 graph（包含内核）	更新 2022.1 编程模型。
同步窗口访问	添加章节，用于解释窗口到窗口的广播以及多重速率设计支持。
基于串流的访问	添加章节，用于解释使用级联串流执行基于串流的访问的方式。
并行使用串流	更新 32 位宏和 64 位宏的列表。
运行时参数支持汇总	移除不再受支持的 AI 引擎到 AI 引擎运行时间参数构造。
多播支持	更新多播支持场景表。
使用 RTL 编程逻辑的设计流程	更新示例，用于反映 2022.1 编程模型更改。
第 12 章：graph 编程模型	更新整章内的编程模型详细信息和示例。
AI 引擎编译器选项	添加新的多重速率选项。 更新 `--Xrouter=<string>` 示例。
映射器和布线器选项	移除 `enableSplitAsBroadcast` 选项（始终开启）。 添加 `disablePathBalancing` 选项。
x86 功能仿真器	添加在 Vitis 分析器中可视化 X86 仿真输出的功能。
设计编译	添加 X86 仿真器选项。
数据快照	添加在 Vitis 分析器中可视化快照的功能。
限制	移除“仿真输出文件处理注意事项”和“adf::headers 约束和 aie_api 包含文件”章节，因为这些 x86 仿真限制已得到解决。
仿真器选项	添加新选项和挂起检测详细信息。
启用第三方仿真器	更新 VCS 详细信息。 添加 Riviera 仿真器信息。
剖析硬件中的 AI 引擎 接口拼块剖析 剖析 AI 引擎、存储器模块和接口拼块	添加用于 DMA write/read_bandwidths 的事件。 添加剖析接口事件的功能。
Vitis 分析器中的 FIFO 深度可视化	添加可视化 Vitis 分析器中的 DMA FIFO 深度（来自仿真 VCD 数据）的功能。
XSDB 流程 XRT 流程	添加指定事件追踪开始时间的功能。 添加定期卸载追踪数据的功能。
使用 Vitis 分析器查看剖析结果	添加接口指标示例。 添加在 Vitis 分析器中整合多个剖析结果的功能。

章节	修订综述
分析硬件中的 AI 引擎状态 生成 AI 引擎状态 在 Vitis 分析器中分析 AI 引擎状态	添加报告并输出硬件中的 AI 引擎状态的功能，并添加在 Vitis 分析器中打开和分析该报告的功能。
以 DFX 平台为目标	添加了除基础平台外还可使用 DFX 平台的功能，并添加了有关在硬件中使用此平台的信息。
用于控制 AI 引擎 graph 的多进程和多线程支持	添加有关 `xrtGraphClose` 和 `xrtDeviceClose` 行为的澄清信息。
平台	更新“平台类型”，以包含 DFX 平台。
性能指标	添加“Show Percentage”按钮描述。
锁定停滞分析 串流停滞分析 级联停滞分析 存储器停滞分析	添加程序计数器 (PC) 选项，该选项允许您在 Vitis 分析器中对来自“Trace”视图的源代码进行交叉探测。
生成流量用于软硬件仿真 以 Python 和 C++ 创建流量生成器	添加在 x86 功能仿真器、AI 引擎仿真器、软件仿真和硬件仿真中使用流量生成器的支持。这些流量生成器可采用 Python、C++ 或 HDL 来编写。
裸机的主机编程 构建裸机系统 Linux 与裸机之间的主机编程支持比较	添加有关裸机软件栈的详细信息。 对在裸机中与在 Linux 操作系统中运行主机应用的功能进行比对。
第 8 章：使用 Vitis 工具流程来集成应用 系统链接 为硬件封装系统	更新以反应 v++ 链接现在可生成 XSA 文件的事实。
第 12 章：AI 引擎硬件剖析和调试方法论	新增有关 AI 引擎硬件剖析和调试方法论的章节。
input_gmio/output_gmio input_plio/output_plio	更新文档，以反映编程模型的更改，包括 `input_gmio`/`output_gmio` 和 `input_plio`/`output_plio`。
其它约束	为多重速率设计添加 `repetition_count` 约束。
2021 年 12 月 17 日 2021.2 版	
第 8 章：窗口和串流数据 API	添加更多受支持的无符号整数数据类型。
指定运行时数据参数	澄清描述。
模型功能特性编程	更改章节标题。
AI 引擎编译器选项	新增 表 12：最优化选项
AI 引擎剖析	新增章节。
graph 吞吐量剖析	添加信息。
剖析硬件中的 AI 引擎	新增章节。
硬件中的事件追踪	新增章节。
硬件事件追踪	新增章节。
在硬件中对事件追踪进行故障排除	新增章节。
2021 年 10 月 22 日 2021.2 版	
AI 引擎组件	更新 。
内核准备	更新 AI 引擎 API。
创建数据流 graph（包含内核）	添加表示 graph 连接的图示。
第 8 章：窗口和串流数据 API	更新 AI 引擎 API 和模板支持的数据类型。
包切换 graph 构造	添加浮点数据示例。
面积位置约束	新增章节。

章节	修订综述
分层约束	添加信息。
模型功能特性编程	新增章节。
AI 引擎编译器选项	新增选项。
第 4 章：对 AI 引擎 graph 应用进行仿真	添加仿真流程相关信息。
数据快照	新增章节。
死锁检测	新增章节。
追踪报告	新增章节。
存储器访问违例和 Valgrind	新增章节。
存储器模型	更新信息。
仿真输出文件处理注意事项	新增章节。
adf::headers 约束和 aie_api 包含文件	新增章节。
软件仿真	新增章节。
仿真器选项	新增选项。
硬件仿真	新增章节。
复用 AI 引擎仿真器选项	添加有关设置 AI 引擎编译器 `workdir` 环境变量以及手动创建仿真选项的信息。
基于 AI 引擎仿真的剖析	新增章节。
受支持的窗口数据类型	更新数据类型。
受支持的串流数据类型	
在 Vitis 分析器中执行 AI 引擎停滞分析	新增章节。
用于控制 AI 引擎 graph 的多进程和多线程支持	新增章节。
AI 引擎错误事件	更新错误以及调试技巧。
运行软件仿真	新增章节。
面积分组约束	更新属性。
创建 AI 引擎 graph 工程和顶层系统工程	更新截屏。
构建和运行系统	更新以添加软件仿真。
第 10 章：调试 AI 引擎应用	添加调试信息。
从 Vitis IDE 进行软件仿真调试	新增章节。
从命令行运行软件仿真	新增章节。
使用调试环境	更新截屏。
观察点	新增章节。
用于软件仿真调试的 Vitis IDE 布局	新增章节。
附录 C: 窗口和串流 API 的非模板版本	添加附录，描述窗口和串流数据类型的非模板版本和 API。
2021 年 7 月 19 日 2021.1 版	
FIFO 位置约束	更新 FIFO 约束示例。
受支持的窗口数据类型	新增主题。
受支持的串流数据类型	新增主题。
在 Vitis IDE 中构建裸机 AI 引擎	更新步骤 4。
2021 年 6 月 16 日 2021.1 版	
运行时比率	新增主题。

章节	修订综述
串流数据类型 读取并递增输入串流 写入并递增输出串流	新增串流类型。
运行时参数支持汇总	添加 AI 引擎 RTP 支持表。
串流开关 FIFO DMA FIFO AI 引擎拼块 DMA 性能	新增 FIFO 主题。
包切换 graph 构造	更新允许的包串流数量。
多播支持	新增主题。
第 11 章：AI 引擎/可编程逻辑集成	更新内容。
硬件仿真流程和硬件流程	移除 `ADF_FRONTEND`。
AI 引擎/PL 与 AI 引擎/NoC 接口之间的性能比较	新增主题。
AI 引擎编译器选项	• 更新堆和栈大小。 • 添加 `--broadcast-enable-core` CDO 选项。 • 更新追踪选项。 • 更新 `xlopt`。
graph 和阵列详细信息	新增章节。
AI 引擎编译器指南	新增主题。
复用 AI 引擎仿真器选项	添加 `--profile`/`AIE_PROFILE` 选项。
启用第三方仿真器	添加仿真器并更新版本。
x86 功能仿真器	更新内容，添加新章节。
在 Vitis 分析器中查看运行汇总	更新内容。
追踪视图数据可视化	新增章节。
运行时事件 API 性能计数器使用汇总	新增主题。
第 7 章：PS 主机应用编程	移除 `ADF_FRONTEND`。
利用 XRT C++ API 控制 AI 引擎 graph	新增主题。
通过 XRT API 报告错误	更新 `xbutil` 作用域。
含 ADF API 和 XRT API 的主机代码参考	更新 `printf`。
PL 内核时钟设置	更新主题。
为 Cortex-A72 处理器编译嵌入式应用	更新代码。 • `aarch64-linux-gnu-g++` 改为 `aarch64-xilinx-linux-g++`
运行硬件仿真	新增章节。
第 9 章：使用 Vitis IDE	更新截屏。
第 11 章：映射器/布线器方法论	新增章节。
事件 API	移除额外的“Enumeration”章节。
FIFO 约束	新增主题。
附录 B: 在 AI 引擎内核中使用 restrict 关键字	更新 C++。

请阅读：重要法律声明

本文向贵司/您所提供的信息（下称“资料”）仅在对赛灵思产品进行选择和使用时参考。在适用法律允许的最大范围内：(1) 资料均按“现状”提供，且不保证不存在任何瑕疵，赛灵思在此声明对资料及其状况不作任何保证或担保，无论是明示、暗示还是法定的保证，包括但不限于对适销性、非侵权性或任何特定用途的适用性的保证；且 (2) 赛灵思对任何因资料发生的或与资料有关的（含对资料的使用）任何损失或赔偿（包括任何直接、间接、特殊、附带或连带损失或赔偿，如数据、利润、商誉的损失或任何因第三方行为造成的任何类型的损失或赔偿），均不承担责任，不论该等损失或者赔偿是何种类或性质，也不论是基于合同、侵权、过失或是其它责任认定原理，即便该损失或赔偿可以合理预见或赛灵思事前被告知有发生该损失或赔偿的可能。赛灵思无义务纠正资料中包含的任何错误，也无义务对资料或产品说明书发生的更新进行通知。未经赛灵思公司的事先书面许可，贵司/您不得复制、修改、分发或公开展示本资料。部分产品受赛灵思有限保证条款的约束，请参阅赛灵思销售条款：https://china.xilinx.com/legal.htm#tos；IP 核可能受赛灵思向贵司/您签发的许可证中所包含的保证与支持条款的约束。赛灵思产品并非为故障安全保护目的而设计，也不具备此故障安全保护功能，不能用于任何需要专门故障安全保护性能的用途。如果把赛灵思产品应用于此类特殊用途，贵司/您将自行承担风险和责任。请参阅赛灵思销售条款：https://china.xilinx.com/legal.htm#tos。

关于与汽车相关用途的免责声明

如将汽车产品（部件编号中含“XA”字样）用于部署安全气囊或用于影响车辆控制的应用（“安全应用”），除非有符合 ISO 26262 汽车安全标准的安全概念或冗余特性（“安全设计”），否则不在质保范围内。客户应在使用或分销任何包含产品的系统之前为了安全的目的全面地测试此类系统。在未采用安全设计的条件下将产品用于安全应用的所有风险，由客户自行承担，并且仅在适用的法律法规对产品责任另有规定的情况下，适用该等法律法规的规定。

版权声明