altera
An Intel Company

FPGA AI Suite

Design Examples User Guide

Updated for FPGA AI Suite: **2024.3**

Online Version **848957**
Send Feedback **2025.03.28**

Contents

1. FPGA AI Suite Design Examples User Guide

The *FPGA AI Suite Design Examples User Guide* describes the design and implementation for accelerating AI inference using the FPGA AI Suite, Intel® Distribution of OpenVINO™ toolkit, and various development boards (depending on the design example).

About the FPGA AI Suite Documentation Library

Documentation for the FPGA AI Suite is split across a few publications. Use the following table to find the publication that contains the FPGA AI Suite information that you are looking for:

Table 1. FPGA AI Suite Documentation Library

Title and Description	
Release Notes Provides late-breaking information about the FPGA AI Suite including new features, important bug fixes, and known issues.	Link
Getting Started Guide Get up and running with the FPGA AI Suite by learning how to initialize your compiler environment and reviewing the various design examples and tutorials provided with the FPGA AI Suite	Link
IP Reference Manual Provides an overview of the FPGA AI Suite IP and the parameters you can set to customize it. This document also covers the FPGA AI Suite IP generation utility.	Link
Compiler Reference Manual Describes the use modes of the graph compiler (`dla_compiler`). It also provides details about the compiler command options and the format of compilation inputs and outputs.	Link
Design Examples User Guide Describes the design and implementation for accelerating AI inference using the FPGA AI Suite, Intel Distribution of OpenVINO toolkit, and various development boards (depending on the design example).	Link

2. FPGA AI Suite Design Examples

The *FPGA AI Suite Design Examples User Guide* provides information about the following design examples.

Use the design example symbol to identify the chapters and sections of this document that apply to a design example. Chapters and sections not prefixed with a symbol apply to all design examples.

Table 2. FPGA AI Suite Design Examples Descriptions

Design Example	Description	Symbol
PCIe-based design example	Demonstrates how OpenVINO toolkit and the FPGA AI Suite support the look-aside deep learning acceleration model. This design example targets the Terasic* DE10-Agilex Development Board (DE10-Agilex-B2E2).	[PCIE]
OFS PCIe-attach design example	Demonstrates the OpenVINO toolkit and the FPGA AI Suite that target Open FPGA Stack (OFS)-based boards. This design example targets the following Open FPGA Stack (OFS)-based boards: • Agilex™ 7 FPGA I-Series Development Kit ES2 (DK-DEV-AGI027RBES) • Intel FPGA SmartNIC N6001-PL Platform (without Ethernet controller)	[OFS-PCIE]
Hostless DDR-Free design examples	Demonstrates hostless DDR-free operation of the FPGA AI Suite IP. Graph filters, bias, and FPGA AI Suite IP configurations are stored in internal memory on the FPGA device. This design example targets the Agilex 7 FPGA I-Series Development Kit ES2 (DK-DEV-AGI027RBES).	[HL-NO-DDR]
Hostless JTAG design example	Demonstrates the step-by-step sequence of configuring FPGA AI Suite IP and starting inference by writing into CSRs directly via JTAG. This design example targets the Agilex 5 FPGA E-Series 065B Premium Development Kit (DK-A5E065BB32AES1).	[HL-JTAG]
SoC design example	Demonstrates how OpenVINO toolkit and the FPGA AI Suite support the CPU-offload deep-learning acceleration model in an embedded system. The design example targets the following development boards: • Agilex 7 FPGA I-Series Transceiver-SoC Development Kit (DK-SI-AGI027FC) • Arria® 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S)	[SOC]

The table that follows provides an overview of the some of the main properties of the design examples.

Table 3. FPGA AI Suite Design Examples Properties Overview

*For the **Stream** column, the entries are defined as follows:

M2M FPGA AI Suite runtime software running on the external host transfers the image (or data) to the FPGA DDR memory.

S2M Streaming input data is copied to FPGA on-device memory. The FPGA AI Suite runtime runs on the FPGA device (HPS or RTL state machine). The runtime is used only to coordinate the data transfer from FPGA DDR memory into the FPGA AI Suite IP.

Direct Data is streamed directly in and out of the FPGA on-chip memory.

Example Design Type	Target FPGA Device	Host	Memory	Stream*	Identifier	Supported Development Kit
PCIe Attached	Agilex 7	External host processor	DDR	M2M	agx7_de10_pcie	Terasic DE10-Agilex Development Board (DE10-Agilex-B2E2)
					agx7_iseries_ofs_pcie	Agilex 7 FPGA I-Series Development Kit ES2 (DK-DEV-AGI027RBES)
					agx7_n6001_ofs_pcie	Intel FPGA SmartNIC N6001-PL Platform (without Ethernet controller)
Hostless DDR-Free	Agilex 7	Hostless	DDR-Free	Direct	agx7_iseries_ddrfree	Agilex 7 FPGA I-Series Development Kit ES2 (DK-DEV-AGI027RBES)
Hostless JTAG-Attach	Agilex 5		DDR	M2M	agx5e_modular_Jtag	Agilex 5 FPGA E-Series 065B Premium Development Kit (DK-A5E065BB32AES1)
SoC	Agilex 7	On-device HPS	DDR	M2M and S2M	agx7_soc_m2m agx7_soc_s2m	Agilex 7 FPGA I-Series Transceiver-SoC Development Kit (DK-SI-AGI027FC)
	Arria 10				a10_soc_m2m a10_soc_s2m	Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S)

2.1. About the PCIe-Based Design Example

The FPGA AI Suite PCIe-based design example demonstrates how the Intel Distribution of OpenVINO toolkit and the FPGA AI Suite support the look-aside deep learning acceleration model.

The PCIe-based design example is implemented with the following components:

- FPGA AI Suite IP
- Intel Distribution of OpenVINO toolkit
- Terasic DE10-Agilex Development Board
- Sample hardware and software systems that illustrate the use of these components

This design example includes pre-built FPGA bitstreams that correspond to pre-optimized architecture files. However, the design example build scripts let you choose from a variety of architecture files and build (or rebuild) your own bitstreams, provided that you have a license permitting bitstream generation.

This design is provided with the FPGA AI Suite as an example showing how to incorporate the IP into a design. This design is not intended for unaltered use in production scenarios. Any potential production application that uses portions of this example design must review them for both robustness and security.

The following sections in this document describe the steps to build and execute the design:

- [PCIE] Building the FPGA AI Suite Runtime on page 14
- [PCIE] Running the Design Example Demonstration Applications on page 16

The following sections in this document describe design decisions and architectural details about the design:

- [PCIE] Design Example Components on page 26

Use this document to help you understand how to create a PCIe example design with the targeted FPGA AI Suite architecture and number of instances and compiling the design for use with the Intel FPGA Basic Building Blocks (BBBs) system.

2.2. About the Open FPGA Stack (OFS) for PCIe Attach Design Examples

The FPGA AI Suite Open FPGA Stack (OFS) for PCIe Attach design examples demonstrate the design and implementation for accelerating AI inference using the FPGA AI Suite, Intel Distribution of OpenVINO toolkit, and boards that support Agilex 7 PCIe Attach OFS:

- Agilex 7 FPGA I-Series Development Kit ES2 (DK-DEV-AGI027RBES)
- Intel FPGA SmartNIC N6001-PL Platform (without Ethernet controller)

  Tip: N6001-PL SmartNIC boards are available through ODM partners. For more information, including ordering information, refer to the SmartNIC N6000-PL product brief.

Use this document to help you understand how to create the OFS for PCIe attach design example with the targeted FPGA AI Suite architecture and number of instances and compiling the design for use with the Intel FPGA Basic Building Blocks (BBBs) system.

The following sections in this document describe the steps to build and execute the design:

- [OFS-PCIE] Getting Started with Open FPGA Stack (OFS) for PCIe Attach Design Examples on page 38

The following sections in this document describe design decisions and architectural details about the design:

- [OFS-PCIE] Design Example Components on page 52

2.3. About the Hostless DDR-Free Design Example

The FPGA AI Suite provides a design example to demonstrate hostless and DDR-free operation of the FPGA AI Suite IP. Graph filters, bias, and FPGA AI Suite IP configurations are stored in on-chip memory on the FPGA device instead of DDR memory on the board.

The DDR-free design example demonstrates how FPGA AI Suite supports the following features:

- DDR-free operation
- Hostless operation (that is, running on the devices without the FPGA AI Suite runtime)
- Streaming of input features
- Streaming of inference results

The DDR-Free design example is implemented with the following components:

- FPGA AI Suite IP
- Agilex 7 FPGA I-Series Development Kit ES2 (DK-DEV-AGI027RBES)
- Sample hardware and software systems that illustrate the use of these components

For more details about DDR-free operation, refer to DDR-Free Operation in the *FPGA AI Suite IP Reference Manual*.

The design example build scripts in [PCIE] Building the FPGA AI Suite Runtime on page 14 let you choose from a variety of architecture files and build your own bitstreams, provided that you have a license permitting bitstream generation.

This design is provided with the FPGA AI Suite as an example showing how to incorporate the FPGA AI Suite IP into a DDR-Free design. This design is not intended for unaltered use in production scenarios. Any potential production application that uses portions of this design example must be reviewed for both robustness and security.

The following sections in this document describe the steps to build and execute the design:

- [HL-NO-DDR] Getting Started with the FPGA AI Suite DDR-Free Design Example on page 59
- [HL-NO-DDR] Running the Hostless DDR-Free Design Example on page 60

The following sections in this document describe design decisions and architectural details about the design:

- [HL-NO-DDR] Design Example System Architecture on page 62
- [HL-NO-DDR] Quartus Prime System Console on page 66
- [HL-NO-DDR] JTAG to Avalon MM Host Register Map on page 70
- [HL-NO-DDR] Updating MIF Files on page 71

2.4. About the Hostless JTAG Design Example

The hostless JTAG design example example demonstrates how to instantiate one instance of the FPGA AI Suite IP on an Agilex 5 E-Series device.

It places configurations and data on external DDR memory and allows an external host to interact with the memory and FPGA AI Suite IP via JTAG.

This design example targets the Agilex 5 FPGA E-Series 065B Premium Development Kit (DK-A5E065BB32AES1).

The following sections in this document describe the steps to build and execute the design:

- [HL-JTAG] Getting Started on page 72

The following sections in this document describe design decisions and architectural details about the design:

- [HL-JTAG] Design Example Components on page 79

2.5. About the SoC Design Example

The FPGA AI Suite SoC design example shows how the Intel Distribution of OpenVINO toolkit and the FPGA AI Suite support the CPU-offload deep learning acceleration model in an embedded system

The SoC design examples are implemented with the following components:

- FPGA AI Suite IP
- Intel Distribution of OpenVINO toolkit
- The community-supported OpenVINO ARM plugin
- Sample hardware and software systems that illustrate the use of these components
- Arm*-Linux build scripts for the Arria 10 SX SoC and Agilex 7 I-Series SoC FPGA hard processor system (HPS) built using Yocto frameworks

For an easier initial experience, these design examples include prebuilt FPGA bitstreams and a Linux-compiled system image that correspond to pre-optimized FPGA AI Suite architecture files.

You can copy this disk-image to an SD card and insert the card into a supported FPGA development kit. Additionally, you can use the design example scripts to choose from a variety of architecture files and build (or rebuild) your own bitstreams, subject to IP licensing limitations.

The following sections in this document describe the steps to build and execute the design:

- [SOC] FPGA AI Suite SoC Design Example Quick Start Tutorial on page 83
- [SOC] FPGA AI Suite SoC Design Example Run Process on page 106

The following sections in this document describe design decisions and architectural details about the design:

- [SOC] FPGA AI Suite SoC Design Example Build Process on page 108
- [SOC] FPGA AI Suite SoC Design Example Quartus Prime System Architecture on page 117
- [SOC] FPGA AI Suite SoC Design Example Software Components on page 131
- [SOC] Streaming-to-Memory (S2M) Streaming Demonstration on page 136

Use this document to help you understand how to create a SoC example design with the targeted FPGA AI Suite architecture.

SoC Design Example Execution Models

The SoC design example has two execution models:

- Memory-to-memory (M2M) execution model, which provides a `dla_benchmark` interface to the inference engine, similar to the PCIe-based design examples.

  For design details, refer to [SOC] Memory-to-Memory (M2M) Variant Design on page 118.

- Streaming-to-memory (S2M) execution model that demonstrates a streaming data source

  For simplicity in this design example, the streaming data source is the SoC ARM CPU itself, which streams to a layout transform on the FPGA, but this design illustrates one of the suggested system architectures for any streaming source.

  For design details, refer to [SOC] Streaming-to-Memory (S2M) Variant Design on page 120.

The design example is typically compiled for the S2M execution model, which supports both the M2M and S2M modes. A reduced functionality bitstream is also included as a compilation option, which supports only the M2M execution model.

The SoC design example has been optimized for simplicity, to create a flexible foundation that you can use to build more complex SoC designs.

3. [PCIE] Getting Started with the FPGA AI Suite PCIe-based Design Example

Before starting with the FPGA AI Suite PCIe-based Design Example, ensure that you have followed all the installation instructions for the FPGA AI Suite compiler and IP generation tools and completed the design example prerequisites as provided in the *FPGA AI Suite Getting Started Guide*.

4. [PCIE] Building the FPGA AI Suite Runtime

The FPGA AI Suite PCIe-based Design Example `runtime` directory contains the source code for the OpenVINO plugins and customized versions of the following OpenVINO programs:

- `dla_benchmark`
- `classification_sample_async`
- `object_detection_demo_yolov3_async`
- `segmentation_demo`

The CMake tool manages the overall build flow to build the FPGA AI Suite runtime plugin.

4.1. [PCIE] CMake Targets

The top level CMake build target is the FPGA AI Suite runtime plugin shared library, `libcoreDLARuntimePlugin.so`. It will not be built if the target is the software reference. Details on how to target one of the example design boards or the software emulation are specified in [PCIE] Build Options on page 14. The source files used to build the `libcoreDLARuntimePlugin.so` target are located under the following directories:

- `runtime/plugin/src/`
- `runtime/coredla_device/src/`

The flow also builds additional targets as dependencies for the top-level target. The most significant additional targets are as follows:

- The Input and Output Layout Transform library, `libdliaPluginIOTransformations.a`. The sources for this target are under `runtime/plugin/io_transformations/`.

4.2. [PCIE] Build Options

The `runtime` folder in the design example package contains a script to build the runtime called `build_runtime.sh`.

Issue the following command to run the script:

```
./build_runtime.sh <command_line_options>
```

Where <*command_line_options*> are defined in the following table:

Table 4. Command Line Options for the `build_runtime.sh` Script

Command	Description
`-h | --help`	Show usage details
`--cmake_debug`	Call cmake with a debug flag
`--verbosity=<number>`	Large numbers add some extra verbosity
`--build_dir=<path>`	Directory where the runtime build should be placed
`--disable_jit`	If this flag is specified, then the runtime will only support the Ahead of Time mode. The runtime will not link to the precompiled compiler libraries. Use this mode when trying to compile the runtime on an unsupported operating system.
`--build_demo`	Adds several OpenVINO demo applications to the runtime build. The demo applications are in subdirectories of the `runtime/directory`.
`--target_de10_agilex`	Target the Terasic DE10-Agilex Development Board.
`--target_emulation`	Target the software emulator model. Specify this option to build the runtime without a board installed. Inference requests are executed by the software emulator model of the FPGA AI Suite IP.
`--aot_splitter_example`	Builds the AOT splitter example utility for the selected target (Terasic DE10-Agilex Development Board). This option builds an AOT file for a model, splits the AOT file into its constituent components (weights, overlay instructions, etc), and the builds a small utility that loads the model and a single image onto the target FPGA board without using OpenVINO. You must set the `$AOT_SPLITTER_EXAMPLE_MODEL` and `$AOT_SPLITTER_EXAMPLE_INPUT` environment variables correctly. For details, refer to "FPGA AI Suite Ahead-of-Time (AOT) Splitter Utility Example Application" in *FPGA AI Suite IP Reference Manual*.

The FPGA AI Suite runtime plugin is built in release mode by default. To enable debug mode, you must specify the `--cmake_debug` option of the script command.

The `-no_make` option skips the final call to the `make` command. You can make this call manually instead.

FPGA AI Suite hardware is compiled to include one or more IP instances, with the same architecture for all instances. Each instance accesses data from a unique bank of DDR:

- The Terasic DE10-Agilex Development Board has four DDR banks and supports up to four instances.

The runtime automatically adapts to the correct number of instances.

If the FPGA AI Suite runtime uses two or more instances, then the image batches are divided between the instances to execute two or more batches in parallel on the FPGA device.

5. [PCIE] Running the Design Example Demonstration Applications

This section describes the steps to run the demonstration application and perform accelerated inference using the PCIe Example Design.

5.1. [PCIE] Exporting Trained Graphs from Source Frameworks

Before running any demonstration application, you must convert the trained model to the Inference Engine format (`.xml, .bin`) with the OpenVINO Model Optimizer.

For details on creating the `.bin/.xml` files, refer to the *FPGA AI Suite Getting Started Guide*.

5.2. [PCIE] Compiling Exported Graphs Through the FPGA AI Suite

The network as described in the `.xml` and `.bin` files (created by the Model Optimizer) is compiled for a specific FPGA AI Suite architecture file by using the FPGA AI Suite compiler.

The FPGA AI Suite compiler compiles the network and exports it to a `.bin` file that uses the same `.bin` format as required by the OpenVINO Inference Engine.

This `.bin` file created by the compiler contains the compiled network parameters for all the target devices (FPGA, CPU, or both) along with the weights and biases. The inference application imports this file at runtime.

The FPGA AI Suite compiler can also compile the graph and provide estimated area or performance metrics for a given architecture file or produce an optimized architecture file.

For more details about the FPGA AI Suite compiler, refer to the *FPGA AI Suite Compiler Reference Manual*.

5.3. [PCIE] Compiling the PCIe-based Example Design

Prepackaged bitstreams are available for the PCIe Example Design. If the prepackaged bitstreams are installed, they are installed in `demo/bitstreams/`.

To build example design bitstreams, you must have a license that permits bitstream generation for the IP, and have the correct version of Quartus® Prime software installed. Use the `dla_build_example_design.py` utility to create a bitstream.

For more details about this command, the steps it performs, and advanced command options, refer to Build Script and to the *FPGA AI Suite Getting Started Guide*.

5.4. [PCIE] Programming the FPGA Device (Agilex 7)

You can program the Terasic DE10-Agilex Development Board board using the `fpga_jtag_reprogram` tool.

For details, refer to "FPGA AI Suite Quick Start Tutorial" in the *FPGA AI Suite Getting Started Guide*.

5.5. [PCIE] Performing Accelerated Inference with the `dla_benchmark` Application

You can use the `dla_benchmark` demonstration application included with the FPGA AI Suite runtime to benchmark the performance of image classification networks.

5.5.1. [PCIE] Inference on Image Classification Graphs

The demonstration application requires the OpenVINO device flag to be either `HETERO:FPGA,CPU` for heterogeneous execution or `HETERO:FPGA` for FPGA-only execution.

The `dla_benchmark` demonstration application runs five inference requests (batches) in parallel on the FPGA, by default, to achieve optimal system performance. To measure steady state performance, you should run multiple batches (using the `niter` flag) because the first iteration is significantly slower with FPGA devices.

The `dla_benchmark` demonstration application also supports multiple graphs in the same execution. You can place more than one graphs or compiled graphs as input, separated by commas.

Each graph can have either a different input dataset or use a commonly shared dataset among all graphs. Each graph requires an individual `ground_truth_file` file, separated by commas. If some `ground_truth_file` files are missing, the `dla_benchmark` continues to run and ignore the missing ones.

When multi-graph is enabled, the `-niter` flag represents the number of iterations for each graph, so the total number of iterations becomes `-niter` × number of graphs.

The `dla_benchmark` demonstration application switches graphs after submitting `-nireq` requests. The request queue holds the number of requests up to `-nireq` × number of graphs. This limit is constrained by the DMA CSR descriptor queue size (64 per hardware instance).

The board you use determines the number of instances that you can compile the FPGA AI Suite hardware for:

- For the Terasic DE10-Agilex Development Board, you can compile up to four instances with the same architecture on all instances.

Each instance accesses one of the DDR banks on the board and executes the graph independently. This optimization enables multiple batches to run in parallel, limited by the number of DDR banks available. Each inference request created by the demonstration application is assigned to one of the instances in the FPGA plugin.

To enable memory-mapped device (MMD) debug messages when you run the `dla_benchmark` demonstration application. set the ACL_PCIE_DEBUG environment variable as follows:

```
ACL_PCIE_DEBUG=1
```

Also, you can test full DDR write and read back functionality when the `dla_benchmark` demonstration application runs by setting the COREDLA_RUNTIME_MEMORY_TEST environment variable as follows:

```
COREDLA_RUNTIME_MEMORY_TEST=1
```

To ensure that batches are evenly distributed between the instances, you must choose an inference request batch size that is a multiple of the number of FPGA AI Suite instances. For example, with two instances, specify the batch size as six (instead of the OpenVINO default of five) to ensure that the experiment meets this requirement.

The example usage that follows has the following assumptions:

- A Model Optimizer IR `.xml` file is in `demo/models/public/resnet-50-tf/FP32/`
- An image set is in `demo/sample_images/`
- The board is programmed with a bitstream that corresponds to `AGX7_Performance.arch`

```
binxml=$COREDLA_ROOT/demo/models/public/resnet-50-tf/FP32

imgdir=$COREDLA_ROOT/demo/sample_images

cd $COREDLA_ROOT/runtime/build_Release

./dla_benchmark/dla_benchmark \
   -b=1 \
   -m $binxml/resnet-50-tf.xml \
   -d=HETERO:FPGA,CPU \
   -i $imgdir \
   -niter=4 \
   -plugins ./plugins.xml \
   -arch_file $COREDLA_ROOT/example_architectures/AGX7_Performance.arch \
   -api=async \
   -groundtruth_loc $imgdir/TF_ground_truth.txt \
   -perf_est \
   -nireq=8 \
   -bgr
```

The following example shows how the FPGA AI Suite IP can dynamically swap between graphs. This example usage assumes that another Model Optimizer IR `.xml` file has been placed in `demo/models/public/resnet-101-tf/FP32/`. It also assumes that another image set has been placed into `demo/sample_images_rn101/`. In this case, `dla_benchmark` only evaluates the classification accuracy of Resnet50 because we did not provide ground truth for the second graph (ResNet101).

```
binxml1=$COREDLA_ROOT/demo/models/public/resnet-50-tf/FP32
binxml2=$COREDLA_ROOT/demo/models/public/resnet-101-tf/FP32
imgdir1=$COREDLA_ROOT/demo/sample_images
imgdir2=$COREDLA_ROOT/demo/sample_images_rn101
cd $DEVELOPER_PACKAGE_ROOT/runtime/build_Release
./dla_benchmark/dla_benchmark \
   -b=1 \
   -m $binxml1/resnet-50-tf.xml,$binxml2/resnet-101-tf.xml \
   -d=HETERO:FPGA,CPU \
```

```
-i $imgdir1,$imgdir2 \
-niter=8 \
-plugins ./plugins.xml \
-arch_file $COREDLA_ROOT/example_architectures/AGX7_Performance.arch \
-api=async \
-groundtruth_loc $imgdir1/TF_ground_truth.txt \
-perf_est \
-nireq=8 \
-bgr
```

5.5.2. [PCIE] Inference on Object Detection Graphs

To enable the accuracy checking routine for object detection graphs, you can use the `-enable_object_detection_ap=1` flag.

This flag lets the `dla_benchmark` calculate the mAP and COCO AP for object detection graphs. Besides, you need to specify the version of the YOLO graph that you provide to the `dla_benchmark` through the `–yolo_version` flag. Currently, this routine is known to work with YOLOv3 (graph version is `yolo-v3-tf`) and TinyYOLOv3 (graph version is `yolo-v3-tiny-tf`).

5.5.2.1. [PCIE] The mAP and COCO AP Metrics

Average precision and average recall are averaged over multiple Intersection over Union (IoU) values.

Two metrics are used for accuracy evaluation in the `dla_benchmark` application. The mean average precision (mAP) is the challenge metric for PASCAL VOC. The mAP value is averaged over all 80 categories using a single IoU threshold of 0.5. The COCO AP is the primary challenge for object detection in the Common Objects in Context contest. The COCO AP value uses 10 IoU thresholds of .50:.05:.95. Averaging over multiple IoUs rewards detectors with better localization.

5.5.2.2. [PCIE] Specifying Ground Truth

The path to the ground truth files is specified by the flag `–groundtruth_loc`.

The validation dataset is available on the COCO official website.

The `dla_benchmark` application currently allows only plain text ground truth files. To convert the downloaded JSON annotation file to plain text, use the `convert_annotations.py` script.

5.5.2.3. [PCIE] Example of Inference on Object Detection Graphs

The example that follows makes the following assumptions:

- The Model Optimizer IR `graph.xml` for either YOLOv3 or TinyYOLOv3 is in the current working directory.

  Model Optimizer generates an FP32 version and an FP16 version. Use the FP32 version.
- The validation images downloaded from the COCO website are placed in the `./mscoco-images` directory.
- The JSON annotation file is downloaded and unzipped in the current directory.

To compute the accuracy scores on many images, you can usually increase the number of iterations using the flag `-niter` instead of a large batch size `-b`. The product of the batch size and the number of iterations should be less than or equal to the number of images that you provide.

```
cd $COREDLA_ROOT/runtime/build_Release

python ./convert_annotations.py ./instances_val2017.json \
   ./groundtruth

./dla_benchmark/dla_benchmark \
   -b=1 \
   -niter=5000 \
   -m=./graph.xml \
   -d=HETERO:FPGA,CPU \
   -i=./mscoco-images \
   -plugins=./plugins.xml \
   -arch_file=../../example_architectures/AGX7_Performance.arch \
   -yolo_version=yolo-v3-tf \
   -api=async \
   -groundtruth_loc=./groundtruth \
   -nireq=8 \
   -enable_object_detection_ap \
   -perf_est \
   -bgr
```

5.5.3. [PCIE] Additional `dla_benchmark` Options

The `dla_benchmark` tool is part of the example design and the distributed runtime includes full source code for the tool.

Table 5. Command Line `dla_benchmark` Options

Command Option	Description
`-nireq=<N>`	This option controls the number of simultaneous inference requests that are sent to the FPGA. Typically, this should be at least twice the number of IP instances; this ensures that each IP can execute one inference request while `dla_benchmark` loads the feature data for a second inference request to the FPGA-attached DDR memory.
`-b=<N>` `--batch-size=<N>`	This option controls the batch size. A batch size greater than 1 is created by repeating configuration data for multiple copies of the graph. A batch size of 1 is typically best for latency System throughput for small graphs, when inference operations are offloaded from a CPU to an FPGA, may improve by using a batch greater than 1. On very small graphs, IP throughput may also improve when using a batch greater than 1. The default value is 1.
`-niter=<N>`	Number of batches to run. Each batch has a size specified by the `--batch-size` option. The total number of images processed is the product of the `--batch-size` option value multiplied by the `-niter` option value.

continued...

Command Option	Description
-d=<*STRING*>	Using -d=HETERO:FPGA, CPU causes dla_benchmark to use the OpenVINO heterogeneous plugin to execute inference on the FPGA, with fallback to the CPU for any layers that cannot go to the FPGA. Using -d=HETERO:CPU or -d=CPU executes inference on the CPU, which may be useful for testing the flow when an FPGA is not available. Using -d=HETERO:FPGA may be useful for ensuring that all graph layers are accelerated on the FPGA (and an error is issued if this is not possible).
-arch_file=<*FILE*> --arch=<*FILE*>	This specifies the location of the .arch file that was used to configure the IP on the FPGA. The dla_benchmark will issue an error if this does not match the.arch file used to generate the IP on the FPGA.
-m=<*FILE*> --network_file=<*FILE*>	This points to the XML file from OpenVINO Model Optimizer that describes the graph. The BIN file from Model Optimizer must be kept in the same directory and same filename (except for the file extension) as the XML file.
-i=<*DIRECTORY*>	This points to the directory containing the input images. Each input file corresponds to one inference request. The files are read in order sorted by filename; set the environment variable VERBOSE=1 to see details describing the file order.
-api=[sync|async]	The -api=async option allows dla_benchmark to fully take advantage of multithreading to improve performance. The -api=sync option may be used during debug.
-groundtruth_loc=<*FILE*>	Location of the file with ground truth data. If not provided, then dla_benchmark will not evaluate accuracy. This may contain classification data or object detection data, depending on the graph.
-yolo_version=<*STRING*>	This option is used when evaluating the accuracy of a YOLOv3 or TinyYOLOv3 object detection graph. The options are yolo-v3-tf and yolo-v3-tiny-tf.
-enable_object_detection_ap	This option may be used with an object detection graph (YOLOv3 or TinyYOLOv3) to calculate the object detection accuracy.
-bgr	When used, this flag indicates that the graph expects input image channel data to use BGR order.
-plugins_xml_file=<*FILE*>	*Deprecated:* This option is deprecated and will be removed in a future release. Use the -plugins option instead. This option specifies the location of the file specifying the OpenVINO plugins to use. This should be set to $COREDLA_ROOT/runtime/plugins.xml in most cases. If you are porting the design to a new host or doing other development, it may be necessary to use a different value.
-plugins=<*FILE*>	This option specifies the location of the file that specifies the OpenVINO plugins to use. The default behavior is to read the plugins.xml file from the runtime/ directory. This runs inference on the FPGA device. If you want to run inference using the emulation model, specify -plugins=emulation. If you are porting the design to a new host or doing other development, you might need to use a different value.

continued...

Command Option	Description
`-mean_values=<input_name[mean_values]>`	Uses channel-specific mean values in input tensor creation through the following formula: $\frac{\text{input} - \text{mean}}{\text{scale}}$. The Model Optimizer mean values are the preferred choice and the mean values defined by this option serve as fallback values.
`-scale_values=<input_name[scale_values]>`	Uses channel-specific scale values in input tensor creation through the following formula: $\frac{\text{input} - \text{mean}}{\text{scale}}$. The Model Optimizer scale values are the preferred choice and the scale values defined by this option serve as fallback values.
`-pc`	This option reports the performance counters for the CPU subgraphs, if there is any. No sorting is done on the report.
`-pcsort=[sort|no_sort|simple_sort]`	This option reports the performance counters for the CPU subgraph and sets the sorting option for the performance counter report: • `sort`: Report is sorted by operation time cost • `no_sort`: Report is not sorted • `simple_sort`: Report is sorted by opts time cost but print only executed operations
`-save_run_summary`	Collect performance metrics during inference. These metrics can help you determine how efficient an architecture is at executing a model. For more information, refer to [PCIE] The dla_benchmark Performance Metrics on page 22.

5.5.4. [PCIE] The `dla_benchmark` Performance Metrics

The `-save_run_summary` option makes the `dla_benchmark` demonstration application collect performance metrics during inference. These metrics can help you determine how efficient an architecture is at executing a model.

Note: The `dla_benchmark` application provides throughput in "frames per second". The time per frame (latency) is 1/throughput.

Statistic	Description
Count	The number of times interference was performed. This is set by the `-niter` option.
System duration	The total time between when the first inference request was made to when the last request was finished, as measured by the host program.
IP duration	The total time the spent-on inference. This is reported by the IP on the FPGA.
Latency	The median time of all inference requests made by the host. This includes any overhead from OpenVINO or the FPGA AI Suite runtime.
System throughput	The total throughput of the system, including any OpenVINO or FPGA AI Suite runtime overhead.
Number of hardware instances	The number of IP instances on the FPGA.
Number of network instances	The number graphs that the IP processes in parallel.
IP throughput per instance	The throughput of a single IP instance. This is reported by the IP on the FPGA.

continued...

Statistic	Description
IP throughput per f_{MAX} per instance	The **IP throughput per instance** value scaled by the **IP clock frequency** value.
IP clock frequency	The clock frequency, as reported by the IP running on the FPGA device. The `dla_benchmark` application treats this value as the IP core f_{MAX} value.
Estimated IP throughput per instance	The estimated per-IP throughput, as estimated by the `dla_compiler` command with the `--fanalyze-performance` option.
Estimated IP throughput per fmax per instance	The **Estimated IP throughput per instance** value scaled by the compiler f_{MAX} estimate.

5.5.4.1. [PCIE] Interpreting System Throughput and Latency Metrics

The **System throughput** and **Latency** metrics are measured by the host through the OpenVINO API. These measurements include any overhead that is incurred by both the API and the FPGA AI Suite runtime. They also account for any time spent waiting to make inference requests and the number of available instances.

In general, the system throughput is defined as follows:

$$\text{System Throughput} = \frac{\text{Batch Size} \times \text{Images per Batch}}{\text{Latency}}$$

The **Batch Size** and **Images Per Batch** values are set by the `--batch-size` and `-niter` options, respectively.

For example, consider when `-nireq=1` and there is a single IP instance. The **System throughput** value is approximately the same as the **IP-reported throughput** value because the runtime can perform only one inference at a time. However, if both the `-nireq` and the number of IP instances is greater than one, the runtime can perform requests in parallel. As such, the total system throughput is greater than the individual IP throughput.

In general, the `-nireq` value should be *twice* the number of IP instances. This setting enables the FPGA AI Suite runtime to pipeline inferences requests, which allows the host to prepare the data for the next request while an IP instance is processing the previous request.

5.6. [PCIE] Running the Ported OpenVINO Demonstration Applications

Some of the sample demonstration applications from the OpenVINO toolkit for Linux Version 2023.3 have been ported to work with the FPGA AI Suite. These applications are built at the same time as the runtime when using the `-build_demo` flag to `build_runtime.sh`.

The FPGA AI Suite runtime includes customized versions of the following demo applications for use with the FPGA AI Suite IP and plugins:

- `classification_sample_async`
- `object_detection_demo_yolov3_async`
- `segmentation_demo`

Each demonstration application uses a different graph. The OpenVINO HETERO plugin can fall-back to the CPU for portions of the graph that are not supported with FPGA-based acceleration. However, in a production environment, it may be more efficient to use alternate graphs that execute exclusively on the FPGA.

You can use the example `.arch` files that are supplied with the FPGA AI Suite with the demonstration applications. However, certain example `.arch` files do not enable some of the layer-types used by the graphs associated with the demonstration applications. Using these `.arch` files cause portions of the graph to needlessly execute on the CPU. To minimize the number of layers that are executed on the CPU by the demonstration application, use the following architecture description files located in the `example_architectures/` directory of the FPGA AI Suite installation package to run the demos:

- Agilex 7: `AGX7_Generic.arch`

As specified in [PCIE] Programming the FPGA Device (Agilex 7) on page 17, you must program the FPGA device with the bitstream for the architecture being used. Each demonstration application includes a README.md file specifying how to use it.

When the OpenVINO sample applications are modified to support the FPGA AI Suite, the FPGA AI Suite plugin used by OpenVINO needs to know how to find the `.arch` file describing the IP parameterization by using the following configuration key. The following C++ code is used in the demo for this purpose:

```
ie.SetConfig({ { DLIA_CONFIG_KEY(ARCH_PATH), FLAGS_arch_file } }, "FPGA");
```

The OpenVINO demonstration application `hello_query_device` does not work with the FPGA AI Suite due to low-level hardware identification assumptions.

5.6.1. [PCIE] Example Running the Object Detection Demonstration Application

You must download the following items:

- `yolo-v3-tf` from the OpenVINO Model Downloader. The command should look similar to the following command:

  ```
  python3 <path_to_installation>/open_model_zoo/omz_downloader \
     --name yolo-v3-tf \
     --output_dir <download_dir>
  ```

  From the downloaded model, generate the `.bin`/`.xml` files:

  ```
  python3 <path_to_installation>/open_model_zoo/omz_converter \
     --name yolo-v3-tf \
     --download_dir <download_dir> \
     --output_dir <output_dir> \
     --mo <path_to_installation>/model_optimizer/mo.py
  ```

  Model Optimizer generates an FP32 version and an FP16 version. Use the FP32 version.
- Input video from: https://github.com/intel-iot-devkit/sample-videos.
- The recommended video is `person-bicycle-car-detection.mp4`

To run the object detection demonstration application,

1. Ensure that demonstration applications have been built with the following command:

   ```
   build_runtime.sh -target_de10_agilex -build-demo
   ```

2. Ensure that the FPGA has been configured with the Generic bitstream.
3. Run the following command:

   ```
   ./runtime/build_Release/object_detection_demo/object_detection_demo \
      -d HETERO:FPGA,CPU \
      -i <path_to_video>/input_video.mp4 \
      -m <path_to_model>/yolo_v3.xml \
      -arch_file=$COREDLA_ROOT/example_architectures/AGX7_Generic.arch \
      -plugins $COREDLA_ROOT/runtime/plugins.xml \
      -t 0.65 \
      -at yolo
   ```

Tip: High-resolution video input, such as when using HD camera as input, imposes considerable decoding overhead on the inference engine that can potentially lead to reduced system throughput. Use the the `-input_resolution=<width>x<height>` option that is included in the demonstration application to adjust the input resolution to a level that balances video quality with system performance.

6. [PCIE] Design Example Components

6.1. [PCIE] Build Script

While this design example includes prepackaged bitstreams, you can also use the build script to build bitstreams.

To build the PCIe-based example design, use the `bin/dla_build_example_design.py` script. You can use this script to create an example design with one or multiple FPGA AI Suite IP instances.

The script generates a wrapper that wraps one or more IP instances along with adapters necessary to connect to the Terasic DE10-Agilex BSP.

When specifying an <*architecture_file*>, pay attention to the resource limitations on the FPGA, as well as the number of resources that the board support package (BSP) uses.

The `dla_compiler` tool includes a `--fanalyze-area` option to estimate the resources required for a single IP instance corresponding to an architecture file, as described in the *FPGA AI Suite Compiler Reference Manual*.

Implementing two instances (as is the default for `dla_build_example_design.py`) requires twice the resources.

Table 6. Build Script Resources

Resources	ALMs	M20k	DSPs
Resources available on Agilex 7 F-Series 014 device	487200	7110	4510
Reasonable Target Utilization	80%	90%	90%
Usable Resources	390000	6399	4059

The DE10-Agilex design is only validated for use with Quartus Prime Pro Edition Version 24.3. The Agilex 7 device has significantly more resources and can support up to four IP instances.

6.1.1. [PCIE] Build Script Options

Table 7. Build Script Options

Option	Description
`-a, --archs`	Path to FPGA AI Suite IP Architecture Description File
`--build-dir`	Path to hardware build directory where BSP infrastructure and generated RTL will be located.

continued...

Option	Description
	(default: `coredla/pcie_ed/platform/build_synth`)
`--build`	Option to perform compilation of the PCIe design using Quartus Prime after instantiation (default: False).
`-d, --archs-dir`	Path to directory that contains Architecture Description Files for you to interactively choose from (alternative to '-a')
`-ed, --example-design-id`	To build for the Terasic DE10-Agilex board, specify 3. (default: 3)
`-n, --num-instances`	Number of IP instances to build (default: 2). For the Terasic DE10-Agilex board, this number must be 1, 2, 3, or 4.
`--num-paths`	Number of top critical paths to report after compiling the design (default: `2000`).
`-q, --quiet`	Run script quietly without printing the output of underlying scripts to the terminal.
`--qor-modules`	List of internal modules (instance names) from inside the IP to include in the QoR summary report.
`-s, --seed`	Seed to be used in compiling the design (default: `1`).
`--unlicensed/licensed`	This option is passed to the `dla_create_ip` tool to tell the tool to generate either an unlicensed or licensed copy of the FPGA AI Suite: • **Unlicensed IP:** Unlicensed IP has a limit of 10000 inferences. After 10000 inferences, the unlicensed IP refuses to perform any additional inference and a bit in the CSR is set. For details about the CSR bit, refer to DMA Descriptor Queue in *FPGA AI Suite IP Reference Manual*. • **Licensed IP:** Licensed IP has no inference limitation. If you do not have a license but generate licensed IP, Quartus Primesoftware cannot generate a bitstream. If neither option is specified, then the `dla_create_ip` tool queries the `lmutil` license manager to determine the correct option.
`--wsl`	This option sets the build script to run such that the final Quartus Prime compilation runs in the Windows* environment. After the script sets up the compilation, it prints the instructions to complete the compilation on Windows. *Restriction:* Only supported within a WSL 2 environment and for the DE10-Agilex example design.
`--finalize`	*Restriction:* This option can be used only when following the instructions provided by the build script run with the `--wsl` option.

6.1.2. [PCIE] Script Flow

The following steps describe the internal flow of the `dla_build_example_design.py` script for the Terasic DE10-Agilex Development Board:

1. Runs the `dla_create_ip` script to create an FPGA AI Suite IP for the requested FPGA AI Suite architecture
2. Creates a wrapper around the FPGA AI Suite IP instances and adapter logic
3. Copies in the Terasic BSP, and patches the directory with FPGA AI Suite files. This creates a build directory that has the BSP infrastructure needed to compile the design with Quartus Prime software.
4. Runs the `quartus_sh` command on the FPGA AI Suite `dla_flat_compile.tcl` script to compile the design example with Quartus Prime software to produce an FPGA bitstream.

The bitstream is in the <*build_dir*> directory that you set when running the script (or the default location, if you did not set it). The bitstream file name is `flat.sof`.

The Quartus Prime compilation reports are available in the <*build_dir*>/hw directory. A `build.log` file that has all the output log for running the build script is available in the <*build_dir*> directory. In addition, the achieved FPGA AI Suite clock frequency is the Clock Frequency value in the summary table in the following file:

<*build_dir*>/quartus_summary.txt

6.2. [PCIE] Example Architecture Bitstream Files

The FPGA AI Suite provides example Architecture Files and bitstreams for the PCIe-based Example Design. The bitstreams are distributed as a separate tarball.

6.3. [PCIE] Software Components

The PCIe-based design example contains a sample software stack for the runtime flow.

The following figure, *Software Stacks for FPGA AI Suite Inference*, shows the complete runtime stack.

For the Agilex 7 design example, the following components comprise the runtime stack:

- OpenVINO Toolkit 2023.3 LTS (Inference Engine, Heterogeneous Plugin)
- FPGA AI Suite runtime plugin
- Terasic DE10-Agilex-B2E2 board driver

The design example contains the source files and Makefiles to build the FPGA AI Suite runtime plugin. The OpenVINO component is external and must be manually pre-installed.

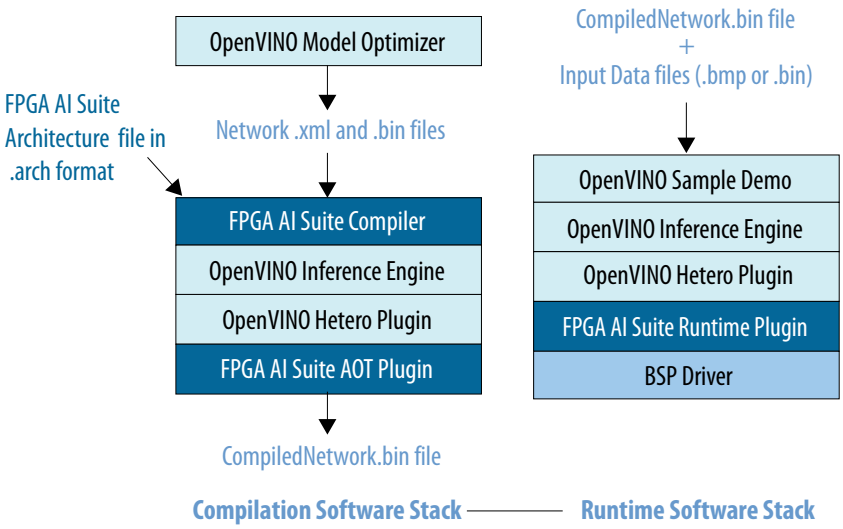
A separate flow compiles the AI network graph using the FPGA AI Suite compiler, as shown in figure *Software Stacks for FPGA AI Suite Inference* below as the Compilation Software Stack.

The compilation flow output is a single binary file called `CompiledNetwork.bin` that contains the compiled network partitions for FPGA and CPU devices along with the network weights. The network is compiled for a specific FPGA AI Suite architecture and batch size. This binary is created on-disk only when using the Ahead-Of-Time flow; when the JIT flow is used, the compiled object stays in-memory only.

An Architecture File describes the FPGA AI Suite IP architecture to the compiler. You must specify the same Architecture File to the FPGA AI Suite compiler and to the FPGA AI Suite PCIe Example Design build script (`dla_build_example_design.py`).

The runtime flow accepts the `CompiledNetwork.bin` file as the input network along with the image data files.

Figure 1. Software Stacks for FPGA AI Suite Inference



The runtime stack cannot program the FPGA with a bitstream. To build a bitstream and program the FPGA devices:

1. Compile the design example. For details, refer to Compiling the PCIe-based Design Example.
2. Program the device with the bitstream. For details, refer to [PCIE] Programming the FPGA Device (Agilex 7) on page 17.

To run inference through the OpenVINO Toolkit on the FPGA, set the OpenVINO device configuration flag (used by the heterogeneous Plugin) to FPGA or HETERO:FPGA,CPU.

6.3.1. [PCIE] OpenVINO FPGA Runtime Plugin

The FPGA runtime plugin uses the OpenVINO Inference Engine Plugin API.

The OpenVINO Plugin architecture is described in the *OpenVINO Developer Guide for Inference Engine Plugin Library*.

The source files are located under `runtime/plugin`. The three main components of the runtime plugin are the Plugin class, the Executable Network class, and the Inference Request class. The primary responsibilities for each class are as follows:

Plugin class

- Initializes the runtime plugin with an FPGA AI Suite architecture file which you set as an OpenVINO configuration key (refer to [PCIE] Running the Ported OpenVINO Demonstration Applications on page 23).
- Contains `QueryNetwork` function that analyzes network layers and returns a list of layers that the specified architecture supports. This function allows network execution to be distributed between FPGA and other devices and is enabled with the HETERO mode.
- Creates an executable network instance in one of the following ways:
  - Just-in-time (JIT) flow: Compiles a network such that the compiled network is compatible with the hardware corresponding to the FPGA AI Suite architecture file, and then loads the compiled network onto the FPGA device.
  - Ahead-of-time (AOT) flow: Imports a precompiled network (exported by FPGA AI Suite compiler) and loads it onto the FPGA device.

Executable Network Class

- Represents an FPGA AI Suite compiled network
- Loads the compiled model and config data for the network onto the FPGA device that has already been programmed with an FPGA AI Suite bitstream. For two instances of FPGA AI Suite, the Executable Network class loads the network onto both instances, allowing them to perform parallel batch inference.
- Stores input/output processing information.
- Creates infer request instances for pipelining multiple batch execution.

Infer Request class

- Runs a single batch inference serially.
- Executes five stages in one inference job – input layout transformation on CPU, input transfer to DDR, FPGA AI Suite FPGA execution, output transfer from DDR, output layout transformation on CPU.
- In asynchronous mode, executes the stages on multiple threads that are shared across all inference request instances so that multiple batch jobs are pipelined, and the FPGA is always active.

Related Information

OpenVINO Developer Guide for Inference Engine Plugin Library

6.3.2. [PCIE] FPGA AI Suite Runtime

The FPGA AI Suite runtime implements lower-level classes and functions that interact with the memory-mapped device (MMD). The MMD is responsible for communicating requests to the driver, and the driver connects to the BSP, and ultimately to the FPGA AI Suite IP instance or instances.

The runtime source files are located under `runtime/coredla_device`. The three most important classes in the runtime are the Device class, the GraphJob class, and the BatchJob class.

Device class

- Acquires a handle to the MMD for performing operations by calling `aocl_mmd_open`.
- Initializes a DDR memory allocator with the size of 1 DDR bank for each FPGA AI Suite IP instance on the device.
- Implements and registers a callback function on the MMD DMA (host to FPGA) thread to launch FPGA AI Suite IP for batch=1 after the batch input data is transferred from host to DDR.
- Implements and registers a callback function (interrupt service routine) on the MMD kernel interrupt thread to service interrupts from hardware after one batch job completes.
- Provides the `CreateGraphJob` function to create a GraphJob object for each FPGA AI Suite IP instance on the device.
- Provides the `WaitForDla(`*`instance id`*`)` function to wait for a batch inference job to complete on a given instance. Returns instantly if the number of batch jobs finished (that is, the number of jobs processed by interrupt service routine) is greater than number of batch jobs waited for this instance. Otherwise, the function waits until interrupt service routine notifies. Before returning, this function increments the number of batch jobs that have been waited for this instance.

GraphJob class

- Represents a compiled network that is loaded onto one instance of the FPGA AI Suite IP on an FPGA device.
- Allocates buffers in DDR memory to transfer configuration, filter, and bias data.
- Creates BatchJob objects for a given number of pipelines and allocates input and output buffers for each pipeline in DDR.

BatchJob class

- Represents a single batch inference job.
- Stores the DDR addresses for batch input and output data.
- Provides `LoadInputFeatureToDdr` function to transfer input data to DDR and start inference for this batch asynchronously.
- Provides `ReadOutputFeatureFromDdr` function to transfer output data from DDR. Must be called after inference for this batch is completed.

6.3.3. [PCIE] BSP Driver

The FPGA AI Suite runtime MMD software uses a driver supplied as part of the BSP to access and interact with the FPGA device.

The source files for the driver are in `runtime/coredla_device/mmd`. The source files contain classes for managing and accessing the FPGA device by using BSP functions for reading/writing to CSR, reading/writing to DDR, and handling kernel interrupts.

BSP Driver for the Agilex 7 Design Example

Contact your Intel representative for information on the driver for the Terasic DE10-Agilex board support package.

6.3.4. [PCIE] Software Interface to the BSP

The interface to the user-space portion of the BSP drivers is centralized in the `MmdWrapper` class, which can be found in the file `$COREDLA_ROOT/runtime/coredla_device/inc/mmd_wrapper.h`.

When porting the runtime to a new board, the team responsible for the new board support must ensure that each of the member functions in MmdWrapper calls into a board-specific implementation function. The team doing this will need to modify the runtime build process and adjacent code.

7. [PCIE] Design Example System Architecture for the Agilex 7 FPGA

The Agilex 7 design example is derived from the BSP provided by the Terasic DE10-Agilex Development Board.

7.1. [PCIE] System Overview

The system consists of the following components connected to a host system via a PCIe interface as shown in the following figure.

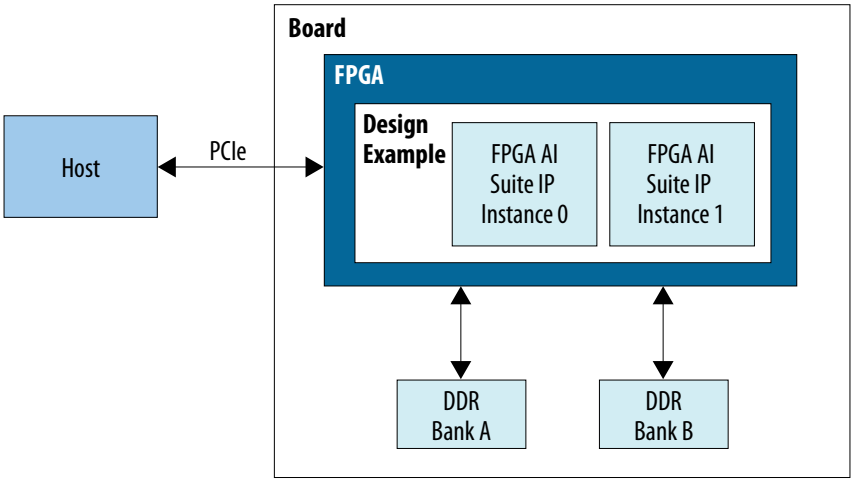
- A board with the FPGA device
- On-board DDR memory

The FPGA image consists of the FPGA AI Suite IP and an additional logic that connects it to a PCIe interface and DDR. The host can read and write to the DDR memory through the PCIe port. In addition, the host can communicate and control the FPGA AI Suite instances through the PCIe connection which is also connected the direct memory access (DMA) CSR port of FPGA AI Suite instances.

The FPGA AI Suite IP accelerates neural network inference on batches of images. The process of executing a batch follows these steps:

1. The host writes a batch of images, weights, and configuration data to DDR where weights can be reused between batches.
2. The host writes to the FPGA AI Suite CSR to start execution.
3. FPGA AI Suite computes the results of the batch and stores them in DDR.
4. Once the computation is complete, FPGA AI Suite raises an interrupt to the host.
5. The host reads back the results from DDR.

Figure 2. FPGA AI Suite Example Design System Overview



7.2. [PCIE] Hardware

This section describes the PCIe--based Example Design in detail.

A top-level view of the design example is shown in FPGA AI Suite Example Design Top Level.

The instances of FPGA AI Suite IP are on the right (`dla_top.sv`). All communication between the FPGA AI Suite IP systems and the outside occurs via the FPGA AI Suite IP DMA. The FPGA AI Suite IP DMA provides a CSR (which also has interrupt functionality) and reader/writer modules which read/write from DDR.

The host communicates with the board through the PCIe protocol. The host can do the following things:

1. Read and write the on-board DDR memory (these reads/writes do not go through FPGA AI Suite IP).
2. Read/write to the FPGA AI Suite IP DMA CSR of the instances.
3. Receive interrupt signals from the FPGA AI Suite IP DMA CSR of both instances.

Each FPGA AI Suite IP instance can do the following things:

1. Read/write to its DDR bank.
2. Send interrupts to the host through the interrupt interface.
3. Receive reads/writes to its DMA CSR.

From the perspective of the FPGA AI Suite, external connections are to the PCIe interface and to the on-board DDR4 memory. The DDR memory is connected directly to `mem.qsys` block, while the PCIe interface is converted into Avalon® memory mapped (MM) interfaces in `pcie_ed.qsys` block for communication with the `mem.qsys` block.

The `mem.qsys` blocks arbitrate the connections to DDR memory between the reader/writer modules in FPGA AI Suite IP and reads/writes from the host. Each FPGA AI Suite IP instance in this design has access to only one of the DDR banks. This design

decision implies that the number simultaneous FPGA AI Suite IP instances that can exist in the design is limited to the number of DDR blocks available on the board. Adding an additional arbiter would relax this restriction and allow additional FPGA AI Suite IP instances.

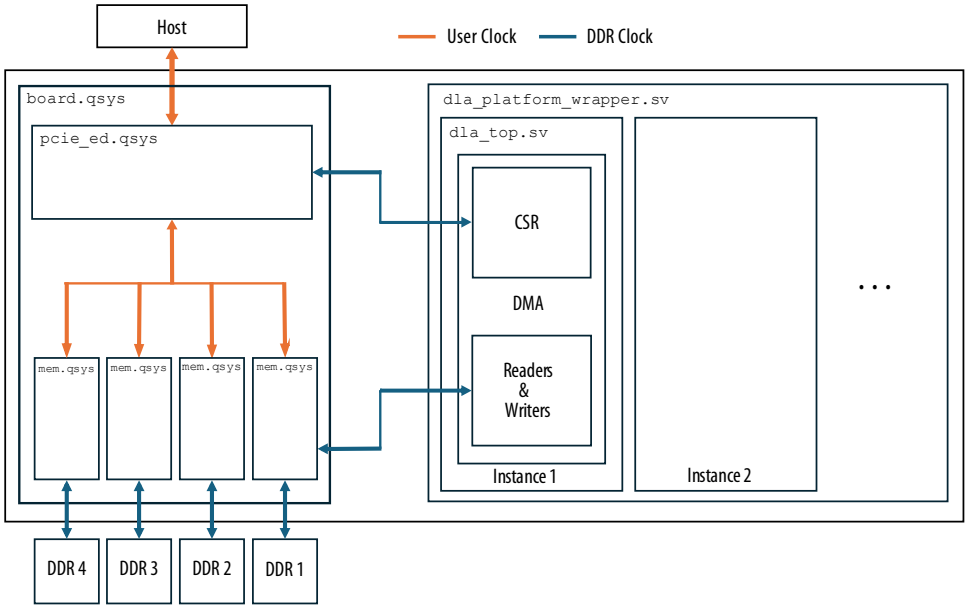
Much of `board.qsys` operates using the Avalon Memory-mapped (MM) interface protocol. The FPGA AI Suite DMA uses AXI protocol, and `board.qsys` has Avalon MM interface to AXI adapters just before each interface is exported to the FPGA AI Suite IP (so that outside of the Platform Designer system it can be connected to FPGA AI Suite IP). Clock crossing is also handled inside of `board.qsys`. For example, the host interface must be brought to the DDR clock to talk with the FPGA AI Suite IP CSR.

There are three clock domains: host clock, DDR clock, and the FPGA AI Suite IP clock. The PCIe logic runs on the host clock. FPGA AI Suite DMA and the platform adapters run on the DDR clock. The rest of FPGA AI Suite IP runs on the FPGA AI Suite IP clock.

FPGA AI Suite IP protocols:

- Readers and Writers: 512-bit data (width configurable), 32-bit address AXI4 interface, 16-word max burst (width fixed).
- CSR: 32-bit data, 11-bit address

Figure 3. FPGA AI Suite Example Design Top Level



The `board.qsys` block contains two major elements; the `pcie_ed.qsys` block and the `mem.qsys` blocks. The `pcie_ed.qsys` block interfaces between the host PCIe data and the `mem.qsys` blocks. The `mem.qsys` blocks interface between DDR memory, the readers/writers, and the host read/write channels.

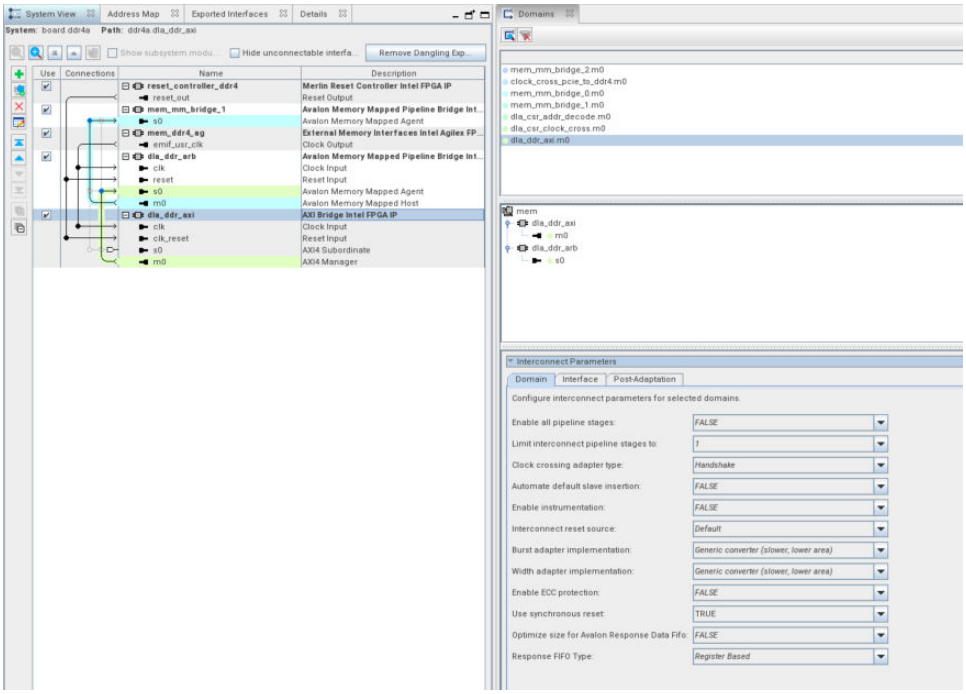
- Host read is used to read data from DDR memory and send it to the host.
- Host write is used to read data from the host into DDR memory.
- The MMIO interface performs several functions:
  - DDR read and write transactions are initiated by the host via the MMIO interface
  - Reading from the AFU ID block. The AFU ID block identifies the AFU with a unique identifier and is required for the OPAE driver.
  - Reading/writing to the DLA DMA CSRs where each instance has its own CSR base address.

Note: Avalon MM/AXI4 adapters in Platform Designer might not close timing.

Platform Designer optimizes for area instead of f_{MAX} by default, so you might need to change the interconnect settings for the inferred Avalon MM/AXI4 adapter. For example, we made some changes as shown in the following figure.

Figure 4. Adjusting the Interconnect Settings for the Inferred Avalon MM/AXI4 Adapter to Optimize for f_{MAX} Instead of Area.

Note: This enables timing closure on the DDR clock.



To access the view in the above figure:

- Within the Platform Designer GUI choose **View -> Domains**. This brings up the **Domains** tab in the top-right window.
- From there, choose an interface (for example, `dla_ddr_axi`).
- For the selected interface, you can adjust the interconnect parameters, as shown on the bottom-right pane.
- In particular, we needed to change **Burst adapter implementation** from **Generic converter (slower, lower area)** to **Per-burst-type converter (faster, higher area)** to close timing on the DDR clock.

This was the only change needed to close timing, however it took several rounds of experimentation to determine this was the setting of importance. Depending on your system, other settings might need to be tweaked.

7.2.1. [PCIE] PLL Adjustment

The design example build script adjusts the PLL driving the FPGA AI Suite IP clock based on the f_{MAX} that the Quartus Prime compiler achieves.

A fully rigorous production-quality flow would re-run timing analysis after the PLL adjustment to account for the small possibility that change in PLL frequency might cause a change in clock characteristics (for example, jitter) that cause a timing failure. A production design that shares the FPGA AI Suite IP clock with other system components might target a fixed frequency and skip PLL adjustment.

8. [OFS-PCIE] Getting Started with Open FPGA Stack (OFS) for PCIe Attach Design Examples

Before starting with the FPGA AI Suite OFS for PCIe attach design example, ensure that you have followed all the installation instructions for the FPGA AI Suite compiler and IP generation tools and completed the design example prerequisites as provided in the *FPGA AI Suite Getting Started Guide*.

The FPGA AI Suite OFS for PCIe attach design example supports Agilex 7 PCIe Attach OFS.

Open FPGA Stack (OFS) Requirements

For Open FPGA Stack (OFS) support, ensure that you have completed the OFS installation and configuration for your board, including the Open Programmable Acceleration Engine (OPAE) and Device Feature List (DFL) as outlined in the *Agilex 7 PCIe Attach OFS Workload Development Guide*.

Additionally, you might need additional environment configuration such as development permissions such as those provided in the `setup_persmissions.sh` script provided by the oneAPI Accelerator Support Package (ASP).

Additional Agilex 7 FPGA I-Series Development Kit Configuration

For the Agilex 7 FPGA I-Series Development Kit, the development kit provides a single 16 GB DIMM that you must replace with two 8 GB DIMMs in the board DIMM Sockets.

This design example was developed and tested to work with the following DIMMs:

- Micron MTA8ATF1G64AZ-2G6E1
- Hynix HMA81GU6JJR8N-VK

Related Information

- Agilex 7 PCIe Attach OFS documentation
- Agilex 7 PCIe Attach OFS Workload Development Guide

8.1. [OFS-PCIE] Building the FPGA AI Suite Runtime

The FPGA AI Suite OFS for PCIe Attach design example `runtime` directory contains the source code for the OpenVINO plugins and the `dla_benchmark` program. The CMake tool manages the overall build flow to build the FPGA AI Suite runtime plugin.

8.1.1. [OFS-PCIE] CMake Targets

The top level CMake build target is the FPGA AI Suite runtime plugin shared library, `libcoreDLARuntimePlugin.so`. It will not be built if the target is the software reference. Details on how to target one of the example design boards or the software emulation are specified in [OFS-PCIE] Build Options on page 39. The source files used to build the `libcoreDLARuntimePlugin.so` target are located under the following directories:

- `runtime/plugin/src/`
- `runtime/coredla_device/src/`

The flow also builds additional targets as dependencies for the top-level target. The most significant additional targets are as follows:

- The Input and Output Layout Transform library, `libdliaPluginIOTransformations.a`. The sources for this target are under `runtime/plugin/io_transformations/`.

8.1.2. [OFS-PCIE] Build Options

To build the runtime for the OFS for PCIe Attach design example:

1. Ensure that you have created a working directory as described in "Creating a Working Directory" in the *FPGA AI Suite Getting Started Guide*.
2. Ensure that the `OPAE_SDK_ROOT` environment variable is set in your build environment. For example, `export OPAE_SDK_ROOT=/usr/`.
3. Run one of the following sets of commands to build the runtime, depending on your Agilex 7 PCIe Attach OFS board:
   - Agilex 7 FPGA I-Series Development Kit

     ```
     cd $COREDLA_WORK/runtime

     ./build_runtime.sh -target_agx7_i_dk
     ```

   - Intel FPGA SmartNIC N6001-PL Platform

     ```
     cd $COREDLA_WORK/runtime

     ./build_runtime.sh -target_agx7_n6001
     ```

   For other `build_runtime.sh` options, refer to [PCIE] Build Options on page 14.

FPGA AI Suite hardware is compiled to include one or more IP instances, with the same architecture for all instances. Each instance accesses data from a unique bank of DDR.

The Agilex 7 FPGA I-Series Development Kit and Intel FPGA SmartNIC N6001-PL Platform both have four DDR banks (two onboard and two DIMM slots) and support up to four FPGA AI Suite IP instances. Install the required DIMMs into each slot that is used. For example, to support four FPGA AI Suite IP instances, you must have both DIMM slots fitted with 8 GB DIMMs. To support two FPGA AI Suite IP instances, you must have the first DIMM slot fitted with an 8GB DIMM.

The four DDR banks are ordered as follows:

1. Onboard DDR 0
2. DIMM slot 0
3. Onboard DDR 1
4. DIMM slot 1

Each DIMM slot can support two FPGA AI Suite IP instances.

The runtime automatically adapts to the correct number of instances.

If the FPGA AI Suite runtime uses two or more instances, then the image batches are divided between the instances to execute two or more batches in parallel on the FPGA device.

8.2. [OFS-PCIE] Running the Design Example Demonstration Applications

This section describes the steps to run the demonstration application and perform accelerated inference using the OFS for PCIe attach design example.

8.2.1. [OFS-PCIE] Setup the OFS Environment for the FPGA Device

Before you can program the FPGA device with the OFS for PCIe attach design example, you must set up the FPGA device with the OFS framework components and ensure that the OPAE drivers on the host system run correctly.

These steps must be done whenever the system hosting the FPGA board is power-cycled or soft-rebooted.

To set up the FPGA device:

1. Ensure that the PCIe bifurcation BIOS setting on the host machine that hosts the FPGA card is set as follows, depending on the target board:
   - Agilex 7 FPGA I-Series Development Kit: x8
   - Intel FPGA SmartNIC N6001-PL Platform: Auto
2. Program the FPGA devices with the `.sof` file for the OFS 2024.2-1 slim FIM for your board:
   - Agilex 7 FPGA I-Series Development Kit: https://github.com/OFS/ofs-agx7-pcie-attach/releases/download/ofs-2024.2-1/iseries-dk-slimfim-images_ofs-2024-2-1.tar.gz
   - Intel FPGA SmartNIC N6001-PL Platform: https://github.com/OFS/ofs-agx7-pcie-attach/releases/download/ofs-2024.2-1/n6001-slimfim-images_ofs-2024-2-1.tar.gz

   Program the FPGA with the following command:

   ```
   quartus_pgm -c 1 -m jtag \
               -o "p;<path to the sof file>/ofs_top.sof@1"
   ```
3. Soft-reboot the machine with the following command:

   ```
   sudo reboot
   ```

A soft reboot is required whenever you program the FPGA with a `.sof` file (SRAM object file) so that the PCIe host can reenumerate the attached devices. A hard reboot or power cycle would require you to reprogram the FPGA device with the earlier command.

4. If you want to use a non-root user to run inference on the FPGA board, complete the following steps:
   a. Set user process resource limits as follows:
      i. Create a rule file at `/etc/security/limits.d/90-intel-fpga-ofs-limits.conf` with the following content:

```
soft memlock unlimited

hard memlock unlimited
```

      ii. Log out of your current session and log back in.
      iii. Run the `ulimit -l` command to ensure that limits are set to `unlimited`.
   b. Enable huge pages help improve the performance of DMA operations between host and FPGA device. Enable huge pages as follows:
      i. Create a rule file at `/etc/sysctl.d/intel-fpga-ofs-sysctl.conf` with the following content:

```
vm.nr_hugepages = 2048
```

      ii. Create a rule file at `/sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages` with the following content:

```
2048
```

   c. Set the permissions for the OFS device feature list (DFL) framework as follows:
      i. Create a rule file at `/etc/udev/rules.d` named `90-intel-fpga-ofs.rules` with the following content:

```
KERNEL=="dfl-fme.[0-9]", ACTION=="add|change", GROUP="root",
MODE="0666", RUN+="/bin/bash -c 'chmod 0666 %S%p/errors/ /dev/%k'"

KERNEL=="dfl-port.[0-9]", ACTION=="add|change", GROUP="root",
MODE="0666", RUN+="/bin/bash -c 'chmod 0666 %S%p/dfl/userclk/
frequency %S%p/errors/* /dev/%k'"
```

      Ensure you enter the content as two lines only. The lines are line-wrapped only due to document formatting restrictions.
      ii. Run the following commands:

```
sudo udevadm control --reload

sudo udevadm trigger /dev/dfl-fme.0

sudo udevadm trigger /dev/dfl-port.0
```

   d. Set the permissions for userspace I/O (UIO) devices as follows:
      i. Create a rule file at `/etc/udev/rules.d` named `uio.rules` with the following content:

```
SUBSYSTEM=="uio" KERNEL=="uio*" MODE="0666"
```

ii. Run the following commands:

```
sudo udevadm control --reload

sudo udevadm trigger --subsystem-match=uio --settle
```

e. Initialize the OPAE SDK.

You must initialize the OPAE SDK after every system power cycle or soft reboot. You can make this initialization persistent by using a `systemd` startup service.

To initialize the OPAE SDK:

i. Determine the PCIe B:d.F (system, bus, device, function) of your board by running the `fpgainfo` command:

```
sudo fpgainfo fme
```

In the output, look for a line similar to the following line:

```
PCIe s:b:d.f                      : 0000:03:00.0
```

ii. Assign the first SR-IOF virtual function to the FPGA board with the following command:

```
sudo pci_device s:b:d.f vf 1
```

iii. Initialize `opae.io` with the following command:

```
sudo opae.io init -d s:b:d.1 <your user name>
```

Note: The original function (*f* in *s:b:d.f*) value that the `fpgainfo` command reported is replaced here by `1`.

5. Ensure that the `OPAE_PLATFORM_ROOT` environment variable points to your OFS FPGA interface manager (FIM) `pr_build_template` directory.

8.2.2. [OFS-PCIE] Exporting Trained Graphs from Source Frameworks.

Before running any demonstration application, you must convert the trained model to the Inference Engine format (`.xml, .bin`) with the OpenVINO Model Optimizer.

For details on creating the `.bin/.xml` files, refer to the *FPGA AI Suite Getting Started Guide*.

8.2.3. [OFS-PCIE] Compiling Exported Graphs Through the FPGA AI Suite

The network as described in the `.xml` and `.bin` files (created by the Model Optimizer) is compiled for a specific FPGA AI Suite architecture file by using the FPGA AI Suite compiler.

The FPGA AI Suite compiler compiles the network and exports it to a `.bin` file that uses the same `.bin` format as required by the OpenVINO Inference Engine.

This `.bin` file created by the compiler contains the compiled network parameters for all the target devices (FPGA, CPU, or both) along with the weights and biases. The inference application imports this file at runtime.

The FPGA AI Suite compiler can also compile the graph and provide estimated area or performance metrics for a given architecture file or produce an optimized architecture file.

For more details about the FPGA AI Suite compiler, refer to the *FPGA AI Suite Compiler Reference Manual*.

8.2.4. [OFS-PCIE] Compiling the OFS for PCIe Attach Design Example

To build example design bitstreams, you must have a license that permits bitstream generation for the IP, and have the correct version of Quartus Prime software installed. Use the `dla_build_example_design.py` utility to create a bitstream.

For more details about this command, the steps it performs, and advanced command options, refer to Build Script and to the *FPGA AI Suite Getting Started Guide*.

Before running the `dla_build_example_design.py` utility, ensure that the `OPAE_PLATFORM_ROOT` environment variable points to your OFS FPGA interface manager (FIM) `pr_build_template` directory. If you do not want to compile your own FIM, you can get prebuilt OFS FIM binaries for boards supported by the Agilex 7 OFS for PCIe Attach reference shells on GitHub at the following URL:

```
https://github.com/OFS/ofs-agx7-pcie-attach/releases/
```

The FPGA AI Suite OFS for PCIe Attach design example is based on the OFS 2024.2-1 release of the reference shells.

The `dla_build_example_design.py` utility generates a wrapper that wraps one or more FPGA AI Suite IP instances along with adapters required to connect to the OFS slim FIM.

Important: The OFS FIM uses FPGA resources as well as the FPGA AI Suite IP instances. Keep the FPGA resource limitations in mind when deciding on how many FPGA AI Suite IP instances to use.

Get an estimate of the FPGA resource required for a single FPGA AI Suite IP instance by using the `--fanalyze-area` option of the `dla_compiler`. Use the single instance values to determine the resources required for the number of instances that you want. For more details, see the `--fanalyze-area` option description in *FPGA AI Suite Compiler Reference Manual*.

For the OFS for PCIe Attach design example, the `dla_build_example_design.py` utility supports the following options:

Option	Description
`-a, --archs`	Path to FPGA AI Suite IP Architecture Description File
`--build-dir`	Path to hardware build directory where BSP infrastructure and generated RTL will be located.
`--build`	Option to perform compilation of the design using Quartus Prime after instantiation (default: False).
`-d, --archs-dir`	Path to directory that contains Architecture Description Files for you to interactively choose from (alternative to '-a')
`-ed, --example-design-id`	To build for the Agilex 7 FPGA I-Series Development Kit, specify AGX7_ISERIES_OFS_PCIE.

continued...

Option	Description
	To build for theIntel FPGA SmartNIC N6001-PL Platform, specify `AGX7_N6001_OFS_PCIE`.
`-n, --num-instances`	Number of IP instances to build (default: 2). To build for the Agilex 7 FPGA I-Series Development Kit, this value can be 1, 2, 3, or 4. To build for theIntel FPGA SmartNIC N6001-PL Platform, this value can be 1 or 2.
`--num-paths`	Number of top critical paths to report after compiling the design (default: `2000`).
`-q, --quiet`	Run script quietly without printing the output of underlying scripts to the terminal.
`-qor-modules`	List of internal modules (instance names) from inside the FPGA AI Suite IP to include in the QoR summary report.
`-s, --seed`	Quartus Prime fitter initial placement seed to use when compiling the design (default: 1).
`--unlicensed/licensed`	This option is passed to the `dla_create_ip` tool to tell the tool to generate either an unlicensed or licensed copy of the FPGA AI Suite IP: • **Unlicensed IP:** Unlicensed IP has a limit of 10000 inferences. After 10000 inferences, the unlicensed IP refuses to perform any additional inference and a bit in the CSR is set. For details about the CSR bit, refer to DMA Descriptor Queue in *FPGA AI Suite IP Reference Manual*. • **Licensed IP:** Licensed IP has no inference limitation. If you do not have a license but generate licensed IP, Quartus Primesoftware cannot generate a bitstream. If neither option is specified, then the `dla_create_ip` tool queries the `lmutil` license manager to determine the correct option.

To generate an FPGA bitstream for the OFS for PCIe Attach design example for the Agilex 7 FPGA I-Series Development Kit with two FPGA AI Suite IP instances, run the following commands:

```
cd $COREDLA_WORK

$COREDLA_ROOT/bin/dla_build_example_design.py \
  -n 2 \
  --archs=$COREDLA_ROOT/example_architectures/AGX7_Generic.arch \
  --build \
  --build-dir=build_generic_2inst \
  --example-design-id="AGX7_ISERIES_OFS_PCIE" \
  --seed=1
```

This command generates a green bitstream (GBS) file called `AGX7_Generic.gbs` that can be found in the `$COREDLA_WORK/build_generics_2inst/` folder.

8.2.5. [OFS-PCIE] Programming the FPGA Green Bitstream

Program the FPGA AI Suite design example green bitstream (`.gbs`) to the devices with the following command:

```
fpgaconf -V <path_to_design_example_green_bitstream(.gbs)_file>
```

For example, to program the FPGA device with the green bitstream file generated by the earlier example command, run the following command:

```
fpgaconf -V $COREDLA_WORK/build_generics_2inst/AGX7_Generic.gbs
```

8.2.6. [OFS-PCIE] Performing Accelerated Inference with the `dla_benchmark` application

You can use the `dla_benchmark` demonstration application included with the FPGA AI Suite runtime to benchmark the performance of image classification networks.

8.2.6.1. [OFS-PCIE] Inference on Image Classification Graphs

The demonstration application requires the OpenVINO device flag to be either `HETERO:FPGA,CPU` for heterogeneous execution or `HETERO:FPGA` for FPGA-only execution.

The `dla_benchmark` demonstration application runs five inference requests (batches) in parallel on the FPGA, by default, to achieve optimal system performance. To measure steady state performance, you should run multiple batches (using the `niter` flag) because the first iteration is significantly slower with FPGA devices.

The `dla_benchmark` demonstration application also supports multiple graphs in the same execution. You can place more than one graphs or compiled graphs as input, separated by commas.

Each graph can have either a different input dataset or use a commonly shared dataset among all graphs. Each graph requires an individual `ground_truth_file` file, separated by commas. If some `ground_truth_file` files are missing, the `dla_benchmark` continues to run and ignore the missing ones.

When multi-graph is enabled, the `-niter` flag represents the number of iterations for each graph, so the total number of iterations becomes `-niter` × number of graphs.

The `dla_benchmark` demonstration application switches graphs after submitting `-nireq` requests. The request queue holds the number of requests up to `-nireq` × number of graphs. This limit is constrained by the DMA CSR descriptor queue size (64 per hardware instance).

The board you use determines the number of instances that you can compile the FPGA AI Suite hardware for. For the Agilex 7 FPGA I-Series Development Kit and Intel FPGA SmartNIC N6001-PL Platform, you can compile up to four instances with the same architecture on all instances. Some large architecture might not fit on the board for four instances, such as `AGX7_Performance_Giant`.

Each instance accesses one of the DDR banks on the board and executes the graph independently. This optimization enables multiple batches to run in parallel, limited by the number of DDR banks available. Each inference request created by the demonstration application is assigned to one of the instances in the FPGA plugin.

To enable memory-mapped device (MMD) debug messages when you run the `dla_benchmark` demonstration application. set the `MMD_ENABLE_DEBUG` environment variable as follows:

```
MMD_ENABLE_DEBUG=1
```

Also, you can test full DDR write and read back functionality when the `dla_benchmark` demonstration application runs by setting the `COREDLA_RUNTIME_MEMORY_TEST` environment variable as follows:

```
COREDLA_RUNTIME_MEMORY_TEST=1
```

To ensure that batches are evenly distributed between the instances, you must choose an inference request batch size that is a multiple of the number of FPGA AI Suite instances. For example, with two instances, specify the batch size as six (instead of the OpenVINO default of five) to ensure that the experiment meets this requirement.

The example usage that follows has the following assumptions:

- A Model Optimizer IR `.xml` file is in `demo/models/public/resnet-50-tf/FP32/`
- An image set is in `demo/sample_images/`
- The board is programmed with a bitstream that corresponds to `AGX7_Performance.arch`

```
binxml=$COREDLA_ROOT/demo/models/public/resnet-50-tf/FP32

imgdir=$COREDLA_ROOT/demo/sample_images

cd $COREDLA_ROOT/runtime/build_Release

./dla_benchmark/dla_benchmark \
   -b=1 \
   -m $binxml/resnet-50-tf.xml \
   -d=HETERO:FPGA,CPU \
   -i $imgdir \
   -niter=4 \
   -plugins ./plugins.xml \
   -arch_file $COREDLA_ROOT/example_architectures/AGX7_Performance.arch \
   -api=async \
   -groundtruth_loc $imgdir/TF_ground_truth.txt \
   -perf_est \
   -nireq=8 \
   -bgr
```

8.2.6.2. [OFS-PCIE] Inference on Object Detection Graphs

To enable the accuracy checking routine for object detection graphs, you can use the `-enable_object_detection_ap=1` flag.

This flag lets the `dla_benchmark` calculate the mAP and COCO AP for object detection graphs. Besides, you need to specify the version of the YOLO graph that you provide to the `dla_benchmark` through the `–yolo_version` flag. Currently, this routine is known to work with YOLOv3 (graph version is `yolo-v3-tf`) and TinyYOLOv3 (graph version is `yolo-v3-tiny-tf`).

8.2.6.2.1. [OFS-PCIE] The mAP and COCO AP Metrics

Average precision and average recall are averaged over multiple Intersection over Union (IoU) values.

Two metrics are used for accuracy evaluation in the `dla_benchmark` application. The mean average precision (mAP) is the challenge metric for PASCAL VOC. The mAP value is averaged over all 80 categories using a single IoU threshold of 0.5. The COCO

AP is the primary challenge for object detection in the Common Objects in Context contest. The COCO AP value uses 10 IoU thresholds of .50:.05:.95. Averaging over multiple IoUs rewards detectors with better localization.

8.2.6.2.2. [OFS-PCIE] Specifying Ground Truth

The path to the ground truth files is specified by the flag `–groundtruth_loc`.

The validation dataset is available on the COCO official website.

The `dla_benchmark` application currently allows only plain text ground truth files. To convert the downloaded JSON annotation file to plain text, use the `convert_annotations.py` script.

8.2.6.2.3. [OFS-PCIE] Example of Inference on Object Detection Graphs

The example that follows makes the following assumptions:

- The Model Optimizer IR `graph.xml` for either YOLOv3 or TinyYOLOv3 is in the current working directory.

  Model Optimizer generates an FP32 version and an FP16 version. Use the FP32 version.
- The validation images downloaded from the COCO website are placed in the `./mscoco-images` directory.
- The JSON annotation file is downloaded and unzipped in the current directory.

To compute the accuracy scores on many images, you can usually increase the number of iterations using the flag `-niter` instead of a large batch size `-b`. The product of the batch size and the number of iterations should be less than or equal to the number of images that you provide.

```
cd $COREDLA_ROOT/runtime/build_Release

python ./convert_annotations.py ./instances_val2017.json \
   ./groundtruth

./dla_benchmark/dla_benchmark \
   -b=1 \
   -niter=5000 \
   -m=./graph.xml \
   -d=HETERO:FPGA,CPU \
   -i=./mscoco-images \
   -plugins=./plugins.xml \
   -arch_file=../../example_architectures/AGX7_Performance.arch \
   -yolo_version=yolo-v3-tf \
   -api=async \
   -groundtruth_loc=./groundtruth \
   -nireq=8 \
   -enable_object_detection_ap \
   -perf_est \
   -bgr
```

8.2.6.3. [OFS-PCIE] Additional dla_benchmark Options

The `dla_benchmark` tool is part of the example design and the distributed runtime includes full source code for the tool.

Table 8. Command Line `dla_benchmark` Options

Command Option	Description
`-nireq=<N>`	This option controls the number of simultaneous inference requests that are sent to the FPGA. Typically, this should be at least twice the number of IP instances; this ensures that each IP can execute one inference request while `dla_benchmark` loads the feature data for a second inference request to the FPGA-attached DDR memory.
`-b=<N>` `--batch-size=<N>`	This option controls the batch size. A batch size greater than 1 is created by repeating configuration data for multiple copies of the graph. A batch size of 1 is typically best for latency System throughput for small graphs, when inference operations are offloaded from a CPU to an FPGA, may improve by using a batch greater than 1. On very small graphs, IP throughput may also improve when using a batch greater than 1. The default value is 1.
`-niter=<N>`	Number of batches to run. Each batch has a size specified by the `--batch-size` option. The total number of images processed is the product of the `--batch-size` option value multiplied by the `-niter` option value.
`-d=<STRING>`	Using `-d=HETERO:FPGA`, CPU causes `dla_benchmark` to use the OpenVINO heterogeneous plugin to execute inference on the FPGA, with fallback to the CPU for any layers that cannot go to the FPGA. Using `-d=HETERO:CPU` or `-d=CPU` executes inference on the CPU, which may be useful for testing the flow when an FPGA is not available. Using `-d=HETERO:FPGA` may be useful for ensuring that all graph layers are accelerated on the FPGA (and an error is issued if this is not possible).
`-arch_file=<FILE>` `--arch=<FILE>`	This specifies the location of the .arch file that was used to configure the IP on the FPGA. The `dla_benchmark` will issue an error if this does not match the `.arch` file used to generate the IP on the FPGA.
`-m=<FILE>` `--network_file=<FILE>`	This points to the XML file from OpenVINO Model Optimizer that describes the graph. The BIN file from Model Optimizer must be kept in the same directory and same filename (except for the file extension) as the XML file.
`-i=<DIRECTORY>`	This points to the directory containing the input images. Each input file corresponds to one inference request. The files are read in order sorted by filename; set the environment variable VERBOSE=1 to see details describing the file order.
`-api=[sync|async]`	The `-api=async` option allows `dla_benchmark` to fully take advantage of multithreading to improve performance. The `-api=sync` option may be used during debug.
`-groundtruth_loc=<FILE>`	Location of the file with ground truth data. If not provided, then `dla_benchmark` will not evaluate accuracy. This may contain classification data or object detection data, depending on the graph.

continued...

Command Option	Description
`-yolo_version=<STRING>`	This option is used when evaluating the accuracy of a YOLOv3 or TinyYOLOv3 object detection graph. The options are `yolo-v3-tf` and `yolo-v3-tiny-tf`.
`-enable_object_detection_ap`	This option may be used with an object detection graph (YOLOv3 or TinyYOLOv3) to calculate the object detection accuracy.
`-bgr`	When used, this flag indicates that the graph expects input image channel data to use BGR order.
`-plugins_xml_file=<FILE>`	*Deprecated:* This option is deprecated and will be removed in a future release. Use the `-plugins` option instead. This option specifies the location of the file specifying the OpenVINO plugins to use. This should be set to `$COREDLA_ROOT/runtime/plugins.xml` in most cases. If you are porting the design to a new host or doing other development, it may be necessary to use a different value.
`-plugins=<FILE>`	This option specifies the location of the file that specifies the OpenVINO plugins to use. The default behavior is to read the `plugins.xml` file from the `runtime/` directory. This runs inference on the FPGA device. If you want to run inference using the emulation model, specify `-plugins=emulation`. If you are porting the design to a new host or doing other development, you might need to use a different value.
`-mean_values=<input_name[mean_values]>`	Uses channel-specific mean values in input tensor creation through the following formula: $\frac{\text{input} - \text{mean}}{\text{scale}}$. The Model Optimizer mean values are the preferred choice and the mean values defined by this option serve as fallback values.
`-scale_values=<input_name[scale_values]>`	Uses channel-specific scale values in input tensor creation through the following formula: $\frac{\text{input} - \text{mean}}{\text{scale}}$. The Model Optimizer scale values are the preferred choice and the scale values defined by this option serve as fallback values.
`-pc`	This option reports the performance counters for the CPU subgraphs, if there is any. No sorting is done on the report.
`-pcsort=[sort|no_sort|simple_sort]`	This option reports the performance counters for the CPU subgraph and sets the sorting option for the performance counter report: • `sort`: Report is sorted by operation time cost • `no_sort`: Report is not sorted • `simple_sort`: Report is sorted by opts time cost but print only executed operations
`-save_run_summary`	Collect performance metrics during inference. These metrics can help you determine how efficient an architecture is at executing a model. For more information, refer to [PCIE] The dla_benchmark Performance Metrics on page 22.

8.2.6.4. [OFS-PCIE] The `dla_benchmark` Performance Metrics

The `-save_run_summary` option makes the `dla_benchmark` demonstration application collect performance metrics during inference. These metrics can help you determine how efficient an architecture is at executing a model.

Note: The `dla_benchmark` application provides throughput in "frames per second". The time per frame (latency) is 1/throughput.

Statistic	Description
Count	The number of times interference was performed. This is set by the `-niter` option.
System duration	The total time between when the first inference request was made to when the last request was finished, as measured by the host program.
IP duration	The total time the spent-on inference. This is reported by the IP on the FPGA.
Latency	The median time of all inference requests made by the host. This includes any overhead from OpenVINO or the FPGA AI Suite runtime.
System throughput	The total throughput of the system, including any OpenVINO or FPGA AI Suite runtime overhead.
Number of hardware instances	The number of IP instances on the FPGA.
Number of network instances	The number graphs that the IP processes in parallel.
IP throughput per instance	The throughput of a single IP instance. This is reported by the IP on the FPGA.
IP throughput per f_{MAX} per instance	The **IP throughput per instance** value scaled by the **IP clock frequency** value.
IP clock frequency	The clock frequency, as reported by the IP running on the FPGA device. The `dla_benchmark` application treats this value as the IP core f_{MAX} value.
Estimated IP throughput per instance	The estimated per-IP throughput, as estimated by the `dla_compiler` command with the `--fanalyze-performance` option.
Estimated IP throughput per fmax per instance	The **Estimated IP throughput per instance** value scaled by the compiler f_{MAX} estimate.

8.2.6.4.1. [OFS-PCIE] Interpreting System Throughput and Latency Metrics

The **System throughput** and **Latency** metrics are measured by the host through the OpenVINO API. These measurements include any overhead that is incurred by both the API and the FPGA AI Suite runtime. They also account for any time spent waiting to make inference requests and the number of available instances.

In general, the system throughput is defined as follows:

$$\text{System Throughput} = \frac{\text{Batch Size} \times \text{Images per Batch}}{\text{Latency}}$$

The **Batch Size** and **Images Per Batch** values are set by the `--batch-size` and `-niter` options, respectively.

For example, consider when `-nireq=1` and there is a single IP instance. The **System throughput** value is approximately the same as the **IP-reported throughput** value because the runtime can perform only one inference at a time. However, if both the `-`

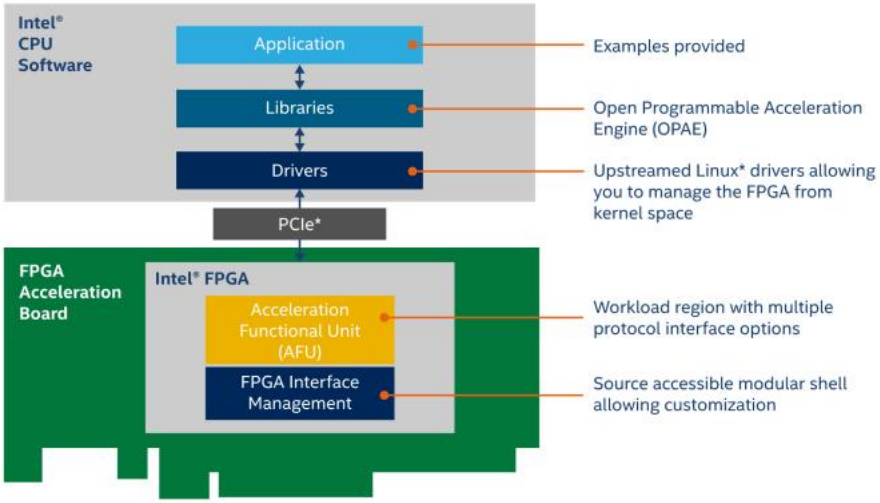
`nireq` and the number of IP instances is greater than one, the runtime can perform requests in parallel. As such, the total system throughput is greater than the individual IP throughput.

In general, the `-nireq` value should be *twice* the number of IP instances. This setting enables the FPGA AI Suite runtime to pipeline inferences requests, which allows the host to prepare the data for the next request while an IP instance is processing the previous request.

9. [OFS-PCIE] Design Example Components

9.1. [OFS-PCIE] Hardware Components

The FPGA AI Suite OFS for PCIe attach design example is based on OFS (Open FPGA Stack). The following diagram shows a high-level view of a typical OFS system/A software stack runs on the host CPU (applications, OFS libraries, FPGA drivers) that connects via a PCIe connection to an FPGA board.



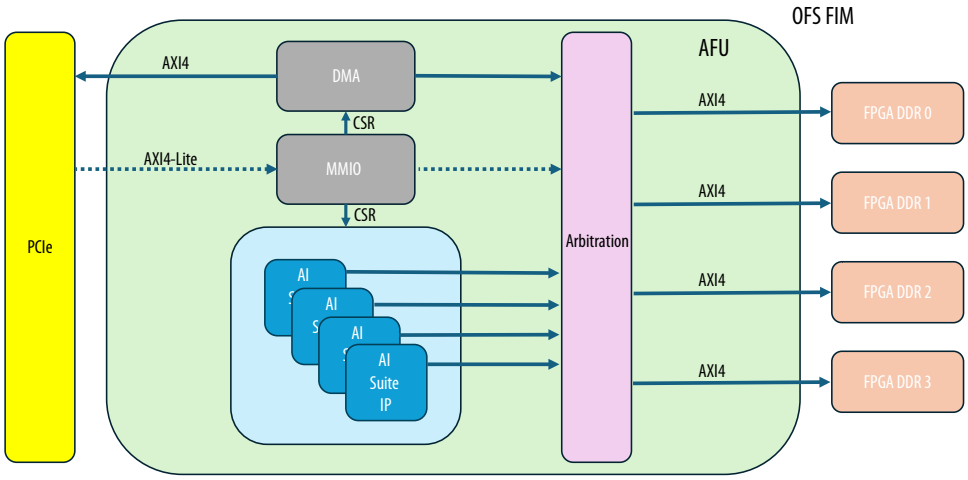
The design example hardware implementation, sits in the AFU (acceleration functional unit) region and uses the OFS FIM (FPGA interface manager) to connect to both the host CPU via a PCIe connection and also to the on-board DDR4 memory.

For more information about OFS, refer to the Open FPGA Stack (OFS) documentation.

The Agilex 7 FPGA I-Series Development Kit has four banks of DDR4 memory on board: two banks are soldered-on 8 GB of DDR4 memory each, two banks are DIMM slots for DDR4 DIMMs. For the design example, the DIMMs must also be 8GB in size each to match the soldered DDR4 memories. Larger size memories are currently not supported.

The Intel FPGA SmartNIC N6001-PL Platform has four banks of DD4 memory on board. All banks are soldered-on 4 GB DDR4 DIMMs.

The following diagram shows how the OFS for PCIe attach design example is implemented within the OFS AFU:



The OFS FIM provides the following external interfaces:

- Towards the host CPU via the PCIe connection two interfaces are exposed:
  - A high-throughput AXI4 agent that initiates reads and writes from the FPGA fabric over the PCIe connection to the host CPU memory that is used by the DMA controller.
  - An AXI4-Lite host so that initiates reads and writes from the host CPU to the FPGA fabric. This interface is used, mainly for configuring the FPGA AI Suite IP CSRs, the DMA CSR, and unaligned MMIO accesses of the FPGA DDRs.
- Towards the on-board FPGA DDR banks.

  Four AXI4 agents, each of which connects to one DDR memory bank, connect to arbitration logic to enable the following paths into the DDR memory banks:
  - DMA controller to DDR
  - MIMO to DDR
  - FPGA AI Suite IP to DDR.

That design example has three different clock domains:

- PCIe core clock (500 MHz)
- DDR core clock (333 MHz)
- User clock (configurable, typically around 600 MHz)

At the entry from the PCIe interface to the AFU, there are clock crossers from the PCIe core clock into the DDR core clock. The DDR core clock is used in all the DMA, CSR and arbitration logic to help benefit timing closure, while always still maintaining the full bandwidth to all four FPGA DDR banks.

The FPGA AI Suite IP runs on a configurable user clock and is set accordingly after the Quartus compile to match the maximum supported frequency of the IP in its chosen configuration. Typically, this frequency is 600 MHz and above for designs with only one FPGA AI Suite IP instance and just below 600 MHz for four FPGA AI Suite IP instances.

Caution: The Intel FPGA SmartNIC N6001-PL Platform card is designed for its specified power budget. If a majority of the FPGA DSPs on the device are operating at 600 MHz, you can exceed this power budget. Exceeding the power budget causes the power regulators on the card to shut the card down, which renders the card invisible on the PCIe bus.

If the card shuts down in this way, the host machine issues a kernel panic (in Linux) and either freezes or reboots automatically.

If this occurs, you must reduce the target frequency of the FPGA AI Suite IP or reduce the number of FPGA AI Suite IP instances in your instantiation of the design example.

9.2. [OFS-PCIE] Software Components

The OFS for PCIe attach design example contains a sample software stack for the runtime flow.

The following figure, *Software Stacks for FPGA AI Suite Inference*, shows the complete runtime stack.

The following components comprise the runtime stack:

- OpenVINO Toolkit 2023.3 LTS (Inference Engine, Heterogeneous Plugin)
- FPGA AI Suite runtime plugin
- OPAE driver 2.13

The design example contains the source files and Makefiles to build the FPGA AI Suite runtime plugin. The OpenVINO and OPAE components are external and must be manually preinstalled.

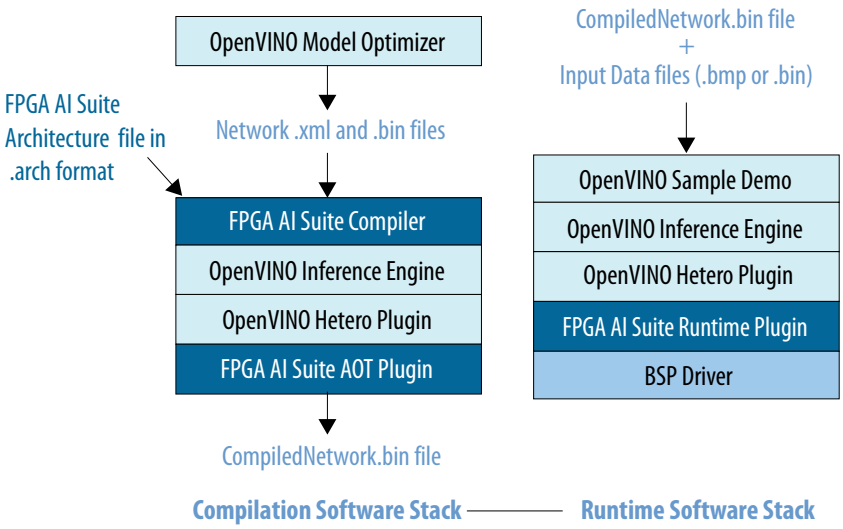
A separate flow compiles the AI network graph using the FPGA AI Suite compiler, as shown in figure *Software Stacks for FPGA AI Suite Inference* below as the Compilation Software Stack.

The compilation flow output is a single binary file called `CompiledNetwork.bin` that contains the compiled network partitions for FPGA and CPU devices along with the network weights. The network is compiled for a specific FPGA AI Suite architecture and batch size. This binary is created on-disk only when using the Ahead-Of-Time flow; when the JIT flow is used, the compiled object stays in-memory only.

An Architecture File describes the FPGA AI Suite IP architecture to the compiler. You must specify the same Architecture File to the FPGA AI Suite compiler and to the FPGA AI Suite PCIe Example Design build script (`dla_build_example_design.py`).

The runtime flow accepts the `CompiledNetwork.bin` file as the input network along with the image data files.

Figure 5. Software Stacks for FPGA AI Suite Inference



The runtime stack cannot program the FPGA with a bitstream. To build a bitstream and program the FPGA devices:

1. Compile the design example. For details, refer to Compiling the PCIe-based Design Example.
2. Program the device with the bitstream. For details, refer to [PCIE] Programming the FPGA Device (Agilex 7) on page 17.

To run inference through the OpenVINO Toolkit on the FPGA, set the OpenVINO device configuration flag (used by the heterogeneous Plugin) to FPGA or HETERO:FPGA,CPU.

9.2.1. [OFS-PCIE] OpenVINO FPGA Runtime Plugin

The FPGA runtime plugin uses the OpenVINO Inference Engine Plugin API.

The OpenVINO Plugin architecture is described in the *OpenVINO Developer Guide for Inference Engine Plugin Library*.

The source files are located under `runtime/plugin`. The three main components of the runtime plugin are the Plugin class, the Executable Network class, and the Inference Request class. The primary responsibilities for each class are as follows:

Plugin class

- Initializes the runtime plugin with an FPGA AI Suite architecture file which you set as an OpenVINO configuration key (refer to [PCIE] Running the Ported OpenVINO Demonstration Applications on page 23).
- Contains `QueryNetwork` function that analyzes network layers and returns a list of layers that the specified architecture supports. This function allows network execution to be distributed between FPGA and other devices and is enabled with the HETERO mode.
- Creates an executable network instance in one of the following ways:
  - Just-in-time (JIT) flow: Compiles a network such that the compiled network is compatible with the hardware corresponding to the FPGA AI Suite architecture file, and then loads the compiled network onto the FPGA device.
  - Ahead-of-time (AOT) flow: Imports a precompiled network (exported by FPGA AI Suite compiler) and loads it onto the FPGA device.

Executable Network Class

- Represents an FPGA AI Suite compiled network
- Loads the compiled model and config data for the network onto the FPGA device that has already been programmed with an FPGA AI Suite bitstream. For two instances of FPGA AI Suite, the Executable Network class loads the network onto both instances, allowing them to perform parallel batch inference.
- Stores input/output processing information.
- Creates infer request instances for pipelining multiple batch execution.

Infer Request class

- Runs a single batch inference serially.
- Executes five stages in one inference job – input layout transformation on CPU, input transfer to DDR, FPGA AI Suite FPGA execution, output transfer from DDR, output layout transformation on CPU.
- In asynchronous mode, executes the stages on multiple threads that are shared across all inference request instances so that multiple batch jobs are pipelined, and the FPGA is always active.

Related Information

OpenVINO Developer Guide for Inference Engine Plugin Library

9.2.2. [OFS-PCIE] FPGA AI Suite Runtime

The FPGA AI Suite runtime implements lower-level classes and functions that interact with the memory-mapped device (MMD). The MMD is responsible for communicating requests to the driver, and the driver connects to the BSP, and ultimately to the FPGA AI Suite IP instance or instances.

The runtime source files are located under `runtime/coredla_device`. The three most important classes in the runtime are the Device class, the GraphJob class, and the BatchJob class.

Device class

- Acquires a handle to the MMD for performing operations by calling `aocl_mmd_open`.
- Initializes a DDR memory allocator with the size of 1 DDR bank for each FPGA AI Suite IP instance on the device.
- Implements and registers a callback function on the MMD DMA (host to FPGA) thread to launch FPGA AI Suite IP for batch=1 after the batch input data is transferred from host to DDR.
- Implements and registers a callback function (interrupt service routine) on the MMD kernel interrupt thread to service interrupts from hardware after one batch job completes.
- Provides the `CreateGraphJob` function to create a GraphJob object for each FPGA AI Suite IP instance on the device.
- Provides the `WaitForDla(`*`instance id`*`)` function to wait for a batch inference job to complete on a given instance. Returns instantly if the number of batch jobs finished (that is, the number of jobs processed by interrupt service routine) is greater than number of batch jobs waited for this instance. Otherwise, the function waits until interrupt service routine notifies. Before returning, this function increments the number of batch jobs that have been waited for this instance.

GraphJob class

- Represents a compiled network that is loaded onto one instance of the FPGA AI Suite IP on an FPGA device.
- Allocates buffers in DDR memory to transfer configuration, filter, and bias data.
- Creates BatchJob objects for a given number of pipelines and allocates input and output buffers for each pipeline in DDR.

BatchJob class

- Represents a single batch inference job.
- Stores the DDR addresses for batch input and output data.
- Provides `LoadInputFeatureToDdr` function to transfer input data to DDR and start inference for this batch asynchronously.
- Provides `ReadOutputFeatureFromDdr` function to transfer output data from DDR. Must be called after inference for this batch is completed.

9.2.3. [OFS-PCIE] BSP Driver

The FPGA AI Suite runtime MMD software uses a driver supplied as part of the BSP to access and interact with the FPGA device.

The source files for the driver are in `runtime/coredla_device/mmd/agx7_ofs_pcie`. The source files contain classes for managing and accessing the FPGA device by using BSP functions for reading/writing to CSR, reading/writing to DDR, and handling kernel interrupts.

9.2.4. [OFS-PCIE] Software Interface to the BSP

The interface to the user-space portion of the BSP drivers is centralized in the `MmdWrapper` class, which can be found in the file `$COREDLA_ROOT/runtime/coredla_device/inc/mmd_wrapper.h`.

When porting the runtime to a new board, the team responsible for the new board support must ensure that each of the member functions in MmdWrapper calls into a board-specific implementation function. The team doing this will need to modify the runtime build process and adjacent code.

10. [HL-NO-DDR] Getting Started with the FPGA AI Suite DDR-Free Design Example

The FPGA AI Suite DDR-Free design example is provided with the FPGA AI Suite. Before starting with the FPGA AI-Suite DDR-Free design example, ensure that you have followed all the installation instructions for the FPGA AI Suite compiler and IP generation tools.

The DDR-Free design example is only validated for use with Quartus Prime Pro Edition Version 24.3.

10.1. [HL-NO-DDR] Hardware Requirements

This design example requires the following hardware:

- Agilex 7 FPGA I-Series Development Kit ES2 (DK-DEV-AGI027RBES)
- Intel FPGA Download Cable

10.2. [HL-NO-DDR] Software Requirements

This design example requires the following software:

- FPGA AI Suite
- Quartus Prime Programmer (either standalone or as part of Quartus Prime Design Suite).
- Quartus Prime System Console (either standalone or as part of Quartus Prime Design Suite).

Ensure that all the binaries included in the Quartus Prime Design Suite are added to your $PATH environment variable so that they can be called from any location.

When you have met these prerequisites, validate that the development kit is connected to the JTAG interface by using the `jtagconfig` utility provided by the Quartus Prime Design Suite. A successful confirmation of the JTAG connection looks like the following example output:

```
$ jtagconfig
1) AGI FPGA Development Kit [1-7.2]
 034BB0DD AGIB027R29A(.|B|C|R0|R1|R3)/..
 020D10DD VTAP10
```

11. [HL-NO-DDR] Running the Hostless DDR-Free Design Example

Procedure

To run the hostless DDR-free design example with a ResNet-18 PyTorch Model:

1. Download and prepare the ResNet-18 PyTorch Model with the OpenVINO Open Model Zoo tools with the following commands:

```
source ~/build-openvino-dev/openvino_env/bin/activate

omz_downloader --name resnet-18-pytorch \
    --output_dir $COREDLA_WORK/demo/models/

omz_converter --name resnet-18-pytorch \
    --download_dir $COREDLA_WORK/demo/models/ \
    --output_dir $COREDLA_WORK/demo/models/
```

   Important: The OpenVINO Open Model Zoo (OMZ) PyTorch models do not include a softmax operation at the end of the model.

2. Generate the parameter ROMs as `.mif` files by running the FPGA AI Suite compiler with the following command:

```
dlac \
  --batch-size=1 \
  --network-file <path/to/graph> \
  --march $COREDLA_ROOT/example_architectures/
AGX7_Streaming_Ddrfree_Resnet18.arch \
  --foutput-format=open_vino_hetero \
  --o <compiler output .bin file name>  \
  --fplugin HETERO:FPGA \
  --dumpdir $COREDLA_WORK/resnet-18-dlac-out/
```

   The `.mif` files are created a subdirectory of the directory specified by the `--dumpdir` option. This subdirectory is called `parameter_rom`.

   For details about creating the `.mif` files required for DDR-free operation, refer to "Generating Artifacts for DDR-Free Operation" in the *FPGA AI Suite Compiler Reference Manual*.

3. Build the example design with the following command:

```
dla_build_example_design.py \
  -n 1 \
  --arch=$COREDLA_ROOT/example_architectures/
AGX7_Streaming_Ddrfree_Resnet18.arch \
  --build --build-dir=<path/to/build/dir> \
  --example-design-id=0_STREAMING \
  --seed=1 \
  --parameter-rom-dir $COREDLA_ROOT/resnet-18-dlac-out/parameter_rom/
```

   Building the example design creates the bitstream needed to program the FPGA device.

For more information about the `dla_build_example_design` command, refer to [PCIE] Build Script on page 26. .

4. Program the FPGA device with the Quartus Prime Programmer.

   The bitstream used to program the device is *<path/to/build/dir>*`/hw/output_files/top.sof`.

   Program the FPGA device with the following command:

   ```
   quartus_pgm -c 1 -m jtag -o "p;top.sof@1"
   ```

   For more information about the Quartus Prime Programmer, refer to *Quartus Prime Pro Edition User Guide: Programmer*.

5. Use the Quartus Prime System Console to run inference on the example design.

   Because this example design is hostless, operations that typically come from the host are performed through Quartus Prime System Console instead. For more information about the Quartus Prime System Console, refer to "Analyzing and Debugging Designs with System Console" in *Quartus Prime Pro Edition User Guide: Debug Tools*.

   Use the System Console to complete the following steps:

   a. Store input features in the FPGA on-chip memory.

   b. Prime the FPGA AI Suite IP registers for inference.

   c. Configure an ingress Modular Scatter-Gather DMA (mSGDMA) core to read the input features from on-chip memory and stream data into the FPGA AI Suite IP.

   d. Configure an egress mSGDMA core to stream data from the FPGA AI Suite IP into on-chip memory.

   e. Read the inference results from on-chip memory.

   The design example provides a System Console script to automate these operations for you. You can find the script in the `$CORDLA_ROOT/runtime/streaming/ed0_streaming_example` folder.

   To use the design example System Console script:

   a. Run the following command:

      ```
      system-console –-script=system_console_script.tcl <path-to-img.bin> \
                     <#-of-inferences> <output-channels> <output-height> \
                     <output-width>
      ```

      The design example Quartus Prime System Console script generates a file called `output.bin` that contains the raw inference results.

   b. (Optional) To measure the performance of the design example, run the following command:

      ```
      system-console –-script=system_console_perf.tcl <path-to-img.bin>
      ```

6. Postprocess the raw inference output for readability with the following command:

   ```
   python3 $COREDLA_ROOT/bin/streaming_post_processing.py <path-to-output.bin>
   ```

   This script cleans the raw output binary file by script some invalid bytes and storing an FP16 formatted `result_hw.txt` file for readability.

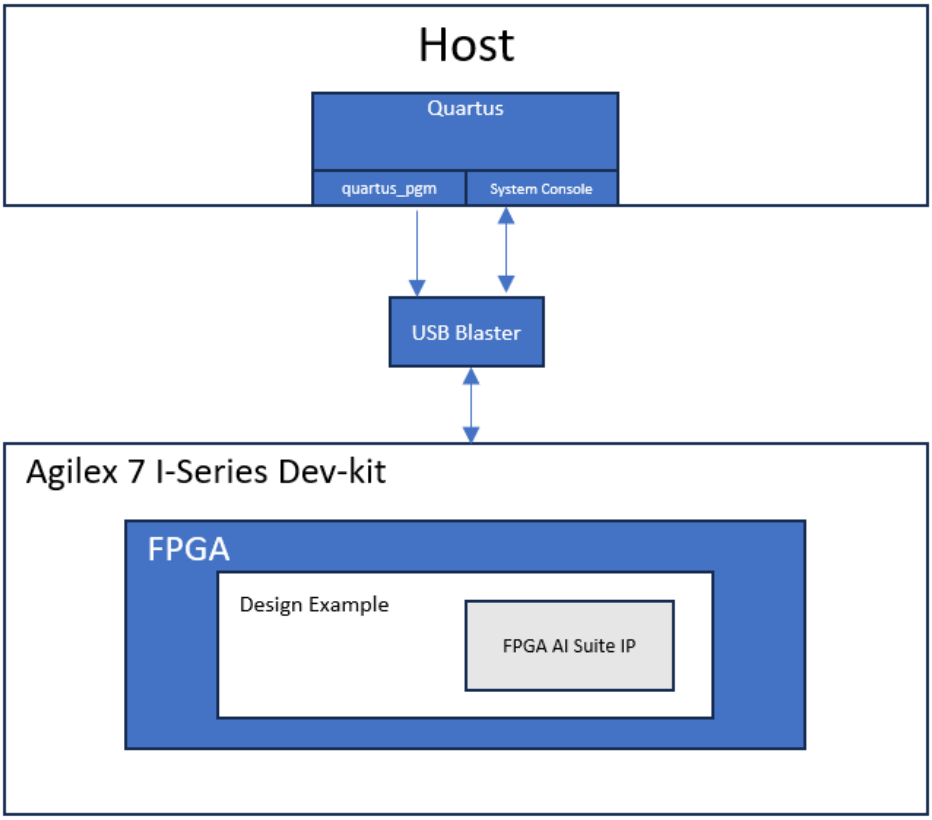
12. [HL-NO-DDR] Design Example System Architecture

12.1. [HL-NO-DDR] System Overview

The FPGA image consists of the FPGA AI Suite IP and additional logic that connects the IP to a JTAG interface. The DDR-Free design example does not use the `dla_benchmark` runtime. Instead, it allows for communication and control of the FPGA AI Suite IP through a JTAG-Quartus Prime System Console connection. In addition, the DDR-Free design example showcases the FPGA AI Suite IP streaming functionality. For more information about feature input and output streaming, refer to "Feature Input and Output Streaming" in *FPGA AI Suite IP Reference Manual*.

The system configuration of this design example is shown in the following block diagram:

Figure 6. DDR-Free Design Example System Configuration



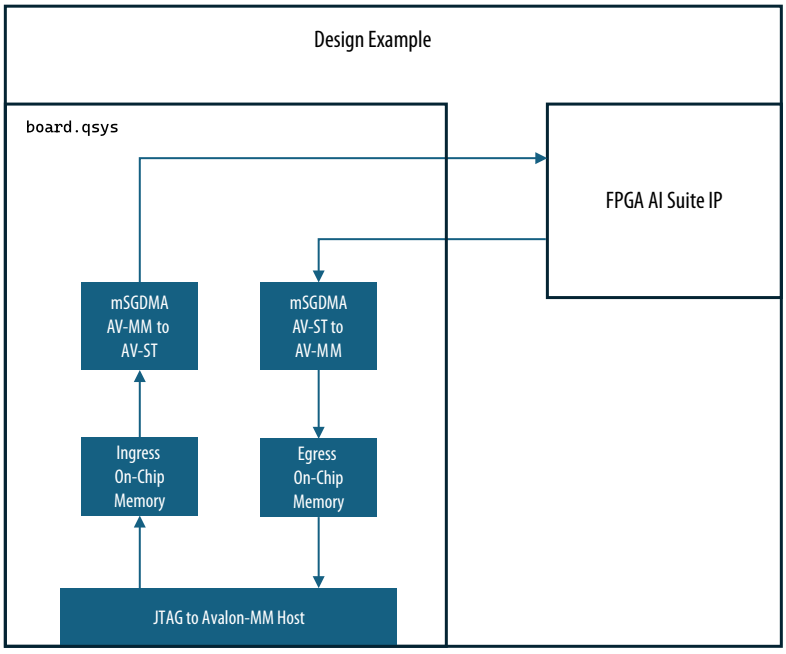
12.2. [HL-NO-DDR] Hardware

This section provides an in-depth description of the design example, focusing on the integration and functionality of the JTAG to Avalon-MM host, the ingress and egress mSGDMA engines, the FPGA AI Suite IP for inference, and the on-chip memory modules. It covers the configuration and control mechanisms, as well as the interaction between different components to achieve efficient AI inference on an FPGA device.

A top-level view of the design example that illustrates the data flow is shown in Figure 7 on page 63. The DDR-Free design example is currently limited to one FPGA AI Suite IP instance.

All components are connected to the JTAG to Avalon-MM host and are memory-mapped on the JTAG bus, allowing for efficient communication and control from the Quartus Prime System Console. Address offsets for each component is provided in [HL-NO-DDR] JTAG to Avalon MM Host Register Map on page 70.

Figure 7. DDR-Free Design Example System Architecture



12.2.1. [HL-NO-DDR] The Modular Scatter-Gather DMA (mSGDMA) Engines

The data flow within the system is orchestrated by the modular scatter-gather DMA (mSGDMA) engines and the FPGA AI Suite IP (which performs the inference computation). The following mSGDMA engines are used in the design example:

- **Ingress mSGDMA**

  The ingress mSGDMA engine performs memory-mapped reads from the on-chip memory and streams the data into the FPGA AI Suite IP. It converts Avalon-MM transactions to Avalon Streaming format.

- **Egress mSGDMA**

  The egress mSGDMA engine receives the streamed inference results from the FPGA AI Suite IP and stores them into the egress on-chip memory using MM operations. It converts Avalon-ST transactions back to Avalon-MM format.

The mSGDMA engines are configured to use 128-bit streaming transfer sizes.

For more information about how to use the modular scatter-gather DMA core, refer to "Modular Scatter-Gather DMA Core" in *Embedded Peripherals IP User Guide*.

12.2.2. [HL-NO-DDR] On-Chip Memory Modules

The on-chip memory modules store input data and final inference results. These memories are accessible via Avalon-MM interfaces. There are two modules: one for staging input memory and one for staging output inference results. These are referred to as ingress and egress on-chip memory, respectively. The sizes of the on-chip memory modules are defined in Table 9 on page 64.

- **Ingress On-Chip Memory**

  This module is dedicated to storing the input data before it is processed by the FPGA AI Suite IP. It serves as the staging area for data that will be read by the ingress mSGDMA engine and streamed into the inference IP.

- **Egress On-Chip Memory**

  This module is used to store the final inference results after they have been processed by the FPGA AI Suite IP. The egress mSGDMA engine writes the inference results from the FPGA AI Suite IP to this memory, making it available for retrieval and further use.

The following table provides the specific sizes allocated for each on-chip memory module, ensuring that the system has adequate storage for both input data and inference result:

Table 9. On-Chip Memory Module Sizes

On-Chip Memory Module	Size (in bytes)
Ingress	524288
Egress	131072

12.2.3. [HL-NO-DDR] Platform Designer System

The on-chip memory modules, mSGDMA engines, and other components are instantiated and interconnected within the `board.qsys` Platform Designer system. This comprehensive system design is then instantiated as an IP block within the `top.sv` file, ensuring seamless integration and efficient operation of the entire design.

12.2.4. [HL-NO-DDR] PLL Adjustment

The design example build script adjusts the PLL driving the FPGA AI Suite IP clock based on the f_{MAX} that the Quartus Prime compiler achieves.

13. [HL-NO-DDR] Quartus Prime System Console

This design example requires user interaction on the host system through Quartus Prime System Console. For more information about the Quartus Prime System Console, refer to "Analyzing and Debugging Designs with System Console" in *Quartus Prime Pro Edition User Guide: Debug Tools*.

The system console user interface communicates over JTAG to a JTAG to Avalon-MM host IP that enables the following functions:

- Read/write to the FPGA AI Suite IP DMA CSR

  For more information about the FPGA AI Suite IP CSR map, refer to "CSR Map and Descriptor Queue" in the *FPGA AI Suite IP Reference Manual*
- Read/write to ingress and egress on-chip memory
- Read/write to ingress and egress modular scatter-gather DMA (mSGDMA) CSR

  For more information about mSGDMA CSR, refer to "Register Map of mSGDMA" in *Embedded Peripherals IP User Guide*.

You can find an example Quartus Prime System Console Tcl script in the following location:

```
$COREDLA_ROOT/runtime/streaming/ed0_streaming_example/system_console_script.tcl
```

Use this only functional testing in hardware. For performance testing, refer to the script described in [HL-NO-DDR] Quartus Prime System Console Performance Script on page 68.

13.1. [HL-NO-DDR] Functionality

The process of executing an inference on the DDR-Free design example involves the following steps in the Quartus Prime System Console. Each step translates to a specific Tcl process in the `system_console_script.tcl` script:

1. Prime the FPGA AI Suite IP's CSR and resets the SGDMA's for streaming inference.

   ```
   initialize_coredla{}
   ```

2. Load raw input features into ingress on-chip memory

   ```
   stage_input{}
   ```

3. Queue a descriptor into the ingress SGDMA for MM to streaming operation

   ```
   queue_ingress_descriptor{}
   ```

4. Queue a descriptor into the egress SGDMA for streaming to MM operation

   ```
   queue_egress_descriptor{}
   ```

5. Reads inference results from the egress on-chip memory

```
read_output{}
```

13.2. [HL-NO-DDR] System Reset

In most FPGA AI Suite Design Examples, system resets are typically managed through software running on a host. However, because this design example operates without a host, this design example uses In-System Sources and Probes to perform a reset via JTAG.

This approach enables remote control of the reset process, ensuring both flexibility and accessibility. For the DDR-Free design example, the reset operation is initiated by writing a reset bit through the system console via JTAG. The following Tcl code snippet demonstrates the reset process.

Figure 8. System-Console TCL Reset Code for DDR-Free Design Example

```
# Initiate reset via source/probe IP
proc assert_reset {} {
 set issp_index 0
 set issp [lindex [get_service_paths issp] 0]
 set claimed_issp [claim_service issp $issp mylib]
 set source_data 0x0
 issp_write_source_data $claimed_issp $source_data
 set source_data 0x1
 issp_write_source_data $claimed_issp $source_data
 }
```

Related Information

"Design Debugging Using In-System Sources and Probes" in *Quartus Prime Pro Edition User Guide: Debug Tools*

13.3. [HL-NO-DDR] Input Data Conversion

This design example streams data into the FPGA AI Suite IP from the ingress on-chip memory. To achieve this, the system console script must stage input data in a `.bin` file format instead of `.bmp` format. The `.bin` file must be in FP16 (half-precision floating point) and organized in HWC (height-width-channel) format.

To facilitate this conversion, refer to the following example Python code. This script reads a `.bmp` file, converts the image data to FP16 format, and saves it in the required `.bin` format.

Figure 9. Python Example Code for `.bmp` to `.bin` Conversion

```
import sys
from PIL import Image
import numpy as np

def convert_image_to_bin(input_image_name):
    # Read the BMP file
    img = Image.open(input_image_name)
    output_file_name = 'array_hwc_fp16.bin'

    # Convert the image to a numpy array
    arr = np.array(img)

    # Convert the image to FP16 format
    arr_fp16 = arr.astype(np.float16)

    # Save the FP16 HWC formatted data to a .bin file
    with open(output_file_name, 'wb') as f:
        arr_fp16.tofile(f)

    print(f"Converted {input_image_name} to {output_file_name}")

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: python bmp_to_bin_converter.py <input_image_name>")
        sys.exit(1)

    input_image_name = sys.argv[1]
    convert_image_to_bin(input_image_name)
```

13.4. [HL-NO-DDR] Quartus Prime System Console Performance Script

Find the Quartus Prime System Console performance TCL script in the following location:

```
$COREDLA_ROOT/runtime/streaming/ed0_streaming_example/
system_console_script_perf.tcl
```

This script measures the throughput of the IP by running 32 inferences and calculating the average performance.

This script is hard-coded to run against a specific CNN graph. To modify this performance script for your design:

1. Run the functional script in [HL-NO-DDR] Quartus Prime System Console on page 66 and note the following items from the start of the script output:
   - Expected number of bytes to be transferred by the FPGA AI Suite output streamer
   - Size of the input file in bytes
2. Update the input length value in the `queue_ingress_descriptor{}` process of the performance script must be updated with the new input size of the file in bytes.

   ```
   master_write_32 $master_path 0x00030028 <input length>
   ```
3. Update the output length value in the `queue_egress_descriptor{}` process with the expected number of bytes transferred by the output streamer.

   ```
   master_write_32 $master_path 0x00030068 <output length>
   ```

4. Update the output length value in the `read_last_output{}` process with the expected number of bytes transferred by the output streamer.

```
master_read_to_file $master_path output0.bin 0x00280000 <output length>
```

14. [HL-NO-DDR] JTAG to Avalon MM Host Register Map

Table 10. JTAG to Avalon MM Host Register Map

IP	Offset	Description
FPGA AI Suite IP 0	0x0003_8000 – 0x0003_87ff	Refer to "CSR Map and Descriptor Queue" in the *FPGA AI Suite IP Reference Manual*
Ingress On-Chip Memory	0x0020_0000 – 0x0027_ffff	Refer to "On-Chip Memory II (RAM or ROM) Intel FPGA IP" in *Embedded Peripherals IP User Guide*.
Egress On-Chip Memory	0x0028_0000 – 0x0029_ffff	
Ingress mSGDMA (MM to Streaming)	CSR: 0x0003_0000 – 0x0003_001f	Refer to "Modular Scatter-Gather DMA Core" in *Embedded Peripherals IP User Guide*.
	Descriptor: 0x0003_0020 – 0x0003_002f	
Egress mSGDMA (Streaming to MM)	CSR: 0x0003_0040 – 0x0003_005f	
	Descriptor: 0x0003_0060 – 0x0003_006f	

15. [HL-NO-DDR] Updating MIF Files

The design example build process uses `*.mif` files to initialize the on-chip M2Ks. The M20Ks store filters, bias, and configuration data on-chip rather than in external memory. The DDR-Free flow allows inference of different graphs using the "update_mif" feature of Quartus Prime software without needing to recompile the bitstream. You must guarantee that the filter, bias, and config cache depth are large enough to hold the new graph parameters and FPGA AI Suite IP configuration.

After a design example is compiled, you can update the contents of the M20Ks through the Quartus Prime tools. The commands regenerate the `top.sof` bitstream file that needs to be reprogrammed on the device.

The Quartus Prime tools do not change the architecture of the FPGA AI Suite IP,. The update only the contents of the on-chip M20Ks that store the graph information and the FPGA AI Suite IP configuration.

To update the contents of the M20K on-chip memory:

1. Recompile the `.mif` files for the new graph as described in [HL-NO-DDR] Running the Hostless DDR-Free Design Example on page 60.
2. Replace the `.mif` files under "*<path/to/build/dir>*`/coredla_ip/intel_ai_ip/verilog/.`" with the files that were created in the previous step.
3. Run the following commands from "*<path/to/build/dir>*`/hw/`":

```
quartus_cdb top -c top --update_mif

quartus_asm --read_settings_files=on --write_settings_files=off top -c top
```

16. [HL-JTAG] Getting Started

The FPGA AI Suite integrates the JTAG design example with OpenVINO and the FPGA AI Suite `dla_benchmark` example application to perform inference.

This section describes the steps to build the necessary components and commands to launch inference with the `dla_benchmark` application.

16.1. [HL-JTAG] Prerequisites

16.1.1. [HL-JTAG] Software Requirements

You should install the FPGA AI Suiteand OpenVINO according to the steps in the *FPGA AI Suite Getting Started Guide*.

To generate the bitstream files for this design example, you require the following components from Quartus Prime Pro Edition Version 2024.3 or later:

- Quartus Prime software
- Agilex 5 E-Series device support

For host-FPGA device communication, and to configure the Agilex 5 FPGA E-Series 065B Premium Development Kit (DK-A5E065BB32AES1), you require the following components from Quartus Prime Pro Edition Version 2024.3 or later:

- Quartus Prime Programmer
- Quartus Prime System Console

The FPGA AI Suite software runtime requires the following Linux packages:

- Gflags 2.2.2 or later
- Boost (libboost) 1.85 or later

16.1.2. [HL-JTAG] Hardware Requirements

The JTAG design example has the following hardware requirement:

- Agilex 5 FPGA E-Series 065B Premium Development Kit (DK-A5E065BB32AES1)

Enable JTAG programming for the development kit as follows:

1. Enable JTAG programming on the development kit board by setting the board switches SW27[1:3] to [OFF, OFF, OFF].
2. On the host computer, ensure that the necessary USB port rules and permissions are enabled by creating a `/etc/udev/rules.d/51-altera-usb-blaster.rules` file with the following content:

```
SUBSYSTEM=="usb", ATTR{idVendor}=="09fb", ATTR{idProduct}=="6001",
MODE="0666"
SUBSYSTEM=="usb", ATTR{idVendor}=="09fb", ATTR{idProduct}=="6002",
MODE="0666"
SUBSYSTEM=="usb", ATTR{idVendor}=="09fb", ATTR{idProduct}=="6003",
MODE="0666"
SUBSYSTEM=="usb", ATTR{idVendor}=="09fb", ATTR{idProduct}=="6010",
MODE="0666"
SUBSYSTEM=="usb", ATTR{idVendor}=="09fb", ATTR{idProduct}=="6810",
MODE="0666"
SUBSYSTEM=="usb", ATTR{idVendor}=="0403", ATTR{idProduct}=="6001",
MODE="0666"
SUBSYSTEM=="usb", ATTR{idVendor}=="0403", ATTR{idProduct}=="6002",
MODE="0666"
SUBSYSTEM=="usb", ATTR{idVendor}=="0403", ATTR{idProduct}=="6003",
MODE="0666"
SUBSYSTEM=="usb", ATTR{idVendor}=="0403", ATTR{idProduct}=="6010",
MODE="0666"
SUBSYSTEM=="usb", ATTR{idVendor}=="0403", ATTR{idProduct}=="6810",
MODE="0666"
```

3. Change the port permission of the JTAG-USB connection to `0666`.

   Use the `lsusb` command to determine the bus and device numbers for adjusting permissions.

   For example, the command `lsusb` outputs the following bus and device path for devices on the development kit,

```
Bus 002 Device 006: ID 09fb:6010 Altera
Bus 002 Device 007: ID 0403:6010 Future Technology Devices International,
Ltd FT2232C/D/H Dual UART/FIFO IC
```

   With this information, adjust the permissions with the following commands:

```
sudo chmod 0666 /dev/bus/usb/002/006

sudo chmod 0666 /dev/bus/usb/002/007
```

16.2. [HL-JTAG] Building the FPGA AI Suite Runtime

Before you can run inference on the design example, build the FPGA AI Suite runtime using the provided `build_runtime.sh` script as follows:

1. Ensure that you have a working directory created as outlined in "Creating a Working Directory" in *FPGA AI Suite Getting Started Guide*.
2. Run the following commands:

```
cd $COREDLA_WORK

source $COREDLA_ROOT/bin/dla_init_local_directory.sh

cd runtime

./build_runtime.sh -target_system_console
```

A successful build of the runtime creates the following objects:

- `<runtime-directory>/build_Release/libcoreDlaRuntimePlugin.so`

  A runtime library that has the APIs for the host to access the FPGA AI Suite IP's registers and memories in the design example.
- `<runtime-directory>/build_Release/system_console_script.tcl`

  A script that the runtime invokes via the Quartus Prime System Console to set up the JTAG services to access the memories and CSR registers in the design example.
- `<runtime-directory>/build_Release/dla_benchmark/dla_benchmark`

  The `dla_benchmark` application.

If you run into the following error, check if `/sbin` appears near the start of the list of paths in your PATH environment variable:

```
Imported target "Boost::filesystem" includes non-existent path "/include"
```

Your PATH environment variable does not need to include `/sbin` for the JTAG design example.

16.3. [HL-JTAG] Building an FPGA Bitstream for the JTAG Design Examples

To generate an FPGA bitstream for the JTAG Design Example, run the following commands:

```
cd $COREDLA_WORK

$COREDLA_ROOT/bin/dla_build_example_design.py \
  -n 1\
  --archs=$COREDLA_ROOT/example_architectures/AGX5_Generic.arch \
  --build \
  --build-dir=build_agx5_jtag_ed \
  --example-design-id="AGX5E_JTAG"
```

This command creates the bitstream file called `AGX5_Generic.sof` that can be found in the `$COREDLA_WORK/build_agx5_jtag_ed/` folder.

Restriction: The QOR reporting utility of the `dla_build_example_design.py` command does not currently support the JTAG Design Example. To inspect the details of the design compilation, such as FPGA resource usage and clock f_{MAX}, consult the reports in the `$COREDLA_WORK /build_agx5_jtag_ed/hw/output_files` directory.

The JTAG design examples supports the following arguments for the `dla_build_example_design.py` command:

Option	Description
`-a, --archs`	Path to FPGA AI Suite IP Architecture Description File
`--build-dir`	Path to hardware build directory where BSP infrastructure and generated RTL will be located.
`--build`	Option to perform compilation of the design using Quartus Prime after instantiation (default: False).
`-d, --archs-dir`	Path to directory that contains Architecture Description Files for you to interactively choose from (alternative to '-a')

continued...

Option	Description
`-ed, --example-design-id`	To build for the Agilex 5 FPGA E-Series 065B Premium Development Kit, specify AGX5E_JTAG.
`-n, --num-instances`	Number of IP instances to build (default: 2). For the JTAG design example, this value must be 1.
`--num-paths`	Number of top critical paths to report after compiling the design (default: `2000`).
`-q, --quiet`	Run script quietly without printing the output of underlying scripts to the terminal.
`--unlicensed/licensed`	This option is passed to the `dla_create_ip` tool to tell the tool to generate either an unlicensed or licensed copy of the FPGA AI Suite: • **Unlicensed IP:** Unlicensed IP has a limit of 10000 inferences. After 10000 inferences, the unlicensed IP refuses to perform any additional inference and a bit in the CSR is set. For details about the CSR bit, refer to DMA Descriptor Queue in *FPGA AI Suite IP Reference Manual*. • **Licensed IP:** Licensed IP has no inference limitation. If you do not have a license but generate licensed IP, Quartus Primesoftware cannot generate a bitstream. If neither option is specified, then the `dla_create_ip` tool queries the `lmutil` license manager to determine the correct option.

16.4. [HL-JTAG] Programming the FPGA Device

Before you program the FPGA device, ensure that the USB-JTAG connection and port permissions are set up as described in [HL-JTAG] Hardware Requirements on page 72.

Program the FPGA device with Quartus Prime Programmer and FPGA bitstream that you generated in [HL-JTAG] Building an FPGA Bitstream for the JTAG Design Examples on page 74 with the following commands:

```
cd $COREDLA_WORK/build_agx5_jtag_ed

quartus_pgm -c 1 -m jtag -o "p;AGX5_Generic.sof"
```

16.5. [HL-JTAG] Preparing Graphs for Inference with FPGA AI Suite

Before running any demonstration application, you must convert the trained model to the Inference Engine format (`.xml, .bin`) with the OpenVINO Model Optimizer.

For details on creating the `.bin/.xml` files, refer to the *FPGA AI Suite Getting Started Guide*.

The network as described in the `.xml` and `.bin` files (created by the Model Optimizer) is compiled for a specific FPGA AI Suite architecture file by using the FPGA AI Suite compiler.

The FPGA AI Suite compiler compiles the network and exports it to a `.bin` file that uses the same `.bin` format as required by the OpenVINO Inference Engine.

This `.bin` file created by the compiler contains the compiled network parameters for all the target devices (FPGA, CPU, or both) along with the weights and biases. The inference application imports this file at runtime.

The FPGA AI Suite compiler can also compile the graph and provide estimated area or performance metrics for a given architecture file or produce an optimized architecture file.

For more details about the FPGA AI Suite compiler, refer to the *FPGA AI Suite Compiler Reference Manual*.

16.6. [HL-JTAG] Performing Inference on the Agilex 5 FPGA E-Series 065B Premium Development Kit

After the FPGA device on the Agilex 5 FPGA E-Series 065B Premium Development Kit has been programmed, you can perform inference.

For this design example, the FPGA AI Suite runtime needs the locations of the following items before performing inference:

- Bitstream file location

  The default bitstream path is `top.sof` located in the `$COREDLA_WORK` directory.

  You can override this default location by specifying the `DLA_SOF_PATH` environment variable.

- Quartus Prime System Console Tcl script (required to properly claim JTAG services)

  The default script is `$COREDLA_WORK/runtime/build_Release/system_console_script.tcl`.

  You can override this value by specifying the `DLA_SYSCON_SOURCE_FILE` environment variable.

Perform inference by running the following command

```
# Modify MODEL to suit your application

export DLA_SOF_PATH=$COREDLA_WORK/build_agx5_jtag_ed/AGX5_Generic.sof
export MODEL=<path-to-model-XML-file>/resnet50.xml

cd $COREDLA_WORK/runtime/build_Release/dla_benchmark

dla_benchmark \
  -b=1 \
  -m=$MODEL \
  -d=HETERO:FPGA,CPU \
  -i <path-to-image-files> \
  -niter=2 \
  -plugins=../../plugins.xml \
  -arch_file= $COREDLA_ROOT/example_architectures/AGX5_Generic.arch \
  -api=async \
  -perf_est \
  -nireq=1 -dump_output -report_lsu_counters -enable_early_access
```

For more information about the `dla_benchmark` application, refer to [PCIE] Performing Accelerated Inference with the dla_benchmark Application on page 17.

16.7. [HL-JTAG] Inference Performance Measurement

The `dla_benchmark` application reports inference duration and throughput for the entire design example as well as for the FPGA AI Suite IP.

To perform one inference iteration, the host performs the following steps:

1. Write input data via JTAG to the DDR memory on the FPGA development board.
2. Programs CSRs on the FPGA AI Suite IP to start inference.
3. Polls the CSRs until the FPGA AI Suite IP completes the inference.
4. Read the output from the DDR memory to the host via JTAG.

The system duration accounts for all these steps above..

In contrast, the IP duration omits the duration of input and output data transfer.

For this design example, system duration is usually much larger than the IP duration because data transfer over JTAG is relatively slow. Thus, the IP duration and throughput better reflect the performance of the FPGA AI Suite IP.

The following output is an example throughput report generated by the `dla_benchmark` application after performing 3925 inferences on a quantized ResNet-18 model:

```
[Step 11/12] Dumping statistics report
count:              3925 iterations
system duration:   464549.5363 ms
IP duration:       17945.7971 ms
latency:            118.2524 ms
system throughput: 8.4490 FPS
number of hardware instances: 1
number of network instances: 1
IP throughput per instance: 218.7142 FPS
IP throughput per fmax per instance: 1.0936 FPS/MHz
IP clock frequency: 200.0000 MHz
```

16.8. [HL-JTAG] Known Issues and Limitations

The JTAG design example has the following known issues and limitations:

- The number of inference request (-nireq) must be 1 when running dla_benchmark with the Agilex 5 E-Series JTAG Example Design.
- The USB-JTAG connection between the host and the FPGA is relatively slow, so the system throughput is much lower than both the measured and estimated IP throughputs per instance.
- For FPGA AI Suite IP configured with large K_{vec} and C_{vec} parallelism, the peak throughput of the DDR4 interface on the Agilex 5 FPGA E-Series 065B Premium Development Kit can become the bottleneck. When you run the `dla_benchmark` application with the flag `-perf_est`, the application provides a throughput estimation that does not fully account for the limited external memory bandwidth on the development kit, so the estimate might be higher than the measured IP throughput per instance.
- On Ubuntu 20 and Ubuntu 22 systems, the runtime might fail when loading model to the FPGA device if the $DLA_SOF_PATH environment variable does not point to the correct bitstream file, or if the Quartus Prime System Console `system-console` command is not present in the $PATH environment variable.

  The Quartus Prime System Console command is in `$QUARTUS_ROOTDIR/syscon/bin`.

  ```
  [Step 5/12] Resizing network to match image sizes and given batch
  [Step 6/12] Configuring input of the model
  [[Step 7/12] Loading the model to the device
  Generating unsupported layer chains graph (./unsupported_layer_chains.dot)
  Using the Tcl setup script at /home/user/coredla-work/runtime/build_Release/
  system_console_script.tcl
  Saving temporary files to /home/user/Downloads
  Segmentation fault (core dumped)
  ```

- You might occasionally see the following Quartus Prime System Console error when running inference with the runtime on this design example:

  ```
  claim_service: Path cannot be found while executing
  "claim_service master $path {}
  "\{${::g_const_master_offset_dla} ${::g_const_master_range_dla} EXCLUSIVE\}""
      procedure "claim_dla_csr_service" line 4)
      invoked from within"claim_dla_csr_service"
  procedure "initialization" line 4)
   invoked from within
  "initialization"
  ```

  To recover from the issue, reprogram the FPGA with the correct bitstream and rerun the `dla_benchmark` application. To reduce the likelihood of this issue, lower the JTAG clock frequency to 16 MHz before running the run the `dla_benchmark` application:

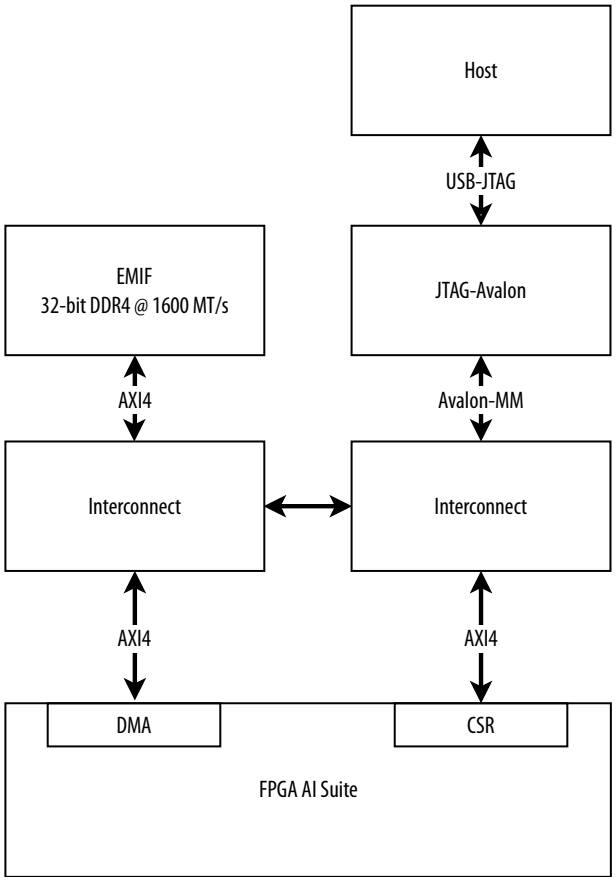
  ```
  jtagconfig --setparam 1 JtagClock 16M
  ```

17. [HL-JTAG] Design Example Components

17.1. [HL-JTAG] Hardware Components

The FPGA AI Suite JTAG design example uses a JTAG-USB connection on the development kit to facilitate host-FPGA transactions. The following diagram shows a simplified block diagram of the design.

Figure 10. Simplified System Design of the JTAG Design Example



The design example stores filter weights, biases, configurations for the FPGA AI Suite IP, graph inputs, outputs, and intermediate data on the lower 2GB of a DDR4 memory interface, similar to the PCIe-based design examples. An external memory interface (EMIF) IP manages the DDR4 memory and toggles the 32-bit interface at 1600 MT/s.

The FPGA AI Suite IP is clocked at 200 MHz and accesses the EMIF IP via a 128-bit AXI4 data interface clocked at 200 MHz. The IP can only access the first 2GB of the memory.

The host uses a JTAG-Avalon Bridge IP to access the DDR4 memory, as well as the CSR registers of the FPGA AI Suite IP via a 32-bit Avalon-MM interface clocked at 100 MHz. As seen from the JTAG-Avalon Bridge IP, the subordinates reside at the addresses outlined in the following table:

Table 11. Subordinate Addresses Accessible from the JTAG-Avalon Interface

External Memory	FPGA AI Suite IP CSRs	
Address Range (Bytes)	0x0000_0000 – 0x7FFF_FFFF	0x8000_0000 – 0x8000_0FFF

To investigate and modify the implementation of the hardware components, review the source files of the system design part of the design example in the `$COREDLA_ROOT/platform/a5e_premium_devkit_jtag` directory.

17.2. [HL-JTAG] Software Components

The JTAG design example relies on the following software components to executed FPGA inference via the dla_benchmark application:

- OpenVINO Toolkit
- FPGA AI Suite Runtime Plugin
- Quartus Prime System Console

When the build target is this design example, the FPGA AI Suite Runtime Plugin instantiates an MMD Wrapper object that converts external memory and FPGA AI Suite CSR access commands into Quartus Prime System Console calls. The MMD wrapper depends on the Boost C++ Libraries. The definitions of the MMD wrapper object and its methods are implemented in `$COREDLA_WORK/runtime/coredla_device/mmd/system_console/mmd_wrapper.cpp`.

Upon instantiation, the MMD wrapper initializes the JTAG services via the routine in a Quartus Prime System Console script: `$COREDLA_WORK/runtime/coredla_device/mmd/system_console/system_console.tcl`.

18. [SOC] FPGA AI Suite SoC Design Example Prerequisites

The SoC design example requires one of the following development kits:

- Agilex 7 FPGA I-Series Transceiver-SoC Development Kit (DK-SI-AGI027FC) (Ordering code DK-SI-AGI027FA or DK-SI-AGI027FC)

  This development kit features an Agilex 7 I-Series SoC device with 4 F-Tiles (OPN: AGIB027R31B1E1V).

  Important: The FPGA AI Suite requires DDR4 memory with an x8 or higher component data width. The RAM device provided with the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit provides only a x4 component data width. For more details and recommended RAM modules, refer to [SOC] Agilex 7 FPGA I-Series Transceiver-SoC Development Kit Hardware Requirements on page 82.

  For more details about this development kit, refer to the following URL: https://www.intel.com/content/www/us/en/products/details/fpga/development-kits/agilex/si-agi027.html
- Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S)

  This development kit features an Arria 10 SX 660 device (OPN: 10AS066N3F40E2SG) with a " -2" speed grade with the included DDR4 HILO memory cards.

  For more details about this development kit, refer to the following URL:

  https://www.intel.com/content/www/us/en/products/details/fpga/development-kits/arria/10-sx.html

In addition, the following hardware components are required:

- SDHC flash card, class 10 speed or faster (minimum 2 GB but 4 GB or more is recommended)
- Mini-USB cable suitable for connecting the development board to a host PC
- Ethernet cable suitable for connecting the development board to a network to provide access from a host PC on the same network

The host PC must use a supported operating system (Red Hat* Enterprise Linux* 8, Ubuntu* 20.04, or Ubuntu 22.04), and must have an internet connection to install the software dependencies.

To build bitstreams, Quartus Prime Pro Edition Version 24.3 must be installed on the host system.

Although the development host system does not need to be the same as the system used to build packages and bitstreams, this guide does not explicitly cover the scenario where they are distinct.

18.1. [SOC] Agilex 7 FPGA I-Series Transceiver-SoC Development Kit Hardware Requirements

The FPGA AI Suite SoC design example requires x8 (or wider) DDR4 memory.

The RAM module provided with the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit does not support the FPGA AI Suite SoC design example because the included RAM module provides only an x4 width.

The design example has been verified on a development kit fitted with a Kingston* x8 RDIMM (KSM32RS8/16MFR). Intel recommends using this memory module to help you successfully use the design example.

19. [SOC] FPGA AI Suite SoC Design Example Quick Start Tutorial

The SoC design example quick start tutorial provides instructions to do the following tasks:

- Build a bitstream and flash card image for the FPGA development kit.
- Run the `dla_benchmark` utility from the example runtime on the SoC FPGA HPS (Arm processor) host. This example runtime uses the memory-to-memory (M2M) model.
- Run the streaming image application that streams data from the HPS Arm processor host to the FPGA device in a way that mimics how data is streamed from any other input source (such as Ethernet, HDMI, or MIPI). This streaming image application uses the steaming-to-memory (S2M) model.

This quick start tutorial assumes that you have reviewed the following sections in the *FPGA AI Suite Getting Started Guide*:

- About the FPGA AI Suite
- Installing the FPGA AI Suite

SoC Design Example Quick Start Tutorial Prerequisites

Before you start the tutorial ensure that you have successfully completed the installation tasks outlined in "Installing the FPGA AI Suite Compiler and IP Generation Tools" in the *FPGA AI Suite Getting Started Guide*.

The remaining sections of the *FPGA AI Suite Getting Started Guide* can help you understand the overall flow of using the FPGA AI Suite, but they are not required to complete this quick start tutorial

19.1. [SOC] Initial Setup

The quick start tutorial instructions assume that you have initialized your environment for the FPGA AI Suite with the `init_env.sh` script from a shell that is compatible with the Bourne shell (`sh`).

The FPGA AI Suite `init_env.sh` script might already be part of your shell login script. If not, then use the following command to initialize your shell environment:

```
source /opt/intel/fpga_ai_suite_2024.3/dla/bin/init_env.sh
```

This command assumes that the FPGA AI Suite is installed in the default location. If you are using an FPGA AI Suite version other than 2024.3, adjust the path to script accordingly.

19.2. [SOC] Initializing a Work Directory

While you can build the design example directly in the $COREDLA_ROOT location, it is better to use a work directory. You can create a work directory as follows:

```
mkdir ~/coredla_work
cd ~/coredla_work
source dla_init_local_directory.sh
```

If you created a work directory while following the instructions in the *FPGA AI Suite Getting Started Guide*, the `dla_init_local_directory.sh` script prompts you to use the `coredla_work.sh` script instead to set the $COREDLA_WORK environment variable.

19.3. [SOC] (Optional) Create an SD Card Image (`.wic`)

An SD card provides the FPGA bitstream and HPS disk image to the SoC FPGA development kit. You can build your own SD card image or use the prebuilt image provided by the FPGA AI Suite SoC design example.

If you want to use the prebuilt image, skip this section and go to [SOC] Writing the SD Card Image (.wic) to an SD Card on page 89.

Important: You cannot build the SD card as `root` due to security checks in the BitBake tool used when creating an SD card image.

19.3.1. [SOC] Installing Prerequisite Software for Building an SD Card Image

Building the SD card image requires the following additional software:

- Quartus Prime Pro Edition Version 24.3
- Ashling* RiscFree* IDE for Altera®
- (Ubuntu only) Ubuntu package `libncurses5`

If you did not install Quartus Prime Pro Edition Version 24.3 when following the instructions in the *FPGA AI Suite Getting Started Guide*, you must install it now.

Building the SD card image also requires tools provided by Ashling RiscFree IDE for Altera. You can install Ashling RiscFree IDE from a separate installation package or part of your Quartus Prime bundled installation package.

You can download the required software from the following URL: https://www.intel.com/content/www/us/en/software-kit/790039/intel-quartus-prime-pro-edition-design-software-version-23-3-for-linux.html.

To install the prerequisite software for building an SD card image:

1. Install Quartus Prime Pro Edition and Ashling RiscFree IDE for Altera
2. (Ubuntu only) Install Ubuntu package `libcurses` with the following command:

```
sudo apt install libncurses5
```

3. Ensure that the QUARTUS_ROOTDIR environment variable is set properly:

```
echo $QUARTUS_ROOTDIR
```

   If the QUARTUS_ROODIR is not set, run the following command:

```
export QUARTUS_ROOTDIR=/opt/intel/intelFPGA_pro/24.3/quartus
```

   If you chose install Quartus Prime in a location other than the default location, adjust the path in `export` command to match your Quartus Prime installation location

4. Ensure your $PATH environment variable includes paths to the installed Quartus Prime and Ashling RiscFree IDE binaries. Adjust the following commands appropriately if you did not install into the default location:

```
export PATH=$PATH:/opt/intel/intelFPGA_pro/24.3/niosv/bin
export PATH=$PATH:/opt/intel/intelFPGA_pro/24.3/nios2eds/bin
export PATH=$PATH:/opt/intel/intelFPGA_pro/24.3/riscfree/toolchain/riscv32-unknown-elf/bin
```

5. Confirm that Quartus Prime Pro Edition Version 24.3 is installed by running the following command:

```
quartus_cmd -v
```

Related Information

- Ashling RiscFree IDE for Altera User Guide
- Intel FPGA Software Installation and Licensing

19.3.2. [SOC] Building the FPGA Bitstreams

The FPGA AI Suite SoC design example also includes prebuilt demonstration FPGA bitstreams. If you want to use the prebuilt demonstration bitstreams in your SD card image, skip ahead to [SOC] Installing HPS Disk Image Build Prerequisites on page 86.

If you build your own bitstreams and do not have an FPGA AI Suite IP license, then your bitstream have a limit of 10000 inferences. After 10000 inferences, the unlicensed IP refuses to perform any additional inference. To reset the limit, reprogram the FPGA device.

Building the FPGA Bitstreams for the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit

To build the FPGA bitstreams for the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit, run the following commands:

```
dla_build_example_design.py \
 -ed 4_AGX7_S2M \
 -n 1 \
 -a $COREDLA_ROOT/example_architectures/AGX7_Performance.arch \
 --build \
 --build-dir $COREDLA_WORK/agx7_perf_bitstream \
 --output-dir $COREDLA_WORK/agx7_perf_bitstream
```

The bitstreams built by these commands support both the M2M execution model and the S2M execution model.

Building the FPGA Bitstreams for the Arria 10 SX SoC FPGA Development Kit

To build the FPGA bitstreams for the Arria 10 SX SoC FPGA Development Kit, run the following commands:

```
dla_build_example_design.py \
 -ed 4_A10_S2M \
 -n 1 \
 -a $COREDLA_ROOT/example_architectures/A10_Performance.arch \
 --build \
 --build-dir $COREDLA_WORK/a10_perf_bitstream \
 --output-dir $COREDLA_WORK/a10_perf_bitstream
```

The bitstreams built by these commands support both the M2M execution model and the S2M execution model.

19.3.3. [SOC] Installing HPS Disk Image Build Prerequisites

The process to build the HPS disk image has additional prerequisites. To install these prerequisites, follow the instructions for your operating system in the following sections:

- Red Hat Enterprise Linux 8 Prerequisites on page 86
- Ubuntu 20 Prerequisites on page 87
- Ubuntu 22 Prerequisites on page 87

Red Hat Enterprise Linux 8 Prerequisites

To install the prerequisites for Red Hat Enterprise Linux 8:

1. Enable additional Red Hat* Enterprise Linux 8 repository and package manager:

```
sudo subscription-manager repos \
  --enable codeready-builder-for-rhel-8-x86_64-rpms
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm
sudo dnf install ./epel-release-latest-8.noarch.rpm epel-release
sudo dnf upgrade
```

2. Install the dependency packages:

```
sudo dnf install gawk wget git diffstat unzip texinfo gcc gcc-c++ make \
chrpath socat cpio python3 python3-pexpect xz iputils python3-jinja2 \
mesa-libEGL SDL xterm python3-subunit rpcgen zstd lz4 perl-open.noarch \
perl-Thread-Queue
```

3. Install packages required to create the flash card image and FPGA AI Suite runtime and dependencies:

```
cd /tmp
mkdir uboot_tools && cd uboot_tools
wget https://kojipkgs.fedoraproject.org/\
vol/fedora_koji_archive02/packages/uboot-tools/2018.03/3.fc28/x86_64/\
uboot-tools-2018.03-3.fc28.x86_64.rpm
sudo dnf install ./uboot-tools-2018.03-3.fc28.x86_64.rpm
sudo dnf install ninja-build fakeroot
sudo python3 -m pip install pylint passlib scons
```

4. Install CMake Version 3.16.3 or later:

```
sudo dnf install openssl-devel
cd /tmp
mkdir cmake && cd cmake
wget https://github.com/Kitware/CMake/releases/\
download/v3.24.3/cmake-3.24.3.tar.gz
```

```
tar zxf cmake-3*tar.gz
cd cmake-3.24.3
./bootstrap --prefix=/usr
make
sudo make install
```

5. Install Make Version 4.3 or later:

```
cd /tmp
mkdir make && cd make
wget https://ftp.gnu.org/gnu/make/make-4.3.tar.gz
tar xvf make-4.3.tar.gz
cd make-4.3
./configure
make
sudo make install
```

6. Add the `/sbin` directory to your $PATH environment variable:

```
export PATH="/sbin:$PATH"
```

Ubuntu 20 Prerequisites

To install the prerequisites for Ubuntu 20:

1. Install the dependency packages:

```
sudo apt install gawk wget git diffstat unzip texinfo gcc build-essential \
chrpath socat cpio python3 python3-pip python3-pexpect xz-utils debianutils \
iputils-ping python3-git python3-jinja2 libegl1-mesa libsdl1.2-dev pylint3 \
xterm python3-subunit mesa-common-dev zstd liblz4-tool device-tree-compiler \
mtools
```

2. Install packages required to create the flash card image and FPGA AI Suite runtime and dependencies:

```
sudo apt install ninja-build u-boot-tools scons fakeroot
```

3. Add the `/sbin` directory to your $PATH environment variable:

```
export PATH="/sbin:$PATH"
```

Ubuntu 22 Prerequisites

To install the prerequisites for Ubuntu 22:

1. Install the dependency packages:

```
sudo apt install gawk wget git diffstat unzip texinfo gcc build-essential \
chrpath socat cpio python3 python3-pip python3-pexpect xz-utils debianutils \
iputils-ping python3-git python3-jinja2 libegl1-mesa libsdl1.2-dev xterm \
python3-subunit mesa-common-dev zstd liblz4-tool device-tree-compiler mtools
```

2. Install packages required to create the flash card image and FPGA AI Suite runtime and dependencies:

```
sudo apt install ninja-build u-boot-tools scons fakeroot
```

3. Add the `/sbin` directory to your $PATH environment variable:

```
export PATH="/sbin:$PATH"
```

19.3.4. [SOC] (Optional) Downloading the ImageNet Categories

By default, the S2M streaming app prints the category associated with each image after inference.

Optionally, you can use human-readable category names as follows:

1. Download the list of ImageNet categories from a source such as the following URL:

```
https://github.com/xmartlabs/caffeflow/blob/master/examples/imagenet/
imagenet-classes.txt
```

2. Place the contents into the following file:

```
$COREDLA_WORK/runtime/streaming/streaming_inference_app/categories.txt
```

19.3.5. [SOC] Building the SD Card Image

The SD card image contains a Yocto Project embedded Linux system, HPS packages, and the FPGA AI Suite runtime.

Building the SD card image requires a minimum of 100GB of free disk space.

The SD card image is build with the `create_hps_image.sh` command, which does the following steps for you:

- Build a Yocto Project embedded Linux system.
- Build additional packages required by the SoC design example runtime, including the OpenVINO and OpenCV runtimes.
- Build the design example runtime.
- Combine all of these items and FPGA bitstreams into an SD card image using `wic`.
- Place the SD card image in the specified output directory.

For more details about the `create_hps_image.sh` command, refer to [SOC] Building the Bootable SD Card Image (.wic) on page 112.

To build the SD card image, run one of the following commands:

- Agilex 7 FPGA I-Series Transceiver-SoC Development Kit

```
cd $COREDLA_WORK/runtime
./create_hps_image.sh \
   -f $COREDLA_WORK/agx7_perf_bitstream/hw/output_files \
   -o <output_dir> -u \
   -m agilex7_dk_si_agi027fa
```

- Arria 10 SX SoC FPGA Development Kit

```
cd $COREDLA_WORK/runtime
./create_hps_image.sh \
   -f $COREDLA_WORK/a10_perf_bitstream/hw/output_files \
   -o <output_dir> -u \
   -m arria10
```

If the command returns errors such as "`bitbake: command not found`", try deleting the `$COREDLA_WORK/runtime/build_Yocto/` directory before rerunning the `create_hps_image.sh` command.

19.4. [SOC] Writing the SD Card Image (`.wic`) to an SD Card

Before running the demonstration, you must create a bootable SD card for the FPGA development kit. You can use either the precompiled SD card image or an SD card image that you created.

The precompiled SD card image (`.wic`) is in the following location:

- Agilex 7 FPGA I-Series Transceiver-SoC Development Kit

```
$COREDLA_ROOT/demo/ed4/agx7_soc_s2m/sd-card/coredla-image-
agilex7_dk_si_agi027fa.wic
```

- Arria 10 SX SoC FPGA Development Kit

```
$COREDLA_ROOT/demo/ed4/a10_soc_s2m/sd-card/coredla-image-arria10.wic
```

If you built your own SD card image following the instructions in [SOC] (Optional) Create an SD Card Image (.wic) on page 84, then your SD card image is located in the directory that you specified for the `-o` option of the `create_hps_image.sh` command.

To write the SD card image to an SD card:

1. Determine the device associated with the SD card on the host by running the following command before and after inserting the SD card:

   ```
   cat /proc/partitions
   ```

   Typical locations for the SD card include `/dev/sdb` or `/dev/sdc`. The rest of these instruction use `/dev/sdx` as the SD card location.

2. Use the `dd` command to write the SD card image as follows:

   ```
   wic_image=<path to SD (.wic) image file>
   sudo umount /dev/sdx*
   sudo dd if=$wic_image of=/dev/sdx bs=1M
   sudo sync
   sudo udisksctl power-off -b /dev/sdx
   ```

After the SD card image is written, insert the SD card into the development kit SD card slot.

If you want to use a Microsoft* Windows system to write the SD card image to the SD card, refer to the GSRD manuals available at the following URL: https://www.rocketboards.org/foswiki/Documentation/GSRD.

19.5. [SOC] Preparing SoC FPGA Development Kits for the FPGA AI Suite SoC Design Example

To prepare an FPGA development kit for the FPGA AI Suite SoC design example:

1. Prepare one of the supported development kits:
   - Prepare the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit.
   - Prepare the Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S).
2. Configure the SoC FPGA development kit UART connection.
3. Determine the SoC FPGA development kit IP address.

19.5.1. [SOC] Preparing the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit

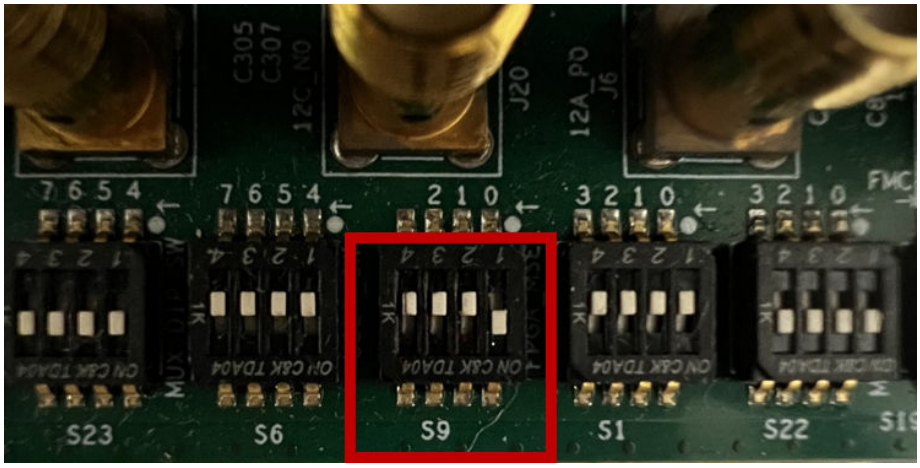
To prepare the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit for the FPGA AI Suite SoC design example:

1. [SOC] Confirming Agilex 7 FPGA I-Series Transceiver-SoC Development Kit Board Set Up on page 90
2. [SOC] Programming the Agilex 7FPGA Device with the JTAG Indirect Configuration (.jic) File on page 91.
3. [SOC] Connecting the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit to the Host Development System on page 93

19.5.1.1. [SOC] Confirming Agilex 7 FPGA I-Series Transceiver-SoC Development Kit Board Set Up

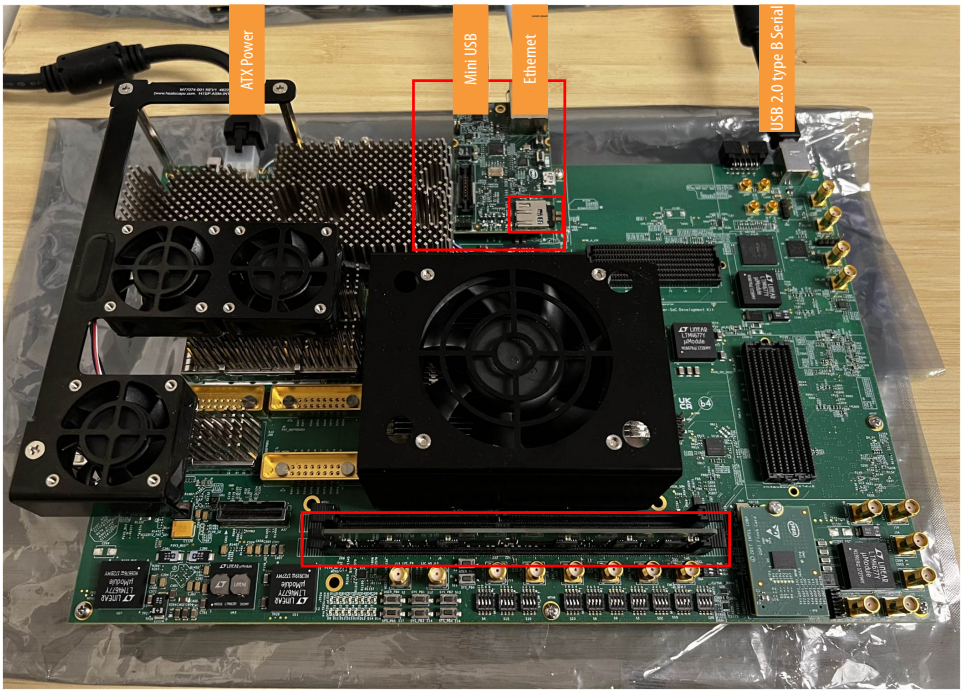
Confirm the board settings as follows:

1. Ensure that the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit has the required DIP switch and jumper settings. The SoC example design requires that all DIP switches have their default settings except for S9 where switch 1 is ON and switches 2,3, and 4 are OFF:



   For more details about default DIP switch and jumper settings, refer to *Agilex 7 FPGA I-Series Transceiver-SoC Development Kit User Guide*.
2. Ensure that the HPS IO48 OOBE daughter card is installed in connector J4 on the development kit, and the SD card with the programmed Yocto image is installed in the daughter card.
3. Ensure that the DDR4 x8 RDIMM is installed in the PCIe slot furthest from the fan. For RDIMM requirements, refer to [SOC] Agilex 7 FPGA I-Series Transceiver-SoC Development Kit Hardware Requirements on page 82.

When configured and connected, the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit should resemble the following image:



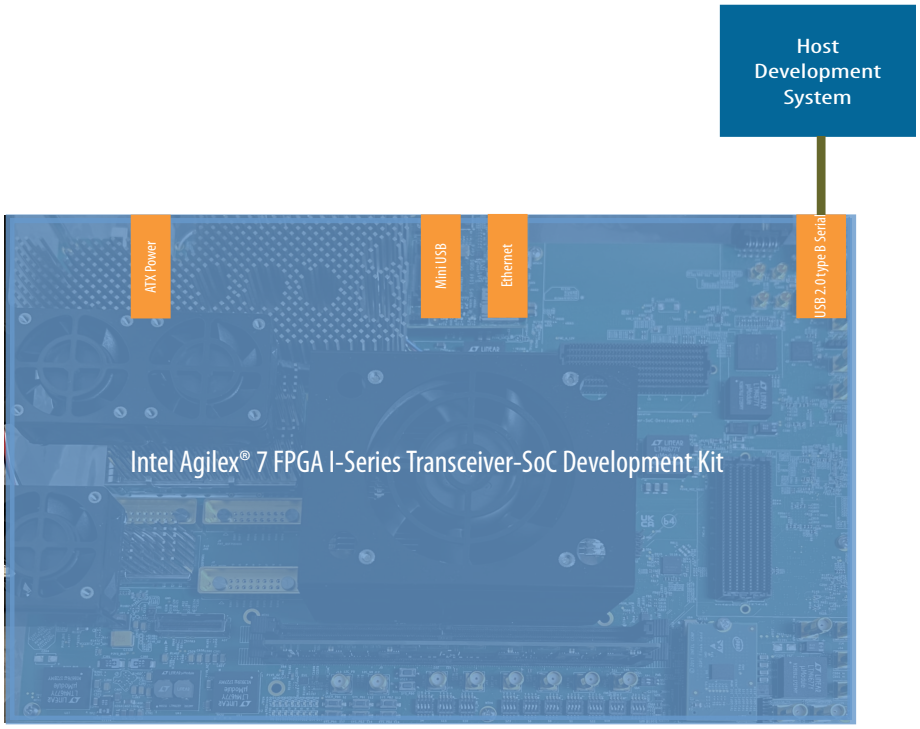
The board connections serve the following purposes:

- The USB 2.0 connector is used to program the FPGA device.
- The Ethernet connector is used for fast data transfer to the HPS.
- The micro USB connector is used to monitor the serial output from, and provide command-line input to the HPS during operation.

19.5.1.2. [SOC] Programming the Agilex 7FPGA Device with the JTAG Indirect Configuration (`.jic`) File

To program the Agilex 7 FPGA device with the JTAG indirect configuration (`.jic`) file:

1. Connect the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit to your host development system via USB 2.0 connection as shown in the following diagram:



2. Program the QSPI with the `.jic` file by running the following commands on the host development system:

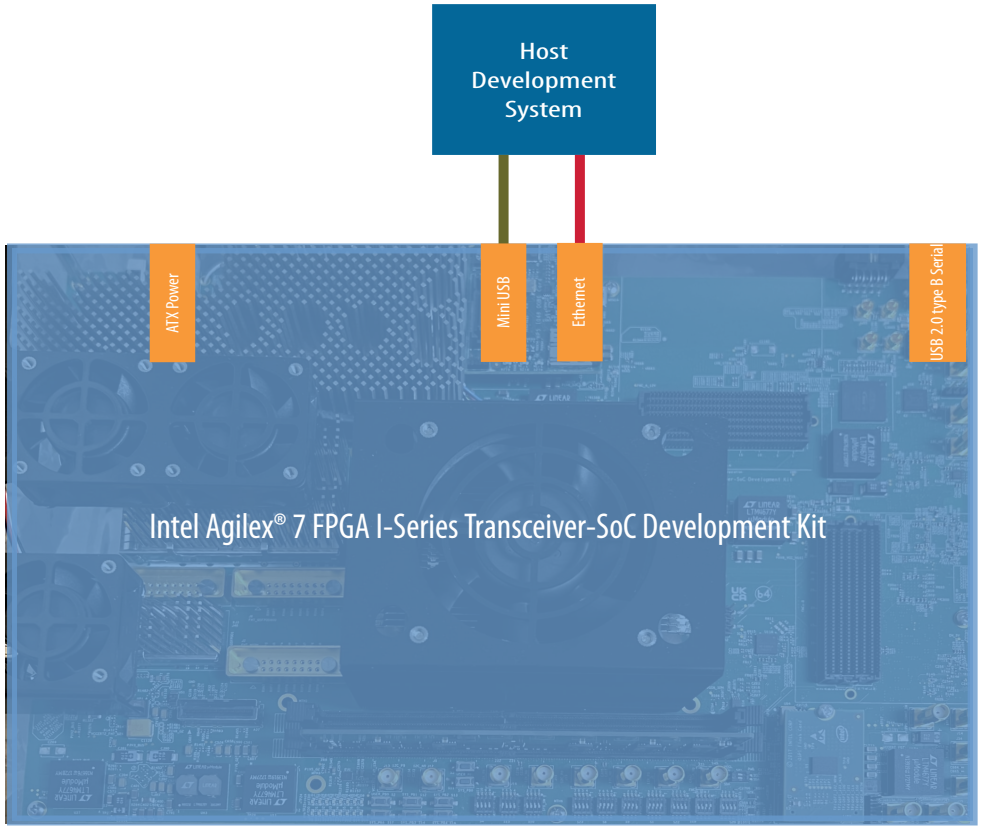
```
cd $COREDLA_ROOT/demo/ed4/agx7_soc_s2m/sd-card/
quartus_pgm -m jtag -o "pvi;u-boot-spl-dtb.hex.jic@<device_number>"
```

   where *<device_number>* is 1 or 2, depending on whether the HPS is already running (that is, the prior state of the device). Use 1 if the HPS is not running, and 2 if the HPS is already running. If you do not know the state of the device, try 1. If that fails, try 2.

   The Agilex 7FPGA device is configured from the QSPI flash at boot time.

19.5.1.3. [SOC] Connecting the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit to the Host Development System

Connect the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit to your host development system via Ethernet and USB UART connections as shown in the following diagram:



19.5.2. [SOC] Preparing the Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S)

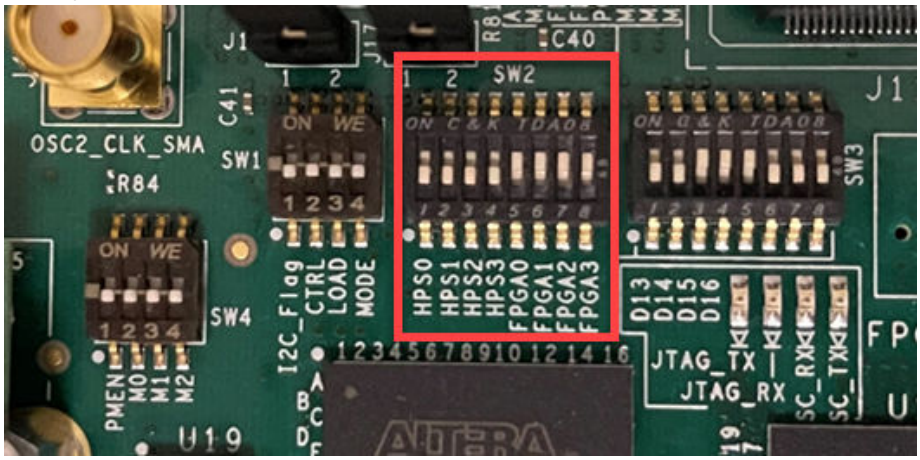
To prepare the Arria 10 SX SoC FPGA Development Kit for the FPGA AI Suite SoC design example:

1. [SOC] Confirming Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S) Board Settings on page 93
2. [SOC] Connecting the Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S) to the Host Development System on page 96.

19.5.2.1. [SOC] Confirming Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S) Board Settings

Confirm the board settings as follows:

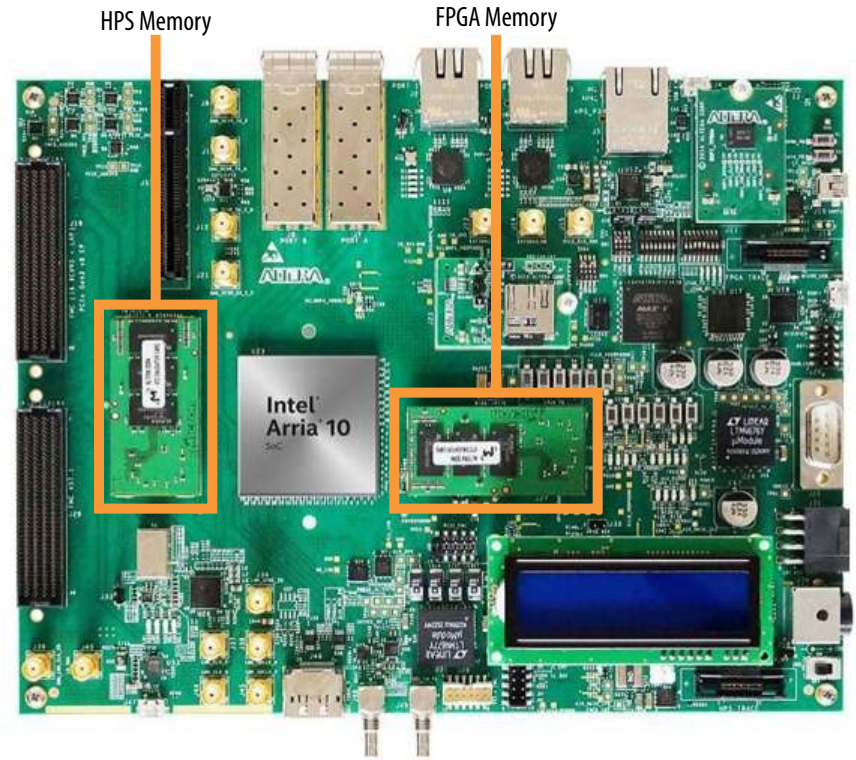
1. Ensure that the Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S) has the required DIP switch and jumper settings. The SoC example design requires that all DIP switches have their default settings except for SW2 switches 5, 6, 7, and 8, which should be switched ON:



   For more details about default DIP switch and jumper settings, refer to *Arria 10 SoC Development Kit User Guide*.

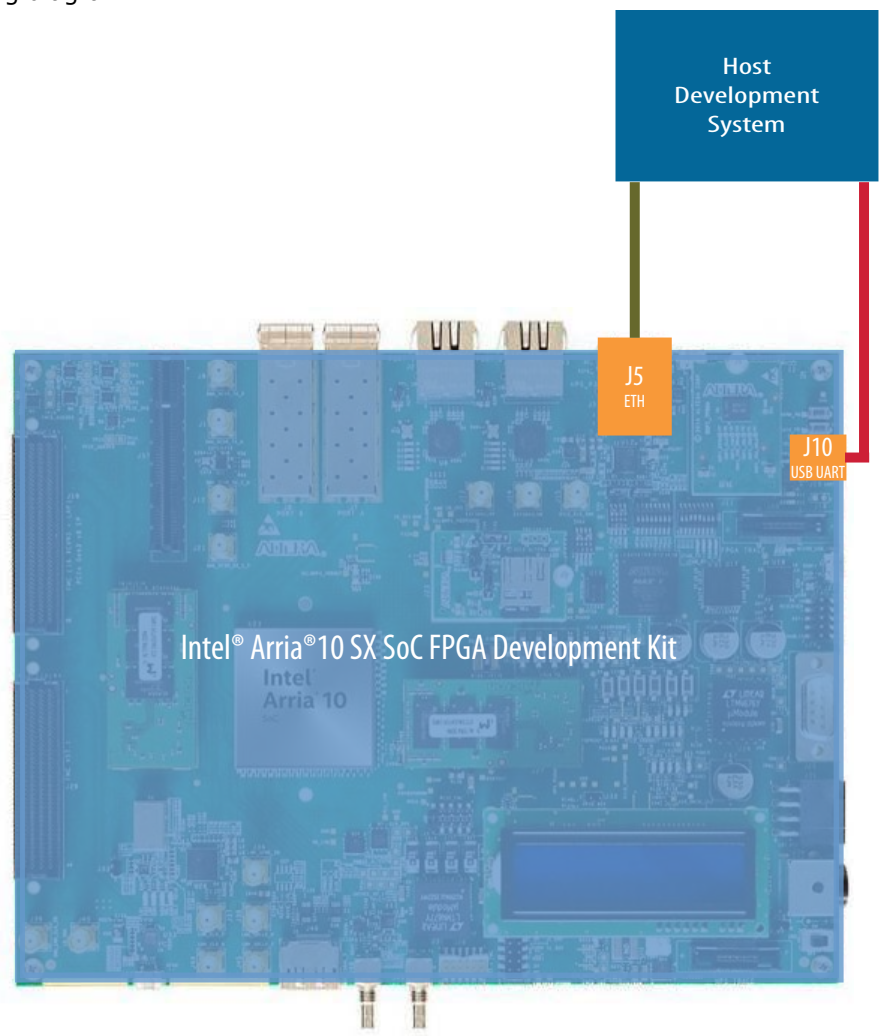
2. Ensure that the HILO cards are fitted correctly.

The Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S) includes two DDR4 HILO cards: the HPS memory (1GB) and the FPGA memory (2GB). Both the HPS Memory and FPGA Memory DDR4 HILO modules must be fitted as shown in the following image:



19.5.2.2. [SOC] Connecting the Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S) to the Host Development System

Connect the Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S) to your host development system via Ethernet and USB UART connections as shown in the following diagram:



19.5.3. [SOC] Configuring the SoC FPGA Development Kit UART Connection

The SoC FPGA development kit boards have USB-to-serial converters that allows the host computer to see the board as a virtual serial port:

- The Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S) has a FTDI USB-to-serial converter chip.
- The Agilex 7 FPGA I-Series Transceiver-SoC Development Kit has a USB-to-serial converter on the IO48 daughter card.

Ubuntu, Red Hat Enterprise Linux, and other modern Linux distributions have built-in drivers for the FTDI USB-to-serial converter chip, so no driver installation is necessary on those platforms.

On Microsoft Windows, the Windows SoC EDS installer automatically installs the necessary drivers. For details, see the SoC GSRD for your SoC FPGA development kit at the following URL: https://www.rocketboards.org/foswiki/Documentation/GSRD

The serial communication parameters are as follows:

- Baud rate: 115200
- Parity: None
- Flow control: None
- Stop bits: 1

On Windows, you can use utilities such as TeraTerm or PuTTY to connect the board. You can configure these utilities from their tool menus.

On Linux, you can use the Minicom utility. Configure the Minicom utility as follows:

1. Determine the device name associated with the virtual serial port on your host development system. The virtual serial port is typically named `/dev/ttyUSB0`.
   a. Before connecting the mini USB cable to the SoC FPGA development kit, determine which USB serial devices are installed with the following command:
      ```
      ls /dev/ttyUSB*
      ```
   b. Connect the mini USB cable from the SoC FPGA development kit to the host development system.
   c. Confirm the new device connection with the `ls` command again:
      ```
      ls /dev/ttyUSB*
      ```
2. If you do not have the Minicom application installed on the host development system, install it now.
   - OnRed Hat Enterprise Linux 8: `sudo yum install minicom`
   - On Ubuntu: `use sudo apt-get install minicom`
3. Configure Minicom as follows:
   a. Start Minicom:
      ```
      sudo minicom -s
      ```
   b. Under **Serial Port Setup** choose the following:
      - Serial Device: **/dev/ttyUSB0** (Change this value to match the system value that you found earlier, if needed)
      - Bps/Par/Bits: **115200 8N1**
      - Hardware Flow Control: **No**
      - Software Flow Control: **No**

      Press **[ESC]** to return to the main configuration menu.
   c. Select **Save Setup as dfl** to save the default setup. Then select **Exit**.

19.5.4. [SOC] Determining the SoC FPGA Development Kit IP Address

To determine the FPGA development kit IP address:

1. Open a Terminal session to the FPGA development kit via the UART connection and log in using the user name `root` and no password.

```
         Starting Record Runlevel Change in UTMP…
[  OK  ] Finished Record Runlevel Change in UTMP.
     9.553286] random: crng init done
[  OK  ] Finished Load/Save Random Seed.
[   10.084845] socfpga-dwmax ff800000.ethernet eth0: Link is Up – 1Gbps/Full
– flow control off
[   10.103287] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready

Poky (Yocto Project Reference Distro) 4.0.2 arria10-62747948036a ttyS0

arria10-62747948036a login:
```

2. Issue a `hostname` command to display the network name of the FPGA development kit board:

```
root@arria10-62747948036a:~# hostname
arria10-62747948036a
root@arria10-62747948036a:~#
```

   In this example, the network name of the board is `arria10-62747948036a`.

   Tip: You need this hostname later on to open an SSH connection to the FPGA development kit.

3. Confirm that you have a connection to the development kit from the development host with the `ping` command. Append the `.local` to the host name when you issue the `ping` command:

```
build-host:$ ping arria10-62747948036a.local -c4
PING arria10-62747948036a (192.168.0.23) 56(84) bytes of data.
64 bytes of data from arria10-62747948036a (192.168.0.23): icmp_seq=1 ttl=63
time=1.66ms
64 bytes of data from arria10-62747948036a (192.168.0.23): icmp_seq=1 ttl=63
time=1.66ms
64 bytes of data from arria10-62747948036a (192.168.0.23): icmp_seq=1 ttl=63
time=1.66ms
64 bytes of data from arria10-62747948036a (192.168.0.23): icmp_seq=1 ttl=63
time=1.66ms

--- arria10-62747948036a ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 1.664/2.037/2.283/0.238 ms
```

You can use the host name when you need to transfer files to the running system by appending the `.local` to the host name. For example, for the host name `arria10-62747948036a`, you can use `arria10-62747948036a.local`.

19.6. [SOC] Adding Compiled Graphs (AOT files) to the SD Card

An AOT file contains instructions for the FPGA AI Suite IP to "execute" in order to perform inference. The M2M design variant and the S2M design variant require different AOT files. The instructions in this section create both AOT files.

To add the compiled graphs to the development kit SD card:

Tip: If you completed the FPGA AI Suite Quick Start Tutorial in the *FPGA AI Suite Getting Started Guide*, you can skip steps 1-3.

1. Create the $COREDLA_WORK directory, if you have not already done so.
2. Prepare OpenVINO Model Zoo and Model Optimizer.
3. Prepare a model.

   Tip: If you completed the FPGA AI Suite Quick Start Tutorial in the *FPGA AI Suite Getting Started Guide*, you have already completed these first three steps.
4. Confirm that you have the following directory:

   ```
   $COREDLA_WORK/demo/models/public/resnet-50-tf/FP32/
   ```

   If you do not have this directory, confirm that you have completed the first three steps.
5. Compile the graphs.
6. Copy the compiled graphs to the SD card.

Related Information

FPGA AI Suite Quick Start Tutorial

19.6.1. [SOC] Preparing OpenVINO Model Zoo

These instructions assume that you have a copy of OpenVINO Model Zoo 2023.3 in your `$COREDLA_WORK/demo/open_model_zoo/` directory.

To download a copy of Model Zoo, run the following commands:

```
cd $COREDLA_WORK/demo
git clone https://github.com/openvinotoolkit/open_model_zoo.git
cd open_model_zoo
git checkout 2023.3.0
```

19.6.2. [SOC] Preparing a Model

A model must be converted from a framework (such as TensorFlow, Caffe, or Pytorch) into a pair of `.bin` and `.xml` files before the FPGA AI Suite compiler (`dla_compiler` command) can ingest the model.

The following commands download the ResNet-50 TensorFlow model and run theOpenVINO Open Model Zoo tools with the following commands:

```
source ~/build-openvino-dev/openvino_env/bin/activate

omz_downloader --name resnet-50-tf \
    --output_dir $COREDLA_WORK/demo/models/

omz_converter --name resnet-50-tf \
    --download_dir $COREDLA_WORK/demo/models/ \
    --output_dir $COREDLA_WORK/demo/models/
```

The `omz_downloader` command downloads the trained model to `$COREDLA_WORK/demo/models` folder. The `omz_converter` command runs model optimizer that converts the trained model into intermediate representation `.bin` and `.xml` files in the `$COREDLA_WORK/demo/models/public/resnet-50-tf/FP32/` directory.

The directory $COREDLA_WORK/demo/open_model_zoo/models/public/resnet-50-tf/ contains two useful files that do not appear in the $COREDLA_ROOT/demo/models/ directory tree:

- The README.md file describes background information about the model.
- The model.yml file shows the detailed command-line information given to Model Optimizer (mo.py) when it converts the model to a pair of .bin and .xml files

For a list OpenVINO Model Zoo models that the FPGA AI Suite supports, refer to the *FPGA AI Suite IP Reference Manual*.

Troubleshooting OpenVINO Open Model Zoo Converter Errors

You might get the following error while running the omz_converter on a TensorFlow model:

```
ValueError: Invalid filepath extension for saving. Please add either a
'.keras' extension for the native Keras format (recommended) or a '.h5'
extension. Use 'model.export(filepath)' if you want to export a SavedModel
for use with TFLite/TFServing/etc.
```

If you get this error, you can follow a process similar to the following example process that convert MobilenetV3 TensorFlow model to an OpenVINO model:

1. Run the following Python code that converts MobileNetv3 to Tensorflow .savedmodel format:

```
import os
import tensorflow as tf

COREDLA_WORK = os.environ.get("COREDLA_WORK")
DOWNLOAD_DIR = f"{COREDLA_WORK}/demo/models/"
OUTPUT_DIR = f"{COREDLA_WORK}/demo/models/"

# Set the image data format
tf.keras.backend.set_image_data_format("channels_last")

# Load the MobileNetV3Large model with the specified weights
model = tf.keras.applications.MobileNetV3Large(
    weights=str(
        f"{DOWNLOAD_DIR}/public/mobilenet-v3-large-1.0-224-tf/
weights_mobilenet_v3_large_224_1.0_float.h5"
    )
)
# Save the model to the specified output directory
model.export(filepath=f"{OUTPUT_DIR}/
mobilenet_v3_large_224_1.0_float.savedmodel")
```

2. Run the following command to convert the TensorFlow .savedmodel format to OpenVINO model format:

```
mo \
--input_model=$COREDLA_WORK/demo/models/
mobilenet_v3_large_224_1.0_float.savedmodel \
--model_name=mobilenet_v3_large_224_1.0_float
```

19.6.3. [SOC] Compiling the Graphs

The precompiled SD card image (.wic) provided with the FPGA AI Suite uses one of the following files as the IP architecture configuration file:

- Agilex 7 FPGA I-Series Transceiver-SoC Development Kit

  ```
  AGX7_Performance.arch
  ```

- Arria 10 SX SoC FPGA Development Kit

  ```
  A10_Performance.arch
  ```

To create the AOT file for the M2M variant (which uses the `dla_benchmark` utility), run the following command:

```
cd $COREDLA_WORK/demo/models/public/resnet-50-tf/FP32
dla_compiler \
--march $COREDLA_ROOT/example_architectures/<IP arch config file> \
--network-file ./resnet-50-tf.xml \
--foutput-format=open_vino_hetero \
--o $COREDLA_WORK/demo/RN50_Performance_b1.bin \
--batch-size=1 \
--fanalyze-performance
```

where *<IP arch config file>* is one of the IP architecture configuration files listed earlier.

To create the AOT file for the S2M variant (which uses the streaming inference app), run the following command:

```
cd $COREDLA_WORK/demo/models/public/resnet-50-tf/FP32
dla_compiler \
--march $COREDLA_ROOT/example_architectures/<IP arch config file> \
--network-file ./resnet-50-tf.xml \
--foutput-format=open_vino_hetero \
--o $COREDLA_WORK/demo/RN50_Performance_no_folding.bin \
--batch-size=1 \
--fanalyze-performance \
--ffolding-option=0
```

where *<IP arch config file>* is one of the IP architecture configuration files listed earlier.

After running either these commands, the compiled models and demonstration files are in the following locations:

Compiled Models	`$COREDLA_WORK/demo/RN50_Performance_b1.bin` `$COREDLA_WORK/demo/RN50_Performance_no_folding.bin`
Sample Images	`$COREDLA_WORK/demo/sample_images/`

Architecture File	`$COREDLA_ROOT/example_architectures/AGX7_Performance.arch`
	or
	`$COREDLA_ROOT/example_architectures/A10_Performance.arch`

19.6.4. [SOC] Copying the Compiled Graphs to the SD card

To copy the required demonstration files to the `/home/root/resnet-50-tf` folder on the SD card:

1. In the serial console, create directories to receive the model data and sample images:

   ```
   mkdir ~/resnet-50-tf
   ```

2. On the development host, use the secure copy (`scp`) command to copy the data to the board:

```
TARGET_IP=<Development Kit Hostname>.local
TARGET="root@$TARGET_IP:~/resnet-50-tf"
demodir=$COREDLA_WORK/demo
scp $demodir/*.bin $TARGET/.
scp -r $demodir/sample_images/ $TARGET/.
scp $COREDLA_ROOT/example_architectures/<architecture file> $TARGET/.
scp $COREDLA_ROOT/build_os.txt $TARGET/../app/
```

   where *<architecture file>* is one of the following files, depending on your development kit:

   - Agilex 7 FPGA I-Series Transceiver-SoC Development Kit

     ```
     AGX7_Performance.arch
     ```

   - Arria 10 SX SoC FPGA Development Kit

     ```
     A10_Performance.arch
     ```

3. [Optional] In the serial console run the `sync` command to ensure that the data is flushed to disk.

19.7. [SOC] Verifying FPGA Device Drivers

The device drivers should be loaded when the HPS boots. Verify that the device drivers are initialized by checking that `uio` files are listed in `/sys/class/uio` by running the following command:

```
ls /sys/class/uio
```

The command should show output similar to the following example:

```
uio0 uio1 uio2
```

If the drivers are not listed, refresh the modules by running the following command before checking again that the drivers are loaded:

```
uio-devices restart
```

19.8. [SOC] Running the Demonstration Applications

Depending on the SoC design example mode that you want run, follow the instructions in one of the following sections:

- [SOC] Running the M2M Mode Demonstration Application on page 102
- [SOC] Running the S2M Mode Demonstration Application on page 104

Important: Running the demonstration applications requires the terminal session that you opened in [SOC] Determining the SoC FPGA Development Kit IP Address on page 98.

19.8.1. [SOC] Running the M2M Mode Demonstration Application

The M2M dataflow model uses the `dla_benchmark` demonstration application. The S2M bitstream supports both the M2M dataflow model and the S2M dataflow model.

You must know the host name of the SoC FPGA development kit. If you do not know the development kit host name, go back to [SOC] Determining the SoC FPGA Development Kit IP Address on page 98 before continuing here.

To run inference on the SoC FPGA development kit:

1. Open an SSH connection to the SoC FPGA development kit:
   a. Start a new terminal session
   b. Run the following command:

      ```
      build-host:$ ssh <devkit_hostname>
      ```

      Where *<devkit_hostname>* is the host name you determined in [SOC] Determining the SoC FPGA Development Kit IP Address on page 98.

      Continuing the example from [SOC] Determining the SoC FPGA Development Kit IP Address on page 98, the following command would open an SSH connection:

      ```
      build-host:$ ssh arria10-62747948036a.local
      ```

2. In the SSH terminal, run the following commands:

   ```
   export compiled_model=~/resnet-50-tf/RN50_Performance_b1.bin
   export imgdir=~/resnet-50-tf/sample_images
   export archfile=~/resnet-50-tf/<architecture file>
   cd ~/app
   export COREDLA_ROOT=/home/root/app
   ./dla_benchmark \
      -b=1 \
      -cm $compiled_model \
      -d=HETERO:FPGA,CPU \
      -i $imgdir \
      -niter=8 \
      -plugin ./plugins.xml \
      -arch_file $archfile \
      -api=async \
      -groundtruth_loc $imgdir/TF_ground_truth.txt \
      -perf_est \
      -nireq=4 \
      -bgr
   ```

   where *<architecture file>* is one of the following files, depending on your development kit:

   - Agilex 7 FPGA I-Series Transceiver-SoC Development Kit

     ```
     AGX7_Performance.arch
     ```

   - Arria 10 SX SoC FPGA Development Kit

     ```
     A10_Performance.arch
     ```

The `dla_benchmark` command generates output similar to the following example output for each step. This example output was generated using an Agilex 7 FPGA I-Series Transceiver-SoC Development Kit.

```
[Step 11/12] Dumping statistics report
count:             8 iterations
system duration:   288.8584 ms
IP duration:       121.6040 ms
latency:           136.0344 ms
system throughput: 27.6952 FPS
number of hardware instances: 1
number of network instances: 1
```

```
IP throughput per instance: 65.7873 FPS
IP throughput per fmax per instance: 0.3289 FPS/MHz
IP clock frequency: 200.0000 MHz
estimated IP throughput per instance: 149.5047 FPS (500 MHz assumed)
estimated IP throughput per fmax per instance: 0.2990 FPS/MHz
[Step 12/12] Dumping the output values
[ INFO ] Dumping result of Graph_0 to result.txt, result.bin, result_meta.json,
and result_tensor_boundaries.txt
[ INFO ] Comparing ground truth file /home/root/quartus_uplift/resnet-50-tf/
sample_images/TF_ground_truth.txt with network Graph_0
top1 accuracy: 100 %
top5 accuracy: 100 %
[ INFO ] Get top results for "Graph_0" graph passed
```

19.8.2. [SOC] Running the S2M Mode Demonstration Application

To run the S2M (streaming) mode demonstration application, you need two terminal connections to the host.

You must know the host name of the SoC FPGA development kit. If you do not know the development kit host name, go back to [SOC] Determining the SoC FPGA Development Kit IP Address on page 98 before continuing here.

To run the streaming demonstration application:

1. Open an SSH connection to the SoC FPGA development kit:
   a. Start a new terminal session
   b. Run the following command:

      ```
      build-host:$ ssh <devkit_hostname>
      ```

      Where *<devkit_hostname>* is the host name you determined in [SOC] Determining the SoC FPGA Development Kit IP Address on page 98.

      Continuing the example from [SOC] Determining the SoC FPGA Development Kit IP Address on page 98, the following command would open an SSH connection:

      ```
      build-host:$ ssh arria10-62747948036a.local
      ```

2. Repeat step 1 to open a second SSH connection to the SoC FPGA development kit.
3. In a terminal session, run the following commands:

   ```
   cd /home/root/app
   ./run_inference_stream.sh
   ```

4. In the other terminal session, run the following commands:

   ```
   cd /home/root/app
   ./run_image_stream.sh
   ```

The first terminal session (where you ran the `run_inference_stream.sh` command) then shows output similar to the following example:

```
root@arria10-ea80b8d770e7:~/app# ./run_inference_stream.sh
Runtime arch check is enabled. Check started...
Runtime arch check passed.
Runtime build version check is enabled. Check started...
Runtime build version check passed.
Ready to start image input stream.
1 - coffee mug, score = 93.9453
2 - acoustic guitar, score = 38.6963
```

```
3 - desktop computer, score = 43.9209
4 - guacamole, score = 99.9512
5 - red wine, score = 55.1758
6 - stopwatch, score = 38.8428
7 - jigsaw puzzle, score = 100
```

For more details about the streaming applications and their command line options, refer to [SOC] Running the Streaming Demonstration on page 142.

19.8.3. [SOC] Troubleshooting the Demonstration Applications

If you receive an error similar to the following error while running either the S2M or M2M applications, check that all the board DIMMs are securely installed:

```
altera-msgdma ff200000.msgdma: dma_sync_wait: timeout! [ 251.812846] DMA Failed
```

20. [SOC] FPGA AI Suite SoC Design Example Run Process

This section describes the steps to run the demonstration application and perform accelerated inference using the SoC design example.

20.1. [SOC] Exporting Trained Graphs from Source Frameworks

Before running any demonstration application, you must convert the trained model to the OpenVINO intermediate representation (IR) format (`.xml`/`.bin`) with the OpenVINO Model Optimizer.

For details on creating the `.xml`/`.bin` files, refer to [SOC] Preparing a Model on page 99, which describes how to create `.xml`/`.bin`) files for ResNet50.

The rest of this guide assumes that the same file locations and file names are used as in [SOC] Preparing OpenVINO Model Zoo on page 99.

The `stream_image_app` used for the S2M variant of the SoC design example assumes that images are 224x224. For details, refer to [SOC] The image_streaming_app Application on page 143.

20.2. [SOC] Compiling Exported Graphs Through the FPGA AI Suite

The network as described in the `.xml` and `.bin` files (created by the Model Optimizer) is compiled for a specific FPGA AI Suite architecture file by using the FPGA AI Suite compiler.

The FPGA AI Suite compiler compiles the network and exports it to a `.bin` file with the format required by the OpenVINO Inference Engine. For instructions on how to compile the `.xml` and `.bin` files into AOT file suitable for use with the FPGA AI Suite IP, refer to [SOC] Compiling the Graphs on page 100

This `.bin` file created by the compiler contains the compiled network parameters for all the target devices (FPGA, CPU, or both) along with the weights and biases. The inference application imports this file at runtime.

The FPGA AI Suite compiler can also compile the graph and provide estimated area or performance metrics for a given architecture file or produce an optimized architecture file.

For the demonstration SD card, the FPGA bitstream has been built using one of the following IP architecture configuration files, so the architecture file for your development kit for compiling the OpenVINO™ Model:

- Agilex 7 FPGA I-Series Transceiver-SoC Development Kit

  ```
  AGX7_Performance.arch
  ```

- Arria 10 SX SoC FPGA Development Kit

  ```
  A10_Performance.arch
  ```

For more details about the FPGA AI Suite compiler, refer to the *FPGA AI Suite Compiler Reference Manual*.

21. [SOC] FPGA AI Suite SoC Design Example Build Process

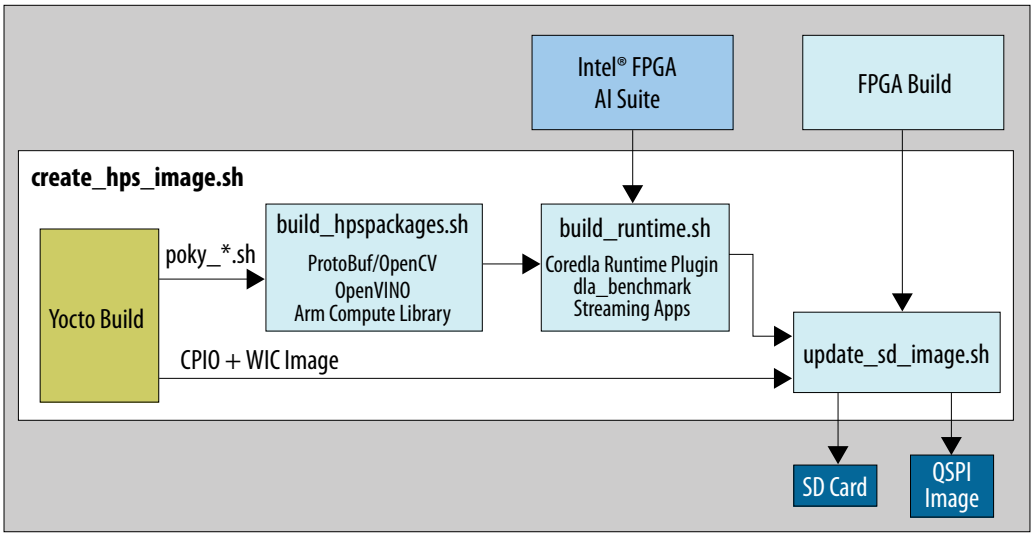
The SoC design example is built around a complete software and hardware solution.

The main building stages are:

- Build a Linux Distribution for a supported SoC FPGA development kit using a cross-compilation flow on a Linux system.
- Build the SoC design example Quartus Prime project
- Combine the FPGA and SoC Linux application and kernel onto an SD card

The following diagram illustrates the overall build process.

Figure 11. FPGA AI Suite SoC Design Example Build Process



21.1. [SOC] Building the Quartus Prime Project

This design example includes prepackaged bitstreams but you can also use the build script provided to build bitstreams with custom architectures or recreate the original bitstreams, subject to IP license limitations.

The Quartus Prime project consists of the FPGA AI Suite IP as well as IP to interface with the HPS and, in the case of the S2M variant, additional flow control IP.

[SOC] FPGA AI Suite SoC Design Example Quick Start Tutorial on page 83 shows how to use a specific architecture configuration file for the FPGA AI Suite IP, but you can use any other device-appropriate configuration file instead.

21.1.1. [SOC] Quartus Prime Build Flow

All FPGA AI Suite design examples are launched at the command line by running the `dla_build_example_design.py` script.

After the build script is invoked, it generates an FPGA AI Suite IP from the provided architecture file, creates an Quartus Prime build directory, builds the Quartus Prime project, and produces a bitstream.

The script has several command-line options to select the SoC design example variants. For details about the build script command options, refer to [SOC] Build Script Options on page 110.

Before launching the script, an architecture file is required. The FPGA AI Suite example architectures are located in directory $COREDLA_ROOT/`example_architectures/`.

Typical launch usage is as follows:

```
dla_build_example_design.py \
  -ed <variant> \
  -a <arch-file> \
  -n 1 \
  --build-dir <build directory> \
  --build \
  --output-dir <output directory>
```

The command options are defined as follows:

- The `-ed` option selects the SoC design example variant to be built. This option is case sensitive. The FPGA AI Suite SoC design examples comes as the following variants:

Table 12. SoC Design Example Variant

Variant setting	Description
`4_AGX7_M2M`	Builds a memory-to-memory (M2M) design for the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit
`4_AGX7_S2M`	Builds a streaming-to-memory (S2M) design for the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit
`4_A10_M2M`	Builds a memory-to-memory (M2M) design for the Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S)
`4_A10_S2M`	Builds a streaming-to-memory (S2M) design for the Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S)

- The `-a` option selects the architecture file.
- The `-n 1` option is the only legal value for this option when you build the SoC design example.
- The `--build-dir` option specifies the Quartus Prime build directory path.
- The `--build` option directs the script to call Quartus Prime and create the bitstreams. The `create_hps_image.sh` script uses these bitstreams when creating the SD card image.
- The `--output-dir` option specified the destination directory folder of the build output.

For a complete list of the build script options, refer to "Build Script Options" in *FPGA AI Suite PCIe-based Design Example User Guide*.

An example of building the Arria 10 S2M variant with the `A10_Performance` architecture in the folder `build_a10_perf` is as follows:

```
dla_build_example_design.py \
 -ed 4_A10_S2M \
 -n 1 \
 -a $COREDLA_ROOT/example_architectures/A10_Performance.arch \
 --build \
 --build-dir $COREDLA_WORK/a10_perf_bitstream \
 --output-dir $COREDLA_WORK/a10_perf_bitstream
```

After the design is built, the output products (`.sof` or `.rbf` files) must be combined with the SoC Linux system in order to be used. This is done in one of the steps in the `create_hps_image.sh` script.

Unlike non-SoC FPGA flows, the `.sof` file is not downloaded to the board via JTAG.

Do not attempt to download the `.sof` file over JTAG because downloading the file over JTAG does not result in a working system. Also, if you attempt to reprogram a running Linux system with a new `.sof` file, the Linux system crashes and the reprogramming results in an unpredictable outcome.

The FPGA device is programmed by booting the Linux system on the SoC via the SD card. For details about combining the build `.sof` file with the SD card image to create a functional solution, refer to [SOC] Building the Bootable SD Card Image (.wic) on page 112.

21.1.1.1. [SOC] Build Synchronization of FPGA with Software

For a system to function correctly, the release version of each FPGA AI Suite component, including the compiler, the runtime, and the FPGA AI Suite IP, must match.

In addition, the AOT file created by the FPGA AI Suite `dla_compiler` command must target the same architecture (`.arch`) file as the FPGA AI Suite IP.

When Quartus Prime compiles the FPGA AI Suite IP, it generates a build-hash that is embedded into the IP. The runtime software checks this build-hash during runtime and if the hashes do not match then the application aborts and displays a mismatch error.

The FPGA AI Suite SoC design example is always built with only one instance of the FPGA AI Suite IP.

21.1.2. [SOC] Build Script Options

The options available in the `dla_build_example_design.py` script are described in "Build Script Options" in *FPGA AI Suite PCIe-based Design Example User Guide*.

The options that are specific to the *FPGA AI Suite* SoC design example are as follows:

Option	Description
`-ed`, `--example-design-id`	Specify the SoC design example variant as follows:

continued...

Option	Description
	• Agilex 7 SoC M2M variant: `4_AGX7_M2M` • Agilex 7 SoC S2M variant: `4_AGX7_S2M` • Arria 10 SoC M2M variant: `4_A10_M2M` • Arria 10 SoC S2M variant, `4_A10_S2M`
`-n, --num-instances`	Number of IP instances to build (default: 1). For SoC designs, this number must be 1.

21.1.3. [SOC] Build Directory

The `dla_build_example_design.py` command creates an Quartus Prime build in the `hw` folder in the directory that you specify with the `--build-dir` command option. The project is named `top.qpf`. You can open this project in Quartus Prime software to review the build logs.

Within the build directory is a collection of command-line scripts that cover different parts of the design example build process. Use these scripts to rebuild parts of the design example if you alter the design. Otherwise, you typically do not need to run these scripts manually as the build process runs them for you.

The scripts provided are as follows:

- `create_project.bash`: This script cleans the build folder and resets the Quartus Prime project back to its default state, ready to be recompiled.
- `generate_sof.bash`: This script launches an Quartus Primer compilation from the command line
- `generate_rbf.bash`: This creates a `.rbf` file that is needed for the Arria 10 FPGA.
- `build_stream_controller.sh`: This script creates the Nios® V `.hex` file. This file holds the compiled Nios software that is embedded into the Nios subsystem.

21.1.3.1. [SOC] The `create_project.bash` Script

The `create_project.bash` script has two command line options:

```
create_project.bash <arch> <ip_path>
```

The arguments must be placed in order.

- The *<arch>* option provides the architecture to be selected when building the SoC Example Design. Use the architecture file that was used as a command option in `dla_build_example_design.py` command.

  The *<arch>* value to specify is extracted from the architecture file name:

Architecture File Name	Command Line Option
`AGX7_Performance.arch`	`AGX7_Performance_AGX7`
`AGX7_FP16_SoftMax.arch`	`AGX_FP16_SoftMax_AGX7`
`A10_Performance.arch`	`A10_Performance_A10`
`A10_FP16_SoftMax.arch`	`A10_FP16_SoftMax_A10`

- The *<ip_path>* option provides a full path to the FPGA AI Suite IP

This script deletes the old output products, cleans the Platform Designer system, and resets the project.

21.1.3.2. [SOC] The `generate_sof.bash` Script

The `generate_sof.bash` script launches a Quartus Prime shell and invokes the build process. Upon completion, a `.sof` file is ready for use.

21.1.3.3. [SOC] The `generate_rbf.bash` Script

The `generate_rbf.bash` script converts the `.sof` file into a `.rbf` (raw binary file) file that can be used by the Linux system.

21.1.3.4. [SOC] The `build_stream_controller.sh` Script

The `build_stream_controller.sh` script builds the Nios V application, and then generates a `.hex` file. The `.hex` file is embedded into the Nios V RAM by the Quartus Prime project. This file must be called `stream_controller.hex` and it must reside alongside the Quartus Prime project files (`top.qpf`).

This script has three command line options that must be entered in order:

```
build_stream_controller <quartus-project-file> <qsys system> <output-file>
```

This script is called as part of the `create_project.bash`, but you can call it manually if you modify the Nios V source code and need a new `.hex` file. The build script is typically called with the same options.

```
build_stream_controller.sh top.qpf qsys/dla.qsys stream_controller.hex
```

21.2. [SOC] Building the Bootable SD Card Image (`.wic`)

To create a bootable SD Card image (`.wic` file), the following components must be built:

- Yocto Image
- Yocto SDK Toolchain
- Arm Cross-Compiled OpenVINO

- Arm Cross-Compiled FPGA AI Suite Runtime Plugin
- Arm Cross-Compiled Demonstration Applications
- FPGA `.sof/.rbf` files

The `create_hps_image.sh` script performs the complete build process and combines all the necessary components into an SD card image that can be written to an SD card. For steps required to write an SD card image to an SD card, refer to [SOC] Writing the SD Card Image (.wic) to an SD Card on page 89.

The SD card image is assembled using Yocto (https://yoctoproject.org).

You must have a build system that meets the minimum build requirements. For details, refer to https://docs.yoctoproject.org/5.0.5/ref-manual/system-requirements.html#supported-linux-distributions.

The commands to install the required packages are shown in [SOC] Installing HPS Disk Image Build Prerequisites on page 86.

To perform the build, run the following commands:

```
cd $COREDLA_WORK/runtime
./create_hps_image.sh \
   -f <bitstream_directory>/hw/output_files \
   -o <output_directory> \
   -u \
   -m <FPGA_target>
```

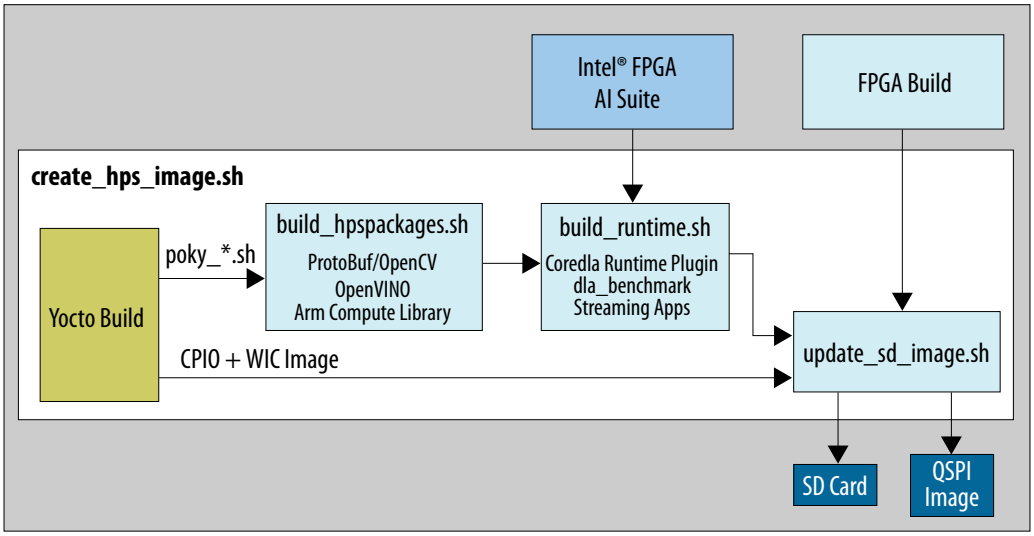
where *<FPGA_target>* is `arria10` or `agilex7_dk_si_agi027fa`

The `create_hps_image.sh` script performs the following steps:

1. Build the Yocto boot SD card image and Yocto SDK toolchain on page 114
2. Build the HPS Packages on page 115
3. Build the runtime on page 115
4. Update the SD card image on page 115

The following diagram illustrates the overall build process performed by the `create_hps_image.sh` script:

Figure 12. FPGA AI Suite SoC Design Example Build Process



Build the Yocto boot SD card image and Yocto SDK toolchain

The FPGA AI Suite Soc design example uses the Yocto Project Poky Distribution.

The Yocto images are based on Golden System Reference Designs, which you can find at the following URL: https://www.rocketboards.org/foswiki/Documentation/GSRD.

To customize the Yocto Poky distribution, modify the recipes found in layer `$COREDLA_ROOT/hps/ed4/yocto/meta-intel-coredla`.

More details can be found in [SOC] Yocto Build and Runtime Linux Environment on page 132.

The defined Yocto Image recipe is `coredla-image` and can be found in `$COREDLA_ROOT/hps/ed4/yocto/meta-intel-coredla/recipes-image/coredla-image.bb`.

A Yocto SDK is also built as part of the build and this SDK is used in subsequent build steps to cross-compile the software for the Arm HPS subsystem:

- **Agilex 7**:

  `$COREDLA_WORK/runtime/build_Yocto/build/tmp/deploy/sdk/poky-glibc-x86_64-coredla-image-armv8a-agilex7_dk_si_agi027fa-toolchain-4.2.3.sh`

- **Arria 10**:

  `$COREDLA_WORK/runtime/build_Yocto/build/tmp/deploy/sdk/poky-glibc-x86_64-lbs-image-poky-cortexa9t2hf-neon-arria10-toolchain-4.1.2.sh`

The SD card image (WIC file) is in the following location:

- **Agilex 7**:

  `$COREDLA_WORK/runtime/build_Yocto/build/tmp/deploy/images/agilex7_dk_si_agi027fa/*`

- **Arria 10**:

  `$COREDLA_WORK/runtime/build_Yocto/build/tmp/deploy/images/arria10/*`

By default, the `create_hps_image.sh` script builds Yocto from scratch. However, if a prebuilt Yocto build folder is available, you can specify the prebuilt Yocto folder via the `-y` option as follows:

```
./create_hps_image.sh \
   -y <prebuilt_Yocto_directory>
   -f <bitstream_directory>/hw/output_files \
   -o <output_directory>
   -u
   -m <FPGA_target>
```

where *<FPGA_target>* is `arria10` or `agilex7_dk_si_agi027fa`

This `-y` option loads the Yocto SDK from *<PREBUILT_YOCTO_DIR>*`/build/tmp/deploy/sdk/` and the `.wic` image from *<PREBUILT_YOCTO_DIR>*`/build/tmp/deploy/images/arria10/` without rerunning a Yocto build.

Build the HPS Packages

The HPS packages are built by the `build_hpspackages.sh` script.

This script cross-compiles OpenCV, OpenVINO, and the Arm-based OpenVINO runtime plugin.

Build the runtime

The runtimes are built by the `build_runtime.sh` script.

This script cross-compiles the OpenVINO FPGA AI Suite runtime plugin and demonstration applications for the SoC devices.

Update the SD card image

The SD card image is updated by the `update_sd_image.sh` script.

This script takes the output products from the previous build steps and builds a bootable SD Card image.

The software binaries are installed to the Ext4 partition under the `/home/root/app` directory. The RTL `fit_spl_fpga.itb` file is copied to the Fat32 partition.

The SD card image is updated only if you specify the `-u` option of the `create_hps_image.sh` command along with the location of the FPGA bitstream directory through the `-f` option.

You can skip updating the SD card while building the rest of the SoC Example Design by omitting the `-f` and `-u` options:

```
./create_hps_image.sh \
   -o <output_directory> \
   -m <FPGA_target>
```

where <*FPGA_target*> is `arria10` or `agilex7_dk_si_agi027fa`

When you skip updating the SD card image, you can build bitstreams and an HPS image (Yocto, HPS packages, FPGA AI Suite runtime) concurrently. You can update the SD card image (`.wic` file) image after all the files are ready:

```
./create_hps_image.sh \
   -y ./build_Yocto
   -f <bitstream_directory>/hw/output_files \
   -o <output_directory> \
   -u \
   -m <FPGA_target>
```

where <*FPGA_target*> is `arria10` or `agilex7_dk_si_agi027fa`

22. [SOC] FPGA AI Suite SoC Design Example Quartus Prime System Architecture

The FPGA AI Suite SoC design examples provide two variants for demonstrating the FPGA AI Suite operation.

All designs are Platform Designer based systems.

There is a single top-level Verilog RTL file for instantiating the Platform Designer system.

These two variants demonstrate FPGA AI Suite operations in the two most common usage scenarios. These scenarios are as follows:

- **Memory to Memory (M2M)**: In this variant, the following steps occur:
  1. The Arm processor host presents input data buffers to the FPGA AI Suite that are stored in a system memory.
  2. The FPGA AI Suite IP performs an inference on these buffers.
  3. The host system collects the inference results.

  This variant demonstrates the simplest use-case of the FPGA AI Suite.
- **Streaming to Memory (S2M)**: This variant offers a superset of the M2M functionality. The S2M variant demonstrates sending streaming input source data into the FPGA AI Suite IP and then collecting the results. An Avalon streaming input captures live input data, stores the data into system memory, and then automatically triggers FPGA AI Suite IP inference operations.

  You can use this variant as a starting point for larger designs that stream input data to the FPGA AI Suite IP with minimal host intervention.

22.1. [SOC] FPGA AI Suite SoC Design Example Inference Sequence Overview

The FPGA AI Suite IP works with system memory. To communicate with the system memory, the FPGA AI Suite has its own multichannel DMA engine.

This DMA engine pulls input commands and data from the system memory. It then writes the output data back to this memory for a Host CPU to collect.

When running inferences, the FPGA AI Suite continually reads and writes intermediate buffers to and from the system memory. The allocation of all the buffer addresses is done by the FPGA AI Suite runtime software library.

Running an inference requires minimal interaction between the host CPU and the IP Registers.

The system memory must be primed with all necessary buffers before starting a machine learning inference operation. These buffers are setup by the FPGA AI Suite runtime library and application that runs on the host CPU.

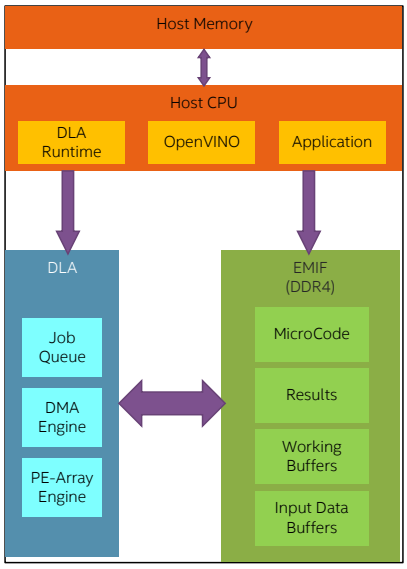
After the setup is complete, the host CPU pushes a job into the IP registers.

The FPGA AI Suite IP now performs a single inference. The job-queue registers in the IP are FIFO based, and the host application can store multiple jobs in the system memory and then prime multiple jobs inside the IP. Each job stores results in system memory and results in a CPU interrupt request.

For each inference operation in the M2M model, the host CPU (HPS) must perform an extensive data transfer from host (HPS) DDR memory to the external DDR memory that is allocated to the FPGA AI Suite IP. As this task has not been FPGA-accelerated in the design, the host operating system and FPGA AI Suite runtime library must manually transfer the data. This step consumes significant CPU resources. The M2M design uses a DMA engine to help with the data transfer from HPS DDR to the allocated DDR memory.

The FPGA AI Suite inference application and library software are responsible for keeping the FPGA AI Suite IP primed with new input data and responsible for consuming the results.

Figure 13. FPGA AI Suite SoC Design Example Inference Sequence Overview



For a detailed overview of the FPGA AI Suite IP inference sequence, refer to the *FPGA AI Suite IP Reference Manual*.

22.2. [SOC] Memory-to-Memory (M2M) Variant Design

The memory-to-memory (M2M) variant of the SoC design example illustrates a technique for embedded (SoC) FPGA AI Suite operations where the input data sets are primarily drawn from a memory or file sources. In this scenario, the data is typically not real time and is processed as fast as possible.

This design combines the HPS SoC FPGA device (Arm Cortex*-A9 on Arria 10 or Arm Cortex-A53 on Agilex 7) with an additional DMA engine to allow for efficient transfer of data to and from the CPU and system memory.

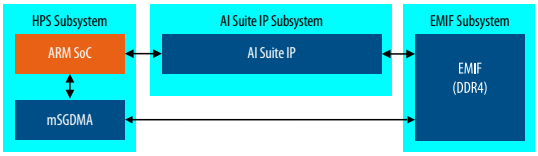
In the M2M design, the source data originally resides within the host CPU domain on an SD card. The application uses the DMA controller to move the host-side data to the device side domain. This movement mimics the process that an application would typically do.

The test program then initiates FPGA AI Suite IP inference operations and wait for the IP to complete its process. Command-line options to the user application define how many inferences are executed.

After the inference operation is completed, the application uses the DMA to transfer the results back from external memory to the host domain. The results are then displayed on the Linux console.

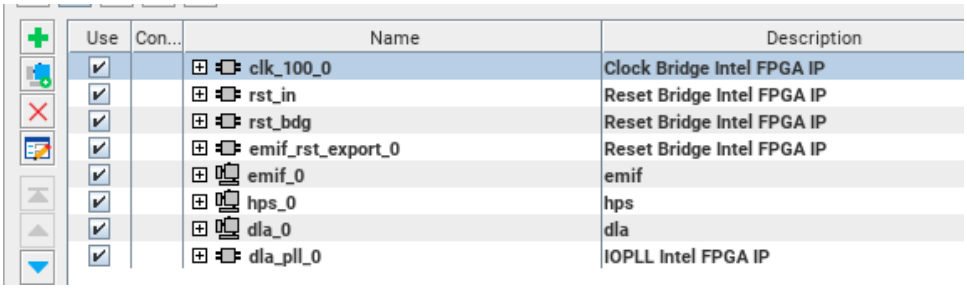
The Intel modular scatter-gather direct memory access (mSGDMA) controller IP provides this DMA facility.

Figure 14. Block Diagram of M2M Variant



The M2M variant appears in Platform Designer as follows:

Figure 15. M2M Variant in Platform Designer



22.2.1. [SOC] The mSGDMA Intel FPGA IP

The modular scatter-gather direct memory access (mSGDMA) Intel FPGA IP used in this design example serves as an example of how you can integrate a DMA into your own system. You can replace this DMA engine by another 3rd party controller.

The FPGA AI Suite runtime software must be modified if you want to use another DMA engine.

22.2.2. [SOC] RAM considerations

An FPGA-based external memory interface is used to store all machine learning input, output, and intermediate data.

The FPGA AI Suite IP uses the DDR memory extensively in its operations.

Typically, you dedicate a memory to the FPGA AI Suite IP and avoid sharing it with the host CPU DDR memory. Although a design can use the host memory, other services that use the DDR memory impact the FPGA AI Suite IP performance and increase non-determinism in inference durations. Consider this impact when you choose to use a shared DDR resource.

The FPGA AI Suite IP requires an extensive depth of memory that prohibits the use of onboard RAM such as M20Ks. Consider the RAM/DDR memory footprint when you design with the FPGA AI Suite IP.

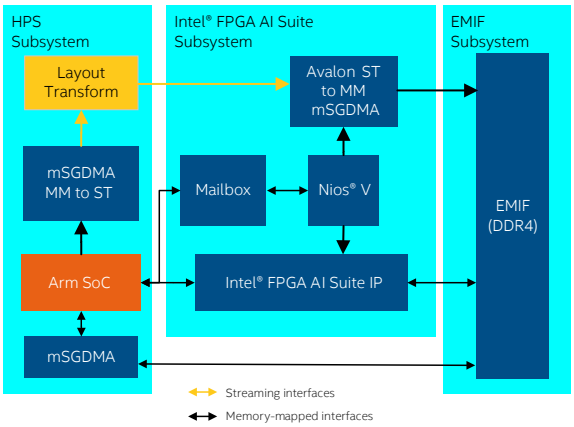
22.3. [SOC] Streaming-to-Memory (S2M) Variant Design

The streaming-to-memory (S2M) variant of the SoC design example builds on top of the M2M design to demonstrate a method of using the FPGA AI Suite IP with continuously streaming input data.

The application example is a typical video stream being processed with ResNet50 to detect physical objects in the images, such as a person, cat, or dog.

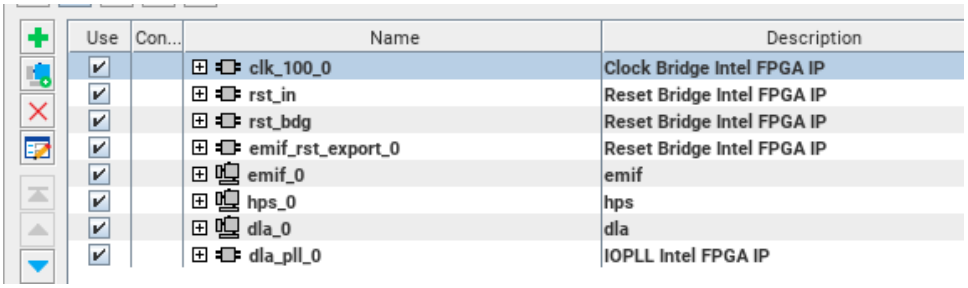
In the example, test images are stored on the SD card file system. These images are loaded into host memory and a DMA (memory-to-streaming) IP is used to create a simulated video stream.

Figure 16. Block Diagram of S2M Variant



The s2M variant appears in Platform Designer as follows:

Figure 17. S2M Variant in Platform Designer



22.3.1. [SOC] Streaming Enablement for FPGA AI Suite

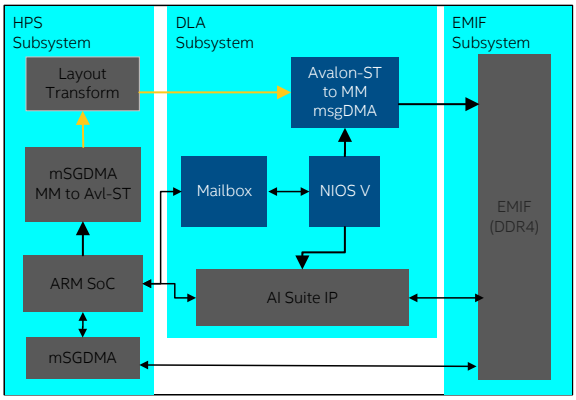
In an M2M system, input buffers are provided by the host CPU. However, in a streaming system (S2M), input buffers are created by an external hardware stream. For the FPGA AI Suite IP to process this external stream, several operations must happen in a coordinated way:

- The raw stream data must pass through a layout-transform IP core to reformat the raw data into an FPGA AI Suite compliant data format
- The formatted data must be written into system memory at specific locations, known only to the host application and the FPGA AI Suite software library at run time.
- The FPGA AI Suite IP job queue must be primed at the correct time, in synchronization with the input stream buffers, such that the FPGA AI Suite IP starts an inference immediately upon a new input buffer becoming ready.

Within Platform Designer, a Nios V based subsystem is added alongside the FPGA AI Suite IP to provide the streaming capabilities. This subsystem highlighted in blue in the block diagram that follows.

In the diagram, the yellow interconnect lines indicate Avalon streaming interfaces, and the black interconnect lines indicate memory-mapped interfaces.

Figure 18. Nios V Streaming Subsystem



22.3.2. [SOC] Nios V Subsystem

Three IP modules make up the Nios V subsystem:

- **mSGDMA** (Avalon streaming to memory-mapped mode). This module is used to take the formatted input data stream and place it into system memory to create the FPGA AI Suite IP input buffers.
- **Mailbox** (On-Chip Memory II Intel FPGA IP). This module is used to provide a communication API between the host-application and the Nios Subsystem. FPGA AI Suite IP command and status message are conveyed through this interface.
- **Nios V processor**. This module manages the FPGA AI Suite IP job-queue, mailbox and mSGDMA buffer allocation. Using the Nios V processor offloads the latency-sensitive ingest and buffer management from the HPS.

All C source-code to the Nios V application is provided. You can modify the Nios software to enable third-party DMA controllers, if required.

22.3.3. [SOC] Streaming System Operation

Two operations must occur in parallel for streaming systems to work:

- Buffers must be managed appropriately. That is, buffers of streaming data must be written into system memory at a specific location ready for the FPGA AI Suite IP to process.
- Inference jobs must be managed appropriately. That is, inference jobs must be primed at the correct time to process the new buffer.

22.3.3.1. [SOC] Streaming System Buffer Management

Before machine learning inference operations can occur, the system requires some initial configuration.

As in the M2M variant, the S2M application allocates sections of system memory to handle the various FPGA AI Suite IP buffers at startup. These include the graph buffer, which contains the weights, biases and configuration, and the input and output buffers for individual inference requests.

Instead of fully managing these buffers, the input-data buffer management is offloaded to the Nios processor. The Nios processor owns the Avalon streaming to memory-mapped mSGDMA, and the processor programs this DMA to push the formatted data into system memory.

As the buffers are allocated at startup, the input buffer locations are written into the mailbox. The Nios V processor then holds onto these buffers until a new set is received. All stream data is now constantly pushed into these buffers in a circular ring-buffer concept.

22.3.3.2. [SOC] Streaming System Inference Job Management

In a M2M system, the host CPU handles pushing jobs into the FPGA AI Suite IP job queue by writing to the IP registers. In the streaming configuration, this task is offloaded to the Nios system and must be done in a coordinated way with the input buffer writing.

In streaming mode, the job-queue management is pushed to the mailbox instead of being managed by the host application. The job queue entry is then received by the Nios processor. After an input buffer is written, the mSGDMA interrupts the Nios processor, and the Nios processor now pushes one job into the FPGA AI Suite IP.

For every buffer stored by the mSGDMA, the Nios processor attempts to start another job.

For more details about the Nios V Stream Controller and the mailbox communication protocol, refer to [SOC] Streaming-to-Memory (S2M) Streaming Demonstration on page 136.

22.3.4. [SOC] Resolving Input Rate Mismatches Between the FPGA AI Suite IP and the Streaming Input

When designing a system, the stream buffer rate should be matched to the FPGA AI Suite IP inferencing rate, so that the input data does not arrive faster than the IP can process it.

The SoC design example has safeguards in the Nios subsystem for when the input data rate exceeds the FPGA AI Suite processing rate.

To prevent input buffer overflow (potentially writing to memory still being processed by the FPGA AI Suite IP), the Nios subsystem has a buffer dropping technique built into it. If the subsystem detects that the FPGA AI Suite IP is falling behind, it starts dropping input buffers to allow the IP to catch up.

Using mailbox commands, the host application can check the queue depth level of the Nios subsystem and see if the subsystem needs to drop input data.

Depending on the buffer processing requirements of a design, dropping input data might not be considered a failure. It is up to you to ensure that the IP inference rate meets the needs of the input data.

If buffer dropping is not desired, you can try to alleviate buffer dropping and increase FPGA AI Suite IP performance with the following options:

- Configure a higher performance `.arch` file (IP configuration), which requires more FPGA resource. The `.arch` can be customized for the target machine learning graphs.
- Increase the system clock-speed.
- Reduce the size of the machine learning network, if possible.
- Implement multiple instances of the FPGA AI Suite IP and multiplex input data between them.

22.3.5. [SOC] The Layout Transform IP as an Application-Specific Block

The layout transformation IP in the S2M design is provided as RTL source as an example layout transformation within a video inferencing application.

The flexibility of the FPGA AI Suite and the scope of projects it can support means that a layout transformation IP cannot serve all inference applications.

Each target application typically requires its own layout transformation module to be designed. System architectures need to budget for this design effort within their project.

Input data to the FPGA AI Suite IP must be formatted in memory so that the data matches the structure of the IP PE array and uses FP16 values.

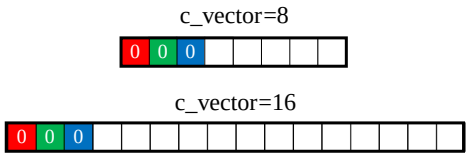
The structure of the PE array is defined by the architecture file and the `c_vector` parameter setting describes the number of layers required for the input buffer. Typical `c_vector` values are 8, 16, and 32.

When considering streaming data, the `c_vector` can be understood in comparison to the number of input channels of data that is present. For example, video has red, green, and blue channels that make up each pixel color. The following diagram shows how the video channels map to the input data stream required by the FPGA AI Suite IP.

Figure 19. Input Data Steam Mapping

The S2M design demonstrates an example of video streaming. Pixel data is sent through the layout-transform as RGB pixels, where each color is considered an input channel of data.

As the input data comprise only three channels of input data, the input data must be padded with zeros for any unused channels. The following diagram shows an example of two architectures, one with `c_vector` value of 8 and another with `c_vector` value of 16.



In the first example where `c_vector` is set to 8, the first pixel of RGB is placed on the input stream filling the first 3 channels, but there are 5 more channels remaining that must be initialized. These are filled with zero (represented by the white squares). This padded stream is then fed into the Nios subsystem.

This example layout transform does not support input folding. Input folding is an input preprocessing step that reduces the amount of zero padding in the `c_vector`. This folding then enables more efficient use of the dot product engine in the FPGA AI Suite IP. The efficiency gains can be significant depending on the graph and C_VEC. For more details, refer to "Input Folding" in *FPGA AI Suite IP Reference Manual*.

Related Information

"Parameter: `c_vector`" in *FPGA AI Suite IP Reference Manual*

22.3.5.1. [SOC] Layout Transform Considerations

Pixels are typically 8-bit integer values, and the FPGA AI Suite requires FP16 values. As well as the `c_vector` padding, the layout transformation module converts the integer values to floating-point values.

The S2M example is a video-oriented demonstration. For networks such as ResNet50, the input pixel data must further be manipulated with a "mean" and "variance" value. The layout transformation module performs basic operation of Y=A*B+C operation on each pixel to meet the needs of a ResNet50 graph trained for ImageNet.

22.3.5.2. [SOC] Layout Transform IP Register Map

The layout transform IP has the following register address space:

Table 13. Register Address Space of Layout Transformation IP

Register	Address Range	Description
Control	0x00	Main Control Register
C-Vector	0x04	Global C-Vector Control
Reserved	0x08 .. 0x03f	
Variance	0x40 .. 0x7f	Variance values per plane
Mean	0x80 .. 0xbf	Mean Values per plane
Reserved	0xc0 .. 0xff	

Control Register (0x00)

This is the global control register. When altering other CSR registers, software should issue a reset to the LT module via this register to commission the new settings. The stream then generates outputs based on the new settings and flush out all stale data.

Attempting to alter the configuration registers of the LT during active streaming generates undefined output data.

Table 14. Control Register Layout

Bit Location	Register Description	Attributes
0	**Reset** '1' = **In reset**: All streaming input data is discarded and no output data generated. '0' = **Running**: Streaming input data produces LT output data.	RW
31:1	Reserved	RO

C-Vector Register (0x04)

This register should be configured to match the C-Vector value of the architecture built into the FPGA AI Suite. This value need only be written once at startup as the architecture of the FPGA AI Suite is fixed at build time.

Table 15. C-Vector Register Layout

Bit Location	Register Description	Attributes
5:0	**C-vector** Value must match architecture of the DLA as defined in the .arch file	RW
31:6	Reserved	RO

Variance Registers (0x40 .. 0x7F)

Each plane has a unique variance value. Software must configure a value for each plane. The values are stored in FP32 format.

There are 16 registers in this section, where each register relates to a given plane. This register is write-only and returns 0xFFFFFFFF when reading.

Table 16. Variance Registers Layout

Bit Location	Register Description	Attributes
31:0	**FP32 formatted Variance value per plane**	WO

Mean Registers (0x80 .. 0xBF)

Each plane has a unique a mean value. Software must configure a value for each plane. The values are stored in FP32 format.

There are 16 registers in this section, where each register relates to a given plane. This register is write-only and returns 0xFFFFFFFF when reading.

Table 17. Mean Registers Layout

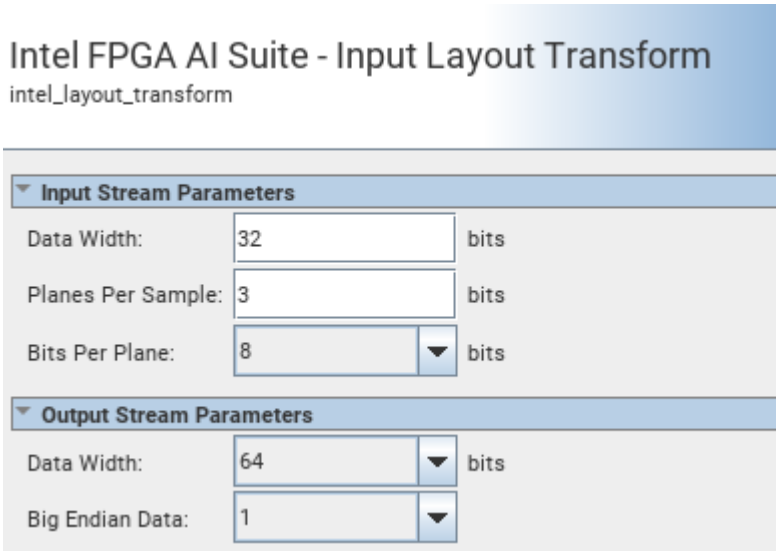
Bit Location	Register Description	Attributes
31:0	**FP32 formatted Mean value per plane**	WO

22.3.5.3. [SOC] Layout Transform Configuration Options

The example layout transform has a range of parameters to adjust to the data width based on the number of input planes being processed.

A maximum of 16 CSR mean and variance values are supported. The **Planes per sample** field sets this upper threshold.

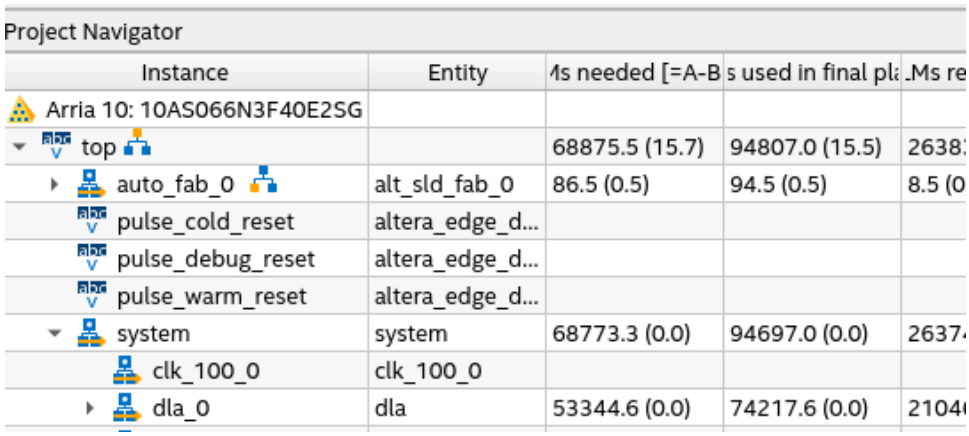
All output data is in FP16 format which is the expected input format for the FPGA AI Suite.



22.4. [SOC] Top Level

After the Quartus Prime project has finished compiling, the design should look similar to the following image in the Quartus Prime Project Navigator:

Figure 20. SoC Design Example Hierarchy



The top-level Verilog file and HPS configuration is derived directly from the GSRD designs located at RocketBoards.org:

- For more information about the GSRD for the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit, refer to the following URL https://www.rocketboards.org/foswiki/Documentation/AgilexSoCGSRDSIAGI027
- For more information about the GSRD for the Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S), refer to the following URL: https://www.rocketboards.org/foswiki/Documentation/arria10SoCGSRD

The GSRD designs have been modified to include the FPGA AI Suite IP. All unnecessary logic has been removed, which provides a concise design example.

The main FPGA AI Suite SoC design example is contained within a single Platform Designer system, called **system**. Double-click this node in the Quartus Prime Project Navigator to launch Platform Designer.

Related Information

- *GSRD for Agilex 7 I-Series Transceiver-SoC DevKit (4x F-Tile)* at RocketBoards.org
- *Arria 10 SoC GSRD* at RocketBoards.org

22.4.1. [SOC] Clock Domains

There are three main clocks within this design. All the clocks are considered asynchronous to each other. The SDC file provided has the clocking constraints for this design.

The design clocks are as follows:

- **100MHz Board clock**

  This clock is used for all mSGDMA infrastructure and CPU CSR interfaces. The HPS AXI interfaces all run off this clock.

- **200MHz DLA clock**

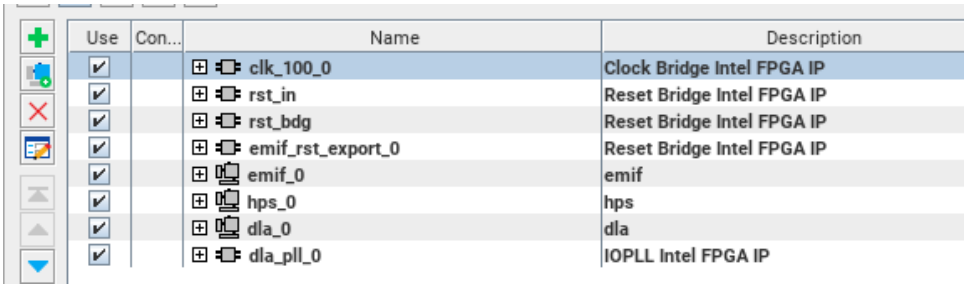
  This clock is used only by the FPGA AI Suite IP. It feeds the `dla_clk` pin and is used inside FPGA AI Suite IP PE array.

- **266MHZ DDR Clock**

  This is used for the DDR controller and interconnect between the DLA and DDR. This interface is used by the DLA to transfer workloads back and forth to system memory.

22.5. [SOC] The SoC Design Example Platform Designer System

At the center of the SoC design example is the Platform Designer system.



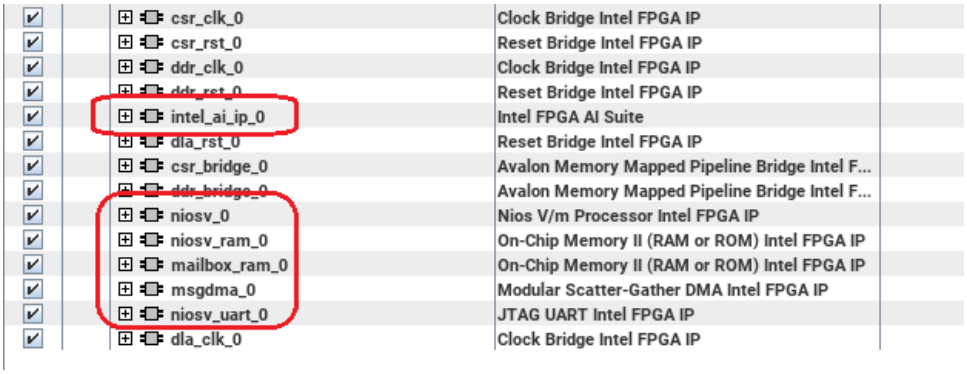
In Platform Designer, the SoC design example is separated into three hierarchical layers, the:

- **emif_0** : This layer contains the FPGA DDR4 External Memory Interface
- **dla_0** : This layer contains all the DLA IP and infrastructure IP
- **hps_0** : This layer contains all the ARM-HPS, ARM-EMIF and infrastructure IP for the ARM

The division of hierarchy demonstrates the sections of the design that are relevant to the solution. For example, if you want to target another board with a different external memory interface, you need to edit only the **emif_0** layer.

22.5.1. [SOC] The dla_0 Platform Designer Layer (`dla.qsys`)

The **dla_0** layer contains the FPGA AI Suite IP and the Nios V subsystem to provide streaming capabilities.

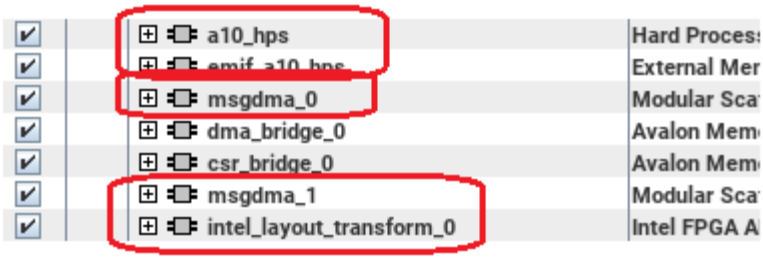


When incorporating the FPGA AI Suite IP into a custom design, you can use the `dla.qsys` file as a starting point for the new design.

22.5.2. [SOC] The hps_0 Platform Designer Layer (hps.qys)

The **hps_0** layer contains the HPS, an mSGDMA instance (`msgdma_0`) for the FPGA AI Suite runtime, and an mSGDMA instance (`msgdma_1`) for the streaming generation app (S2M variant only).

The example layout transform is also located here and can be replaced by your version.



22.6. [SOC] Fabric EMIF Design Component

The design provides a 266MHz DDR4-64Bit Avalon-based memory controller. This EMIF is used solely by the DLA.

The FPGA AI Suite IP memory interface is configured to be 512 bits wide. The EMIF interface is setup to complement this configuration.

22.7. [SOC] PLL Configuration

The FPGA AI Suite IP is designed to operate at high f_{MAX} rates in Intel FPGA devices. The SoC design example provides an IOPLL that provides the IP with a fast clock.

In the design example the external board 100 MHz reference clock is fed into the PLL and a 200 MHz output clock is produced. This clock feeds the FPGA AI Suite IP directly.

You can remove or alter this PLL if you want to profile different performance curves of the system.

The FPGA AI Suite `dla_benchmark` application has no runtime method to dynamically determine the PLL operating frequency in the SOC design. The frequency of 200 MHz has been set as a constant in this application source code. If you alter the PLL frequency, you must also alter the `dla_benchmark` application to match the new clock frequency. This matching ensures that the benchmark metrics accurately reflect the performance.

See `dla_mmd_get_coredla_clock_freq` in `$COREDLA_WORK/runtime/coredla_device/mmd/hps_platform/acl_hps.cpp` and change the return value accordingly.

23. [SOC] FPGA AI Suite SoC Design Example Software Components

The FPGA AI Suite SoC design example contains a software environment for the runtime flow.

The software environment for the supported FPGA development kits consists of the following components:

- Yocto build and runtime Linux environment
- Intel Distribution of OpenVINO toolkit Version 2023.3 LTS (Inference Engine, Heterogeneous plugin)
- OpenVINO Arm CPU plugin
- FPGA AI Suite runtime plugin
- MMD hardware library

The FPGA AI Suite SoC design example contains the source files, Makefiles, and scripts to cross compile all the software for the supported FPGA development kit. The Yocto SDK provides the cross compiler, and is the first component that must be built.

The machine learning network graph is compiled separately using the OpenVINO Model Optimizer and the FPGA AI Suite compiler (`dla_compiler`) command. When you compile the graph for the FPGA AI Suite SoC design example, ensure that you specify the `--foutput-format=open_vino_hetero` and `-o <path_to_file>/CompiledNetwork.bin` options.

The AOT file from the FPGA AI Suite compiler contains the compiled network partitions for FPGA and CPU devices along with the network weights. The network is compiled for a specific FPGA AI Suite architecture and batch size.

The SoC flow does not support the Just-In-Time (JIT) flow because Arm libraries are not available for the FPGA AI Suite compiler.

An architecture file (`.arch`) describes the FPGA AI Suite IP architecture to the compiler. You must specify the same architecture file to the FPGA AI Suite compiler and to the FPGA AI Suite design example build script (`dla_build_example_design.py`).

The runtime stack cannot program the FPGA with a bitstream. The bitstream must be built into the SD card (`.wic`) image that is used to program the flash card, as described in [SOC] (Optional) Create an SD Card Image (.wic) on page 84 and [SOC] Writing the SD Card Image (.wic) to an SD Card on page 89.

The runtime inference on the SoC FPGA device uses the OpenVINO Arm CPU plugin. To enable fallback to the OpenVINO Arm CPU plugin for graph layers that are not supported on the FPGA, the device flag must be set to `HETERO:FPGA,CPU` during the AOT compile step and when you run the `dla_benchmark` command.

In some cases, a layer might be supported by the FPGA even though the OpenVINO Arm CPU plugin does not support the layer. This support is handled by the HETERO plugin and the layer is executed on the FPGA as expected. As an example, 3D convolution layers are not supported by the OpenVINO Arm CPU plugin but still work properly provided that the `.arch` file used for the FPGA AI Suite IP configuration has enabled support for 3D convolutions.

Related Information

- "Running the Graph Compiler" in *FPGA AI Suite Getting Started Guide*
- "Compiling a Graph" in *FPGA AI Suite Compiler Reference Manual*
- "Architecture Description File Parameters" in *FPGA AI Suite IP Reference Manual*
- "Compilation Options (`dla_compiler` Command Options)" in *FPGA AI Suite Compiler Reference Manual*

23.1. [SOC] Yocto Build and Runtime Linux Environment

The Linux runtime and build environment is based on the Yocto build system. Yocto uses a build model based on the concept of layers and recipes.

The Yocto layers and recipes for FPGA AI Suite SoC design example are in `$COREDLA_ROOT/hps/ed4/yocto/`. The recipes extend the standard Golden System Reference Design (GSRD) that is used as the basis for the SoC design example system. For further details on setting up Yocto for Intel SoC devices see the GSRD documentation.

The rest of this section describes key recipes in the `metal-intel-fpga-coredla` layer.

Related Information

- https://yoctoproject.org
- https://www.rocketboards.org/foswiki/Documentation/GSRD

23.1.1. [SOC] Yocto Recipe: `recipes-core/images/coredla-image.bb`

This Yocto recipe customizes the image used on the SoC design example.

The `IMAGE_INSTALL:append` section defines extra packages for the FPGA AI Suite SoC example design. In particular, the `msgdma-userio`, `uio-devices`, and `kernel-modules` recipes are enabled to support the UIO and mSGDMA. The `WKS_FILES:*` section specifies which definition is used for building the SD card image.

23.1.2. [SOC] Yocto Recipe: `recipes-bsp/u-boot/u-boot-socfpga_%.bbappend`

This Yocto recipe appends to the `meta-intel-fpga/recipes-bsp` recipe and enables the FPGA to SDRAM bridge, if it is required for the device target. This bridge is not required for Arria 10 designs.

On devices that require the bridge, the bridge allows mSGDMA to access the HPS SDRAM. This access exposes the full HPS SDRAM to the FPGA device.

23.1.3. [SOC] Yocto Recipe: recipes-drivers/msgdma-userio/msgdma-userio.bb

This Yocto recipe enables an out-of-kernel (userspace) build of the `msgdma-userio` character driver used for transfer data to and from the HPS to the FPGA DDR memory. This driver calls into the DMA_ENGINE API exposed by the `altera-msgdma` kernel driver. The kernel module `altera-msgdma.ko` is enabled in `recipe-kernel/linux` kernel configuration file: `recipes-kernel/linux/files/enable-coredla-mod.cfg`.

Device tree settings are set to assign base address, IRQ for the `altera-msgdma` and associated to the `msgdma-userio` driver. These can be found in `recipes-kernel/linux/files/coredla-dts.patch`.

23.1.4. [SOC] Yocto Recipe: recipes-drivers/uio-devices/uio-devices.bb

This Yocto recipe installs a service which starts up the uio drivers within the system.

The `uio_pdrv_genirq` driver provides user mode access to mapping and umapping CSR registers in the FPGA AI Suite IP.

The kernel modules `uio.ko` and `uio_pdev_genirq.ko` are enabled in the `recipe-kernel/linux` kernel configuration file: `recipes-kernel/linux/files/enable-mod.cfg`.

The device tree settings are set to assign base addresses and IRQs for the FPGA AI Suite IP, the stream controller, and the layout transform module. These can be found in `recipes-kernel/linux/files/coredla-dts.patch`.

23.1.5. [SOC] Yocto Recipe: recipes-kernel/linux/linux-socfpga-lts_%.bbappend

This Yocto recipe applies the `0001-altera-msgdma.patch`, which does the following fixes:

- Set the FPGA DDR `src` and `dest` addresses to allow memory to and from the device to work correctly.
- Fixes the calculation of the number of descriptors used for a transfer on an Arria 10 device.

For Agilex 7 devices, the `agilex-dts.patch` patch enables necessary drivers in the device tree.

For Arria 10 devices, the `coredla-dts.patch` patch enables necessary drivers in the device tree.

This recipe also includes `enable-coredla-mod.cfg`, which is the kernel configuration file to enable `altera-msgdma driver`, `uio`, and `uio_pdrv_genirq` drivers.

23.1.6. [SOC] Yocto Recipe: `recipes-support/devmem2/devmem2_2.0.bb`

This Yocto recipe downloads, compiles, and installs The `devmem2` utility from https://github.com/radii/devmem2 for use in debugging designs. The `devmem2` utility is a simple program to read/write from/to any memory location.

23.1.7. [SOC] Yocto Recipe: `wic`

This Yocto recipe contains two files that define the layout of the SD card image. One file is used for Arria 10 devices, and the other for Stratix® 10 devices (not currently supported by the SoC design example). The partitions are as follows:

vfat	Storage for the Linux kernel, device tree, FPGA image, and u-boot
ext4	Root file system
raw	Arria 10 only. A custom raw partition labeled "a2". This is used for the first-stage boot loader.

23.2. [SOC] FPGA AI Suite Runtime Plugin

The FPGA AI Suite runtime plugin is described in "OpenVINO FPGA Runtime Plugin" in *FPGA AI Suite PCIe-based Design Example User Guide* .

Note that the FPGA AI Suite on Arm CPUs does not support the JIT (Just-In-Time) flow.

23.3. [SOC] Runtime Interaction with the MMD Layer

The FPGA AI Suite runtime uses the MMD layer to interact with the memory-mapped device. In the SoC Example Design, the MMD layer communicates via Linux kernel drivers described in [SOC] MMD Layer Hardware Interaction Library on page 134.

The key classes and structure of the runtime are described in "FPGA AI Suite Runtime" in *FPGA AI Suite PCIe-based Design Example User Guide*.

23.4. [SOC] MMD Layer Hardware Interaction Library

Runtime communication with CSR registers for the stream controller and for the FPGA AI Suite IP happens via the UIO driver. See The Userspace I/O HOWTO for more information on the UIO communication model.

FPGA AI Suite IP graph weights and instructions (from the AOT file) are transferred from host DDR memory to EMIF DDR memory (allocated to the FPGA FPGA AI Suite IP) via the mSGDMA-USERIO driver (a custom kernel driver, see [SOC] Yocto Recipe: recipes-drivers/msgdma-userio/msgdma-userio.bb on page 133). This driver is also used to send microcode and images. Lastly, inference results are transferred into host DDR via the mSGDMA-USERIO driver.

The source files for the library are in `runtime/coredla_device/mmd/hps_platform/`. The files contain classes for managing and accessing the FPGA AI Suite IP, the stream controller, the layout transform module, and DMA by UIO and MSGDMA-USERIO drivers. The remainder of this section describes the key classes.

23.4.1. [SOC] MMD Layer Hardware Interaction Library Class `mmd_device`

This class has the following responsibilities:

- Acquire the FPGA AI Suite IP, the stream controller (S2M only) and the mSGDMA
- Register interrupt callback for the FPGA AI Suite IP
- Provide read/write CSR and DDR functionality to the FPGA AI Suite runtime
- Linux device discovery.

The class attempts discovery of the following UIO devices, each from the `/sys/class/uio/ui*/` namespace.

coredla0	This represents the FPGA AI Suite IP CSR registers.
stream_controller0	If present (S2M only), this represents the Stream Controller CSR registers.
layout_transform0	If present (S2M only), this represents the Layout Transform CSR registers. Note that this device is not directly controlled from the runtime.

The class also attempts to discover the following mSGDMA-USERIO devices.

/dev/msgdma_coredla0	This represents the mSGDMA used to transfer weights, instructions, microcode, and, during M2M operation, image data.
/dev/msgdma_stream0	This represents the mSGDMA used to transfer images to the Layout Transform during the S2M mode of operation.

23.4.2. [SOC] MMD Layer Hardware Interaction Library Class `uio_device`

The Linux device tree is used for UIO devices. The following responsibilities are assumed by this file:

- Acquire and maps/unmap the FPGA CSRs to user mode, using the `sysfs` entries for UIO (`/sys/class/uio/uio*/`).
- Acquire and register a callback for the interrupt from the FPGA AI Suite IP.
- Provide `read_block()` and `write_block()` functions for accessing the CSRs.

23.4.3. [SOC] MMD Layer Hardware Interaction Library Class `dma_device`

The Linux device tree is used for mSGDMA-USERIO devices. These devices provide a Linux character device for simple read/write access to/from the FPGA EMIF via `fseek()`, `fread()`, and `fwrite()`.

The following responsibilities are assumed by this class:

- Acquire the mSGDMA-USERIO driver interface.
- Provide `read_block()` and `write_block()` functions for transfers from and to the FPGA AI Suite assigned DDR memory

24. [SOC] Streaming-to-Memory (S2M) Streaming Demonstration

A typical use case of the FPGA AI Suite IP is to run inferences on live input data. For instance, live data can come from a video source such as an HDMI IP core and stream to the FPGA AI Suite IP to perform image classification on each frame.

For simplicity, the S2M demonstration only simulates a live video source. The streaming demonstration consists of the following applications that run on the target SoC device:

- **`streaming_inference_app`**

  This application loads and runs a network and captures the results.

- **`image_streaming_app`**

  This application loads bitmap files from a folder on the SD card and continuously sends the images to the EMIF, simulating a running video source

The images are passed through a layout transform IP that maps the incoming images from their frame buffer encoding to the layout required by the FPGA AI Suite IP.

There is a module called the stream controller that runs on a Nios V microcontroller that controls the scheduling of the source images to the FPGA AI Suite IP.

The `streaming_inference_app` application creates OpenVINO inference requests. Each inference request is allocated memory on the EMIF for input and output buffers. This information is sent to the stream controller when the inference requests are submitted for asynchronous execution.

In its running state, the stream controller waits for input buffers to arrive from the `image_streaming_app` application. When the buffer arrives, the stream controller programs the FPGA AI Suite IP with the details of the received input buffer, which triggers the FPGA AI Suite IP to run an inference.

When an inference is complete, a completion count register is incremented within the FPGA AI Suite IP CSRs. This counter is monitored by the currently executing inference request in the `streaming_inference_app` application, and is marked as complete when the increment is detected. The output buffer is then fetched from the EMIF and the FPGA AI Suite IP portion of the inference is now complete.

Depending on the model used, there might be further processing of the output by the OpenVINO HETERO plugin and OpenVINO Arm CPU plugin. After the complete network has finished processing, a callback is made to the application to indicate the inference is complete.

The application performs some post processing on the buffer to generate the results and then resubmits the same inference request back to OpenVINO, which lets the stream controller use the same input/output memory block again.

24.1. [SOC] Nios Subsystem

The stream controller module runs autonomously in a Nios V microcontroller. An interface to the module is created by the FPGA AI Suite OpenVINO plugin when an "external streaming" flag is enabled by the inference application. At startup, the interface checks that the stream controller is present by sending a ping message and waiting for a response.

The following sections describe details of the various messages that are sent between the plugin and the stream controller, along with their packet structure.

24.1.1. [SOC] Stream Controller Communication Protocol

The FPGA AI Suite OpenVINO plugin running on the HPS system includes a `coredla-device` component which in turn has a stream controller interface if the "external streaming" flag is enabled by the inference application. This stream controller interface manages the communications from the HPS end to the stream controller microcode module running on the Nios V microcontroller.

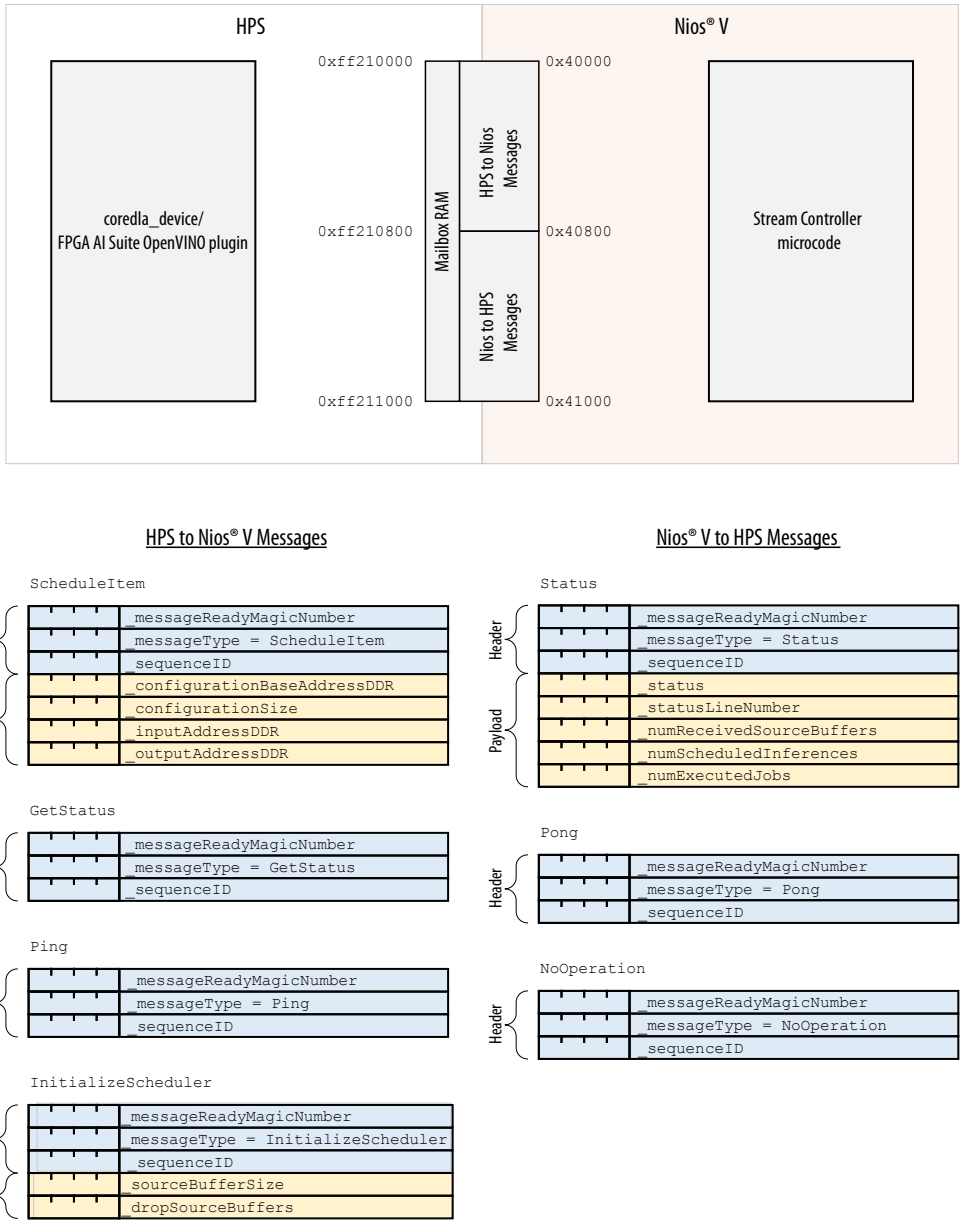
Messages are sent between the HPS and the Nios V microcontroller using the mailbox RAM which is shared between the two. In the HPS, this RAM is at physical address 0xff210000, and in the Nios V microcontroller, it is at address 0x40000. The RAM is 4K bytes. The lower 2K is used to send messages from the HPS to the Nios V microcontroller, and the upper 2K is used to send messages from the Nios V microcontroller to the HPS.

Message flow is always initiated from the HPS end, and the Nios V microcontroller always responds with a message. Therefore, after sending any message the HPS end waits until it receives a reply message. This can contain payload data (for example, status information) or just a "no operation" message with no payload.

Each message has a 3 x uint32_t header, which consists of a `messageReadyMagicNumber` field, a `messageType` field, and a `sequenceID` field. This header is followed by a payload, the size of which depends on the `messageType`. The `messageReadyMagicNumber` field is set to the value of `0x55225522` when the message is ready to be received

When a message is to be sent, all of the buffer apart from the `messageReadyMagicNumber` is first written to the mailbox RAM. The `sequenceID` increments by 1 with every message sent. Then the `messageReadyMagicNumber` is written. The sending end then waits for the value of `messageReadyMagicNumber` to change to the value of the `sequenceID`. This is set by the stream controller microcode module and indicates that the message has been received and processed by the receiving end.

Figure 21. Stream Controller Mailbox RAM and Message Packets



24.1.2. [SOC] Buffer Flow in Streaming Mode using Nios V Software Scheduler

24.1.2.1. [SOC] Review of M2M mode

To explain how the buffers are managed in streaming mode, it can help to review the existing flow for M2M mode.

The inference application loads source images from `.bmp` files into memory allocated from its heap. These buffers are 224x224x3 uint8_t samples (150528 bytes). During the load, the BGR channels are rearranged from interleaved channels into planes.

OpenVINO inference requests are created by the application using the inference engine. These inference requests allocate buffers in the on-board EMIF memory. The size of each of these buffers is the size of the input buffer plus the size of the output buffer. The input buffer size depends FPGA AI Suite IP parameters (specified in the `.arch` file) for which the graph was compiled.

The BGR planar image buffers are attached as input blobs to these OpenVINO inference requests, which are then scheduled for execution.

Preprocessing Steps

In M2M mode, the preprocessing steps are performed in software.

- The samples are converted to 32-bit floating point, and the mean G, B and R values of the imagenet dataset are subtracted from each sample accordingly.
- The samples are converted to 16-bit floating point.
- A layout transform then maps these samples into a larger buffer which has padding, in the layout expected by the FPGA AI Suite.

Inference Execution Steps

- The transformed image is written directly to board memory at its allocated address.
- The FPGA AI Suite IP CSR registers are programmed to schedule the inference.
- The FPGA AI SuiteOpenVINO plugin monitors the completion count register (located on the FPGA AI Suite IP), either by polling or receiving an interrupt, and waits until the count increments.
- The results buffer (2048 bytes) is read directly from the EMIF on the board to HPS memory.

Postprocessing Steps

- The samples in the results buffer (1001 16-bit floating point values) are converted to 32-bit floating point.
- The inference application receives these buffers, sorts them, and collects the top five results.

24.1.2.2. [SOC] External Streaming Mode Buffer Flow

External streaming mode is enabled in the inference application by setting the configuration value `DLIA_CONFIG_KEY(EXTERNAL_STREAMING)` to `CONFIG_VALUE(YES)` in the OpenVINO FPGA plugin.

In streaming mode, the inference application does not handle any of the input buffers. It still must create inference requests, which allocate the input and output buffers in the EMIF memory as before, but no input blobs are attached to the inference requests.

When the inference request is executed, there are no preprocessing steps required, since they do not have any input blobs.

Inference Execution Steps

- Instead of writing a source buffer directly to its allocated address, a `ScheduleItem` command is sent to the Nios V stream controller which contains details of the input buffer EMIF address.
- The FPGA AI Suite IP CSR registers are **not** programmed by the plugin.
- The plugin waits for the completion count register to increment as before.
- The results buffer is read directly from the board as before.

Postprocessing Steps

- The samples in the results buffer (1001 16-bit floating point values) are converted to 32-bit floating point.
- The inference application receives these buffers, sorts them, and collects the top five results.
- The same inference request is rescheduled with the inference engine.

24.1.2.3. [SOC] Nios V Stream Controller State Machine Buffer Flow

When the network is loaded into the `coredla_device`, if external streaming has been enabled, a connection to the Nios V processor is created and an `InitializeScheduler` message is sent. This message resets the stream controller and sets the size of the raw input buffers and the drop/receive ratio of buffers from the input stream.

The inference application queries the plugin for the number of inference requests to create. When scheduled with the inference engine, these send `ScheduleItem` commands to the stream controller, and a corresponding `CoreDlaJobItem` is created. The `CoreDlaJobItem` keeps details of the input buffer address and size and has flags to indicate if it has a source buffer and to indicate if it has been scheduled for inference on the FPGA AI Suite IP. The `CoreDlaJobItem` instances are handled as if they are in a circular buffer.

When the Nios V stream controller has received a `ScheduleItem` command from all of the inference requests and created a `CoreDlaJobItem` instance for each of them, it changes to a running state, which arms the mSGDMA stream to receive buffers, and sets a pointer `pFillingImageJob` that identifies which of the buffers is the next to be filled.

It then enters a loop, waiting for two types of event:

- A buffer is received through the mSGDMA, which is detected by a callback from an ISR.
- A message is received from the HPS.

New Buffer Received

The `pFillingImageJob` pointer is marked as now having a buffer.

If the next job in the circular buffer does not have a buffer, the `pFillingImageJob` pointer is moved on and the mSGDMA is armed again to receive the next buffer at the address of this next job.

If it does have a buffer, the FPGA AI Suite IP cannot keep up with the input buffer rate, so the `pFillingImageJob` does not move and the mSGDMA is armed to capture the next buffer at the same address. This means that the previous input buffer is dropped and is not processed by the FPGA AI Suite IP.

Buffers that have not been dropped can now be scheduled for inference on the FPGA AI Suite IP provided that the IP has fewer than two jobs in its pipeline.

Scheduling a job for execution means programming the CSR registers with the configuration address, the configuration size, and the input buffers address in DDR memory. This programming also sets the flag on the job so the controller knows that the job has been scheduled.

Message Received

If the message is a `ScheduleItem` message type then an inference request has been scheduled by the inference application.

This request happens only if a previous inference request has been completed and rescheduled. The number of jobs in the FPGA AI Suite IP pipeline has decreased by 1, so another job can potentially be scheduled for inference execution, providing it has an input buffer assigned.

If there are no jobs available with valid input buffers, then the FPGA AI Suite IP is processing buffers faster than they are being received by the mSGDMA stream, and consequently all input buffers are processed (that is, none are dropped).

24.2. [SOC] Building the Stream Controller Module

The stream controller is built as part of the steps described in [SOC] Installing HPS Disk Image Build Prerequisites on page 86. For system development that extends the FPGA AI Suite SoC design example, you might want to compile the stream controller module independently.

The stream controller module source code can be found in the distribution, in the `runtime/coredla_device/stream_controller/` directory.

There is a script `build.sh` in the source code directory that builds a binary `.hex` file. This file is then used by Quartus Prime when building the firmware to embed the microcode module.

The script should be run from a Nios V command shell, which is part of Quartus Prime. It requires a Quartus Prime project file and a Quartus Prime `.qsys` file. For this design example, the project file is `top.qpf`, and the Platform Designer file is `dla.qsys`.

An example command to build the stream controller module is as follows:

```
./build.sh top.qpf dla.qsys stream_controller.hex
```

24.3. [SOC] Building the Streaming Demonstration Applications

The two streaming demonstration applications, `streaming_inference_app` and `image_streaming_app`, are built as part of the runtime and are included on the SD card image for the target device in the directory `/home/root/app/`.

24.4. [SOC] Running the Streaming Demonstration

You need two terminals connected to the target device, one for each of the streaming applications.

One can be a serial terminal, and the other can be an SSH connection from a desktop PC. For details, refer to [SOC] Running the S2M Mode Demonstration Application on page 104.

24.4.1. [SOC] The `streaming_inference_app` Application

The `streaming_inference_app` application is an OpenVINO-based application. It loads a given precompiled ResNet50 network, then creates inference requests that are executed asynchronously by the FPGA AI Suite IP.

The resulting tensors are captured from the EMIF using the mSGDMA controller. The postprocessing required in the software involves converting the output tensors to floating point, assigning the values to the appropriate image classification, sorting the results, and selecting the top 5 classification results.

For each inference, the result is displayed on the terminal, and the results for each inference up to the 1000th one are logged in a `results.txt` file in the application folder.

The application depends on the following shared libraries . The system build adds these libraries to the directory `/home/root/app` on the SD card image, along with the application binary and a `plugins.xml` file that defines the plugins available to OpenVINO.

- `libhps_platform_mmd.so`
- `libngraph.so`
- `libinference_engine.so`
- `libinference_engine_transformations.so`
- `libcoreDLAHeteroPlugin.so`
- `libcoreDlaRuntimePlugin.so`

You also need a compiled network binary file and an `.arch` file (which describes the FPGA AI Suite IP parameterization) to run inferences. These have been copied to the `/home/root/resnet-50-tf` directory.

For example, a ResNet50 model compiled for an Arria 10 might have the following files:

- `RN50_Performance_no_folding.bin`
- `A10_Performance.arch`

Before running the application, set the `LD_LIBRARY_PATH` shell environment variable to define the location of the shared libraries:

```
root@arria10-1ac87246f24f:~# cd /home/root/app
root@arria10-1ac87246f24f:~# export LD_LIBRARY_PATH=.
```

Use the `--help` of the `streaming_inference_app` command to display the command usage:

```
# ./streaming_inference_app -help
Usage:
        streaming_inference_app -model=<model> -arch=<arch> -device=<device>

Where:
        <model>    is the compiled model binary file, eg /home/root/resnet-50-tf/
RN50_Performance_no_folding.bin
        <arch>     is the architecture file, eg /home/root/resnet-50-tf/
A10_Performance.arch
        <device>   is the OpenVINO device ID, eg HETERO:FPGA or HETERO:FPGA,CPU
```

Start the streaming inference app with a command like this:

```
# ./streaming_inference_app \
-model=/home/root/resnet-50-tf/RN50_Performance_no_folding.bin \
-arch=/home/root/resnet-50-tf/A10_Performance.arch \
-device=HETERO:FPGA
```

The distribution includes a shell script utility called `run_inference_stream.sh` which calls this command above.

Note that the layout transform IP core does not support folding on the input buffer. For streaming, you must use models that have been compiled by the `dla_compiler` command with the `--ffolding-option=0` command line option specified.

24.4.2. [SOC] The `image_streaming_app` Application

The `image_streaming_app` application loads images from the SD card, programs the layout transform IP, and then transfers the buffers via a streaming mSGDMA interface to the device. The buffers are sent at regular intervals at a frequency set by one of the command line options. Buffers are continually sent until the program is stopped with Ctrl+C.

The `image_streaming_app` application works only with `.bmp` files with dimensions of 224x224 pixels. The `.bmp` files can be either 24 or 32 bits per pixel format. If they are 24 bits, the buffers are padded to make them 32 bits per pixel. This format is expected by the input of the layout transform IP .

The command usage from the `-help` command option is as follows:

```
root@agilex7:~/app# ./image_streaming_app --help
Usage:
 image_streaming_app [Options]

Options:
-images_folder=folder     Location of bitmap files. Defaults to working folder.
-image=path               Location of a single bitmap file for single inference.
-send=n                   Number of images to stream. Default is 1 if -image is
set, otherwise infinite.
-rate=n                   Rate to stream images, in Hz. n is an integer. Default
is 30.
-width=n                  Image width in pixels, default = 224
-height=n                 Image height in pixels, default = 224
-c_vector=n               C vector size, default = 32
-blue_variance=n          Blue variance, default = 1.0
-green_variance=n         Green variance, default = 1.0
-red_variance=n           Red variance, default = 1.0
```

```
-blue_shift=n             Blue shift, default = -103.94
-green_shift=n            Green shift, default -116.78
-red_shift=n              Red shift, default = -123.68
```

The distribution includes a shell script utility called `run_image_stream.sh` that calls the `image_streaming_app` command with a rate of 50 Hz and default layout transform settings.

Once both applications are running, the `streaming_inference_app` application outputs the results to the terminal and a `results.txt` file.

Example output from the `streaming_inference_app` application is as follows:

```
root@agilex7:~/app# ./run_inference_stream.sh
Runtime version check is enabled.
[ INFO ] Architecture used to compile the imported model: AGX7_Performance
Using licensed IP
Read hash from bitstream ROM...
Read build version string from bitstream ROM...
Read arch name string from bitstream ROM...
Runtime arch check is enabled. Check started...
Runtime arch check passed.
Runtime build version check is enabled. Check started...
Runtime build version check passed.
Ready to start image input stream.
1 - class ID 776, score = 58.4
2 - class ID 968, score = 90.7
3 - class ID 769, score = 97.8
4 - class ID 769, score = 97.8
5 - class ID 872, score = 99.8
6 - class ID 954, score = 94.4
7 - class ID 954, score = 94.4
8 - class ID 776, score = 58.4
9 - class ID 872, score = 99.8
10 - class ID 968, score = 90.7
11 - class ID 776, score = 58.4
12 - class ID 968, score = 90.7
13 - class ID 769, score = 97.8
14 - class ID 769, score = 97.8
15 - class ID 872, score = 99.8
16 - class ID 954, score = 94.4
17 - class ID 954, score = 94.4
18 - class ID 776, score = 58.4
19 - class ID 872, score = 99.8
20 - class ID 968, score = 90.7
^C
Ctrl+C detected. Shutting down application
```

The resulting `results.txt` file from the example is as follows:

```
root@agilex7:~/app# cat results.txt
Result: image[1]
1. class ID 776, score = 58.4
2. class ID 683, score = 27.6
3. class ID 513, score = 10.2
4. class ID 432, score = 1.8
5. class ID 558, score = 1.6

Result: image[2]
1. class ID 968, score = 90.7
2. class ID 901, score = 1.1
3. class ID 868, score = 1.0
4. class ID 899, score = 1.0
5. class ID 725, score = 0.9

Result: image[3]
1. class ID 769, score = 97.8
```

```
2. class ID 845, score = 0.5
3. class ID 587, score = 0.4
4. class ID 798, score = 0.1
5. class ID 618, score = 0.1

Result: image[4]
1. class ID 769, score = 97.8
2. class ID 845, score = 0.5
3. class ID 587, score = 0.4
4. class ID 798, score = 0.1
5. class ID 618, score = 0.1
```

848957 | 2025.03.28

Send Feedback

A. FPGA AI Suite Example Designs User Guide Archives

For the latest and previous versions of this user guide, refer to FPGA AI Suite Example Designs User Guide. If an FPGA AI Suite software version is not listed, the user guide for the previous software version applies.

B. FPGA AI Suite Example Designs User Guide Revision History

Document Version	FPGA AI Suite Version	Changes
2025.03.31	2024.3	Initial release. This initial release merges the content from the following publications: • FPGA AI Suite PCIe-based Design Example User Guide • FPGA AI Suite SoC Design Example User Guide The revision histories of these guides is included here for completeness. This initial release adds information about the following design examples: • JTAG Design Example • DDR-Free Hostless Design Example • OFS for PCIe Attach Design Example

B.1. FPGA AI Suite PCIe-based Design Example User Guide Document Revision History

Document Version	FPGA AI SuiteVersion	Changes
2024.12.16	2024.3	• Various minor corrections and typo fixes.
2024.11.25	2024.3	• References to "software reference" have been replaced with "software emulator" or "software emulation.". As part of this change, the `--target_reference` option of the `build.runtime.sh` script has been renamed to `--target_emulation`.
2024.07.31	2024.2	• Added "The `dla_benchmark` Performance Metrics". • Added "Interpreting System Throughput and Latency Metrics". • Revised "Additional `dla_benchmark` Options". • Removed references to the Intel PAC with Arria 10 GX FPGA. The PCIe-based design example no longer supports the Intel PAC with Arria 10 GX FPGA. • Replaced references to the `-plugins_xml_file` option with the `-plugins` option. The `-plugins_xml_file` option is deprecated and will be removed in a future release. • Revised "Script Flow". • Revised "Hardware".
2024.03.29	2024.1	• Added new options to "Additional `dla_benchmark` Options". • Added `segmentation_demo` to "Running the Ported OpenVINO Demonstration Applications".
2023.12.01	2023.3	• Added `--wsl` and `--finalize` options to "Build Script Options".
2023.09.06	2023.2.1	• Updated supported OpenVINO version to 2022.3.1 LTS.

continued...

Document Version	FPGA AI SuiteVersion	Changes
2023.07.03	2023.2	• Updated "Software Components". • Renamed "OPAE Driver" to "BSP Driver" and revised the content in the topic. • Updated supported OpenVINO version to 2022.3 LTS. • Updated OpenVINO installation paths to `/opt/intel/openvino_2023`. • Updated FPGA AI Suite installation paths to `/opt/intel/fpga_ai_suite_2023.2`. • Changed occurrences of `tools/downloader/downloader.py` to `omz_downloader`. • Changed occurrences of `tools/downloader/converter.py` to `omz_converter`.
2023.04.05	2023.1	• Renamed the `dlac` command. The FPGA AI Suite compiler command is now `dla_compiler`. • Updated the Intel Agilex product family name to "Intel Agilex® 7."
2022.12.23	2022.2	• Removed the `-f` build script (`dla_create_and_build_pcie_ed.py`) option. • Renamed the `benchmark_app` to `dla_benchmark`. • Renamed `dla_create_and_build_pcie_ed.py` to `dla_build_example_design.py`
2022.05.26	2022.1.1	• Updated description of how to run the OpenVINO demos.
2022.04.27	2022.1	• Additional updates for Intel Agilex device support. • Updated the locations of graphs from the Getting Started tutorial.
2021.09.10	2021.2	• Added updates for initial Intel Agilex device support.
2021.04.30	2021.1	• Added additional demonstration programs. • Various corrections and updates.
2020.12.04	2020.2	• Initial release.

B.2. FPGA AI Suite SoC Design Example User Guide Document Revision History

Document Version	FPGA AI SuiteVersion	Changes
2024.12.16	2024.3	• Added "Yocto Recipe: `recipes-support/devmem2/devmem2_2.0.bb`". • Revised "Yocto Recipe: `recipes-kernel/linux/linux-socfpga-lts_%.bbappend`" (formerly "octo Recipe: `recipes-kernel/linux/linux-socfpga-lts_5.15.bbappend`")
2024.11.25	2024.3	• Updated required Quartus Prime Pro Edition version to Version 24.3. • Updated "Building the Bootable SD Card Image (`.wic`)" to Yocto project version 5.0.5.
2024.07.31	2024.2	• Revised "Agilex 7 FPGA I-Series Transceiver-SoC Development Kit Hardware Requirements". • Updated required Quartus Prime Pro Edition version to Version 24.2.
2024.03.29	2024.1	• Updated the document for Ubuntu 22.04 support. • Updated required Quartus Prime Pro Edition version to Version 23.4.

continued...

Document Version	FPGA AI SuiteVersion	Changes
2024.02.12	2023.3.1	• Updated the document for Agilex 7 FPGA I-Series Transceiver-SoC Development Kit support, including the following new topics: — "Preparing the Intel Agilex 7 FPGA I-Series Transceiver-SoC Development Kit " — "Confirming Intel Agilex 7 FPGA I-Series Transceiver-SoC Development Kit Board Set Up" — "Programming the Intel Agilex 7FPGA Device with the JTAG Indirect Configuration (.jic) File" — "Connecting the Intel Agilex 7 FPGA I-Series Transceiver-SoC Development Kit to the Host Development System"
2023.12.01	2023.3	• Updated required Quartus Prime Pro Edition version to Version 23.3.
2023.09.06	2023.2.1	• Updated supported OpenVINO version to 2022.3.1 LTS.
2023.07.03	2023.2	• Updated supported OpenVINO version to 2022.3 LTS. • Updated OpenVINO installation paths to `/opt/intel/openvino_2023`. • Updated FPGA AI Suite installation paths to `/opt/intel/fpga_ai_suite_2023.2`. • Changed occurrences of `tools/downloader/downloader.py` to `omz_downloader`. • Changed occurrences of `tools/downloader/converter.py` to `omz_converter`.
2023.04.05	2023.1	• Initial release.