See all versions
of this document

Vitis AI Library User Guide

UG1354 (v3.5) June 29, 2023

Table of Contents

Chapter 1

Introduction

About this Document

Related Libraries

The following AMD Vitis™ AI Libraries are related to this document.

Table 1: **Vitis AI Library Package List**

No	Package Name	Version
1	`vitis-ai-runtime-3.5.0.tar.gz`	r3.5.0
2	`vitis_ai_2023.1-r3.5.0.tar.gz`	r3.5.0
3	`DPUCV2DX8G_xclbins_3_5_0.tar.gz`	r3.5.0
4	`sdk-2023.1.0.0.sh`	2023.1

Intended Audience

The target users of Vitis AI Libraries are as follows:

- Users who want to use pre-trained AMD models to quickly build applications.
- Users who use their own models that are trained with their own datasets under the Vitis AI Library support network list.
- Users who have custom models, similar to models supported by the Vitis AI Libraries, and use the Vitis AI post-processing library.

Note: If you have custom models that are completely different from the models supported by the Vitis AI Library or have specialized post-processing requirements, you can use the Vitis AI Library implementations for reference.

Document Navigation

This document describes how to install, use, and develop applications with the Vitis AI Library.

- Chapter 1: Introduction provides a high-level overview of the Vitis AI Library. This chapter provides a clear understanding of the Vitis AI Library in general, its framework, supported networks, and supported hardware platforms.

- Chapter 2: Installation describes how to install the Vitis AI Library and run the examples. The information in this chapter will help you quickly set up the host and target environments, compile, and execute the Vitis AI Library-related examples.
- Chapter 3: Libraries and Samples describes each model library supported by the Vitis AI Library. This chapter provides an understanding of the model libraries supported by the Vitis AI Library, the purpose of each library, how to test the library with images or videos, and how to test the performance of the library.
- Chapter 4: Programming Examples describes how to develop applications with the Vitis AI Library. This chapter provides an understanding of the following:
  - Developing applications using the Vitis API
  - Developing applications using your models
  - Customizing pre-processing
  - Using the configuration file as pre-processing and post-processing parameters
  - Using the post-processing library in the Vitis AI Library
  - Implementing your post-processing code
  - Using the `xdputil` tool for DPU and xmodel debug
  - Implementing and registering custom operators
- Chapter 5: Application Demos describes how to set up a test environment and run the application demos. There are two application demos provided with the Vitis AI Library.
- Chapter 6: Programming APIs describes how to find the programming APIs.
- Chapter 7: Performance describes the performance of the Vitis AI Library on different boards.
- Chapter 8: API Reference describes the Vitis AI Library APIs.

Navigating Content by Design Process

AMD Adaptive Computing documentation is organized around a set of standard design processes to help you find relevant content for your current development task. All AMD Versal™ adaptive SoC design process Design Hubs and the Design Flow Assistant materials can be found on the Xilinx.com website. This document covers the following design processes:

- **Embedded Software Development:** Creating the software platform from the hardware platform and developing the application code using the embedded CPU. Also covers XRT and Graph APIs. Topics in this document that apply to this design process include:
  - Chapter 2: Installation
  - Chapter 4: Programming Examples

    - Chapter 5: Application Demos

- **Hardware, IP, and Platform Development:** Creating the PL IP blocks for the hardware platform, creating PL kernels, functional simulation, and evaluating the AMD Vivado™ timing, resource use, and power closure. Also involves developing the hardware platform for system integration. Topics in this document that apply to this design process include:

    - Chapter 4: Programming Examples

- **System Integration and Validation:** Integrating and validating the system functional performance, including timing, resource use, and power closure. Topics in this document that apply to this design process include:

    - Chapter 7: Performance

Overview

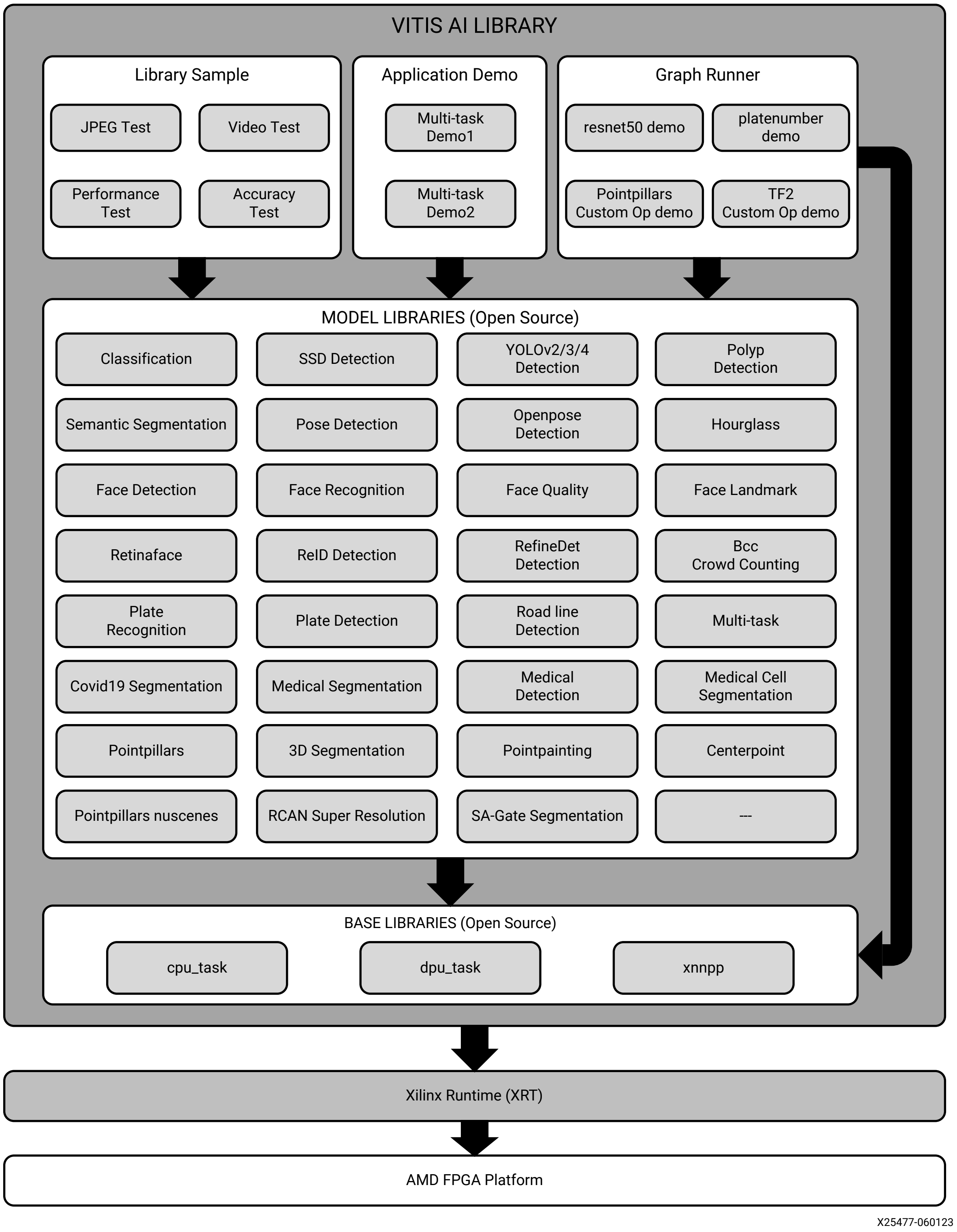
The Vitis AI Library is a set of high-level libraries and APIs built for efficient AI inference with the Deep-Learning Processor Unit (DPU). It is built based on the Vitis AI runtime with unified APIs, and it supports XRT 2023.1.

The Vitis AI Library provides an easy-to-use and unified interface by encapsulating many efficient and high-quality neural networks. This simplifies the use of deep-learning neural networks, even for users without knowledge of deep-learning or FPGAs. The Vitis AI Library allows you to focus more on the development of your applications, rather than the underlying hardware.

For the intended audience for the Vitis AI Library, refer to the About this Document section.

The Vitis AI Library has four parts as shown in the following block diagram.

Figure 1: **Vitis AI Library Block Diagram**



- **Base libraries:** The base libraries provide the basic programming interface with the DPU and the available post-processing modules of each model.

  - *dpu_task* is the interface library for DPU operations.
  - *cpu_task* is the interface library for operations that are assigned to the CPU.
  - *xnnpp* is the post-processing library for each model, with built-in modules such as optimization and acceleration.
- **Model libraries:** The model libraries implement most of the open-source neural network deployment including common types of networks, such as classification, detection, segmentation, and others. These libraries provide an easy-to-use and fast development method with a unified interface, which apply to the AMD models or custom models.
- **Library samples:** The library test samples are used to quickly test and evaluate the model libraries.
- **Application demos:** The application demos show you how to use the Vitis AI Library to develop applications.

Features

The Vitis AI Library features include:

- A full-stack application solution
- Optimized pre-processing and post-processing functions/libraries
- Open-source model libraries
- Unified operation interface with the DPU and the pre-processing and post-processing interface of the model
- Practical, application-based model libraries, pre-processing and post-processing libraries, and application examples

Vitis AI Library v3.5 Release Notes

This section contains information regarding the features and updates of the Vitis AI Library 3.5 release.

Key Features And Enhancements

This Vitis AI Library release includes the following key features and enhancements:

- **New Board Support:** AMD Versal™ VEK280, V70 evaluation board are supported in this release.

- **ONNX Runtime Enhanced:** ONNX Runtime Vitis AI Execution Provider (Vitis AI EP) is provided to hardware-accelerated AI inference with DPUs. It supports both C++ and Python API in this release.
- **New Model Libraries:** Following are the new model libraries supported in this release.
  - YOLOv7 Detection
  - YOLOv8 Detection
  - 2DUnet
- **New Model Support:**
  - Added four new PyTorch models
  - Added one new TensorFlow2 model

Changes

For Vitis AI 3.5, only the VEK280, V70 evaluation board are supported and verified. For Zynq UltraScale+ MPSoC and VCK190, VCK5000-Prod boards, refer to Vitis AI 3.0.

Compatibility

The Vitis AI Library 3.5 is tested with the following images.

- `amd-vek280-dpu-v2023.1-v3.5.0.img.gz`

Device Support

The following platforms and evaluation boards (EVB) are supported by the Vitis AI Library 3.5.

Table 2: **Edge Device Support**

Platform	EVB	Version
Versal Edge series VE2802	AMD VEK280	ES1

Table 3: **Data Center Board Support**

Accelerator Cards
Versal AI Core series V70 Data Center development kit

Limitations

- Zynq UltraScale+ MPSoC and VCK190 are not updated in Vitis AI 3.5 and will be updated in the next release.
- VCK5000-Prod is no longer supported. Use Vitis AI 3.0 if using VCK5000-Production cards.
- Alveo U50 and U280 cards are no longer supported. Use Vitis AI 1.4.x if using these cards.

- Due to limitations of the Docker environment, MultiTask demos cannot run in DRM mode for Data Center targets.

Models Supported by Vitis AI Library v3.5

Model Support

The following models are supported by this version of the Vitis AI Library.

Table 4: **Models Supported by the Vitis AI Library**

No.	Neural Network	VEK280	V70	Application
1	inception_v1_tf	Y	Y	Image Classification
2	inception_v3_tf	Y	Y	
3	inception_v4_2016_09_09_tf	Y	Y	
4	mobilenet_v1_0_25_128_tf	Y	Y	
5	mobilenet_v1_1_0_224_tf	Y	Y	
6	mobilenet_v2_1_0_224_tf	Y	Y	
7	mobilenet_v2_1_4_224_tf	Y	Y	
8	resnet_v1_101_tf	Y	Y	
9	resnet_v1_152_tf	Y	Y	
10	resnet_v1_50_tf	Y	Y	
11	vgg_16_tf	Y	Y	
12	vgg_19_tf	Y	Y	
13	ssd_mobilenet_v1_coco_tf	Y	Y	Object Detection
14	ssd_mobilenet_v2_coco_tf	Y	Y	
15	yolov3_voc_tf	Y	Y	
16	mlperf_ssd_resnet34_tf	Y	Y	
17	resnet50_pt	Y	Y	Image Classification
18	squeezenet_pt	Y	Y	
19	inception_v3_pt	Y	Y	
20	pointpillars_kitti_12000_0_pt pointpillars_kitti_12000_1_pt	Y	Y	Point Cloud
21	MLPerf_resnet50_v1.5_tf	Y	Y	Image Classification

Table 4: **Models Supported by the Vitis AI Library** *(cont'd)*

No.	Neural Network	VEK280	V70	Application
22	RefineDet-Medical_EDD_tf	Y	Y	Medical Detection
23	resnet_v2_50_tf	Y	Y	Image Classification
24	resnet_v2_101_tf	Y	Y	
25	resnet_v2_152_tf	Y	Y	
26	resnet50_tf2	Y	Y	
27	inception_v3_tf2	Y	Y	
28	efficientNet-edgetpu-S_tf	Y	Y	
29	efficientNet-edgetpu-M_tf	Y	Y	
30	efficientNet-edgetpu-L_tf	Y	Y	
31	pointpillars_nuscenes_40000_64_0_pt pointpillars_nuscenes_40000_64_1_pt	Y	Y	3D object detection
32	FADNet_0_pt FADNet_1_pt FADNet_2_pt	N/A	N/A	Depth Estimation
33	rcan_pruned_tf	Y	Y	Super Resolution
34	efficientnet-b0_tf2	N/A	N/A	Classification
35	HardNet_MSeg_pt	Y	Y	Polyp Segmentation
36	ofa_resnet50_0_9B_pt	Y	Y	Classification
37	SESR_S_pt	Y	Y	Image Super-Resolution
38	ofa_depthwise_res50_pt	Y	Y	Classification
39	FADNet_pruned_0_pt FADNet_pruned_1_pt FADNet_pruned_2_pt	N/A	N/A	Depth Estimation
40	PSMNet_pruned_0_pt PSMNet_pruned_1_pt PSMNet_pruned_2_pt	N/A	N/A	
41	mobilenet_v3_small_1_0_tf2	N/A	N/A	Classification
42	ssr_pt	Y	Y	Spectral Remove
43	chen_color_resnet18_pt	Y	Y	Classification

Table 4: **Models Supported by the Vitis AI Library** *(cont'd)*

No.	Neural Network	VEK280	V70	Application
44	face_mask_detection_pt	Y	Y	Face mask Detection
45	ofa_rcan_latency_pt	Y	Y	Super Resolution
46	vehicle_make_resnet18_pt	Y	Y	Classification
47	vehicle_type_resnet18_pt	Y	Y	Classification
48	ofa_yolo_pt	Y	Y	Object Detection
49	ofa_yolo_pruned_0_30_pt	Y	Y	
50	ofa_yolo_pruned_0_50_pt	Y	Y	
51	efficientdet_d2_tf	N/A	N/A	
52	superpoint_tf	Y	N/A	SLAM
53	hfnet_tf	Y	N/A	SLAM
54	movenet_ntd_pt	Y	Y	Pose Estimation
55	yolov3_coco_416_tf2	Y	Y	Object Detection
56	yolov4_leaky_416_tf	Y	Y	
57	yolov4_leaky_512_tf	Y	Y	
58	HRNet_pt	N/A	N/A	Segmentation
59	xilinxSR_pt	N/A	N/A	Super Resolution
60	yolov4_csp_pt	Y	Y	Object Detection
61	yolov5_nano_pt	Y	Y	
62	yolov5s6_pt	Y	Y	
63	yolov5_large_pt	Y	N/A	
64	yolox_nano_pt	Y	Y	
65	yolov6_pt	Y	Y	
66	3D-Unet_pt	N/A	N/A	Medical Segmentation
67	FADNet_v2_0_pt FADNet_v2_1_pt FADNet_v2_2_pt	Y	Y	Depth Estimation
68	FADNet_v2_pruned_0_pt FADNet_v2_pruned_1_pt FADNet_v2_pruned_2_pt	Y	N/A	

Table 4: **Models Supported by the Vitis AI Library** *(cont'd)*

No.	Neural Network	VEK280	V70	Application
69	unet2d_tf2	Y	Y	Segmentation
70	yolov5l_pt	Y	N/A	Object Detection
71	yolov5m_pt	Y	Y	
72	yolov7_pt	Y	Y	
73	yolov8m_pt	Y	Y	

Notes:

1. Networks with the suffix "_tf" or "_tf2" were trained on TensorFlow.
2. Networks with the suffix "_pt" were trained on PyTorch.

Chapter 2

Installation

Downloading the Vitis AI Library

The AMD Vitis™ AI Library package can be obtained from the Vitis AI repository on GitHub.

RECOMMENDED: *Use a data center accelerator card or an evaluation board that supports the Vitis AI Library to become familiar with the product. See the AI Developer Hub for more details about evaluation boards that support the Vitis AI Library. See the Alveo Accelerator Cards product page for more details about Alveo cards.*

Vitis AI 3.5 supports the VEK280 Evaluation Board and the V70 Development Card for AI Inference.

For the following evaluation boards, refer to Vitis AI 3.0:

- AMD Zynq™ UltraScale+™ MPSoC ZCU102 Evaluation Board
- AMD Zynq™ UltraScale+™ MPSoC ZCU104 Evaluation Board
- AMD Versal™ VCK190 Evaluation Board
- AMD Kria™ KV260 Vision AI Starter Kit
- AMD Versal™ VCK5000 Development Card

Setting Up the Host

For Edge

To set up the host for Edge device development, follow these steps:

1. Clone the Vitis AI repository:

```
$ cd ~
$ git clone https://github.com/Xilinx/Vitis-AI
```

2. Install the cross-compilation system environment.

   ```
   $ cd Vitis-AI/board_setup/vek280
   $ ./host_cross_compiler_setup.sh
   ```

   Note:

   - `~/petalinux_sdk_2023.1` path is recommended for the installation. Regardless of the path you choose for the installation, make sure the path has read-write permissions. In this section, it is installed in `~/petalinux_sdk_2023.1`.
   - For Zynq UltraScale+ MPSoC and VCK190 boards, refer to Vitis AI 3.0: https://github.com/Xilinx/Vitis-AI/tree/v3.0.

3. When the installation is complete, follow the prompts and execute the following command:

   ```
   $ source ~/petalinux_sdk_2023.1/environment-setup-cortexa72-cortexa53-
   xilinx-linux
   ```

   Note: If you close the current terminal, you need to re-execute the above instructions in the new terminal interface.

4. To compile the library sample in the Vitis AI Library using `classification` as an example, execute the following command.

   ```
   $ cd ~/Vitis-AI/examples/vai_library/samples/classification
   $ bash -x build.sh
   ```

   The executable program is now produced.

5. To modify the library source code, view and modify it under `~/Vitis-AI/src/vai_library`.

   Before compiling the Vitis AI libraries, confirm the compiled output path. The default output path is `$HOME/build`.

   To change the default output path, modify the `build_dir_default` in `cmake.sh`. For example, you can change from `build_dir_default=$HOME/build/build.${target_info}/${project_name}` to `build_dir_default=/workspace/build/build.${target_info}/${project_name}`.

   Note: If you want to modify the `build_dir_default`, modify `$HOME` only.

6. Build the libraries all at once by executing the following command.

   ```
   $ cd ~/Vitis-AI/src/vai_library
   $ ./cmake.sh --clean
   ```

   After compiling, you can find the generated libraries in the `build_dir_default` location. If you want to change the compilation rules, check and change the `cmake.sh` in the library directory.

   Note: All the source codes, samples, demos, and header files can be found in the `~/Vitis-AI/src/vai_library` location.

For Data Center (Versal V70 Card)

Set up the host by running the Docker image.

1. Clone the Vitis AI repository.

   ```
   $ git clone https://github.com/Xilinx/Vitis-AI
   $ cd Vitis-AI
   ```

2. Run the Docker container according to the instructions in the Docker installation guide.

   ```
   $ ./docker_run.sh -X xilinx/vitis-ai-<pytorch|tensorflow2|tensorflow>-
   cpu:latest
   ```

   Note: A `workspace` folder is created by the Docker runtime system and is mounted in the `/workspace` folder of the Docker runtime system.

3. Place the program, data, and other files to be developed in the `workspace` folder. After the Docker system starts, locate them in the `/workspace` folder of the Docker system.

   WARNING! *Do not put the files in any other path of the Docker system. They will be erased after you exit the Docker system.*

4. Select the model for your platform. You can find the download links for the latest models in the yaml files of the model in the `Vitis-AI/model_zoo/model-list` location.

   - If the `/usr/share/vitis_ai_library/model` folder does not exist, create it first.

     ```
     $ sudo mkdir -p /usr/share/vitis_ai_library/models
     ```

   - For the Versal V70 Prod card, take `resnet_v1_50_tf` as an example.

     ```
     $ wget https://www.xilinx.com/bin/public/openDownload?
     filename=resnet_v1_50_tf-v70-DPUCV2DX8G-r3.5.0.tar.gz -O
     resnet_v1_50_tf-v70-DPUCV2DX8G-r3.5.0.tar.gz
     $ tar -xzvf resnet_v1_50_tf-v70-DPUCV2DX8G-r3.5.0.tar.gz
     $ sudo cp resnet_v1_50_tf /usr/share/vitis_ai_library/models -r
     ```

5. Download the xclbin package from here. Untar it, select the Versal card DPU IP and install it.

   ```
   $ sudo tar -xzvf DPUCV2DX8G_xclbins_3_5_0.tar.gz -C /
   $ export XLNX_VART_FIRMWARE=/opt/xilinx/overlaybins/DPUCV2DX8G/V70/
   dpu_DPUCV2DX8G_250M_xilinx_v70_gen5x8_qdma_base_2.xclbin
   ```

6. If there is more than one card installed on the server and you want to which cards will be used for inference, you can set `XLNX_ENABLE_DEVICES` to achieve this function. The following is the usage of `XLNX_ENABLE_DEVICES`:

   - `export XLNX_ENABLE_DEVICES=0`: Only use device 0 for DPU.
   - `export XLNX_ENABLE_DEVICES=0,1,2`: Use device 0, device 1, and device 2 for DPU.
   - If you do not set this environment variable, all devices are used for DPU (default).

7. To compile the library sample in the `Vitis AI Library` , take `classification` for example, execute the following command:

   ```
   $ cd /workspace/examples/vai_library/samples/classification
   $ bash -x build.sh
   ```

   The executable program is now produced.

8. To modify and compile the library source under `/workspace/src/vai_library`.

   Before compiling the `Vitis AI Library`, confirm the compiled output path. The default output path is `$HOME/build`.

   If you want to change the default output path, modify the `build_dir_default` in `cmake.sh`. For example, change from `build_dir_default=$HOME/build/build.${target_info}/${project_name}` to `build_dir_default=/workspace/build/build.${target_info}/${project_name}`.

   Note: If you want to modify the `build_dir_default`, modify `$HOME` only.

9. Execute the following command to build the libraries all at once:

   ```
   $ cd /workspace/src/vai_library
   $ conda activate base
   $ ./cmake.sh --clean
   ```

   After compiling, you can find the generated `Vitis AI Library` under `build_dir_default`. If you want to change the compilation rules, check and change the `cmake.sh` in the library's directory.

Vitis AI Library File Locations

The following table shows the file locations after the installation is complete.

Table 5: **File Location List**

Files	Location
Source code of the libraries	`/workspace/src/vai_library`
Samples	`/workspace/examples/vai_library/samples`
Apps	`/workspace/examples/vai_library/apps`
Test	`/workspace/src/vai_library/[model]/test`

Notes:

- `/workspace/` is the path to extract the Vitis AI Library compressed package in the Docker system.
- "Samples" is used for rapid application construction and evaluation, and it is for users.
- "Apps" provides more practical examples for user development, and it is for users.
- "Test" is a test example for each model library which is for library developers.

Setting Up the Target

There are three steps to set up the target.

1. Install the board image.
2. Install the AI model package.
3. Install the Vitis AI Library package.

***Note*:** The version of the board image should be 2023.1 or above.

Step 1: Installing the Board Image

Pre-built images for AMD platforms are available in the following locations:

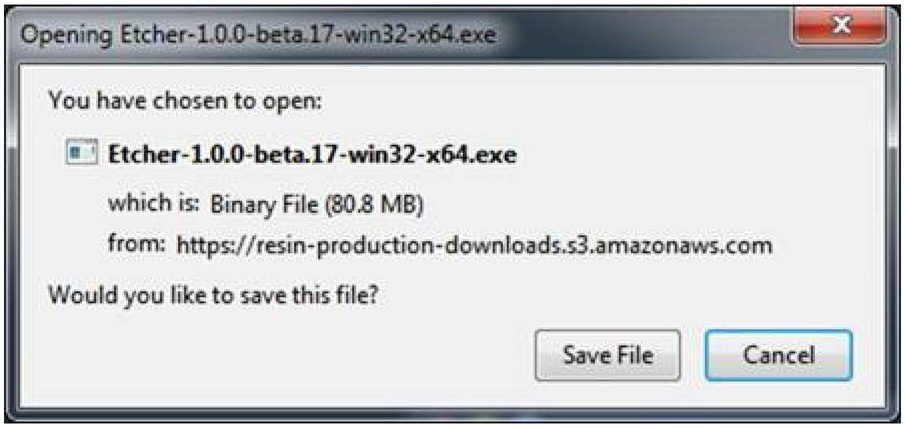
- The pre-built image for the VEK280 evaluation kit can be downloaded from here.

  Pre-built images for ZCU102, ZCU104, KV260, and VCK190 are not updated in Vitis AI 3.5. The following pre-built images are from Vitis AI 3.0.

  - The pre-built image for the ZCU102 evaluation kit can be downloaded from here.
  - The pre-built image for the ZCU104 evaluation kit can be downloaded from here.
  - The pre-built image for the KV260 starter kit can be downloaded from here.
  - The pre-built image for the VCK190 evaluation board can be downloaded from here.

One suggested software application for flashing the SD card is Balena Etcher. It is a cross-platform tool for flashing OS images to SD cards, available for Windows, Linux, and Mac systems. The following example runs on Windows.

1. Download Balena Etcher from: https://etcher.io/ and save the file as shown in the following figure.



Opening Etcher-1.0.0-beta.17-win32-x64.exe

You have chosen to open:

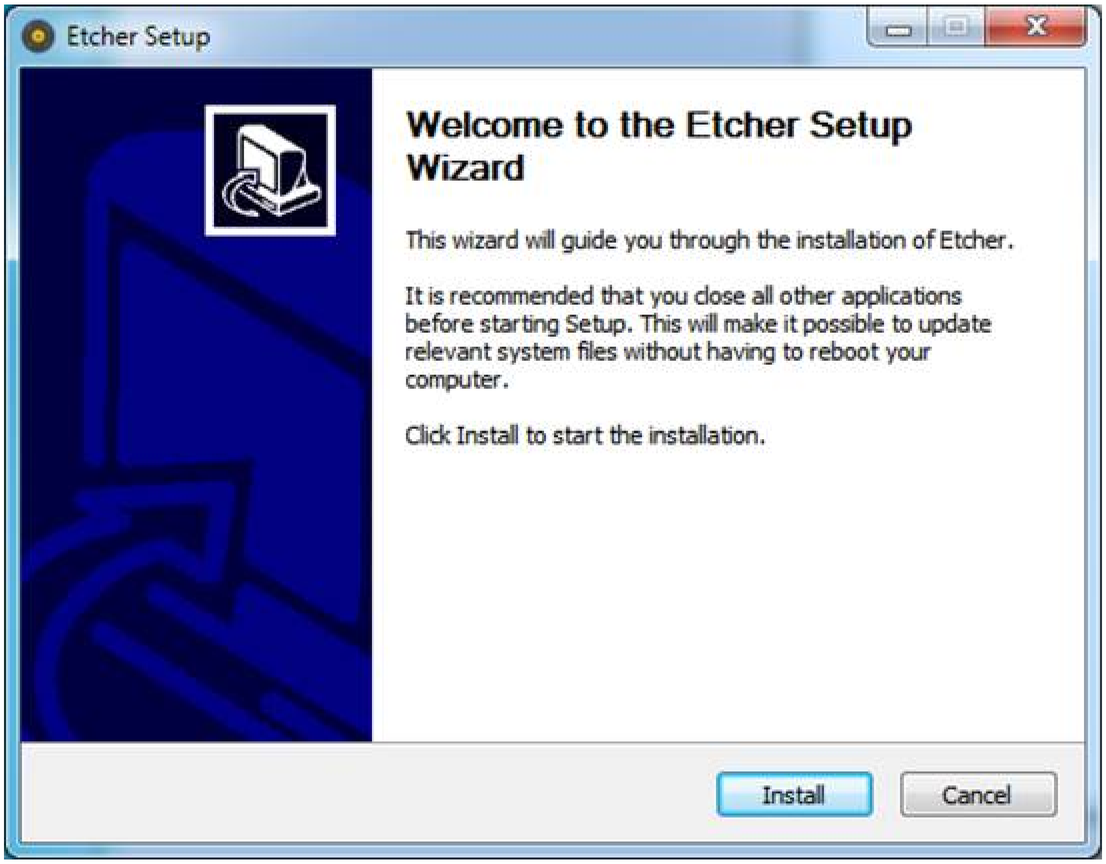
Etcher-1.0.0-beta.17-win32-x64.exe

which is: Binary File (80.8 MB)

from: https://resin-production-downloads.s3.amazonaws.com

Would you like to save this file?

2. Install Balena Etcher, as shown in the following figure.



Etcher Setup

Welcome to the Etcher Setup Wizard

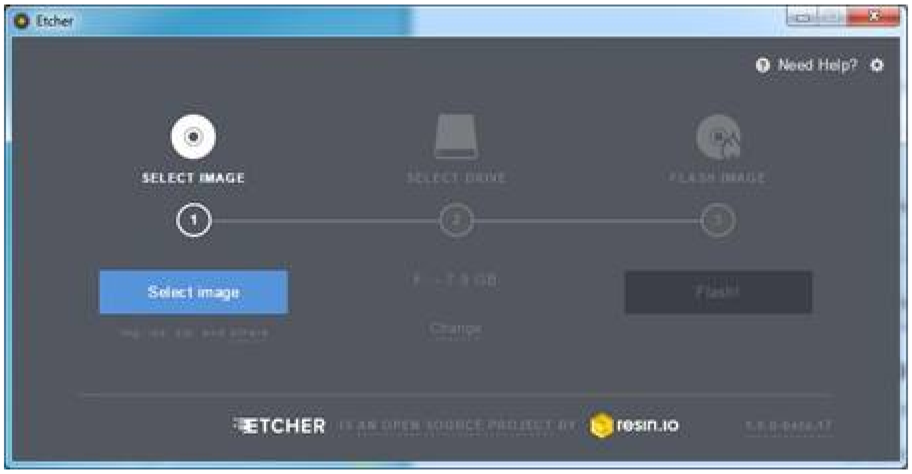
This wizard will guide you through the installation of Etcher.

It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

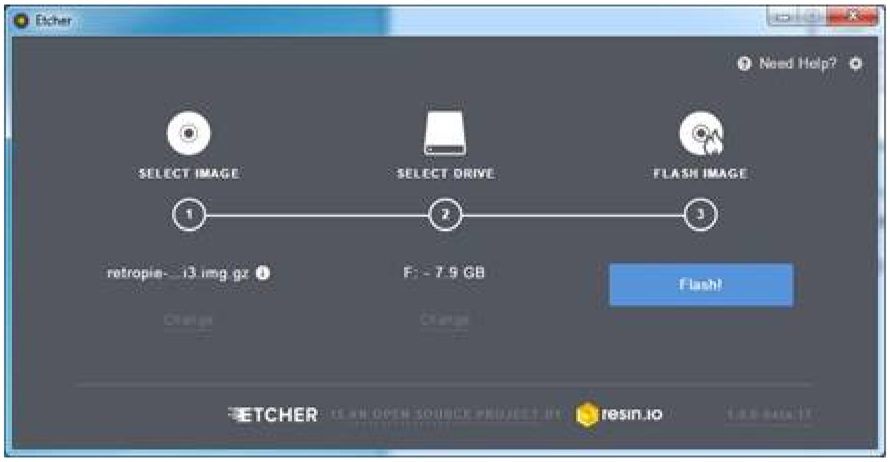
Click Install to start the installation.

3. Eject any external storage devices such as USB flash drives and backup hard disks. This makes it easier to identify the SD card. Then, insert the SD card into the slot on your computer, or into the reader.
4. Run the Etcher program by double-clicking on the Etcher icon shown in the following figure, or select it from the Start menu.

Etcher launches, as shown in the following figure.



5. Select the image file by clicking **Select Image**. You can select a .zip or .gz compressed file.
6. Etcher tries to detect the SD drive. Verify the drive designation and the image size.
7. Click **Flash!**.



8. Insert the SD card with the image into the destination board.
9. Plug in the power and boot the board using the serial port to access the system.
10. Set up the IP information of the board using the serial port.

    You can now access the board using SSH.

Step 2: Installing the AI Model Package

The Vitis AI runtime packages and the Vitis AI Library samples and models are compiled into the pre-built Vitis AI board images. Therefore, you do not have to install the Vitis AI runtime packages and the model package when the target is using a Vitis AI pre-built image. However, you can still install the model on a custom target or on the official image by following these steps:

1. For each model, there is a yaml file that describes all the details about the model. The yaml file contains download links for the various AMD target boards. Choose your model and the desired platform and download it.
2. Copy the downloaded file to the target using `scp` with the following command.

   ```
   $ scp <model>.tar.gz root@IP_OF_BOARD:~/
   ```

   If the target board is connected to the Internet, you can also use the `wget` command to download the model directly to the board.
3. Log in to the target board (using ssh or serial port) and install the model package.
4. If the `/usr/share/vitis_ai_library/model` folder does not exist on the target, create it first.

   ```
   # mkdir -p /usr/share/vitis_ai_library/models
   ```

5. Install the model on the target board.

   ```
   # tar -xzvf <model>.tar.gz -C /usr/share/vitis_ai_library/models
   ```

   By default, the models are located in the `/usr/share/vitis_ai_library/models` directory on the target.

Step 3: Installing the AI Library Package

The Vitis AI runtime packages and the Vitis AI Library samples and models are compiled into the pre-built Vitis AI board images. You do not have to install Vitis AI runtime packages and model packages when the target is using a Vitis AI pre-built image. However, you can still install the Vitis AI runtime on a custom target or on the official image by following these steps:

1. Download the `vitis-ai-runtime-3.5.0.tar.gz` from here. Untar it and copy the following files to the target using the `scp` command.

   ```
   $ tar -xzvf vitis-ai-runtime-3.5.0.tar.gz
   $ scp -r vitis-ai-runtime-3.5.0/2023.1/aarch64/centos root@IP_OF_BOARD:~/
   ```

   If the target board is connected to the Internet, you can also use the `wget` command to download the package directly to the board.

   ***Note*:** You can take the RPM package as a normal archive, and extract the contents on the host side if you only need some of the libraries. Only the model libraries can be separated into independent libraries. The other libraries are common. The operation command is as follows:

   ```
   $ rpm2cpio libvitis_ai_library-3.5.0-r<x>.aarch64.rpm | cpio -idmv
   ```

2. Log in to the board using ssh.

   You can also use the serial port to log in.

3. For Zynq UltraScale+ MPSoCs, run the `zynqmp_dpu_optimize.sh` script on the board.

   ```
   # cd ~/dpu_sw_optimize/zynqmp/
   # ./zynqmp_dpu_optimize.sh
   ```

4. Install the Vitis AI Library.

   ```
   # cd ~/centos
   # bash setup.sh
   ```

   You can also execute the following command to install the library one by one.

   ```
   # cd ~/centos
   # rpm -ivh --force libunilog-3.5.0-r<x>.aarch64.rpm
   # rpm -ivh --force libxir-3.5.0-r<x>.aarch64.rpm
   # rpm -ivh --force libtarget-factory-3.5.0-r<x>.aarch64.rpm
   # rpm -ivh --force libvart-3.5.0-r<x>.aarch64.rpm
   # rpm -ivh --force libvitis_ai_library-3.5.0-r<x>.aarch64.rpm
   ```

   Note: To install all of the rpms that are in a single directory, run `rpm -ivh --force *.rpm --nodeps`.

After the installation is complete, the directories are as follows:

- The library files are stored in the `/usr/lib` location.
- The header files are stored in the `/usr/include/vitis/ai` location.

Running Vitis AI Library Examples

Before running the Vitis AI Library examples on Edge or Data Center, download the vitis_ai_library_r3.5.0_images.tar.gz and vitis_ai_library_r3.5.0_video.tar.gz packages. The images or videos used in the following example can be found in both packages.

For Edge

The Vitis AI runtime packages, and Vitis AI Library samples and models are compiled into the pre-built Vitis AI board images. You can run the examples directly. If you have a new program, compile it on the host side and copy the executable program to the target.

1. Copy `vitis_ai_library_r3.5.0_images.tar.gz` and `vitis_ai_library_r3.5.0_video.tar.gz` from host to the target using the `scp` command as shown below:

   ```
   [Host]$scp vitis_ai_library_r3.5.0_images.tar.gz root@IP_OF_BOARD:~/
   [Host]$scp vitis_ai_library_r3.5.0_video.tar.gz root@IP_OF_BOARD:~/
   ```

2. Untar the image and video packages on the target.

```
cd ~
tar -xzvf vitis_ai_library_r3.5*_images.tar.gz -C Vitis-AI/examples/
vai_library
tar -xzvf vitis_ai_library_r3.5*_video.tar.gz -C Vitis-AI/examples/
vai_library
```

3. Enter the extracted directory of the example on the target board and then compile the example. Take `classification` as an example.

```
cd ~/Vitis-AI/examples/vai_library/samples/classification
```

4. Run the example.

```
./test_jpeg_classification resnet50_pt sample_classification.jpg
```

   Note: It supports batch mode. If the DPU batch number is more than 1, you can also run the following command.

```
./test_jpeg_classification resnet50_pt <img1_url> [<img2_url> ...]
```

5. View the running results.

   There are two ways to view the results. One is to view the results by printing the information. The other way is to view the images by downloading the `0_sample_classification_result.jpg` image.

6. To run the video example, run the following command:

```
./test_video_classification resnet50_pt video_input.webm -t 8
```

   where, `video_input.webm` is the name of the video file for input and `-t` is the number of threads. You must prepare the video file yourself.

   Note:

   - Pre-built Vitis AI board images only support video file input in the `webm` or `raw` format. If you want to use a video file in a format that is not natively supported, you have to install the relevant packages, such as the `ffmpeg` package, on the target.
   - When a display is used as a sink for the post-processed video, the performance will be limited to the maximum frame rate supported by the display interface on the target. This might not reflect the maximum performance, a fact that particularly important when you have enabled multi-threading to benchmark maximum frame rates. However, you can test the maximum inference performance of the Vitis AI Libraries by issuing the following command:

```
env DISPLAY=:0.0 DEBUG_DEMO=1 ./test_video_classification \
resnet50_pt 'multifilesrc location=~/video_input.webm \
! decodebin  !  videoconvert ! appsink sync=false' -t 2
```

7. To test the program with a USB camera as input, run the following command:

```
./test_video_classification resnet50_pt 0 -t 4
```

   Here, 0 is the first USB camera device node. If you have multiple USB cameras, the value is 1,2,3, etc., where, `-t` is the number of threads.

IMPORTANT! *Enable X11 forwarding with the following command (suppose in this example that the host machine IP address is 192.168.0.10) when logging in to the board using an SSH terminal because the test_video examples require a Linux windows subsystem target to work properly.*

```
export DISPLAY=192.168.0.10:0.0
```

8. To test the performance of the model, run the following command:

   ```
   ./test_performance_classification resnet50_pt
   test_performance_classification.list -t 8 -s 60
   ```

   Here, `-t` is the number of threads and `-s` is the number of seconds.

   To view a complete listing of command line options for the executable, run the command with the '`-h`' switch.

9. To run the demo, refer to Chapter 5: Application Demos.

For Data Center (Versal V70 Card)

To run an example for the Versal V70 card, use these steps:

1. After downloading the Vitis AI Library, navigate to the `Vitis-AI` directory, and then start Docker.
2. Enter the directory of the sample and then compile it. Take `resnet50_pt` as an example.

   ```
   cd /workspace/examples/vai_library/samples/classification
   bash -x build.sh
   ```

3. Run the sample.

   ```
   ./test_jpeg_classification resnet50_pt sample_classification.jpg
   ```

   If you want to run the program in batch mode, which means that the DPU processes multiple images simultaneously, you have to compile the entire Vitis AI Library according to the instructions in the Setting Up the Host section. Then the batch mode program will be generated in the `build_dir_default` location. Enter `build_dir_default`, and execute the following command.

   ```
   ./test_classification_batch resnet50_pt <img1_url> [<img2_url> ...]
   ```

4. To run the video example, run the following command:

   ```
   ./test_video_classification resnet50_pt <video_input.mp4> -t 8
   ```

   Here, `video_input.mp4` is the video file that you have to prepare for input and `-t` is the number of threads.

5. To test the performance of the model, run the following command:

   ```
   ./test_performance_classification resnet50_pt
   test_performance_classification.list -t 8 -s 60
   ```

Here, `-t` is the number of threads and `-s` is the number of seconds.

To view a complete listing of command line options for the executable, run the command with the '`-h`' switch.

***Note*:**

- The performance test program is automatically run in batch mode.
- If you run the examples in a hetergeneous V70 system, configure the SoftMax env using "export XLNX_ENABLE_C_SOFTMAX=1".

Support

You can visit the Vitis AI Library forum on the AMD website for topic discussions, knowledge sharing, FAQs, and requests for technical support.

Chapter 3

Libraries and Samples

Caffe Framework

AMD Vitis™ AI contains the following neural network libraries based on the Caffe framework:

- Classification
- Face Detection
- SSD Detection
- Pose Detection
- Semantic Segmentation
- Road Line Detection
- YOLOv2 Detection
- YOLOv3 Detection
- YOLOv4 Detection
- Openpose Detection
- RefineDet Detection
- ReID Detection
- MultiTask
- Face Recognition
- Plate Detection
- Plate Recognition
- Medical Segmentation

TensorFlow Framework

Vitis AI contains the following neural network libraries based on the TensorFlow framework:

- Classification
- SSD Detection
- YOLOv3 Detection
- Medical Detection

- Semantic Segmentation
- RCAN Super Resolution
- EfficientDet_D2
- SuperPoint
- HFNet
- 2DUnet

PyTorch Framework

Vitis AI supports the following type of neural network libraries based on the PyTorch framework.

- Classification
- ReID Detection
- Face Recognition
- Semantic Segmentation
- PointPillars
- Medical Segmentation
- 3D Segmentation
- PointPillars_nuscenes: Surround-view
- Centerpoint: 4D radar-based 3D detection
- PointPainting: Image-lidar sensor fusion
- Depth Estimation
- Bayesian Crowd Counting
- MultiTask V3
- Polyp Segmentation
- UltraFast Road Line Detection
- FairMot
- PSMNet
- SOLO
- CLOCs
- OCR
- Textmountain Detection
- Vehicle Classification
- OFA_YOLO Detection

- Monodepth2
- YOLOv5 Detection
- BEVDet Detection
- cFlownet
- YOLOv6 Detection
- YOLOv7 Detection
- YOLOv8 Detection

The related libraries are open-source and can be modified as needed. The source code is available on GitHub.

The Vitis AI Library provides test images and video test sequences for all the above networks. In addition, the Vitis AI Library package provides the corresponding performance test application. For video-based testing, use the raw video sequences for evaluation. The use of encoded video sequences for evaluation is not recommended as software decoders implemented on Arm processors may exhibit decode jitter which may affect the accuracy of evaluation.

Note: For Edge devices, all sample applications execute on the target but can be cross-compiled on the host or on the target.

Model Library

After the model package is installed on the target, all the models are stored under `/usr/share/vitis_ai_library/models/`. Each model is stored in a separate folder, which is composed of the following files, by default:

- `[model_name].xmodel`
- `[model_name].prototxt`

Note: The elf model is not supported by the Vitis AI Library in the VAI 1.3 and later releases.

Take the "inception_v1" model as an example. `inception_v1.xmodel` is the model data. `inception_v1.prototxt` is the parameter of the model.

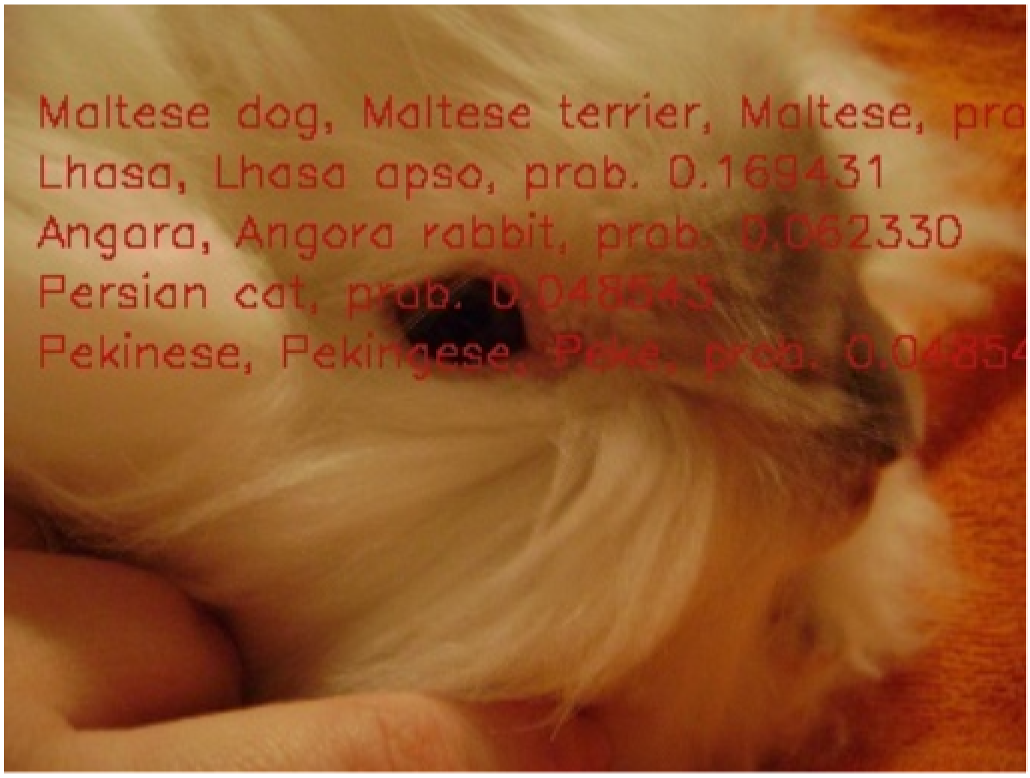
Note: The name of the model directory should be the same as the model name.

Model Type

Classification

The Classification library is used to classify images. Such neural networks are trained on the 1000 class ILSVRC subset of the ImageNet dataset and can predict a per-class probability that the object in the image is a member of each class. The Vitis AI Library supports classification networks including, but not limited to, ResNet18, ResNet50, Inception_v1, Inception_v2, Inception_v3, Inception_v4, VGG, mobilenet_v1, mobilenet_v2, and Squeezenet. The input is a picture with an object and the output is the top-K most probable category.

Figure 2: **Classification Example**



```
Maltese dog, Maltese terrier, Maltese, pro
Lhasa, Lhasa apso, prob. 0.169431
Angora, Angora rabbit, prob. 0.062330
Persian cat, prob. 0.048543
Pekinese, Pekingese, Peke, prob. 0.04854
```

The following table lists the classification models supported by the Vitis AI library.

Table 6: **Classification Models**

No	Model Name	Framework
1	inception_resnet_v2_tf	TensorFlow
2	inception_v1_tf	
3	inception_v3_tf	
4	inception_v4_2016_09_09_tf	
5	mobilenet_v1_0_25_128_tf	
6	mobilenet_v1_0_5_160_tf	
7	mobilenet_v1_1_0_224_tf	
8	mobilenet_v2_1_0_224_tf	
9	mobilenet_v2_1_4_224_tf	
10	resnet_v1_101_tf	
11	resnet_v1_152_tf	
12	resnet_v1_50_tf	
13	vgg_16_tf	
14	vgg_19_tf	
15	mobilenet_edge_1_0_tf	
16	mobilenet_edge_0_75_tf	
17	inception_v2_tf	
18	MLPerf_resnet50_v1.5_tf	
19	resnet50_tf2	
20	mobilenet_1_0_224_tf2	
21	inception_v3_tf2	
22	resnet_v2_50_tf	
23	resnet_v2_101_tf	
24	resnet_v2_152_tf	
25	efficientnet-b0_tf2	
26	efficientNet-edgetpu-S_tf	
27	efficientNet-edgetpu-M_tf	
28	efficientNet-edgetpu-L_tf	
29	mobilenet_v3_small_1_0_tf2	
30	efficientnet_lite_tf2	
31	resnet50	Caffe
32	resnet18	
33	inception_v1	
34	inception_v2	
35	inception_v3	
36	inception_v4	
37	mobilenet_v2	
38	squeezenet	

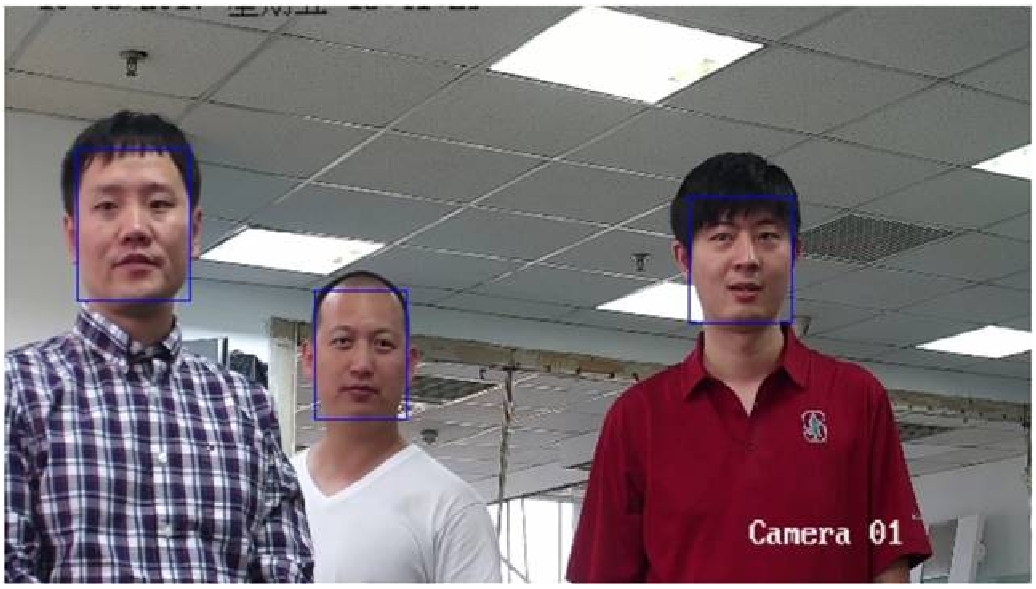
Table 6: **Classification Models** *(cont'd)*

No	Model Name	Framework
39	resnet50_pt	PyTorch
40	squeezenet_pt	
41	inception_v3_pt	
42	ofa_resnet50_0_9B_pt	
43	person-orientation_pruned_558m_pt	
44	ofa_depthwise_res50_pt	
45	chen_color_resnet18_pt	

Face Detection

The Face Detection library uses the DenseBox neural network to detect human faces. The input is a picture with the faces you want to detect and the output is a vector containing the coordinates and probability for each bounding box. The following image shows the result of face detection.

Figure 3: **Face Detection Example**



The following table lists the face detection models supported by the AI Library.

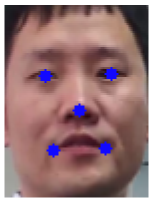
Table 7: **Face Detection Models**

No	Model Name	Framework
1	densebox_320_320	Caffe
2	densebox_640_360	

Face Landmark Detection

The Face Landmark network is used to detect five key points on a human face. The five points include the left eye, the right eye, the nose, the left corner of the lips, and the right corner of the lips. This network is used to correct face direction (what this means is if a face is not directly facing the camera (e.g., tilted 20 degrees left or right), it is "adjusted" to face the camera directly) before face feature extraction. The input image should be a face that is detected by the face detection network. The output of the network is the five key points. The five key points are normalized. The following image shows the result of face detection.

Figure 4: **Face Landmark Detection Example**



The following table lists the face landmark models supported by the AI Library.

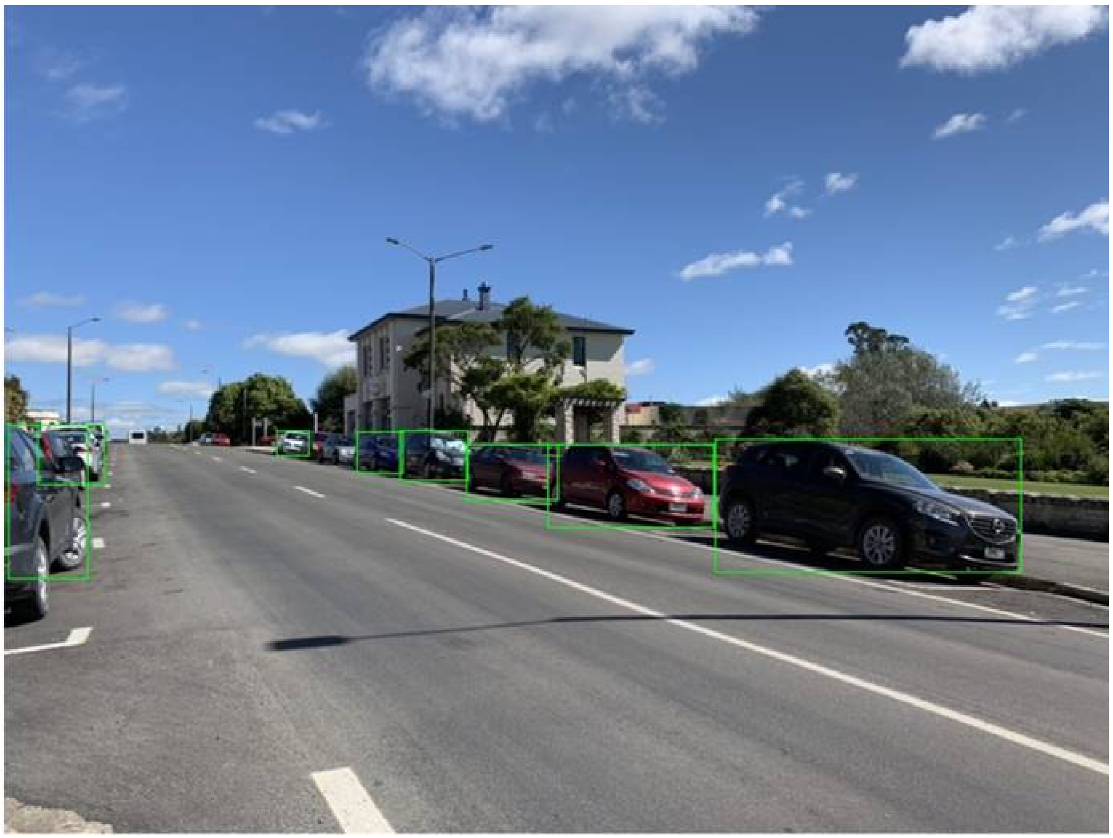
Table 8: **Face Landmark Models**

No	Model Name	Framework
1	face_landmark	Caffe

SSD Detection

The SSD Detection library is commonly used with the SSD neural network. SSD is a neural network that is used to detect objects. The input is a picture with some objects you want to detect. The output is a structure containing the coordinates and probability for each bounding box. The following image shows the result of SSD detection.

Figure 5: **SSD Detection Example**



The following table lists the SSD detection models supported by the Vitis AI Library.

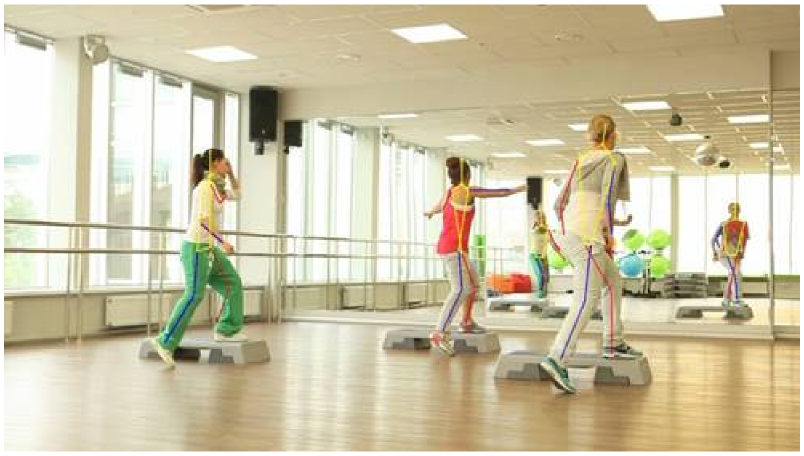
Table 9: **SSD Models**

No	Model Name	Framework
1	ssd_mobilenet_v1_coco_tf	TensorFlow
2	ssd_mobilenet_v2_coco_tf	
3	ssd_resnet_50_fpn_coco_tf	
4	mlperf_ssd_resnet34_tf	
5	ssdlite_mobilenet_v2_coco_tf	
6	ssd_inception_v2_coco_tf	
7	ssd_pedestrian_pruned_0_97	Caffe
8	ssd_traffic_pruned_0_9	
9	ssd_adas_pruned_0_95	
10	ssd_mobilenet_v2	

Pose Detection

The Pose Detection library is used to detect the posture of the human body. This library includes a neural network that can identify 14 key points on the human body. The input is normally a cropped region that was detected by a pedestrian detection neural network such as SSD or RefineDet available in the Vitis AI Model Zoo. The output is a structure containing the coordinates of each point. The following image shows the result of pose detection.

Figure 6: **Pose Detection Example**



The following table lists the pose detection model supported by the Vitis AI Library.

Table 10: **Pose Detection Model**

No	Model Name	Framework
1	sp_net	Caffe

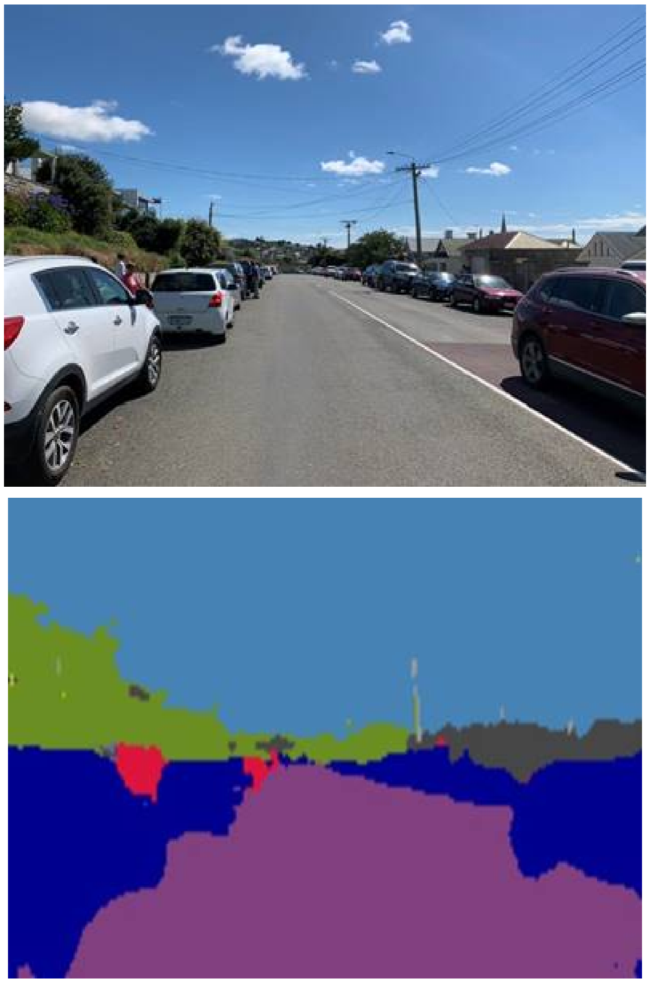
Note: If the input image is arbitrary and you do not know the coordinates of the person, perform a cascaded person detection (pose detection pipeline is required). See the `test_jpeg_posedetect_with_ssd.cpp` file. The input for test_jpeg_posedetect_ssd can be any image with or without a person in it. If there is a person in the image, this cascaded pipeline will first detect the person with SSD, then send the position of the person as the input for posedetect. If the detection network does not localize any person in the image, posedetect does not run. As test_jpeg_posedetect only performs posedetect, the input image must have at least one person. If you input an image without a person for test_jpeg_posedetect, it will throw an error. See the `test_jpeg_posedetect.cpp` file.

Semantic Segmentation

Semantic segmentation assigns a semantic category to each pixel in the input image, that is, it classifies pixels as part of an object, say, a car, a road, a tree, a horse, etc. Libsegmentation is a segmentation library that can be used in ADAS applications. It offers simple interfaces for a developer to deploy segmentation tasks on an AMD target.

The following is an example of semantic segmentation, where "blue-gray" denotes the sky, "green" denotes trees, "red" denotes people, "dark blue" denotes cars, "plum" denotes the road, and "gray" denotes structures.

Figure 7: **Semantic Segmentation Example**



The following table lists the semantic segmentation models supported by the Vitis AI Library.

Table 11: **Semantic Segmentation Models**

No	Model Name	Framework
1	fpn	Caffe
2	FPN-resnet18_Endov	
3	semantic_seg_citys_tf2	TensorFlow
4	mobilenet_v2_cityscapes_tf	

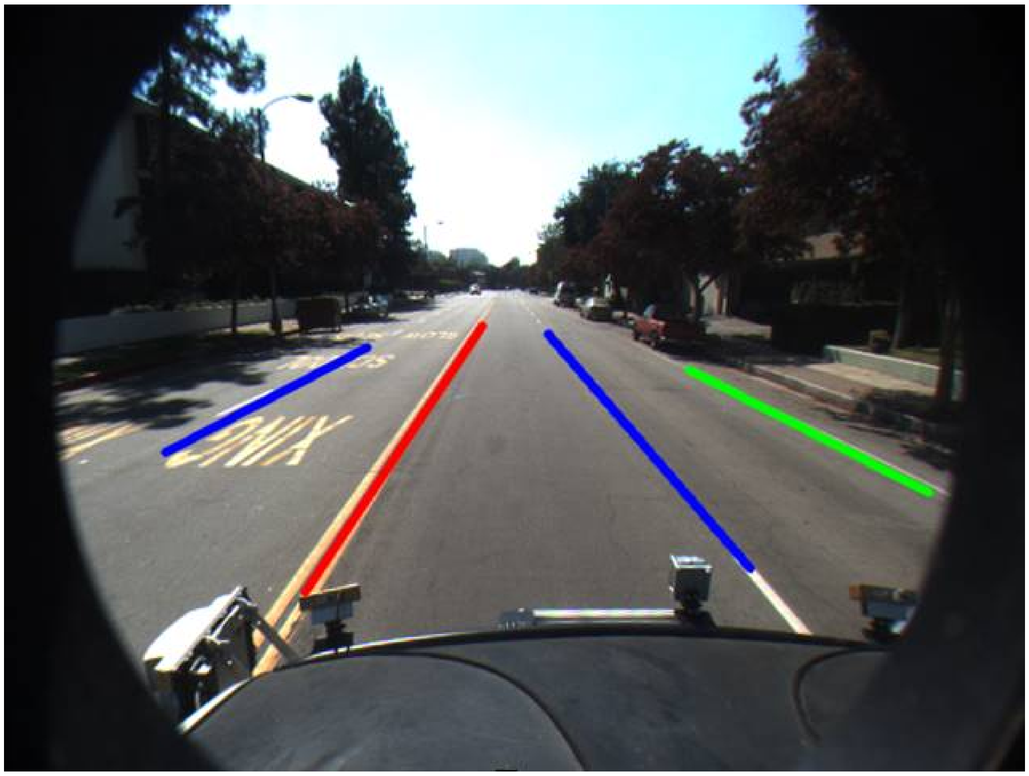
Table 11: **Semantic Segmentation Models** *(cont'd)*

No	Model Name	Framework
5	SemanticFPN_cityscapes_pt	PyTorch
6	ENet_cityscapes_pt	
7	unet_chaos-CT_pt	
8	SemanticFPN_Mobilenetv2_pt	
9	HRNet_pt	

Road Line Detection

The Road Line Detection library is used to draw lane lines in ADAS applications. Each lane line is represented by a number representing the category. A vector<Point> is used to draw the lane line. In the test code, a color map is used. Different types of lane lines are represented by different colors. The point is stored in the container vector, and the polygon interface `cv::polylines()` of OpenCV is used to draw the lane line. The following image shows the result of road line detection.

Figure 8: **Road Line Detection Example**



The following table lists the road line detection models supported by the Vitis AI Library.

Table 12: **Road Line Detection Models**

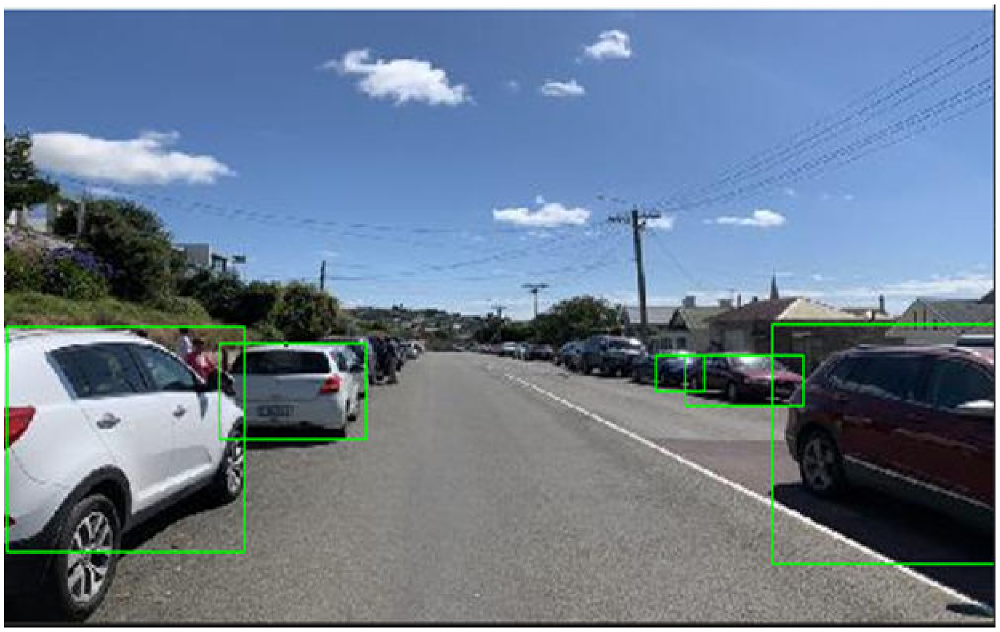
No	Model Name	Framework
1	vpgnet_pruned_0_99	Caffe

Note: The input to the Vitis AI Library implementation of this network is fixed at 480x640. Inputs of different dimensions must be resized.

YOLOv3 Detection

YOLOv3 is a neural network used to detect objects. The input is a picture with one or more objects and the output is a vector of the result struct which is composed of the detected information. The following image shows the result of YOLOv3 detection.

Figure 9: **YOLOv3 Detection Example**



The following table lists the YOLOv3 detection models supported by the Vitis AI library.

Table 13: **YOLOv3 Detection Models**

No	Model Name	Framework
1	yolov3_voc_tf	TensorFlow
2	yolov3_adas_pruned_0_9	Caffe
3	yolov3_voc	
4	yolov3_bdd	
5	tiny_yolov3_vmss	
6	yolov3_coco_416_tf2	TensorFlow2

YOLOv4 Detection

YOLOv4 is an upgraded version of YOLOv3 and does the same thing as YOLOv3. The following table lists the YOLOv4 detection models supported by the Vitis AI Library.

Table 14: **YOLOv4 Detection Models**

No	Model Name	Framework
1	yolov4_leaky_spp_m	Caffe
2	yolov4_leaky_spp_m_pruned_0_36	
3	face_mask_detection_pt	PyTorch
4	yolov4_csp_pt	
5	yolov4_leaky_416_tf	TensorFlow
6	yolov4_leaky_512_tf	TensorFlow

YOLOv2 Detection

YOLOv2 does the same thing as YOLOv3, which is an upgraded version of YOLOv2. The following table lists the YOLOv2 detection models supported by the Vitis AI Library.

Table 15: **YOLOv2 Detection Models**

No	Model Name	Framework
1	yolov2_voc	Caffe
2	yolov2_voc_pruned_0_66	
3	yolov2_voc_pruned_0_71	
4	yolov2_voc_pruned_0_77	

Openpose Detection

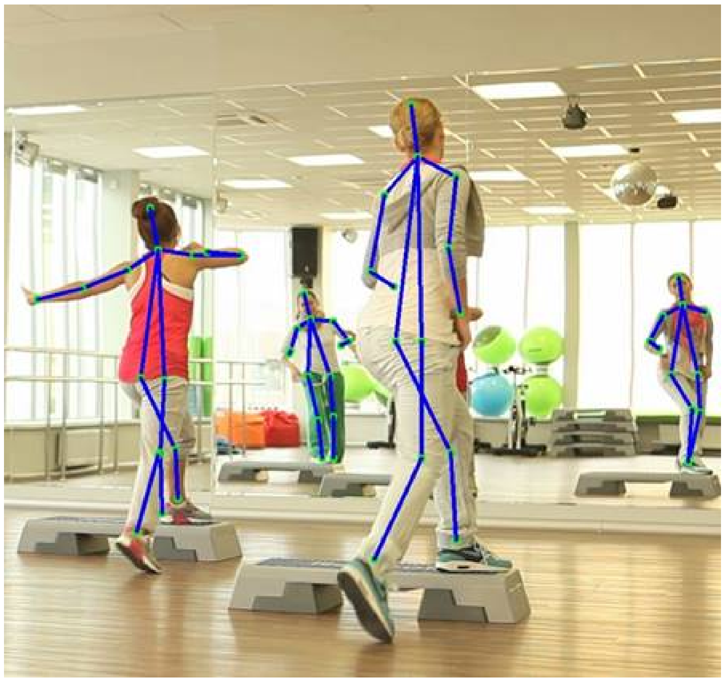
The Openpose Detection library is used to detect the posture of the human body. The posture is represented by an array of 14 key points as shown below:

```
0: head, 1: neck, 2: L_shoulder, 3:L_elbow, 4: L_wrist, 5: R_shoulder,
6: R_elbow, 7: R_wrist, 8: L_hip, 9: L_knee, 10: L_ankle, 11: R_hip,
12: R_knee, 13: R_ankle
```

The input of the network is 368x368. The following image shows the result of openpose detection.

RECOMMENDED: *Use a square picture for input. If you need to detect pictures of other size ratios, use a network with the same input size ratio.*

Figure 10: **Openpose Detection Example**



The following table lists the Openpose detection models supported by the Vitis AI Library.

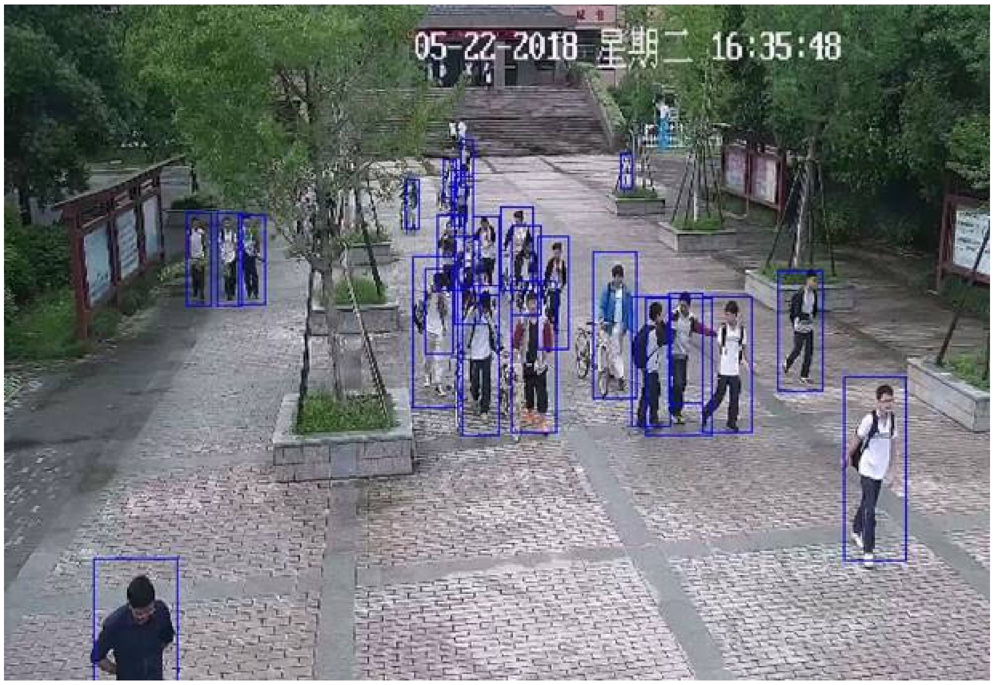
Table 16: **Openpose Detection Models**

No	Model Name	Framework
1	openpose_pruned_0_3	Caffe

RefineDet Detection

RefineDet is a neural network that is used to detect human bodies. The input is a picture with some individuals that you would like to detect. The output is a vector of the resulting structure that contains each box's information. The following image shows the result of RefineDet detection:

Figure 11: **RefineDet Detection Example**



The following table lists the RefineDet detection models supported by the Vitis AI Library.

Table 17: **RefineDet Detection Models**

No	Model Name	Framework
1	refinedet_pruned_0_8	Caffe
2	refinedet_pruned_0_92	
3	refinedet_pruned_0_96	
4	refinedet_baseline	
5	refinedet_VOC_tf	TensorFlow

ReID Detection

The task of person re-identification is to identify a person of interest at any time or place. This is done by extracting the image feature and comparing the features. Images of the same person should have similar features and have small feature distances, while images of different persons have large feature distances. Given a queried image and a pile of candidate images, the image that has the smallest feature distance is identified as the same person as the queried image. The following table lists the ReID detection models supported by the Vitis AI Library.

Table 18: **ReID Detection Models**

Number	Model Name	Framework
1	reid	Caffe

Table 18: **ReID Detection Models** *(cont'd)*

Number	Model Name	Framework
2	personreid-res18_pt	PyTorch
3	personreid-res50_pt	
4	facereid-large_pt	
5	facereid-small_pt	

MultiTask

The MultiTask library is appropriate for a model that has multiple subtasks. The MultiTask model in the Vitis AI Library has two subtasks: semantic segmentation and SSD detection. The following table lists the MultiTask models supported by the Vitis AI Library.

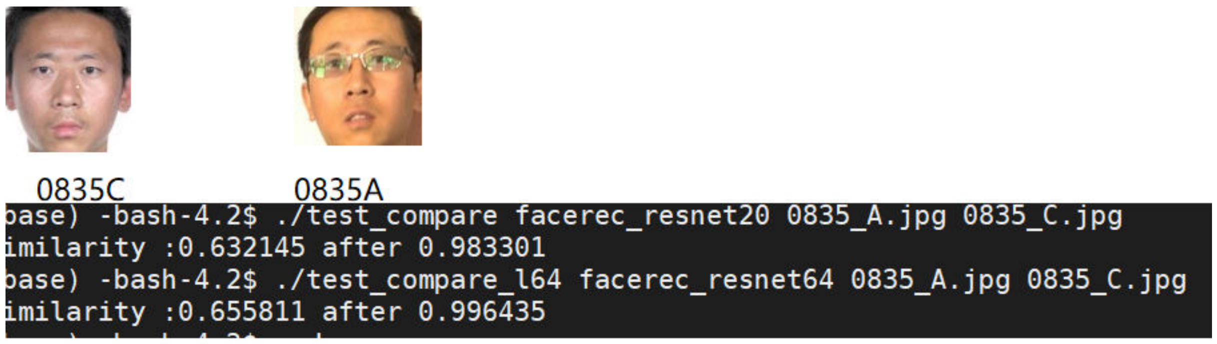
Table 19: **MultiTask Models**

Number	Model Name	Framework
1	multi_task	Caffe
2	MT-resnet18_mixed_pt	PyTorch

Face Recognition

The face feature models are used for face recognition. They can extract the features of a person's face. The output of these models is 512 features. If you have two different images and you want to know if they are of the same person, use these models to extract features of the two images, and then use calculation functions and mapped functions to get the similarity of the two images.

Figure 12: **Face Recognition Example**



```
base) -bash-4.2$ ./test_compare facerec_resnet20 0835_A.jpg 0835_C.jpg
imilarity :0.632145 after 0.983301
base) -bash-4.2$ ./test_compare_l64 facerec_resnet64 0835_A.jpg 0835_C.jpg
imilarity :0.655811 after 0.996435
```

The following table lists the face recognition models supported by the Vitis AI Library.

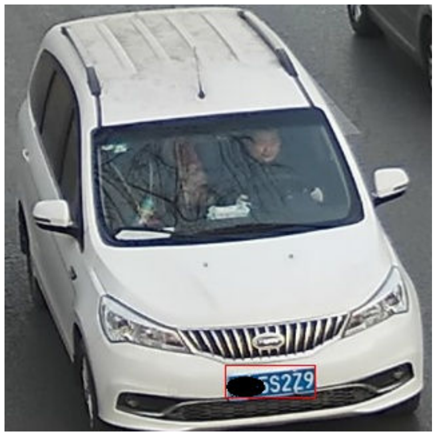
Table 20: **Face Recognition Models**

No	Model Name	Framework
1	facerec_resnet20	Caffe
2	facerec_resnet64	
3	facerec-resnet20_mixed_pt	PyTorch

Plate Detection

The Plate Detection library uses the DenseBox neural network to detect license plates. The input is a picture of the vehicle that is detected by the SSD and the output is a structure containing the plate location information. The following image shows the result of the plate detection.

Figure 13: **Plate Detection Example**



The following table lists the plate detection models supported by the Vitis AI Library.

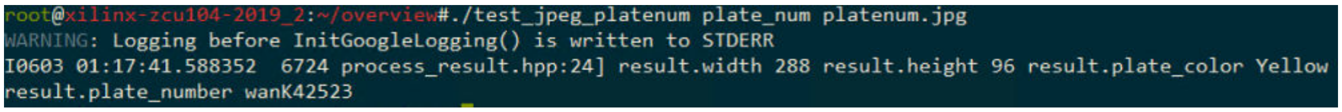
Table 21: **Plate Detection Models**

No	Model Name	Framework
1	plate_detect	Caffe

Plate Recognition

The Plate Recognition library uses a classification network to recognize license plate numbers (Chinese license plates only). The input is a picture of the license plate that is detected by plate detect. The output is a structure containing license plate number information. The following image shows the result of the plate recognition.

Figure 14: **Plate Recognition Example**



```
root@xilinx-zcu104-2019_2:~/overview#./test_jpeg_platenum plate_num platenum.jpg
WARNING: Logging before InitGoogleLogging() is written to STDERR
I0603 01:17:41.588352  6724 process_result.hpp:24] result.width 288 result.height 96 result.plate_color Yellow
result.plate_number wanK42523
```

The following table lists the plate recognition models supported by the Vitis AI Library.

Table 22: **Plate Recognition Models**

No	Model Name	Framework
1	plate_num	Caffe

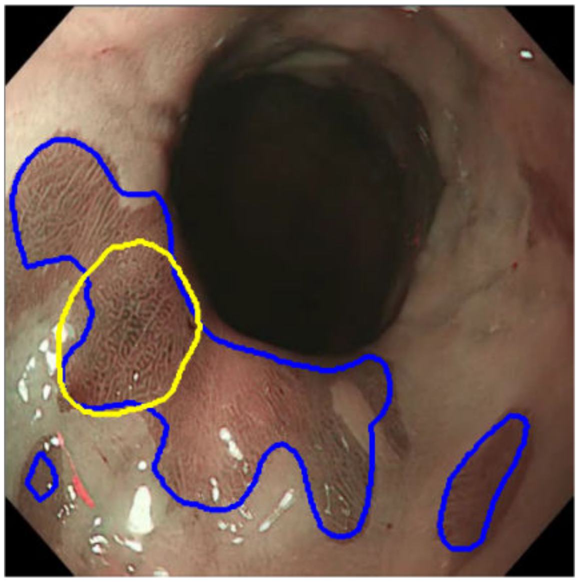
Medical Segmentation

Endoscopy is a common clinical procedure for the early detection of cancers in hollow organs such as nasopharyngeal cancer, esophageal adenocarcinoma, gastric cancer, colorectal cancer, and bladder cancer. Accurate and temporally consistent localization and segmentation of diseased region-of-interests enable precise quantification and mapping of lesions from clinical endoscopy videos, which is critical for monitoring and surgical planning.

The medical segmentation model is used to classify diseased region-of-interests in the input image. It can be classified into many categories, including BE, cancer, HGD, polyp, and suspicious.

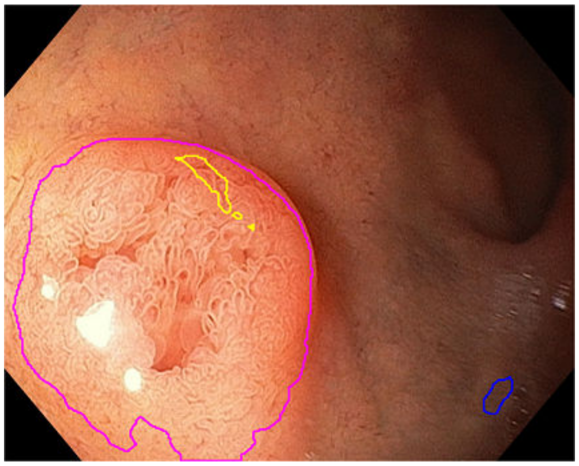
Libmedicalsegmentation is a segmentation library that can be used in the segmentation of multiclass diseases in endoscopy. It offers simple interfaces for developers to deploy segmentation tasks on AMD devices. The following is an example of medical segmentation, where the goal is to mark the diseased region.

Figure 15: **Marking the Diseased Region**



The following is an example of semantic segmentation, where the goal is to predict class labels for each pixel in the image.

Figure 16: **Medical Segmentation Example**



The following table lists the medical segmentation models supported by the Vitis AI Library.

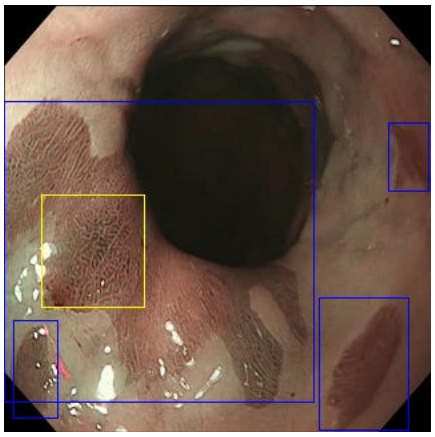
Table 23: **Semantic Segmentation Models**

No	Model Name	Framework
1	FPN_Res18_Medical_segmentation	Caffe

Medical Detection

The RefineDet model is based on vgg16. It is used for medical detection and can detect five types of diseases, namely, BE, cancer, HGD, polyp, and suspicious from an input endoscopy image like the Endoscopy Disease Detection and Segmentation database (EDD2020).

Figure 17: **Medical Detection Example**



The following table lists the medical detection models supported by the Vitis AI Library.

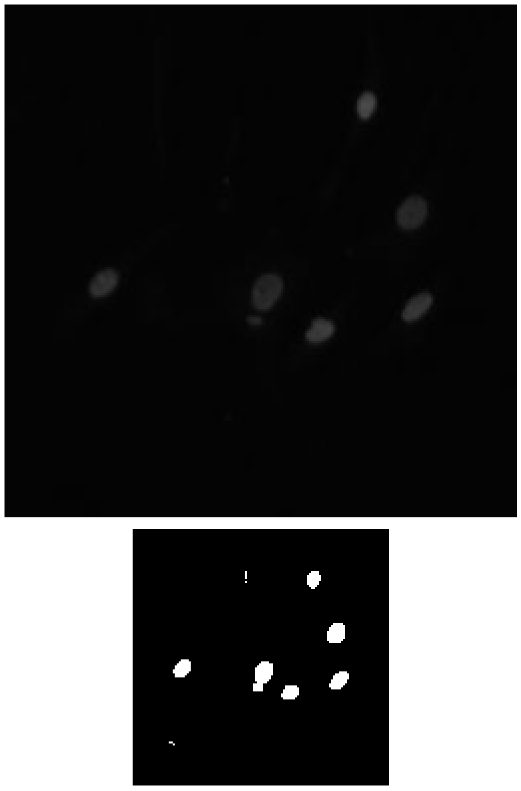
Table 24: **Semantic Detection Models**

No	Model Name	Framework
1	RefineDet-Medical_EDD_tf	TensorFlow

Medical Cell Segmentation

The nucleus is an organelle present within all eukaryotic cells, including human cells. Aberrant nuclear shape can be used to identify cancer cells, for example, pap smear tests for the diagnosis of cervical cancer. Medical segmentation cell models offer nuclear segmentation in digital microscopic tissue images which can enable extraction of high-quality features for nuclear morphometric and other analyses in computational pathology. The following images show the results of cell segmentation.

Figure 18: **Medical Cell Segmentation Examples**



The following table lists the Medical Cell Segmentation models supported by the Vitis AI Library.

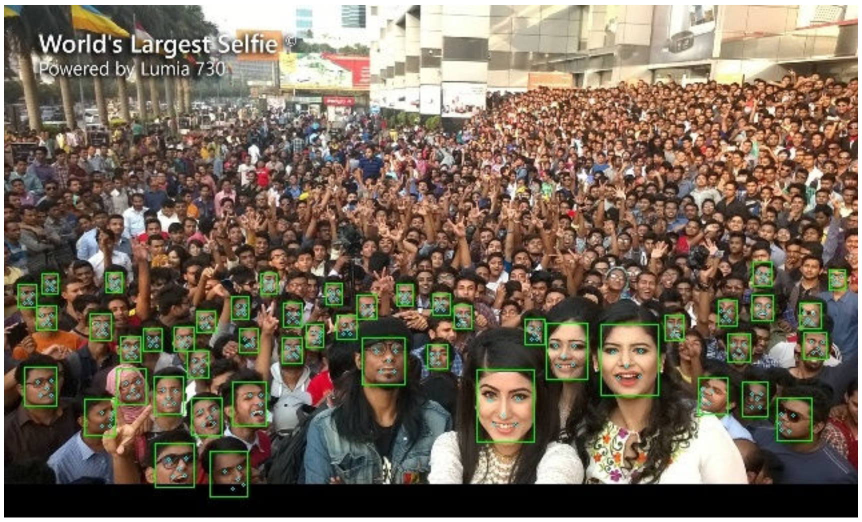
Table 25: **Medical Cell Segmentation Models**

No	Model Name	Framework
1	medical_seg_cell_tf2	TensorFlow

Retinaface

This retinaface network is used to detect human face and face landmarks. The input is a picture with some faces you would like to detect and the output contains face positions, scores, and landmarks of faces.

Figure 19: **Retinaface Detection Example**



The following table lists the retinaface detection models supported by the Vitis AI Library.

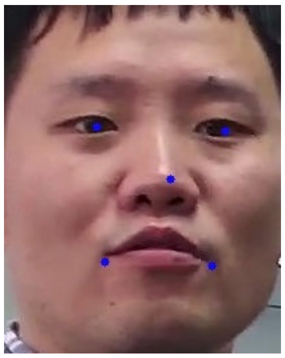
Table 26: **Retinaface Detection Models**

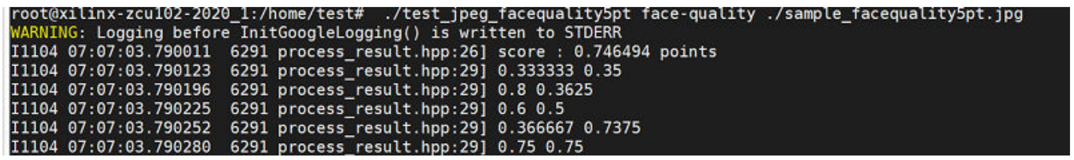
No	Model Name	Framework
1	retinaface	Caffe

Face Quality

The Face Quality library uses the face quality network to detect the quality score of a face. If a person is facing the camera with no obstructions, the score is high. On the contrary, a blurry or face in profile will get a low score. The scores range from 0 to 1. It also provides face landmark positions. The input is a face that is detected by the face detect network and the output contains a quality score and five landmark key points.

Figure 20: **Face Quality Example**





```
root@xilinx-zcu102-2020_1:/home/test#  ./test_jpeg_facequality5pt face-quality ./sample_facequality5pt.jpg
WARNING: Logging before InitGoogleLogging() is written to STDERR
I1104 07:07:03.790011  6291 process_result.hpp:26] score : 0.746494 points
I1104 07:07:03.790123  6291 process_result.hpp:29] 0.333333 0.35
I1104 07:07:03.790196  6291 process_result.hpp:29] 0.8 0.3625
I1104 07:07:03.790225  6291 process_result.hpp:29] 0.6 0.5
I1104 07:07:03.790252  6291 process_result.hpp:29] 0.366667 0.7375
I1104 07:07:03.790280  6291 process_result.hpp:29] 0.75 0.75
```

The following table lists the face quality models supported by the Vitis AI Library.

Table 27: **Face Quality Models List**

No	Model Name	Framework
1	face-quality	Caffe
2	face-quality_pt	PyTorch

Hourglass Pose Detection

The Hourglass library is used to detect the posture of the human body. It is represented by an array of 16 joint points. Joint points are arranged in order:

```
0 - r ankle, 1 - r knee, 2 - r hip, 3 - l hip, 4 - l knee, 5 - l ankle,
6 - pelvis, 7 - thorax, 8 - upper neck, 9 - head top, 10 - r wrist,
11 - r elbow, 12 - r shoulder, 13 - l shoulder, 14 - l elbow, 15 - l wrist
```

This network can detect the posture of only one person in the input image. The input of the network is 256x256. The following image shows the result of hourglass detection.

RECOMMENDED: *Use a square picture for input. If you need to detect pictures of other size ratios, use a network with the same input size ratio.*

The following table lists the hourglass models supported by the Vitis AI Library.

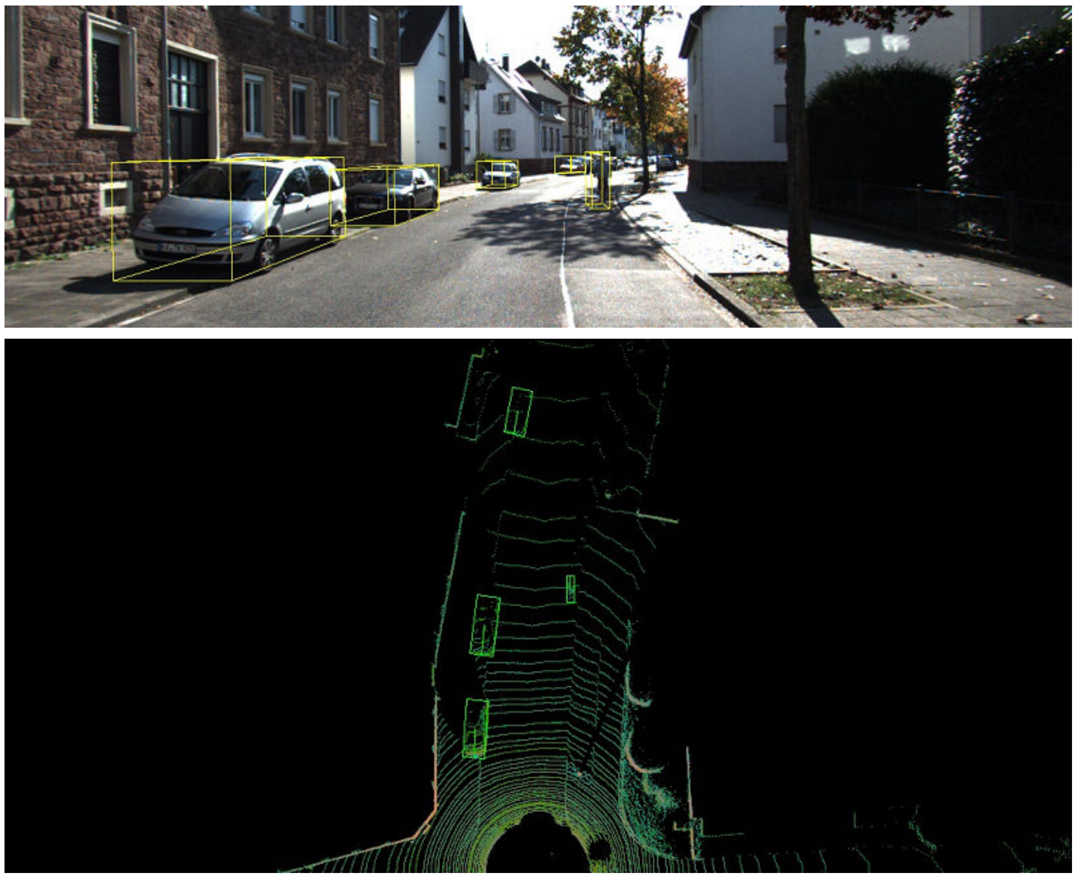
Table 28: **Hourglass Models**

No	Model Name	Framework
1	hourglass-pe_mpii	Caffe

PointPillars

Object detection in point clouds is an important aspect of many robotics applications such as autonomous driving. The PointPillars model is a novel deep network and encoder that can be trained end-to-end on LiDAR point clouds. It offers the best architecture for 3D object detection from LiDAR. The following image shows the result of a pointpillar test.

Figure 21: **PointPillars Test Example**



The following table lists the PointPillars models supported by the Vitis AI Library.

Table 29: **PointPillars Models**

No	Model Name	Framework
1	pointpillars_kitti_12000_0_pt	PyTorch
2	pointpillars_kitti_12000_1_pt	PyTorch

3D Segmentation

The 3D segmentation library can support the SalsaNext model, which is used for the uncertainty-aware semantic segmentation of a full 3D LiDAR point cloud in real-time. SalsaNext is the next version of SalsaNet which has an encoder-decoder architecture, where the encoder unit has a set of ResNet blocks and the decoder unit combines upsampled features from the residual blocks.

The following table lists the3D segmentation models supported by the Vitis AI library.

Table 30: **3D Segmentation Models**

No	Model Name	Framework
1	salsanext_pt	PyTorch
2	salsanext_v2_pt	PyTorch

Covid19 Segmentation

The Covid19 segmentation library can support the COVID-Net model which is a deep convolutional neural network design tailored for the detection of COVID-19 cases from chest X-ray (CXR) images.

The following table lists the Covid19 segmentation model supported by the Vitis AI Library.

Table 31: **Covid19 Segmentation Model**

No	Model Name	Framework
1	FPN-resnet18_covid19-seg_pt	PyTorch

Bayesian Crowd Counting

Bayesian Crowd Counting is a neural network that is used for counting the number of individuals in a crowd. The input is a picture of a crowd whose size you would like to estimate. The output is the estimated number of individuals in the crowd.

The following table lists the BCC models supported by the Vitis AI library.

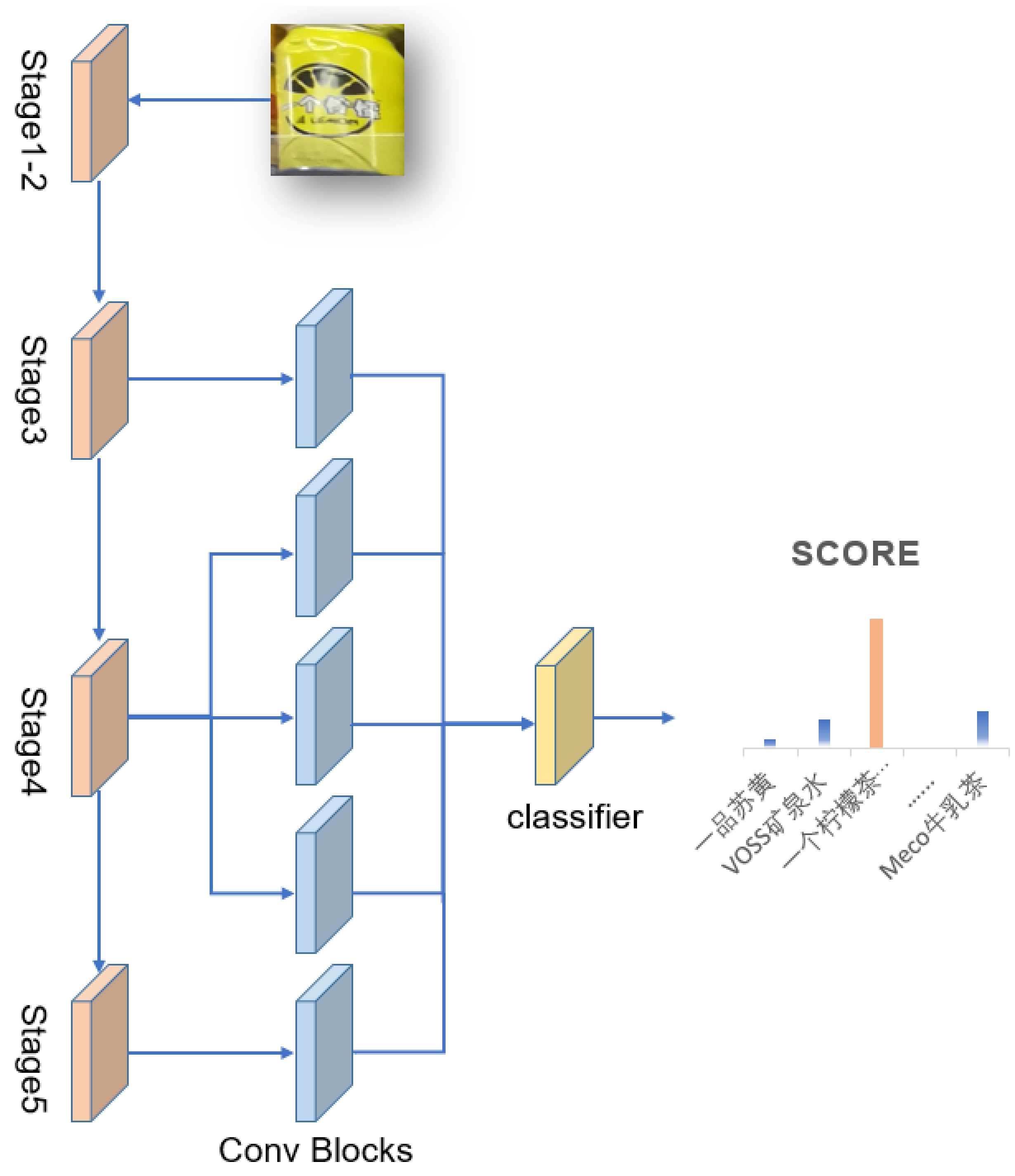
Table 32: **BCC Models**

No	Model Name	Framework
1	bcc_pt	PyTorch

Product Recognition

PMG model can be used for fine-grained goods product recognition, for example, RP2K dataset. The model is ResNet18-based and the detailed model structure is shown in the picture below. On the rp2k dataset, this model can achieve 96.4% top-1 float accuracy with 13.82M parameters and 2.28G Flops. Model final deployment and quantized top-1 accuracies are 96.19% and 96.18%, respectively.

Figure 22: **Product Recognition Example**



The following table lists the PMG models supported by the Vitis AI Library.

Table 33: **PMG Models**

No	Model Name	Framework
1	pmg_pt	PyTorch

SA-Gate Segmentation

SA-Gate is a neural network that is used for indoor segmentation. The input is a pair of an RGB image and an HHA map generated with the depth map. The output is a heat map where each pixel is predicted with a semantic category, like chair, bed, and other objects typically found indoors.

The following image shows the result of SA-Gate segmentation.

Figure 23: **SA-Gate segmentation Test Example 1**

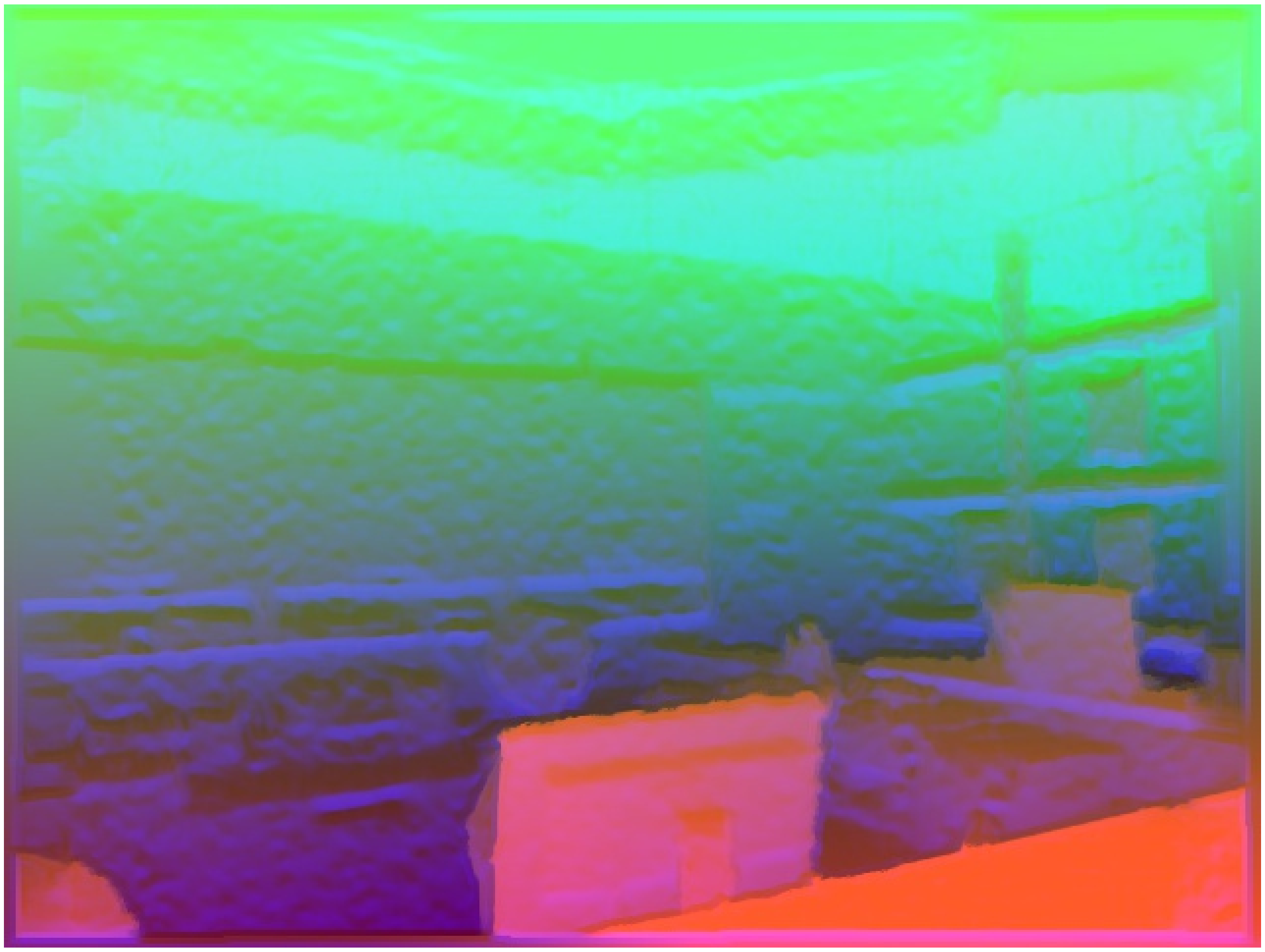


Figure 24: **SA-Gate segmentation Test Example 2**

Figure 25: **SA-Gate segmentation Test Example 3**

The following table lists the SA-Gate models supported by the Vitis AI Library.

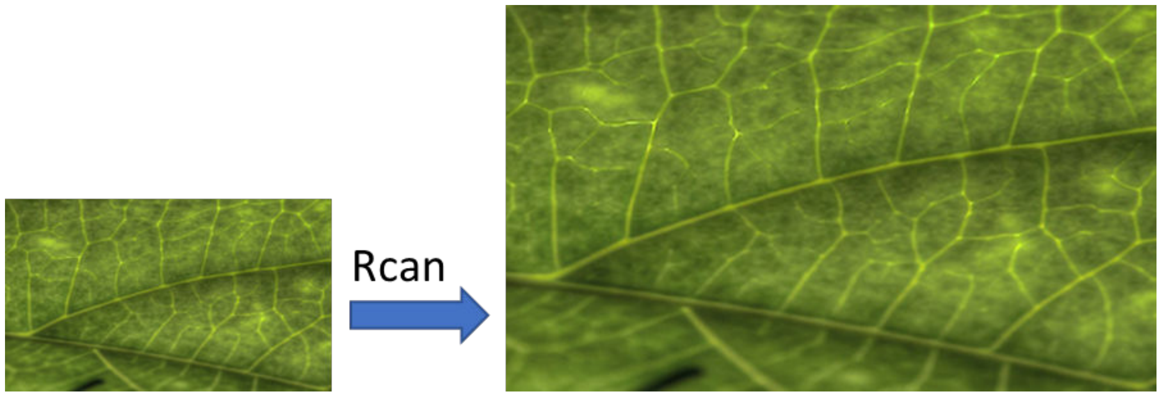
Table 34: **SA-Gate Segmentation Models**

No	Model Name	Framework
1	SA_gate_pt	PyTorch

RCAN Super Resolution

RCAN model is a super-resolution network. The corresponding high-resolution image is reconstructed from the low-resolution image. Based on the original image, the length and width are enlarged by two times. It has important application value in the fields of monitoring equipment, satellite images, and medical imaging. The following images show the result of RCAN. The image is still clear after zooming in.

Figure 26: **Production Recognition Example**



The following table lists the RCAN super-resolution models supported by the Vitis AI Library.

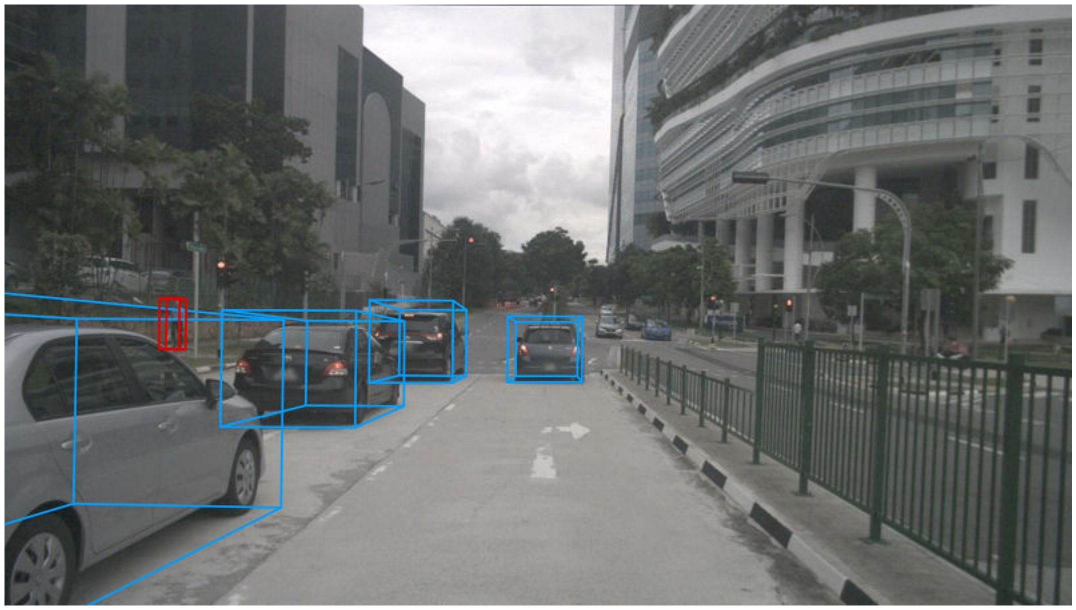
Table 35: **RCAN Super Resolution Models**

No	Model Name	Framework
1	rcan_pruned_tf	TensorFlow
2	ofa-rcan_pt	PyTorch
3	drunet_pt	
4	SESR_S_pt	
5	ofa_rcan_latency_pt	
6	xilinxSR_pt	

PointPainting

For AD/ADAS systems, sensor-fusion algorithms play a significant role in providing high-quality perception and increasing the safety level for driving. PointPainting provides a sensor-fusion framework that takes advantage of 2D semantic segmentation and 3D object detection models. First, a network is applied to the camera images for semantic segmentation. Based on the semantic information and calibration information (on camera and LiDAR), the LiDAR point clouds are projected to the images and fused with the semantic information to get the painted point clouds. Finally, the painted point clouds are consumed by the 3D object detector to achieve better perception.

Figure 27: **PointPainting Example**



The following table lists the PointPainting models supported by the Vitis AI library.

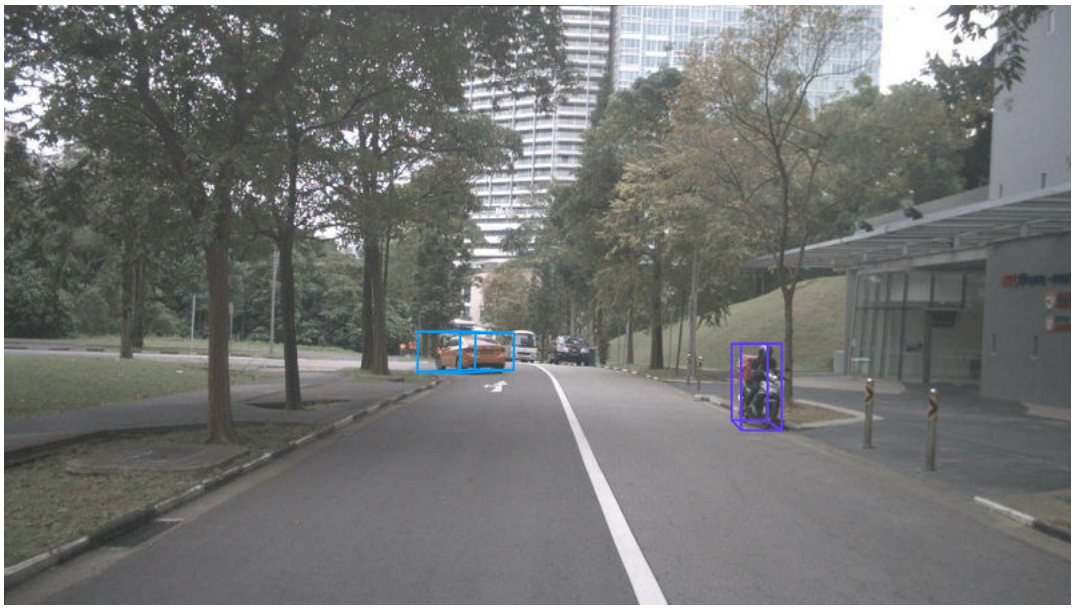
Table 36: **PointPainting Models**

No	Model Name	Framework
1	pointpainting_nuscenes_40000_64_0_pt	PyTorch
2	pointpainting_nuscenes_40000_64_1_pt	PyTorch
3	semanticfpn_nuimage_576_320_pt	PyTorch

PointPillars_nuscenes

PointPillars is an efficient network for real-time 3D object detection on the point cloud. Trained on the nuScenes dataset, this model gives 3D bounding boxes and speed prediction for ten classes (including some kinds of vehicles, pedestrians, barriers, and traffic cones) in the surround-view range. With multisweep point clouds as input, PointPillars can achieve higher accuracy of 3D object detection and speed estimation at the cost of increasing the complexity of the pre-processing part.

Figure 28: **PointPillars_nuscenes Example**



The following table lists the PointPillars_nuscenes models supported by the Vitis AI library.

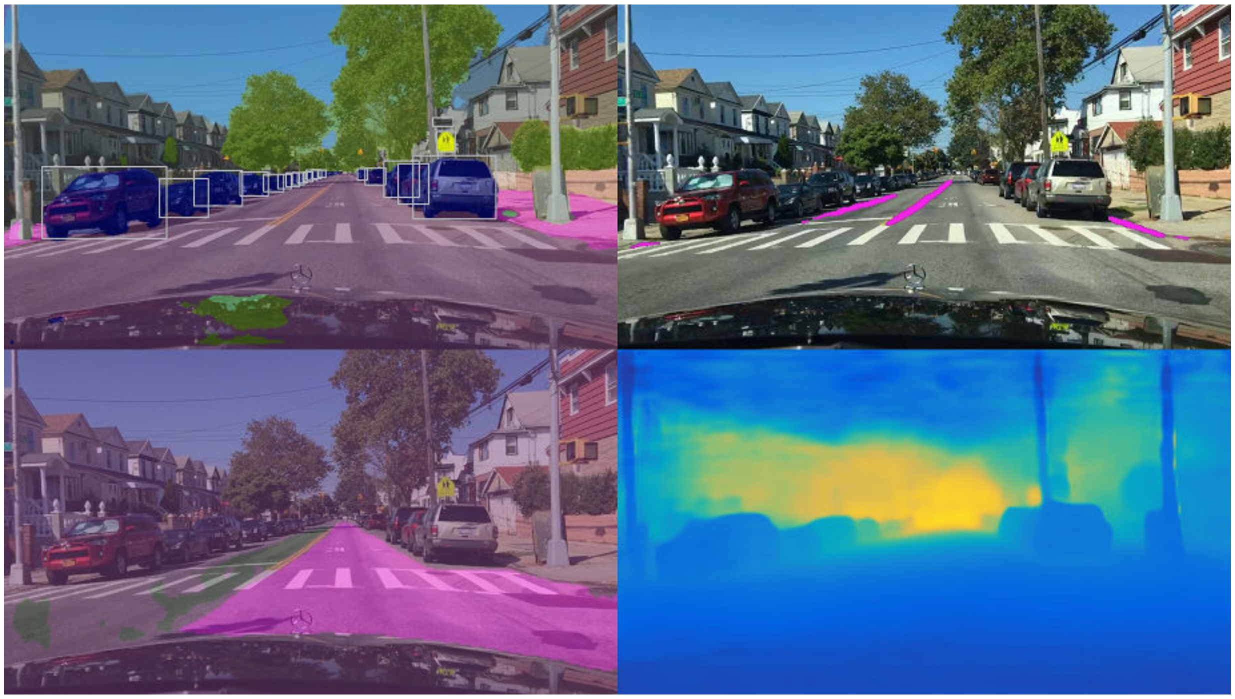
Table 37: **PointPillars_nuscenes Models**

No	Model Name	Framework
1	pointpillars_nuscenes_40000_64_0_pt	PyTorch
2	pointpillars_nuscenes_40000_64_1_pt	PyTorch

MultiTask V3

MultiTask V3 aims to do different tasks in autonomous driving scenarios simultaneously while achieving good performance and efficiency. The tasks include object detection, segmentation, lane detection, drivable area segmentation, and depth estimation, which are important components of the autonomous driving perception module.

Figure 29: **Multi-task V3 Example**



The following table lists the multi-task v3 models supported by the Vitis AI library.

Table 38: **Multi-task V3 Models**

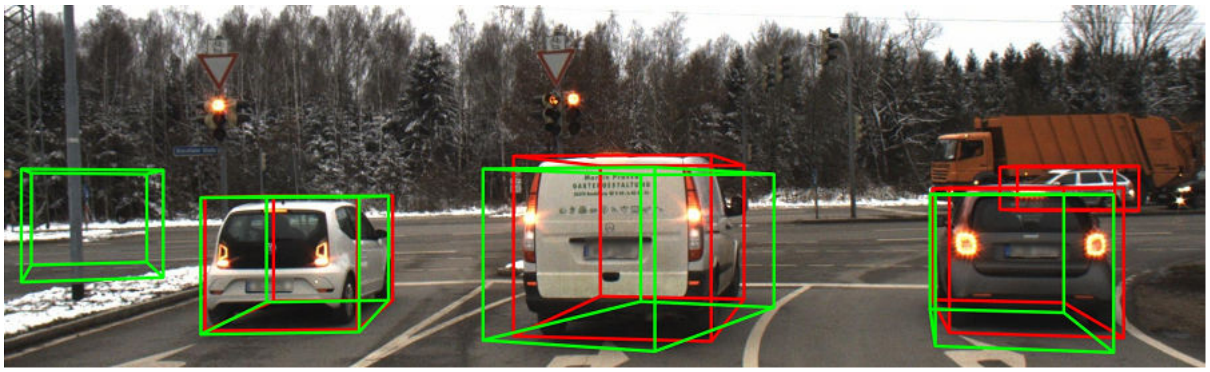
No	Model Name	Framework
1	multi_task_v3_pt	PyTorch

Centerpoint

4D radar is a high-resolution long-range radar sensor that not only detects the distance, relative speed, and azimuth of objects, but also their height above the road level. Unlike LiDAR, it works well in all weather conditions, including fog and heavy rain. A state-of-the-art anchor-free 3D object detector CenterPoint is used. It is trained on the 4D radar data of the open dataset Astyx. Because the annotated samples are limited and the 4D radar point clouds are sparse, the 3D bounding box prediction is naturally not so good. It is observed that although vehicles near ego cars could be correctly detected, there are still some false positive predictions and some objects at longer distances that could not be detected. 4D radar object detection and fusion with camera image could boost the performance by a large margin.

The Centerpoint model is used for 4D radar detection and the following figure shows the result of the Centerpoint model.

Figure 30: **Centerpoint Example**



The following table lists the Centerpoint models supported by the Vitis AI library.

Table 39: **Centerpoint Models**

No	Model Name	Framework
1	centerpoint_0_pt	PyTorch
2	centerpoint_1_pt	PyTorch

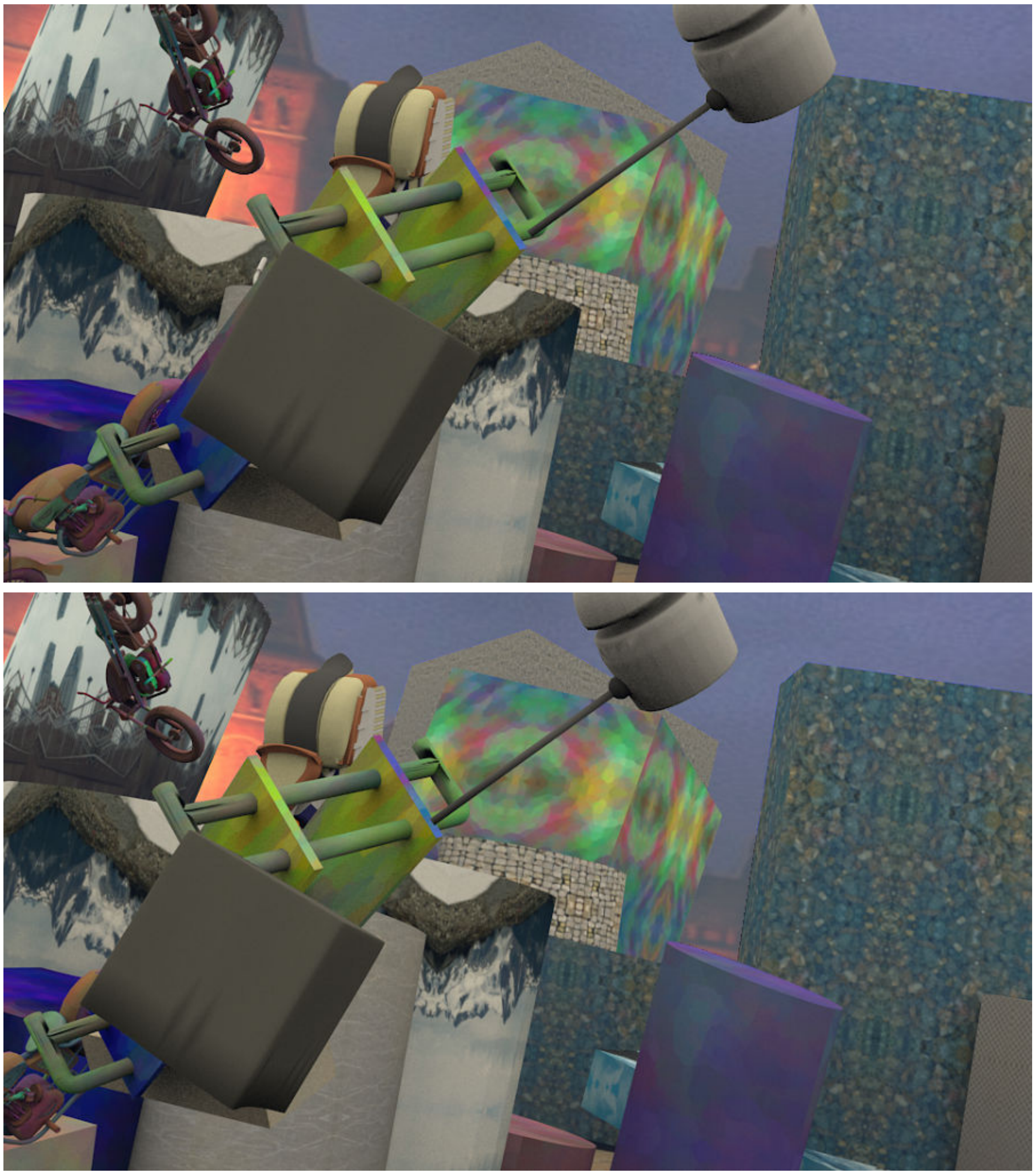
Depth Estimation

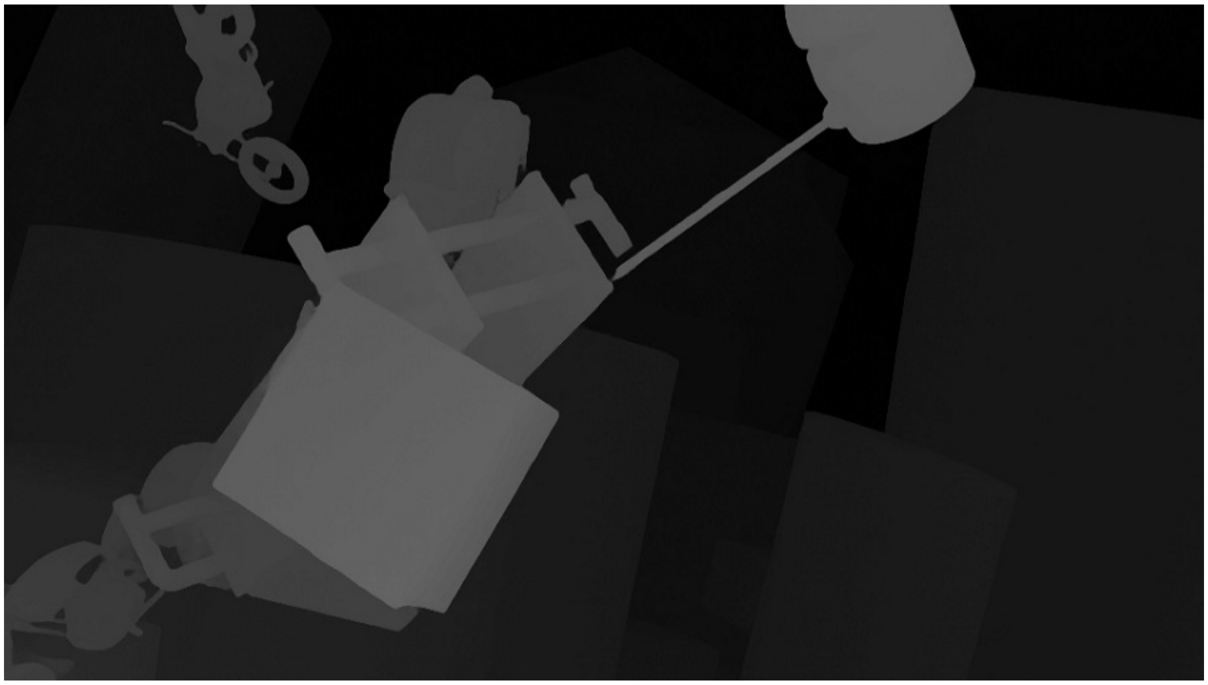
FADNet is a model used for depth estimation. It is a fast and accurate network for disparity estimation. It has three main features:

1. It exploits efficient 2D-based correlation layers with stacked blocks to preserve fast computation.
2. It combines the residual structures to make the deeper model easier to learn.
3. It contains multiscale predictions to exploit a multiscale weight scheduling training technique to improve the accuracy.

The following images show the result of depth estimation. The first image is the left camera image input, the second image is the right camera image input and the third image is the running result of the FADNet model.

Figure 31: **FADNet Depth Estimation Example**





The following table lists the depth estimation models supported by the Vitis AI library.

Table 40: **Depth Estimation Models**

No	Model Name	Framework
1	FADNet_0_pt	PyTorch
2	FADNet_1_pt	
3	FADNet_2_pt	
4	FADNet_v2_0_pt	
5	FADNet_v2_1_pt	
6	FADNet_v2_2_pt	
7	FADNet_v2_pruned_0_pt	
8	FADNet_v2_pruned_1_pt	
9	FADNet_v2_pruned_2_pt	

YOLOX Detection

YOLOX is an anchor-free version of YOLO. It has a simpler design but better performance and aims to bridge the gap between the research and industrial communities. The input size of the model is 640*640, and the output is the score and coordinates of the object.

The following table lists the YOLOX detection models supported by the Vitis AI Library.

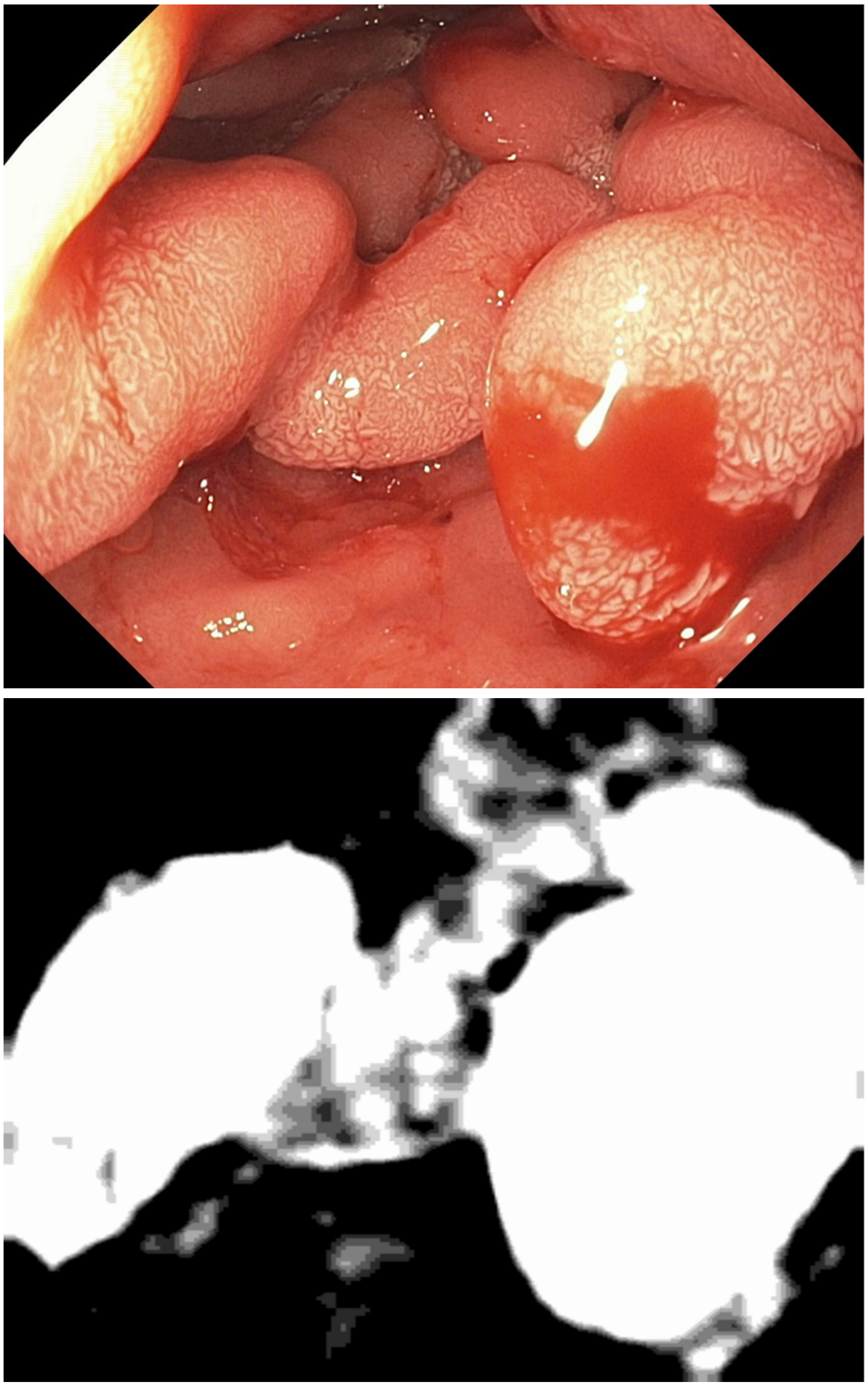
Table 41: **TSD YOLOX Detection Models**

No	Model Name	Framework
1	tsd_yolox_pt	PyTorch
2	yolox_nano_pt	

Polyp Segmentation

HarDNet-MSEG is a new convolution neural network for polyp segmentation. It consists of a backbone and a decoder. The backbone is a low memory traffic CNN called HarDNet68, which has been successfully applied to various CV tasks including image classification, object detection, multi-object tracking, and semantic segmentation. The decoder part is inspired by the Cascaded Partial Decoder, which is known for fast and accurate salient object detection. The following image shows the result of Polyp Segmentation.

Figure 32: **Polyp Segmentation Example**



The following table lists the Polyp Segmentation models supported by the Vitis AI library.

Table 42: **Polyp Segmentation Models**

No	Model Name	Framework
1	HardNet_MSeg_pt	PyTorch

UltraFast Road Line Detection

UltraFast Road Line Detection is a lane detection method that treats the process of lane detection as a row-based selection problem using global features. It can run at high FPS with comparable performance. The input is an image with a lane in it and the output is a structure holding the lane information. The following image shows the result of the UltraFast road line detection.

Figure 33: **UltraFast Road Line Detection Example**

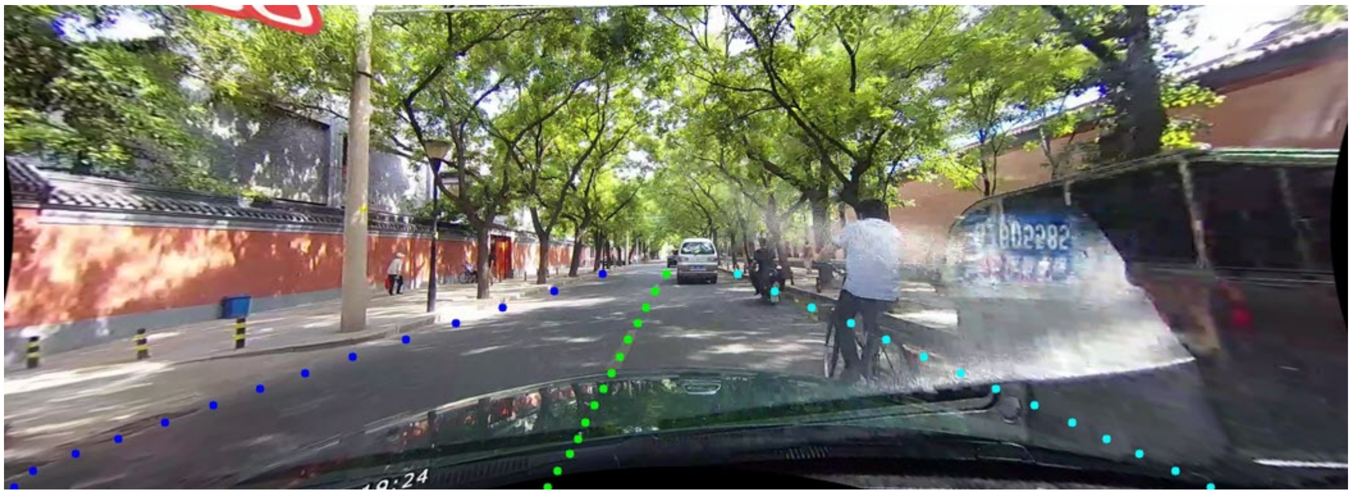


Table 43: **UltraFast Road Line Detection Model**

No	Model Name	Framework
1	ultrafast_pt	PyTorch

SSR

Specular reflections that often appear in the endoscopy images can negatively impact the surgeon's observation and judgment. The SSR model is an end-to-end network that can be used to remove the specular reflections in the endoscopy images thereby improving the image quality.

The following table lists the SSR model supported by the Vitis AI Library.

Table 44: **SSR Model**

No	Model Name	Framework
1	ssr_pt	PyTorch

FairMot

Fairmot is a multi-task model that can detect and get the re-ID features of the detected object at the same time. FairMot detects the person in the picture and provides the features of the detected target. This model can be used for tracking.

The following table lists the FairMot detection models supported by the Vitis AI Library.

Table 45: **FairMot Detection Models**

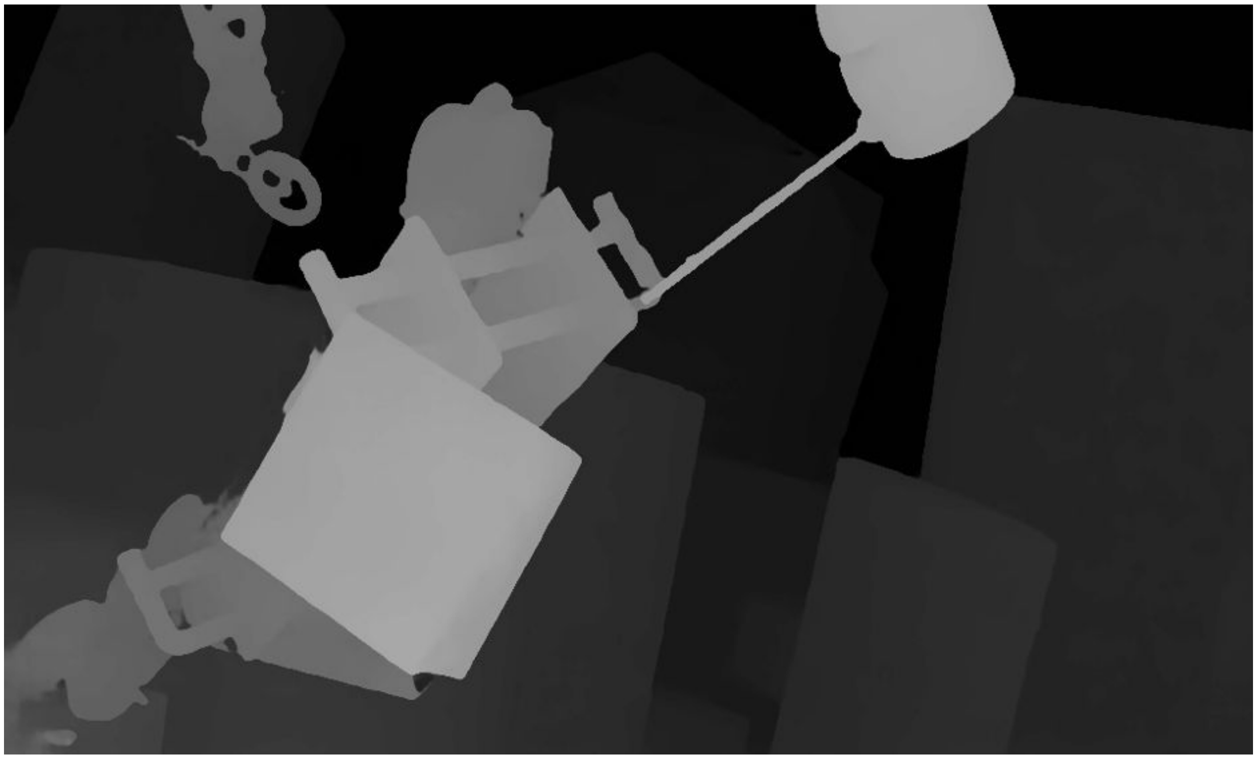
No	Model Name	Framework
1	FairMot_pt	PyTorch

PSMNet

PSMNet is a pyramid stereo matching network that can be used for depth estimation. It consists of two main modules: spatial pyramid pooling and 3D CNN. The spatial pyramid pooling module takes advantage of the capacity of global context information by aggregating context in different scales and locations to form a cost volume. The 3D CNN regularizes cost volume using stacked multiple hourglass networks with intermediate supervision.

The following image shows the result of PSMNet.

Figure 34: **PSMNet Depth Estimation Example**



The following table lists the PSMNet models supported by the Vitis AI Library.

Table 46: **PSMNet Models**

No	Model Name	Framework
1	PSMNet_0_int	PyTorch
2	PSMNet_1_int	PyTorch
3	PSMNet_2_int	PyTorch

C2D2 Coverage Prediction

Colonoscopy Coverage Deficiency via Depth algorithm, or C2D2, is a machine learning-based approach for improving colonoscopy coverage. The C2D2 network is a cascading structure. The inputs are 300 serialized gray images and the output is coverage. The `C2D2_Lite_0_pt` model is responsible for extracting the features of each image and the `C2D2_Lite_1_pt` model predicts a coverage value by inputting the characteristics of 300 pictures.

The following table lists the C2D2 Coverage Prediction models supported by the Vitis AI Library.

Table 47: **C2D2 Model**

No	Model Name	Framework
1	C2D2_Lite_0_pt	PyTorch
2	C2D2_Lite_1_pt	PyTorch

SOLO

Segment objects by locations (SOLO) is a simple and flexible framework applied for accomplishing instance segmentation in digital image processing and computer vision tasks. It is based on the notion of "instance categories" for instance segmentation in which each pixel within an instance of an object is assigned a category based on its location and size.

The following table lists the SOLO model supported by the Vitis AI Library.

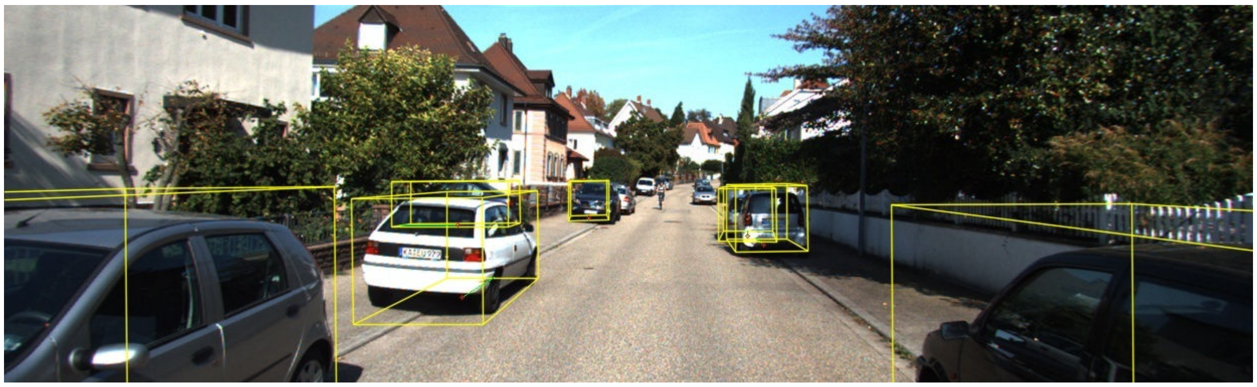
Table 48: **SOLO Models List**

No	Model Name	Framework
1	solo_pt	PyTorch

CLOCs

CLOCs is a novel Camera-LiDAR fusion method for 3D object detection in autonomous driving. Being fed with the predictions from the 2D detection pipeline (with camera image as input) and 3D detection pipeline (with LiDAR point cloud as input) in parallel, a light-weight fusion network is trained to fuse the 2D/3D prediction properly and refine the scores of the 3D detection results. CLOCs decouples the 2D/3D pipelines in the fusion framework, making it convenient to adopt different 2D/3D pipelines to strike a balance between accuracy and efficiency. The following images show the result of CLOCs.

Figure 35: **CLOCs Example**



The following table lists the CLOCs model supported by the Vitis AI Library.

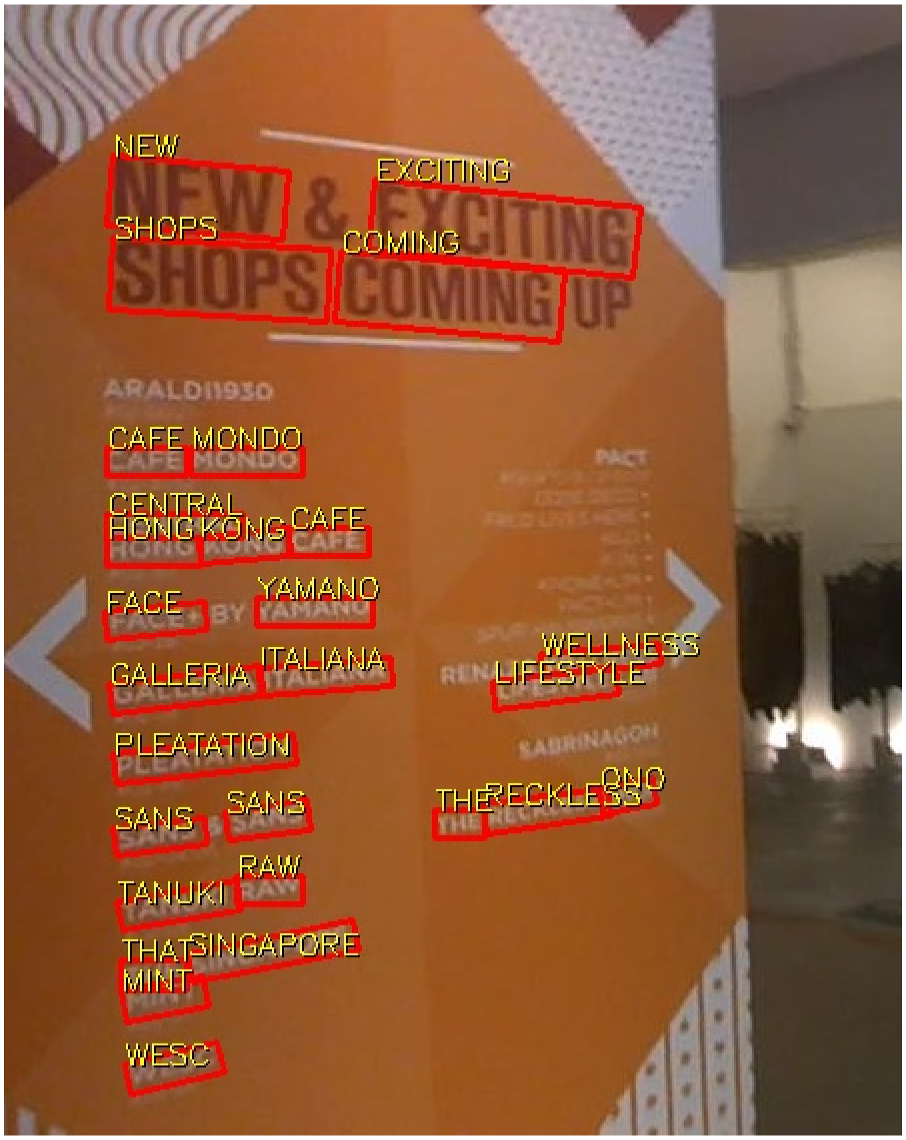
Table 49: **CLOCs Models List**

No	Model Name	Framework
1	clocs_pointpillars_kitti_0_pt	PyTorch
2	clocs_pointpillars_kitti_1_pt	
3	clocs_fusion_cnn_pt	
4	clocs_yolox_pt	

OCR

This network is used for optical character recognition which comprises of text detection and text recognition. The network is composed of a ResNet-FPN feature extractor, a detection branch, and a recognition branch. The model is trained by the ICDAR-2017 dataset. The input is an image containing some character. The output is a structure that includes the words recognized and their position. The following image shows the result of OCR.

Figure 36: **OCR Example**



The following table lists the OCR models supported by the Vitis AI Library.

Table 50: **OCR model**

No	Model Name	Framework
1	ocr_pt	PyTorch

Textmountain Detection

This network is used for multilingual text detection. The network is composed of a ResNet-FPN feature extractor and a detection predictor. The model is trained by ICDAR-2017. The input is an image containing some text. The output is a structure that includes the words detected and their position. The following image shows the result of Textmountain model.

Figure 37: **Textmountain Detection**

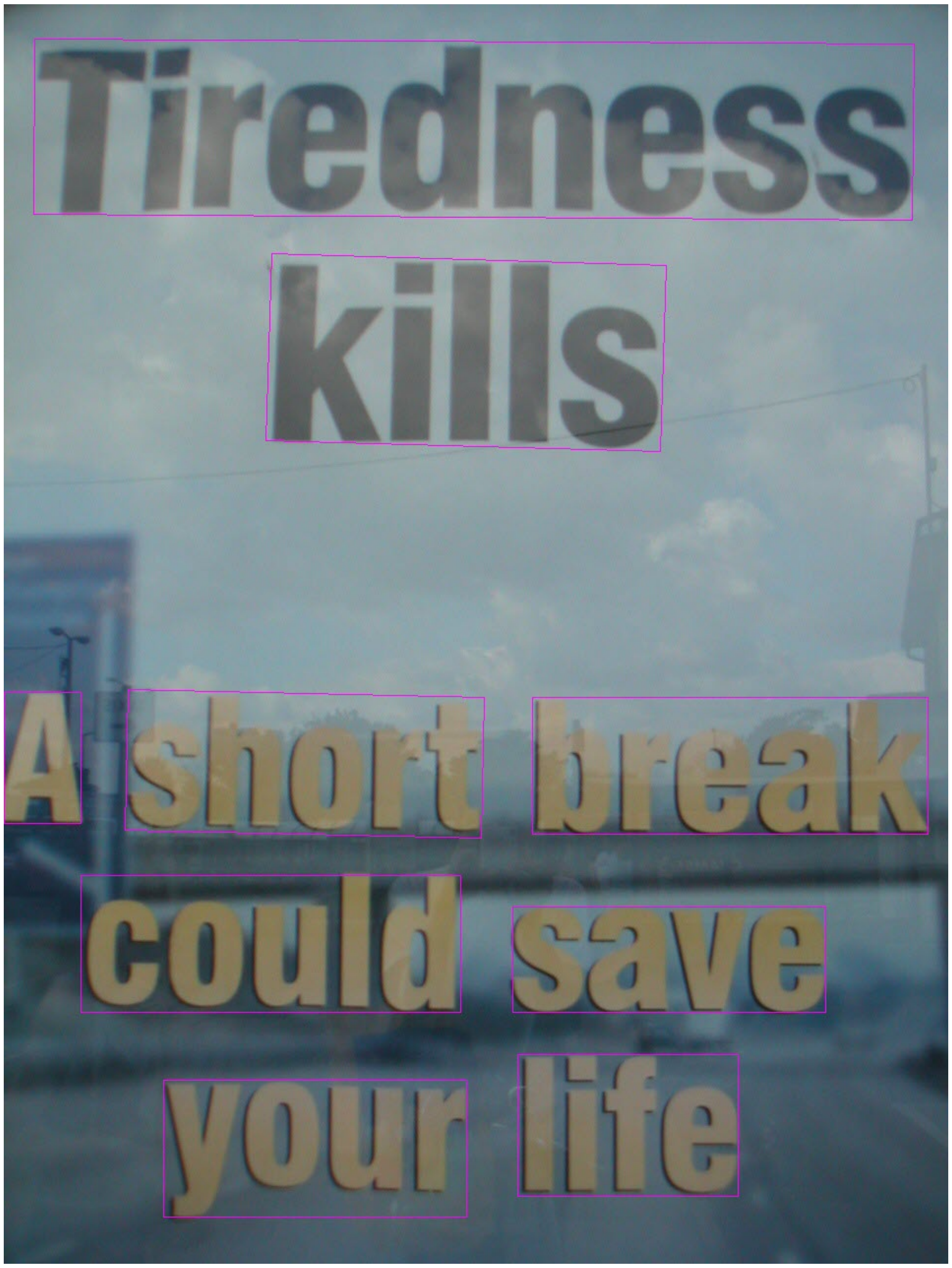


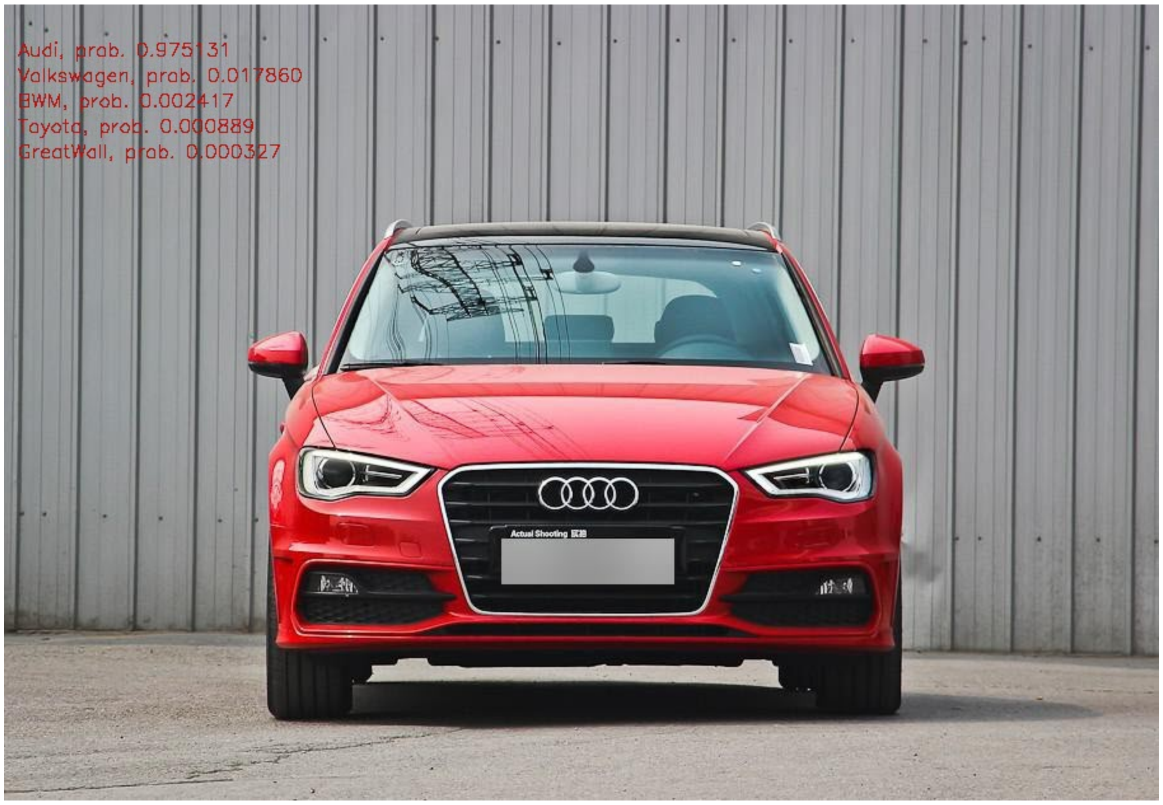
Table 51: **Textmountain Model**

No	Model Name	Framework
1	textmountain_pt	PyTorch

Vehicle Classification

The Vehicle Classification library is used to classify vehicle images (vehicle make or vehicle type). Such neural networks are trained on CompCars and they can identify the objects from 163 classes or 12 classes. The Vitis AI Library integrates networks including `vehicle_make_resnet18_pt` and `vehicle_type_resnet18_pt` into AMD libraries. The input is a picture with an object and the output is the top-K most probable category.

Figure 38: **Vehicle Classification Example**



```
Audi, prob. 0.975131
Volkswagen, prob. 0.017860
BWM, prob. 0.002417
Toyota, prob. 0.000889
GreatWall, prob. 0.000327
```

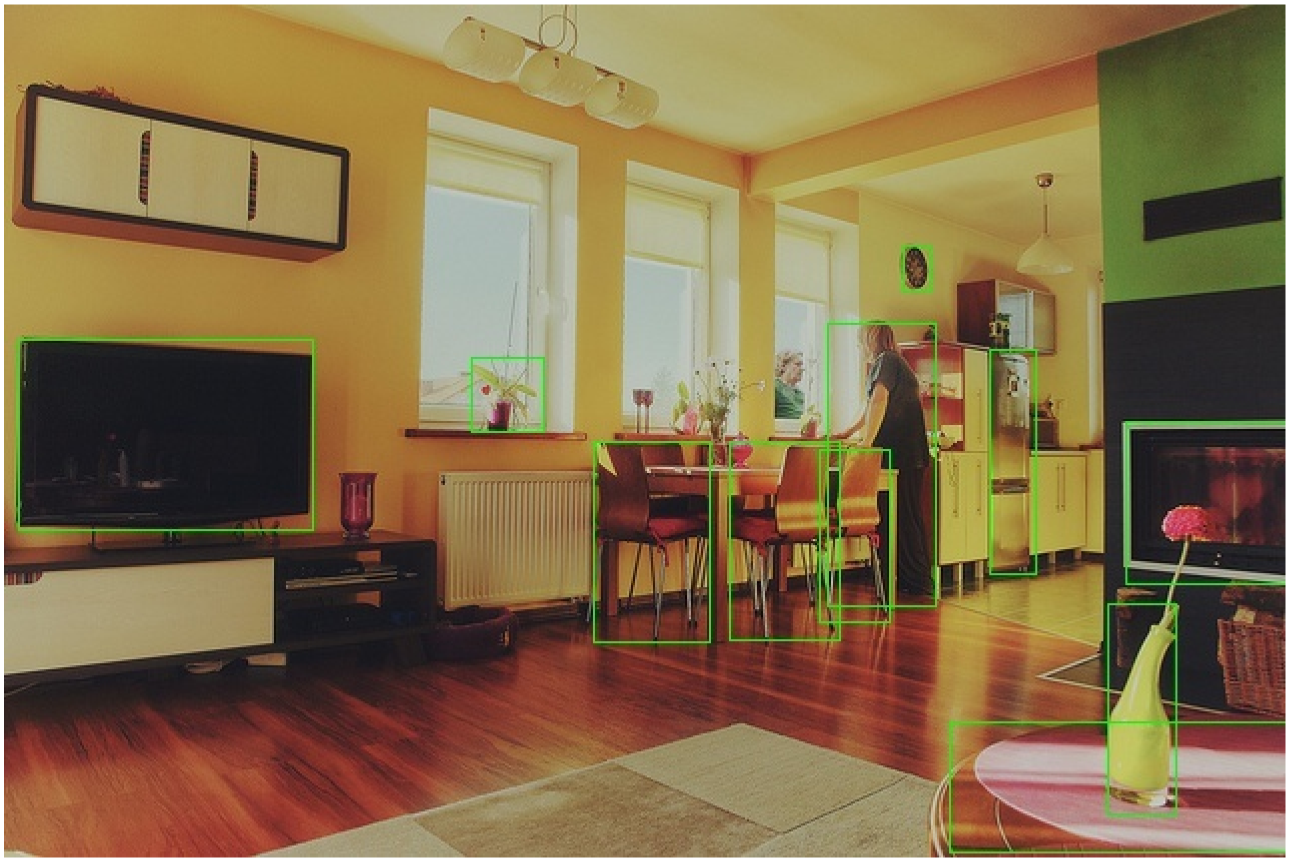
Table 52: **Vehicle Classification Models**

No	Model Name	Framework
1	vehicle_type_resnet18_pt	PyTorch
2	vehicle_make_resnet18_pt	PyTorch

OFA_YOLO Detection

OFA_YOLO is a neural network used to detect objects. The input is a 640x640 picture with one or more objects and the output is a vector of the result struct which is composed of the detected information. The following image shows the result of OFA_YOLO detection.

Figure 39: **OFA_YOLO Detection Example**



The following table lists the OFA_YOLO detection models supported by the Vitis AI Library.

Table 53: **OFA_YOLO Detection Models**

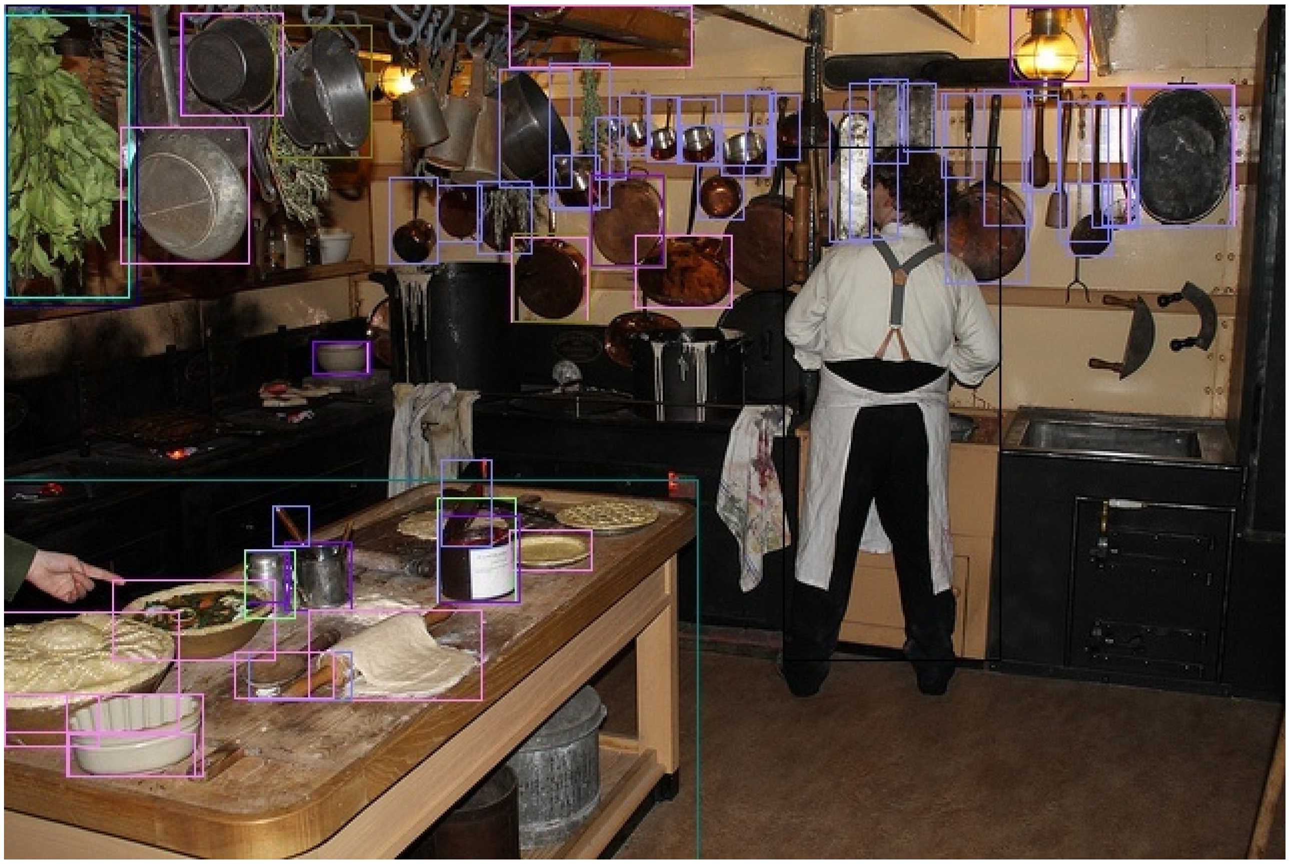
No	Model Name	Framework
1	ofa_yolo_pt	PyTorch
2	ofa_yolo_pruned_0_30_pt	
3	ofa_yolo_pruned_0_50_pt	

EfficientDet_D2

EfficientDet is a one-stage detector built by the Google Brain team. Building upon EfficientNet and incorporating a novel bi-directional feature network and new scaling rules achieve high accuracy with fewer computation operations and is widely used in the production environment.

The following image shows the result of EfficientDet_D2.

Figure 40: **EfficientDet_D2 Example**



The following table lists the EfficientDet_D2 models supported by the Vitis AI Library.

Table 54: **EfficientDet_D2 Models**

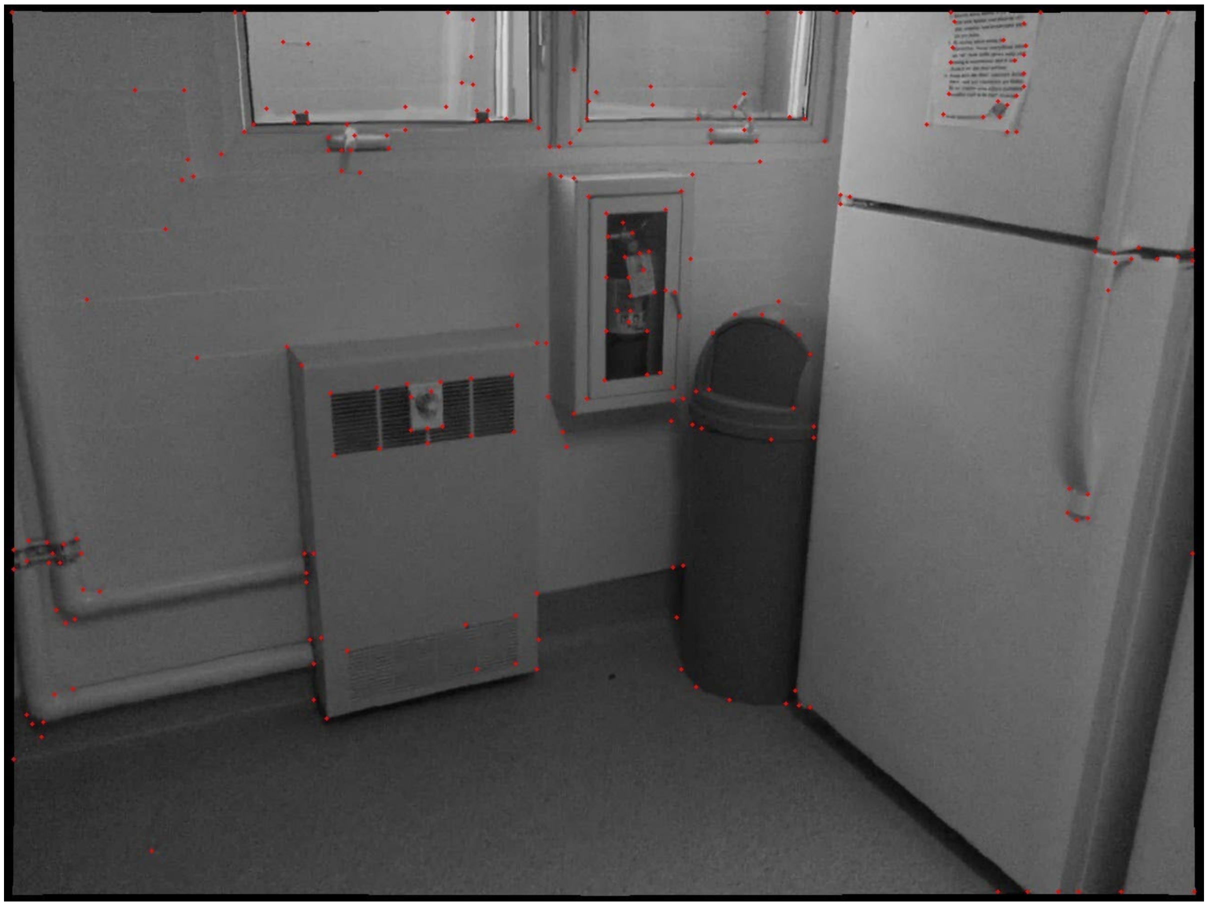
No	Model Name	Framework
1	efficientdet_d2_tf	TensorFlow1

SuperPoint

SuperPoint is a network based on self-supervised interest point detection and description. It's suitable for a large number of multiple-view geometry problems in computer vision. For more details about SuperPoint model, refer to https://arxiv.org/abs/1712.07629.

The following image shows the result of SuperPoint.

Figure 41: **SuperPoint Example**



The following table lists the SuperPoint models supported by the Vitis AI Library.

Table 55: **SuperPoint Models**

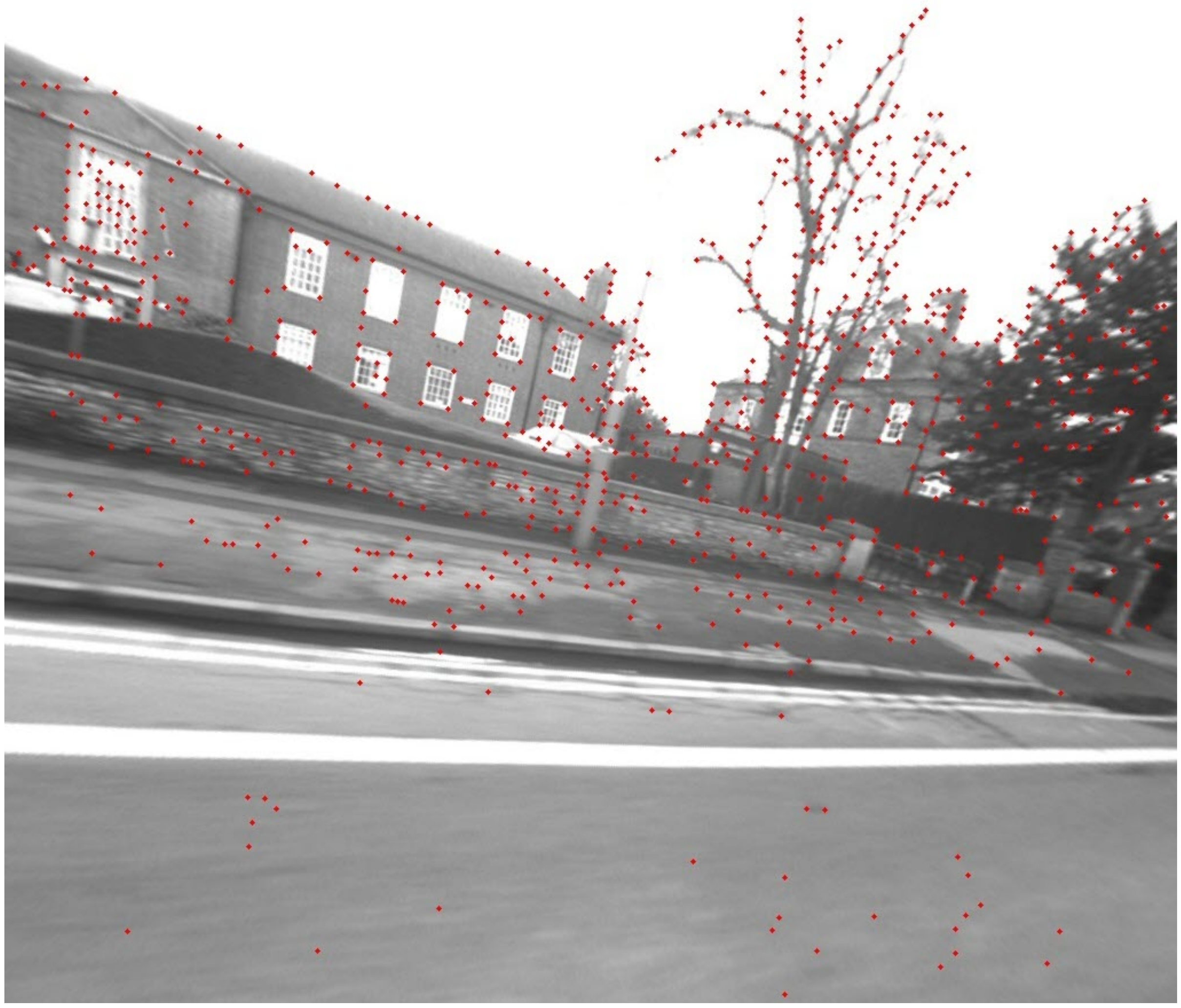
No	Model Name	Framework
1	superpoint_tf	TensorFlow1

HFNet

HF-Net is a hierarchical localization approach based on a monolithic CNN that simultaneously predicts local features and global descriptors for accurate 6-DoF localization. 6-DoF visual localization method is accurate, scalable, and efficient, using HF-Net, a monolithic deep neural network for descriptor extraction. The proposed solution achieves state-of-the-art accuracy on several large-scale public benchmarks while running in real-time. For more details about HF-Net, refer to https://arxiv.org/abs/1812.03506.

The following image shows the result of HFNet.

Figure 42: **HFNet Example**



The following table lists the HFNet models supported by the Vitis AI Library.

Table 56: **HFNet Models**

No	Model Name	Framework
1	hfnet_tf	TensorFlow

Movenet Detection

The library is used to detect posture of the human body. It is represented by a array of 17 joint points. Joint points are arranged in the following order.

```
0: 'nose', 1: 'left_eye', 2: 'right_eye', 3: 'left_ear', 4: 'right_ear',
5: 'left_shoulder', 6: 'right_shoulder', 7: 'left_elbow', 8: 'right_elbow',
9: 'left_wrist', 10 : 'right_wrist', 11: 'left_hip', 12: 'right_hip',
13: 'left_knee', 14: 'right_knee', 15: 'left_ankle', 16: 'right_ankle'
```

This network can only detect for one person and the input of this network is 192x192.

***Note*:** Use a square picture for input. To detect pictures with other size ratios, use a network with the same input size ratio.

The following table lists the Movenet detection models supported by the Vitis AI Library.

Table 57: **Movenet Detection Models**

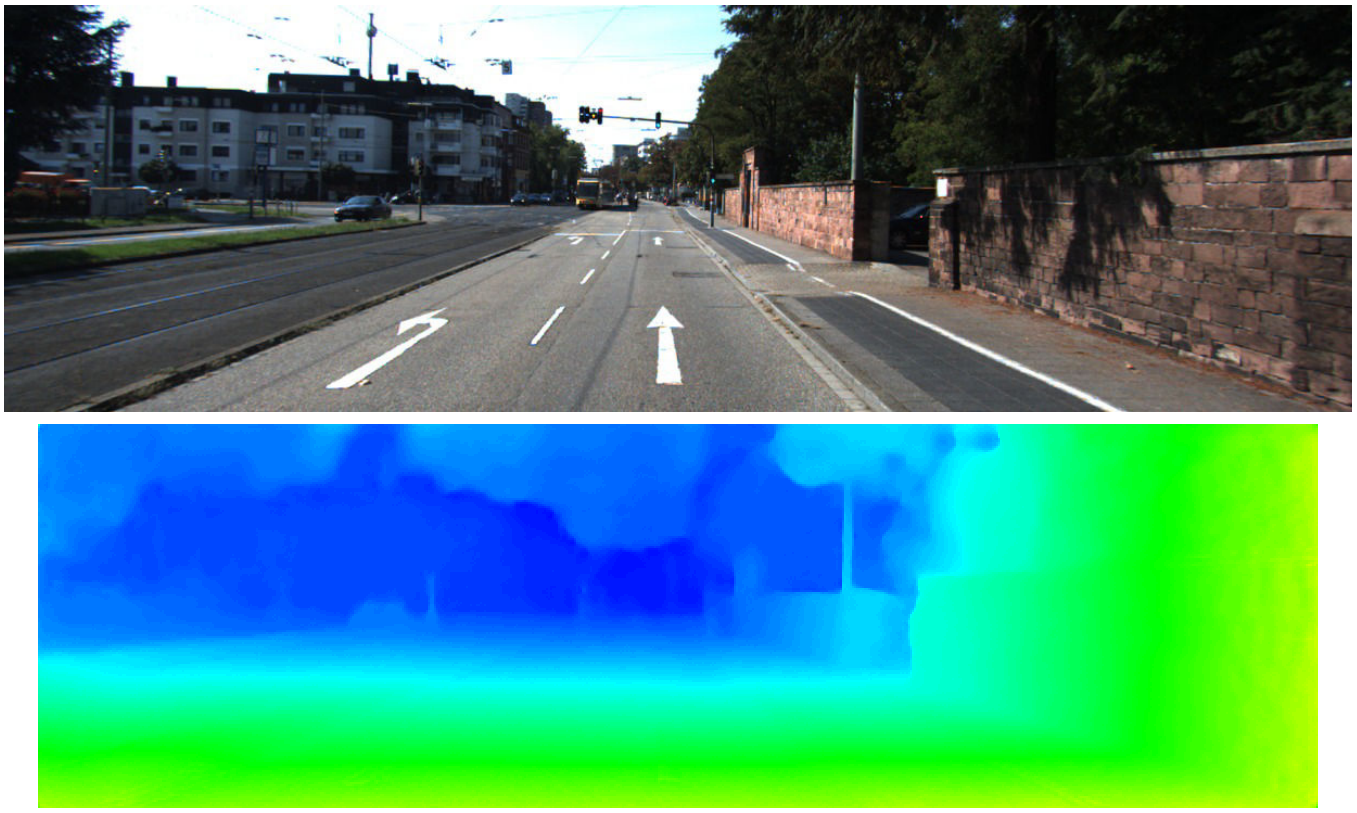
No	Model Name	Framework
1	movenet_ntd_pt	PyTorch

Monodepth2

The monodepth2 model infers a dense depth image from an input image. It has a set of improvements, which result in both quantitatively and qualitatively improved depth maps compared to competing self-supervised methods.

The following image shows the result of Monodepth2.

Figure 43: **Monodepth2 Example**



The following table lists the Monodepth2 model supported by the Vitis AI Library.

Table 58: **Monodepth2 Model**

No	Model Name	Framework
1	monodepth2_pt	PyTorch

YOLOv5 Detection

YOLOv5 is an upgraded version of YOLOv4 and does the same thing as YOLOv4. YOLOv5 uses the PyTorch framework.

The following table lists the YOLOv5 detection models supported by the Vitis AI Library.

Table 59: **YOLOv5 Detection Models**

No	Model Name	Framework
1	yolov5_large_pt	PyTorch
2	yolov5s6_pt	
3	yolov5_nano_pt	
4	yolov5_m_pt	
5	yolov5l_pt	

BEVDet Detection

Autonomous driving considers its surroundings for decision making. This is one of the most complex scenarios in visual perception. BEVDet is a powerful and scalable paradigm for multi-camera 3D object detection that can be deployed on AMD FPGAs. For more details about the BEVDet model, see BEVDet: High-performance Multi-camera 3D Object Detection in Bird-Eye-View.

The following image shows the result of BEVDet Detection.

Figure 44: **BEVDet Detection Example**

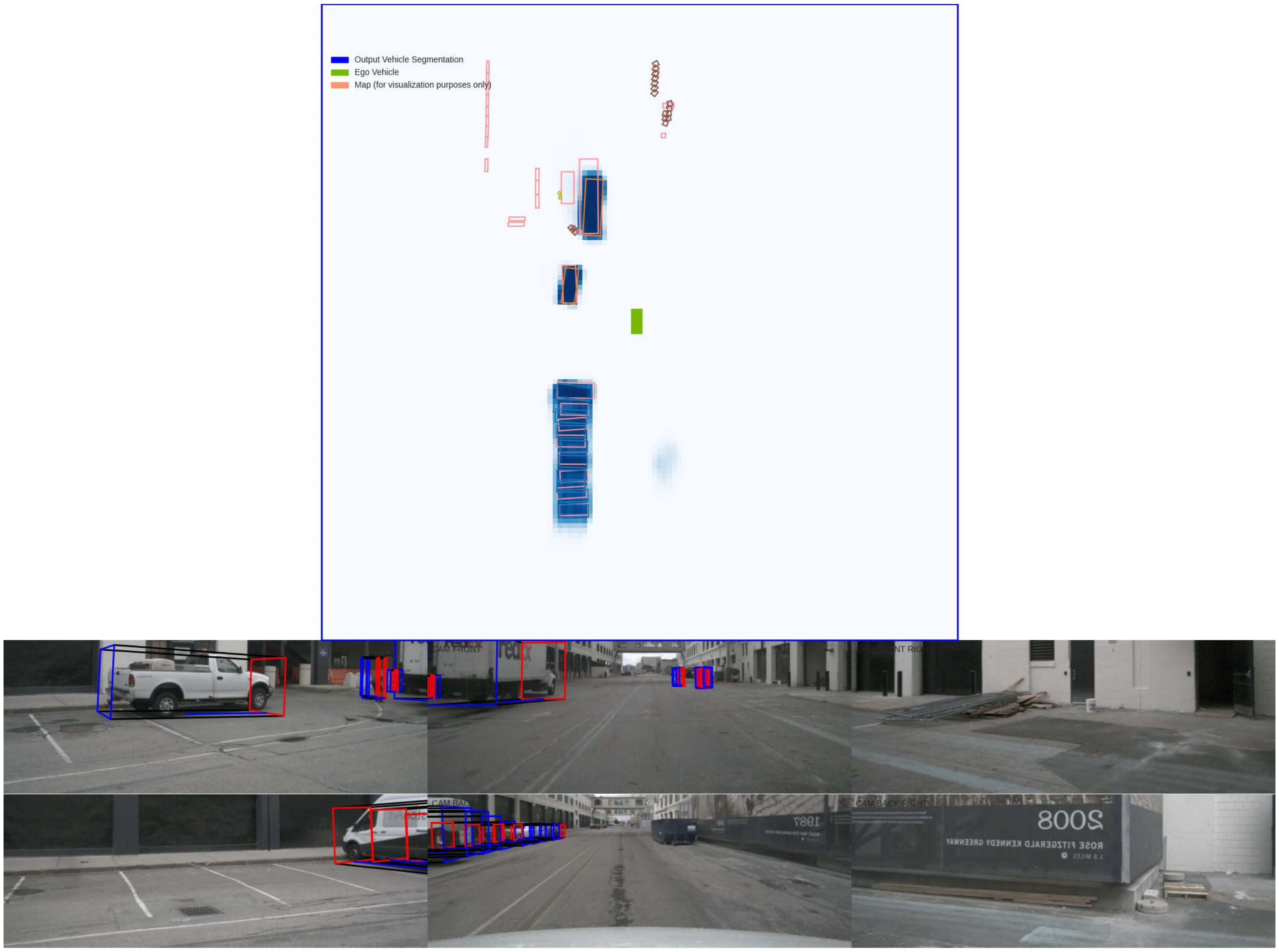

Output Vehicle Segmentation
Ego Vehicle
Map (for visualization purposes only)
CAM FRONT
CAM BACK
CAM BACK RIGHT

Table 60: **BEVDet Model**

No	Model Name	Framework
1	bevdet_pt	PyTorch

cFlownet

cFlownet is a novel conditional generative model that is based on conditional normalizing flow (cFlow). The fundamental idea is to increase the expressivity of the cVAE by introducing a cFlow transformation step after the encoder. This yields improved approximations of the latent posterior distribution, allowing the model to capture richer segmentation variations. For more details about cFlownet model, refer to https://arxiv.org/abs/2006.02683.

The following table lists the cFlownet model supported by the Vitis AI Library.

Table 61: **cFlownet Model**

No	Model Name	Framework
1	cflownet_pt	PyTorch

3D U-Net Segmentation

3D U-Net was introduced shortly after U-Net to process volumetric data which is abundant in medical data analysis. It is based on the previous architecture which consists of an encoder part to analyze the whole image and a decoder part to produce full resolution segmentation. 3D U-Net takes 3D volume as inputs and applies 3D convolution, 3D maxpooling and 3D up-convolutional layers unlike 2D U-Net which has an entirely 2D architecture. For more details about 3D U-Net, refer to 3D U-Net: Learning Dense Volumetric Segmentation from Sparse Annotation.

The following table lists the 3D U-Net model supported by the Vitis AI Library.

Table 62: **3D U-Net Models**

No	Model Name	Framework
1	3D-Unet_pt	PyTorch

YOLOv6 Detection

YOLOv6 has a series of models for various industrial scenarios, including N/T/S/M/L. Architectures vary considering the model size for better accuracy-speed trade-off. Bag-of-Freebies methods, such as self-distillation and additional training epochs, are introduced to further improve the performance. For industrial deployment, QAT with channel-wise distillation and graph optimization is used to pursue extreme performance. For more details, refer to https://arxiv.org/abs/2209.02976.

The following table lists the YOLOv6 detection model supported by the Vitis AI Library.

Table 63: **YOLOv6 Detection Model**

No	Model Name	Framework
1	yolov6m_pt	PyTorch

YOLOv7 Detection

YOLOv7 surpasses all known object detectors in both speed and accuracy. For more details, refer to https://arxiv.org/abs/2207.02696

The following table lists the YOLOv7 detection model supported by the Vitis AI Library.

Table 64: **YOLOv6 Detection Model**

No	Model Name	Framework
1	yolov7_pt	PyTorch

YOLOv8 Detection

YOLOv8 is a cutting-edge, state-of-the-art (SOTA) model that builds upon the success of previous YOLO versions and introduces new features and improvements to further boost performance and flexibility. YOLOv8 is designed to be fast, accurate, and easy to use, making it an excellent choice for a wide range of object detection and tracking, instance segmentation, image classification and pose estimation tasks.

The following table lists the YOLOv8 detection model supported by the Vitis AI Library.

Table 65: **YOLOv8 Detection Model**

No	Model Name	Framework
1	yolov8m_pt	PyTorch

2DUnet

UNet is the most commonly used and simplest segmentation model. It is simple, efficient, easy to understand, easy to build, and can be trained from small datasets. The original intention of UNet is to solve the problem of medical image segmentation. In terms of solving the task of cell-level segmentation, it won multiple firsts in the ISBI cell tracking competition in 2015. After that, UNet has been widely used in various directions of semantic segmentation (such as satellite image segmentation, industrial defect detection, etc.) due to its outstanding segmentation effect.

The following table lists the 2DUnet model supported by the Vitis AI Library.

Table 66: **2DUnet Model**

No	Model Name	Framework
1	unet2d_tf2	Tensorflow2

Model Samples

For models based on VART, the samples are located in the `~/Vitis-AI/examples/vai_library/samples` folder. For models based on the ONNX Runtime, the samples are located in the `~/Vitis-AI/examples/vai_library/samples_onnx` folder. Each sample has the following four types of test samples:

- test_jpeg_[model type]
- test_video_[model type]
- test_performance_[model type]
- test_accuracy_[model type]

Take YOLOv3 as an example.

1. Choose one of the following YOLOv3 models before you run the YOLOv3 detection example:
   - yolov3_bdd
   - yolov3_voc
   - yolov3_voc_tf
2. Ensure that the following test programs exist:
   - test_jpeg_yolov3
   - test_video_yolov3
   - test_performance_yolov3
   - test_accuracy_yolov3_bdd
   - test_accuracy_yolov3_adas_pruned_0_9
   - test_accuracy_yolov3_voc
   - test_accuracy_yolov3_voc_tf

   If the executable program does not exist, cross-compile it on the host, then copy the executable program to the target.
3. To test the image data, execute the following command:

   ```
   #./test_jpeg_yolov3 yolov3_voc_tf sample_yolov3.jpg
   ```

   The result is printed on the terminal. You can also view the output image: `sample_yolov3_result.jpg`.
4. To test the video data, execute the following command:

   ```
   #./test_video_yolov3 yolov3_voc_tf video_input.mp4 -t 8
   ```
5. To test the model performance, execute the following command:

   ```
   #./test_performance_yolov3 yolov3_voc_tf test_performance_yolov3.list -t
   8
   ```

   The result is printed on the terminal.
6. To test the model accuracy, prepare your image dataset, image list file, and the ground truth of the images. Then execute the following command:

   ```
   #./test_accuracy_yolov3_voc_tf [image_list_file] [output_file]
   ```

After the `output_file` is generated, a script file is needed to automatically compare the results. Finally, the accuracy result can be obtained.

Model Accuracy Test

To test the model accuracy on the board, you must take into account the following factors.

- Image dataset
- Model
- Accuracy test program
- Ground truth file of the image dataset
- Accuracy extraction comparison script

ResNet50 Example

Take `resnet50` as an example.

1. Get the image dataset and the ground truth file of the dataset.

   You can get the image dataset information from the Model Zoo content on GitHub. `resnet50` uses the imagenet dataset.

2. Get the model.

   ```
   wget https://www.xilinx.com/bin/public/openDownload?
   filename=resnet_v1_50_tf-vek280-r3.5.0.tar.gz -O resnet_v1_50_tf-vek280-
   r3.5.0.tar.gz
   ```

3. Copy the model to the board.

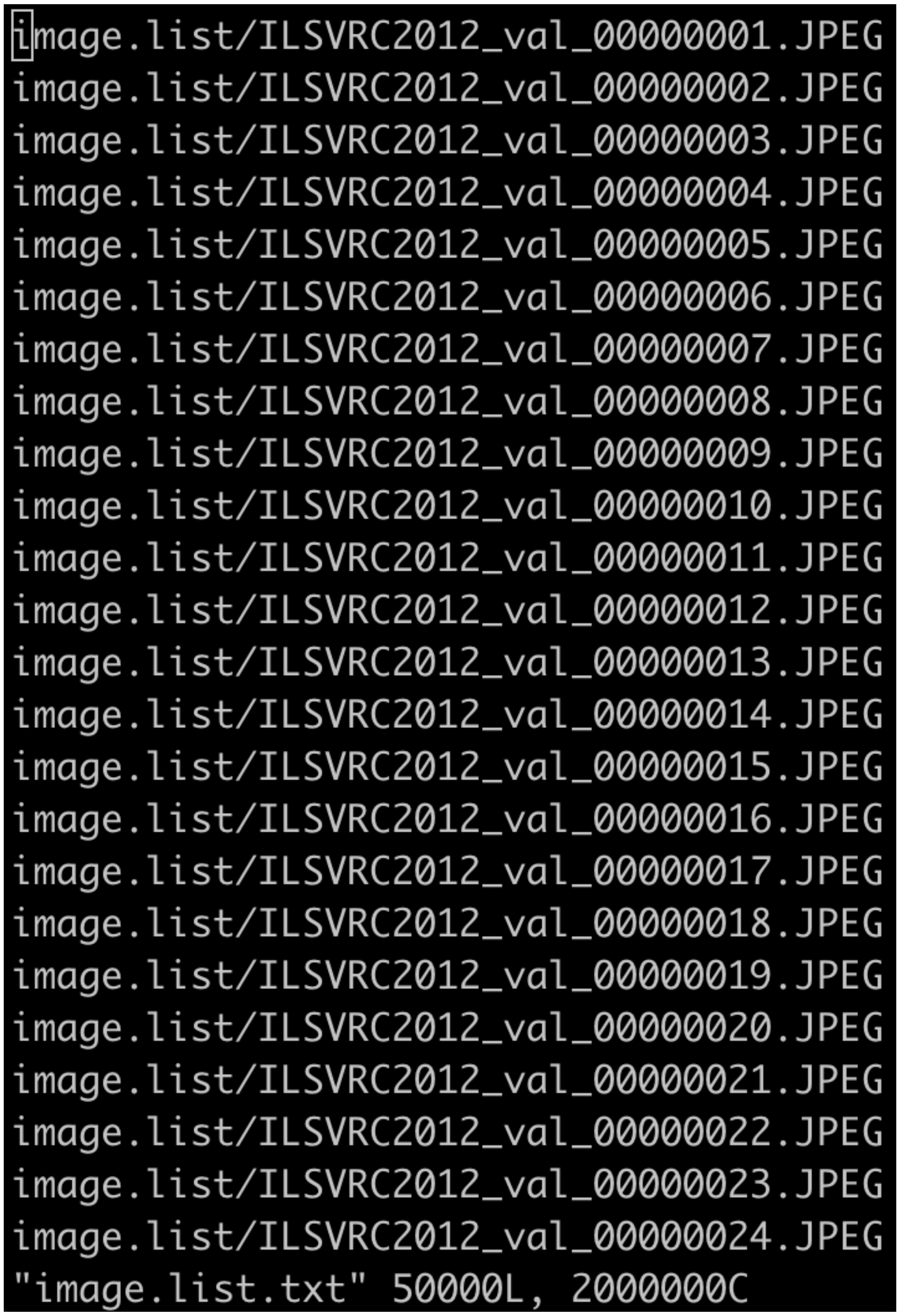
   ```
   scp resnet_v1_50_tf-vek280-r3.5.0.tar.gz root@IP_OF_BOARD:~/
   ```

   It includes `resnet50` and `resnet50_acc` models.

4. Untar it on the board.

   ```
   tar -xzvf resnet_v1_50_tf-vek280-r3.5.0.tar.gz -C /usr/share/vitis-ai-
   library/models
   ```

5. Set the path and image name of the image dataset to a file, such as `image.list.txt`.



```
image.list/ILSVRC2012_val_00000001.JPEG
image.list/ILSVRC2012_val_00000002.JPEG
image.list/ILSVRC2012_val_00000003.JPEG
image.list/ILSVRC2012_val_00000004.JPEG
image.list/ILSVRC2012_val_00000005.JPEG
image.list/ILSVRC2012_val_00000006.JPEG
image.list/ILSVRC2012_val_00000007.JPEG
image.list/ILSVRC2012_val_00000008.JPEG
image.list/ILSVRC2012_val_00000009.JPEG
image.list/ILSVRC2012_val_00000010.JPEG
image.list/ILSVRC2012_val_00000011.JPEG
image.list/ILSVRC2012_val_00000012.JPEG
image.list/ILSVRC2012_val_00000013.JPEG
image.list/ILSVRC2012_val_00000014.JPEG
image.list/ILSVRC2012_val_00000015.JPEG
image.list/ILSVRC2012_val_00000016.JPEG
image.list/ILSVRC2012_val_00000017.JPEG
image.list/ILSVRC2012_val_00000018.JPEG
image.list/ILSVRC2012_val_00000019.JPEG
image.list/ILSVRC2012_val_00000020.JPEG
image.list/ILSVRC2012_val_00000021.JPEG
image.list/ILSVRC2012_val_00000022.JPEG
image.list/ILSVRC2012_val_00000023.JPEG
image.list/ILSVRC2012_val_00000024.JPEG
"image.list.txt" 50000L, 2000000C
```

6. Run the accuracy test program on the board.

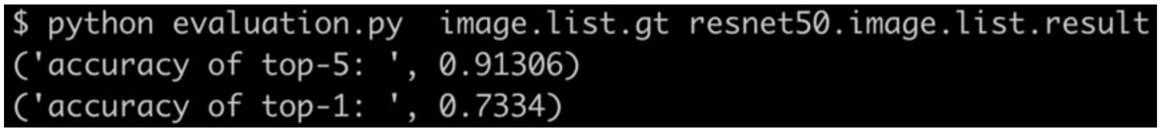
```
cd ~/Vitis-AI/examples/vai_library/samples/classification/
./test_accuracy_classification_mt resnet_v1_50_tf image.list.txt
resnet50.image.list.result
```

Note: The accuracy test loads the `resnet_v1_50_tf_acc` model besides the `resnet_v1_50_tf` model. The only difference between the two models is the model prototxt file.

After the accuracy test program running finished, the result file `resnet50.image.list.result` will be generated.

7. Copy the `resnet50.image.list.result` to the host.
8. Run the corresponding accuracy extraction comparison script to get the final accuracy.

```
python evaluation.py image.list.gt resnet50.image.list.result
```



```
$ python evaluation.py  image.list.gt resnet50.image.list.result
('accuracy of top-5: ', 0.91306)
('accuracy of top-1: ', 0.7334)
```

The following is the code of `evaluation.py`

```
#!/usr/bin/env python


import sys
# argv[1] must groundtruth
readfile=open(sys.argv[1], 'r')
readfile1=open(sys.argv[2], 'r')


dic_val={}


m = 0
for line in readfile:
    temp = line.strip('/').split()
    key = temp[0]
    value = int(temp[1])
    dic_val[key] = value
    m = m + 1


n = 0
for line1 in readfile1:
    temp = line1.strip('/').split()
    if temp[0] in dic_val and int(temp[1]) == dic_val[temp[0]]:
        # print int(temp[1]), dic_val[temp[0]]
        n = n + 1


#print m
#print n
readfile1.close()
readfile2=open(sys.argv[2], 'r')
rate = float(n)/float(m)
print("accuracy of top-5: ", rate)


l = 0
a = 0
for line2 in readfile2:
    a = a + 1
    if (a%5 != 1) : continue
    temp = line2.strip('/').split()
    if temp[0] in dic_val and int(temp[1]) == dic_val[temp[0]]:
        l = l + 1
rate1 = float(l)/float(m)
```

Note: For the accuracy extraction comparison script, see Vitis AI Model Zoo.

Chapter 4

Programming Examples

Application requirements can be categorized into three categories:

- Using the models that are provided by the AMD Vitis™ AI Library to build your own application.
- Using your own custom models which are similar to the models in the Vitis AI Library.
- Using new models that are totally different from the models in the Vitis AI Library.

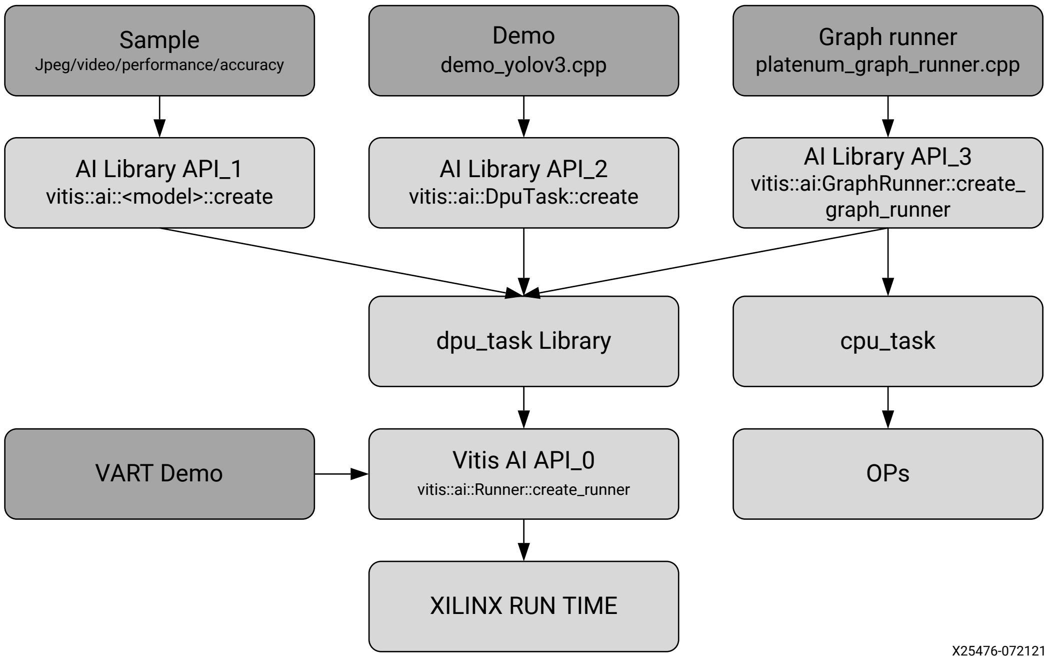
This chapter describes the development steps for the first two cases. For the third case, you can use the Vitis AI Library samples and implementation for reference. This chapter provides information on:

- Customizing preprocessing
- Using the configuration file to specify the preprocessing and postprocessing parameters
- Using the Vitis AI Library's postprocessing library
- Implementing user postprocessing code
- Working with the `xdputil` tool

The following figure shows the relationship between the Vitis AI Library APIs and their corresponding examples. There are four kinds of APIs in this release:

- Vitis AI Library API_0 based on VART
- Vitis AI Library API_1 based on AI Library
- Vitis AI Library API_2 based on DpuTask
- Vitis AI Library API_3 based on Graph_runner

Figure 45: **The Diagram of AI Library API**


Sample
Jpeg/video/performance/accuracy
Demo
demo_yolov3.cpp
Graph runner
platenum_graph_runner.cpp
AI Library API_1
vitis::ai::<model>::create
AI Library API_2
vitis::ai::DpuTask::create
AI Library API_3
vitis::ai:GraphRunner::create_graph_runner
dpu_task Library
cpu_task
VART Demo
Vitis AI API_0
vitis::ai::Runner::create_runner
OPs
XILINX RUN TIME
X25476-072121

Choosing an API for Your Application

Use the following recommendations to choose an API for your application.

- If the model has been split into several subgraphs, API_3 `Graph_runner` is recommended for model deployment.
- If the model has a custom op, API_3 `Graph_runner` is recommended for model deployment.
- If you want to get the best performance and you are a beginner at using AI algorithms, such as model, preprocessing, and postprocessing, API_1 `AI_Library` is recommended.
- If you want to use AMD models to quickly build applications, API_1 `AI_Library` is recommended.
- If you have your own models that are retrained using your own data under the Vitis AI library support network list, API_1 `AI_Library` is recommended.
- If you want to use your custom preprocessing or postprocessing algorithms, API_2 `DpuTask` is recommended.
- If you want to develop and apply AI algorithms on multiple platforms and you are an advanced user of AI algorithms, API_0 `VART` is recommended.

Developing with Vitis AI API_0

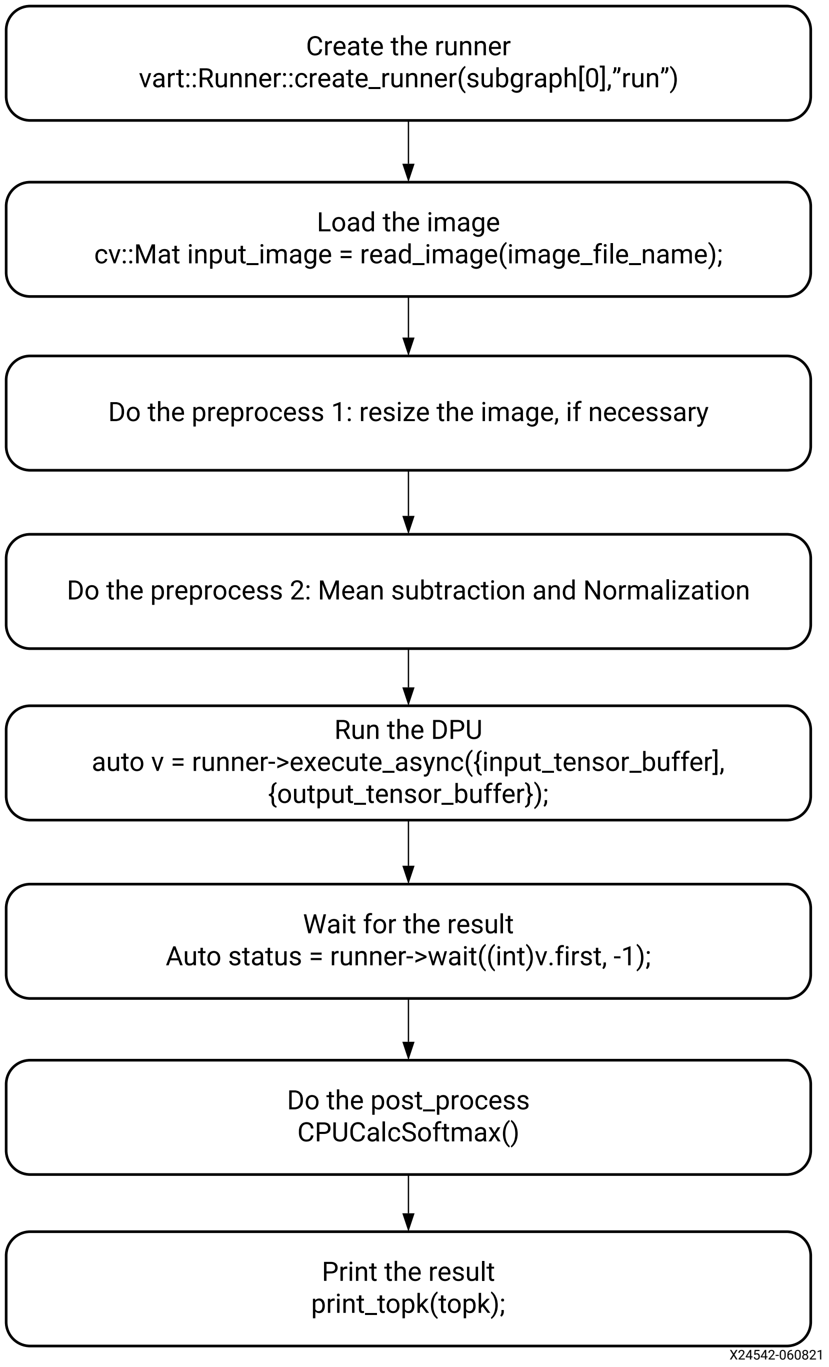
1. Install the cross-compilation system on the host side. Refer to Chapter 2: Installation for more information.
2. Download the model that you want to use, such as `resnet50`, and copy it to the board using the `scp` command.
3. Install the model on the target side.

   ```
   tar -xzvf <model>.tar.gz
   cp -r <model> /usr/share/vitis_ai_library/models
   ```

   By default, the models are located in the `/usr/share/vitis_ai_library/models` directory on the target side.

   ***Note*:** You do not need to install the AMD model package if you want to use your own model.
4. Git clone the corresponding Vitis AI Library from https://github.com/Xilinx/Vitis-AI.
5. Create a folder under your workspace. The following example uses `classification`.

   ```
   mkdir classification
   ```
6. Create the `demo_classification.cpp` source file. The main flow is shown in the following figure. See `Vitis-AI/examples/vai_runtime/resnet50/src/main.cc` for a complete code example.


Create the runner
vart::Runner::create_runner(subgraph[0],"run")
Load the image
cv::Mat input_image = read_image(image_file_name);
Do the preprocess 1: resize the image, if necessary
Do the preprocess 2: Mean subtraction and Normalization
Run the DPU
auto v = runner->execute_async({input_tensor_buffer],
{output_tensor_buffer});
Wait for the result
Auto status = runner->wait((int)v.first, -1);
Do the post_process
CPUCalcSoftmax()
Print the result
print_topk(topk);
X24542-060821

7. Create a `build.sh` file as shown below, or copy one from the Vitis AI Library demo and modify it.

```
#/bin/sh
CXX=${CXX:-g++}
$CXX -O2 -fno-inline -I. -o demo_classification demo_classification.cpp -
lopencv_core -lopencv_video -lopencv_videoio -lopencv_imgproc -
lopencv_imgcodecs -lopencv_highgui -lglog -lxir -lunilog -lpthread -
lvart-runner
```

8. Cross-compile the program.

   ```
   sh -x build.sh
   ```

9. Copy the executable program to the target board using the `scp` command.

   ```
   scp demo_classification root@IP_OF_BOARD:~/
   ```

10. Execute the program on the target board. Before running the program, make sure the target board has the Vitis AI Library installed, and prepare the images you want to test.

    ```
    ./demo_classification /usr/share/vitis_ai_library/models/resnet50/
    resnet50.xmodel resnet50_0 demo_classification.jpg
    ```

Developing with Vitis AI API_1

Vitis AI API_1 is a set of high-level API-based libraries involving different vision tasks including classification, detection, and segmentation. It is optimized for the whole algorithm flow, including pre-process and post-process, and supports the models in the AI Model Zoo.

1. Download the corresponding Docker image from https://github.com/Xilinx/Vitis-AI.
2. Load and run the Docker Image.
3. Create a folder and place the float model within it on the host side, then use the Vitis AI Quantizer to quantize the model. For more details, see the *Vitis AI User Guide* (UG1414).
4. Use the Vitis AI Compiler to compile the model and generate an `xmodel` file, such as `yolov3_custom.xmodel`. For more information, see the *Vitis AI User Guide* (UG1414).
5. Create the `yolov3_custom.prototxt` file, as shown in the following snippet.

   ```
   model {
     name: "yolov3_custom"
     kernel {
        name: "yolov3_custom"
        mean: 0.0
        mean: 0.0
        mean: 0.0
        scale: 0.00390625
        scale: 0.00390625
        scale: 0.00390625
     }
     model_type : YOLOv3
     yolo_v3_param {
       num_classes: 20
       anchorCnt: 3
       layer_name: "59"
       layer_name: "67"
       layer_name: "75"
       conf_threshold: 0.3
       nms_threshold: 0.45
       biases: 10
       biases: 13
       biases: 16
   ```

```
    biases: 30
    biases: 33
    biases: 23
    biases: 30
    biases: 61
    biases: 62
    biases: 45
    biases: 59
    biases: 119
    biases: 116
    biases: 90
    biases: 156
    biases: 198
    biases: 373
    biases: 326
    test_mAP: false
  }
}
```

6. Create the `yolov3_custom` folder. Put the `yolov3_custom.xmodel` and `yolov3_custom.prototxt` files in the `yolov3_custom` folder.
7. Copy the `yolov3_custom` folder to `/usr/share/vitis_ai_library/models` on the target.
8. Use the sample code for `Yolov3` for the application code. There is no need to modify it.

```
int main(int argc, char* argv[]) {
  std::string model = argv[1];
  return vitis::ai::main_for_jpeg_demo(
    argc, argv,
    [model] {
      return vitis::ai::YOLOv3::create(model);
    },
    process_result, 2);
}
```

9. Cross-compile the program and generate an executable file called `test_jpeg_yolov3`.

```
cd Vitis-AI/examples/vai_library/samples/yolov3
sh -x build.sh
```

10. Copy the executable program to the target board using `scp`.

```
scp test_jpeg_yolov3 root@IP_OF_BOARD:~/
```

11. Install the VART runtime on the target. For details, refer to Step 3: Installing the AI Library Package.
12. Execute the program on the target board to get the following results.

```
./test_jpeg_yolov3 yolov3_custom sample.jpg
```

Note: When you develop with Vitis AI API_1, prepare the `model.xmodel` and `model.protxt` files. Run the model with the Vitis AI Library samples. You can find all the Vitis AI Library samples in the `Vitis-AI/examples/vai_library/samples` folder.

Developing with User Model and AI Library API_2

To use your own models, your model framework should be within the scope of the Vitis AI Library. This section shows you how to deploy a retrained YOLOv3 Caffe model to the ZCU102 platform based on the Vitis AI Library.

1. Download the corresponding Docker image from https://github.com/Xilinx/Vitis-AI.
2. Load and run the Docker.
3. Create a folder and place the float model within it on the host side, then use the Vitis AI Quantizer to quantize the model. For more details, see the *Vitis AI User Guide* (UG1414).
4. Use the Vitis AI Compiler to compile the model to an `xmodel` file, such as `yolov3_custom.xmodel`. For more information, see the *Vitis AI User Guide* (UG1414).
5. Create the `yolov3_custom.prototxt`, as shown in the following snippet.

```
model {
  name: "yolov3_custom"
  kernel {
     name: "yolov3_custom"
     mean: 0.0
     mean: 0.0
     mean: 0.0
     scale: 0.00390625
     scale: 0.00390625
     scale: 0.00390625
  }
  model_type : YOLOv3
  yolo_v3_param {
    num_classes: 20
    anchorCnt: 3
    layer_name: "59"
    layer_name: "67"
    layer_name: "75"
    conf_threshold: 0.3
    nms_threshold: 0.45
    biases: 10
    biases: 13
    biases: 16
    biases: 30
    biases: 33
    biases: 23
    biases: 30
    biases: 61
    biases: 62
    biases: 45
    biases: 59
    biases: 119
    biases: 116
    biases: 90
    biases: 156
    biases: 198
    biases: 373
    biases: 326
    test_mAP: false
  }
}
```

Note: The `<model_name>.prototxt` file is effective only when you use the Vitis AI Library API_1.

The parameter of the model needs to be loaded and read manually by the program when using the Vitis AI Library API_2. See the `Vitis-AI/examples/vai_library/samples/dpu_task/yolov3/demo_yolov3.cpp` file for details.

6. Create the `demo_yolov3.cpp` file. See the `Vitis-AI/examples/vai_library/samples/dpu_task/yolov3/demo_yolov3.cpp` file for reference.
7. Create a `build.sh` file as shown below, or copy one from the Vitis AI Library demo and modify it.

```
#/bin/sh
CXX=${CXX:-g++}
$CXX -std=c++17 -O3 -I. -o demo_yolov3 demo_yolov3.cpp -lopencv_core -
lopencv_video -lopencv_videoio -lopencv_imgproc -lopencv_imgcodecs -
lopencv_highgui -lglog -lxnnpp-xnnpp -lvitis_ai_library-model_config -
lprotobuf -lvitis_ai_library-dpu_task
```

8. Exit the Docker tool system and start the Docker runtime system.
9. Cross-compile the program and generate an executable file called `demo_yolov3`.

```
sh -x build.sh
```

10. Create a model folder in the `/usr/share/vitis_ai_library/models` folder on the target side.

```
mkdir yolov3_custom /usr/share/vitis_ai_library/models
```

Note: `/usr/share/vitis_ai_library/models` is the default location for the program to read the model. You can also place the model folder in the same directory as the executable program.

11. Copy the `yolov3_custom.xmodel` and the `yolov3_custom.prototxt` files to the target and put them in the `/usr/share/vitis_ai_library/models/yolov3_custom` location.

```
scp yolov3_custom.xmodel  yolov3_custom.prototxt root@IP_OF_BOARD:/usr/
share/vitis_ai_library/models/yolov3_custom
```

12. Copy the executable program to the target board using `scp`.

```
scp demo_yolov3 root@IP_OF_BOARD:~/
```

13. Execute the program on the target board to get the following results. Before running the program, ensure that the target board has the Vitis AI Library installed, and prepare the images you want to test.

```
./demo_yolov3 yolov3_custom sample.jpg
```

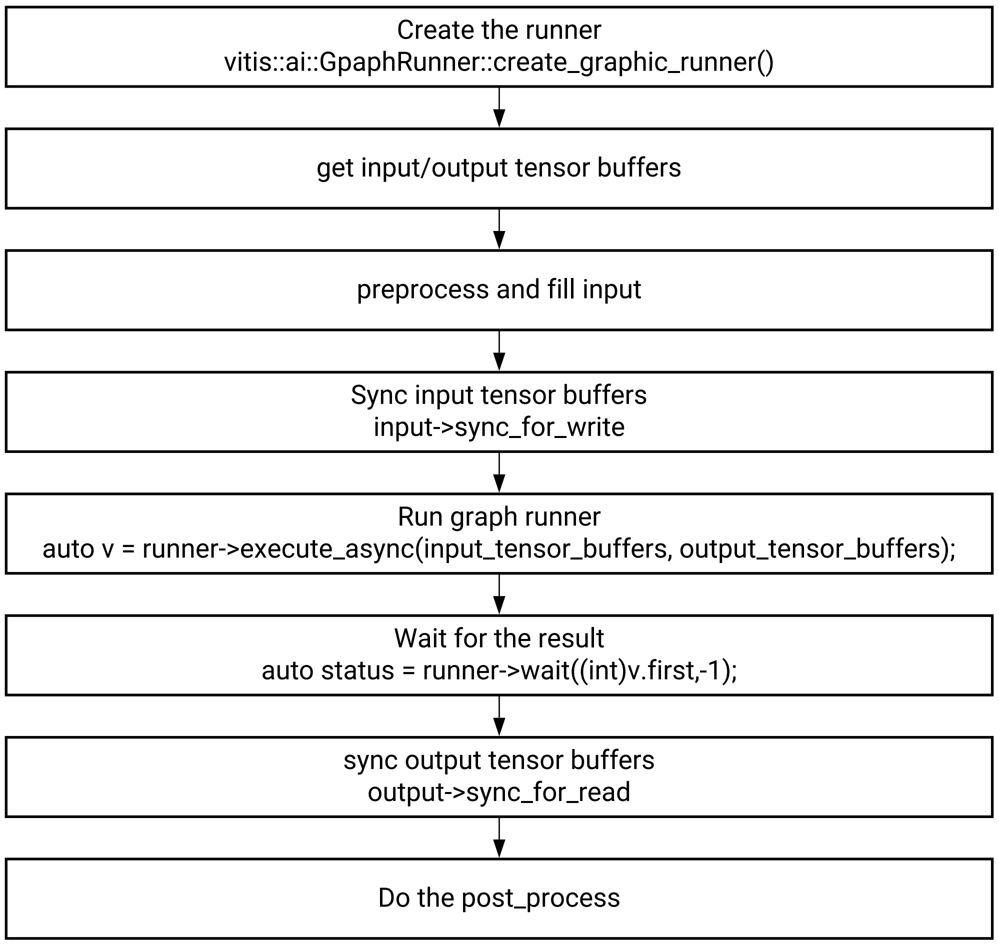
Developing with Vitis AI API_3 (Graph Runner)

If the model is split into multiple subgraphs, you can no longer automatically run it with API_0, API_1, and API_2. You have to deploy the model subgraph by subgraph. Graph runner is the new API for deploying such models. It converts the model into a single graph and makes deployment easier for models with multiple subgraphs. It supports both C++ and Python.

Note: The `Graph Runner` APIs is recommended for custom OP deployment.

1. Git clone the corresponding Vitis AI Library from https://github.com/Xilinx/Vitis-AI.
2. Install the cross-compilation system on the host side. Refer to Chapter 2: Installation for instructions.
3. Check the model to see if it has multiple subgraphs. If yes, you can check whether the operations that are not supported by the DPU are within the scope of supported models. You can find the operations supported by the Vitis AI Library in `Vitis-AI/src/vai_library/cpu_task/ops`.

   Note: If the operation is not in the supported list in `cpu_task`, then you cannot use the `graph_runner` directly. You may encounter an error when you compile the model. You must first solve it, then add the operation under `cpu_task`. You can also refer to the VCK190 Custom Lambda Operator Tutorial to register the custom op and deploy the model with `Graph_runner` APIs.
4. Create the `model_test.cpp` source file. The main flow is shown in the following figure. See the `Vitis-AI/examples/vai_library/samples/graph_runner/platenum_graph_runner/platenum_graph_runner.cpp` file for a complete code example.


Create the runner
vitis::ai::GpaphRunner::create_graphic_runner()
get input/output tensor buffers
preprocess and fill input
Sync input tensor buffers
input->sync_for_write
Run graph runner
auto v = runner->execute_async(input_tensor_buffers, output_tensor_buffers);
Wait for the result
auto status = runner->wait((int)v.first,-1);
sync output tensor buffers
output->sync_for_read
Do the post_process
X25478-062221

5. Create a `build.sh` file as shown below, or copy one from the Vitis AI Library demo and modify it.

```
result=0 && pkg-config --list-all | grep opencv4 && result=1
if [ $result -eq 1 ]; then
    OPENCV_FLAGS=$(pkg-config --cflags --libs-only-L opencv4)
else
    OPENCV_FLAGS=$(pkg-config --cflags --libs-only-L opencv)
fi

CXX=${CXX:-g++}
$CXX -std=c++17 -O2 -I. \
    -o platenum_graph_runner \
    platenum_graph_runner.cpp \
    -lglog \
    -lxir \
    -lvart-runner \
    -lvitis_ai_library-graph_runner \
    ${OPENCV_FLAGS} \
    -lopencv_core \
    -lopencv_imgcodecs \
    -lopencv_imgproc
```

6. Cross-compile the program.

   ```
   sh -x build.sh
   ```

7. Copy the executable program to the target board using the `scp` command.

   ```
   scp test_model root@IP_OF_BOARD:~/
   ```

8. Install the latest VART. For more information, refer to Step 3: Installing the AI Library Package.

9. Execute the program on the target board. Before running the program, ensure that the target board has the Vitis AI Library installed, and prepare the images you want to test.

   ```
   ./model_test <model> <image>
   ```

Developing with the ONNX Runtime

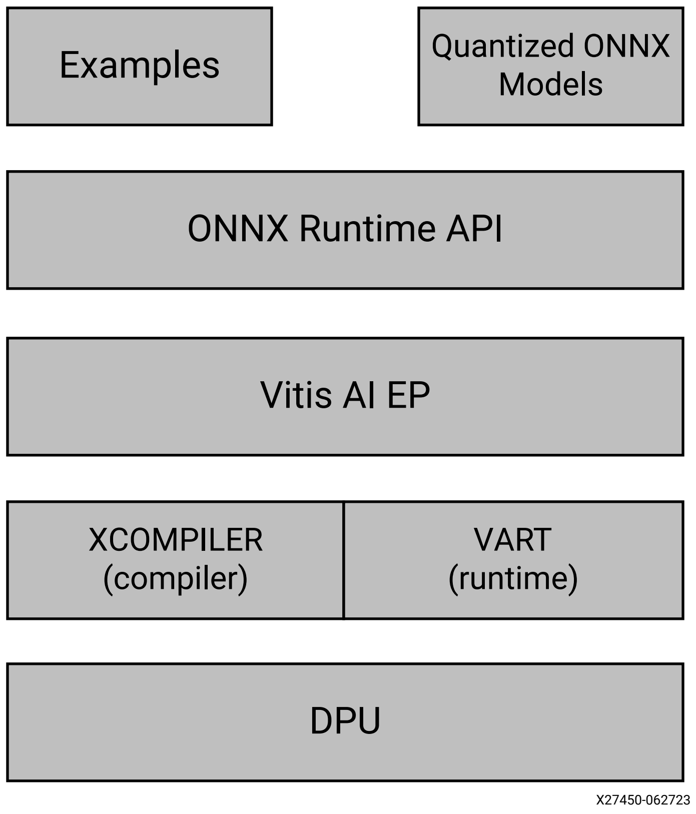
Overview

This section describes how to deploy the quantized ONNX model on the Edge board.

In Vitis AI 3.5, the ONNX Runtime Vitis AI Execution Provider (Vitis AI EP) is provided to hardware-accelerated AI inference with the DPU . It allows you to directly run the quantized ONNX model on the target board. The current Vitis AI EP inside the ONNX Runtime enables acceleration of the neural network model inference using embedded devices such as Zynq UltraScale+ MPSoCs, Versal devices, Versal AI Edge devices, and Kria cards.

The Vitis AI ONNX Runtime Engine (VOE) is the implementation library of Vitis AI EP.

Figure 46: **ONNX Runtime Overview**



Features

The ONNX Runtime has the following features:

- Supports ONNX Opset version 18, ONNX Runtime 1.16.0 and ONNX version 1.13
- Supports C++ and Python APIs (Python version 3)
- In addition to the Vitis AI EP, you can incorporate other execution providers such as ACL EP to accelerate the inference with AMD DPU
- Supports computation on the Arm® 64 Cortex-A72 cores and the supported target is VEK280 in Vitis AI 3.5

The ONNX Runtime offers the following advantages:

- **Versatility:** You could deploy subgraphs on the AMD DPU while using other execution providers such as the Arm NN and Arm ACL for additional operators. This flexibility enables the deployment of models that may not be directly supported on targets boards.
- **Improved performance:** By leveraging specialized execution providers such as the AMD DPU for certain operations and using other providers for the remaining operators, you can achieve optimized performance for their models.

- **Expanded model support:** Enhancing ONNX Runtime enables the deployment of models with operators that are not natively supported by the DPU. By incorporating additional execution providers, you can execute a wide range of models, including those from the ONNX model zoo.

Runtime Options

Vitis AI ONNX Runtime integrates a compiler that compiles the model graph and weights as a micro-coded executable. This executable is deployed on the target accelerator.

The model is compiled when the ONNX Runtime session is started, and compilation must complete prior to the first inference pass. The length of time required for compilation varies, but may take a few minutes to complete. Once the model has been compiled, the model executable is cached and for subsequent inference runs, the cached executable model can optionally be used (details below).

Several runtime variables can be set to configure the inference session as listed in the table below. The `config_file` variable is not optional and must be set to point to the location of the configuration file. The `cacheDir` and `cacheKey` variables are optional.

Table 67: **Runtime Variables**

Runtime Variable	Default Value	Details
config_file	""	Required. The configuration file path, the configuration file vaip_config.json is contained in vitis_ai_2023.1-r3.5.0.tar.gz
cacheDir	/tmp/{user}/vaip/.cache/	Optional. Cache directory
cacheKey	{onnx_model_md5}	Optional. Cache key used to distinguish between different models.

The final cache directory is `{cacheDir}/{cacheKey}`. In addition, environment variables can be set to customize the Vitis AI Execution provider.

Table 68: **Environment Variables**

Environment Variable	Default Value	Details
XLNX_ENABLE_CACHE	1	Whether to use cache, if it is 0, it will ignore the cached executable and the model will be recompiled.
XLNX_CACHE_DIR	/tmp/$USER/vaip/.cache/{onnx_model_md5}	Optional. Configure cache path.

Installing and Deploying

More than 10 deployment examples that are based on the ONNX Runtime are provided in Vitis AI 3.5. You can find the examples in the samples_onnx folder. To deploy the ONNX model using Vitis AI, follow these steps:

1. Git clone the corresponding Vitis AI Library from https://github.com/Xilinx/Vitis-AI.
2. Install the cross-compilation system on the host side. Refer to Chapter 2: Installation for instructions.
3. Prepare the quantized model in ONNX format. Use the Vitis AI Quantizer to quantize the model and output the quantized model in the ONNX format.
4. Download the ONNX runtime package vitis_ai_2023.1-r3.5.0.tar.gz and install it on the target board.

   ```
   tar -xzvf vitis_ai_2023.1-r3.5.0.tar.gz -C /
   ```

   Then, download the voe-0.1.0-py3-none-any.whl and onnxruntime_vitisai-1.16.0-py3-none-any.whl. Make sure the device is online and install them online.

   ```
   pip3 install voe*.whl
   pip3 install onnxruntime_vitisai*.whl
   ```

5. In Vitis AI 3.5, both ONNX Runtime `C++` API and `Python` API are supported. For the details of ONNX Runtime API, refer to ONNX Runtime API docs. The following shows the ONNX model deployment code snippet based on the C++ API.

   ```
   // ...
   #include <experimental_onnxruntime_cxx_api.h>
   // include user header files
   // ...

   auto onnx_model_path = "resnet50_pt.onnx"
   Ort::Env env(ORT_LOGGING_LEVEL_WARNING, "resnet50_pt");
   auto session_options = Ort::SessionOptions();

   auto options = std::unorderd_map<std::string,std::string>({});
   options["config_file"] = "/etc/vaip_config.json";
   // optional, eg: cache path : /tmp/my_cache/abcdefg // Replace abcdefg
   with your model name, eg. onnx_model_md5
   options["cacheDir"] = "/tmp/my_cache";
   options["cacheKey"] = "abcdefg"; // Replace abcdefg with your model
   name, eg. onnx_model_md5

   // Create an inference session using the Vitis AI execution provider
   session_options.AppendExecutionProvider("VitisAI", options);

   auto session = Ort::Experimental::Session(env, model_name,
   session_options);

   auto input_shapes = session.GetInputShapes();
   // preprocess input data
   // ...

   // Create input tensors and populate input data
   std::vector<Ort::Value> input_tensors;
   input_tensors.push_back(Ort::Experimental::Value::CreateTensor<float>(
   input_data.data(), input_data.size(), input_shapes[0]));

   auto output_tensors = session.Run(session.GetInputNames(), input_tensors,
   session.GetOutputNames());
   // postprocess output data
   // ...
   ```

To leverage the Python APIs, use the following example as a reference:

```
import onnxruntime

# Add other imports
# ...

# Load inputs and do preprocessing
# ...

# Create an inference session using the Vitis-AI execution provider

session = onnxruntime.InferenceSession(
'[model_file].onnx',
providers=["VitisAIExecutionProvider"],
provider_options=[{"config_file":"/etc/vaip_config.json"}])

input_shape = session.get_inputs()[0].shape
input_name = session.get_inputs()[0].name

# Load inputs and do preprocessing by input_shape
input_data = [...]
result = session.run([], {input_name: input_data})
```

6. Create a `build.sh` file as shown below, or copy one from the Vitis AI Library ONNX examples and modify it.

```
result=0 && pkg-config --list-all | grep opencv4 && result=1
if [ $result -eq 1 ]; then
    OPENCV_FLAGS=$(pkg-config --cflags --libs-only-L opencv4)
else
    OPENCV_FLAGS=$(pkg-config --cflags --libs-only-L opencv)
fi

lib_x=" -lglog -lunilog -lvitis_ai_library-xnnpp -lvitis_ai_library-
model_config -lprotobuf -lxrt_core -lvart-xrt-device-handle -lvaip-core -
lxcompiler-core -labsl_city -labsl_low_level_hash -lvart-dpu-controller -
lxir -lvart-util -ltarget-factory -ljson-c"
lib_onnx=" -lonnxruntime"
lib_opencv=" -lopencv_videoio -lopencv_imgcodecs -lopencv_highgui -
lopencv_imgproc -lopencv_core "
inc_x=" -I=/usr/include/onnxruntime -I=/install/Release/include/
onnxruntime -I=/install/Release/include -I=/usr/include/xrt "
link_x=" -L=/install/Release/lib"

name=$(basename $PWD)

CXX=${CXX:-g++}
$CXX -O2 -fno-inline -I. \
 ${inc_x} \
 ${link_x} \
 -o ${name}_onnx -std=c++17 \
 $PWD/${name}_onnx.cpp \
 ${OPENCV_FLAGS} \
 ${lib_opencv} \
 ${lib_x} \
 ${lib_onnx}
```

7. Cross-compile the program.

```
sh -x build.sh
```

8. Copy the executable program and the quantized ONNX model to the target board using the `scp` command.

9. Execute the program on the target board. Before running the program, ensure that the target board has the Vitis AI Library installed, and prepare the images you want to test.

```
./resnet50_onnx <Onnx model> <image>
```

Note: For the ONNX model deployment, the input model is the quantized ONNX model. If the environmental variable `WITH_XCOMPILER` is on, it will do the model compiling online first when you run the program. It can take some time to compile the model.

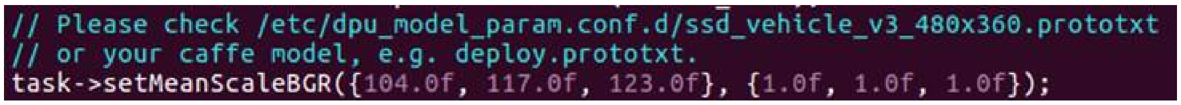
Customizing Pre-Processing

Before convolution neural network processing, image data generally needs to be pre-processed. The basics of some pre-processing techniques that can be applied to any kind of data are as follows:

- Mean subtraction
- Normalization
- PCA and Whitening

Call the `setMeanScaleBGR` function to implement the mean subtraction and normalization, as shown in the following figure. See the `Vitis-AI/src/vai_library/dpu_task/include/vitis/ai/dpu_task.hpp` file for details.

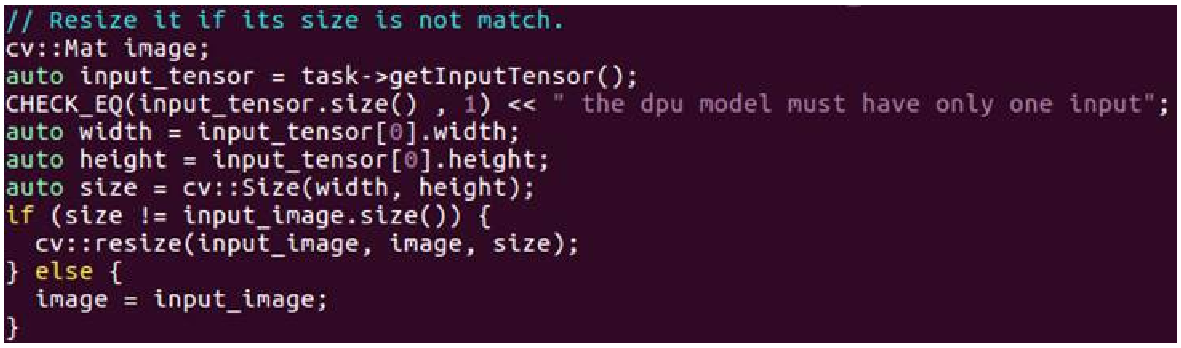
Figure 47: **setMeanScaleBGR Example**



```
// Please check /etc/dpu_model_param.conf.d/ssd_vehicle_v3_480x360.prototxt
// or your caffe model, e.g. deploy.prototxt.
task->setMeanScaleBGR({104.0f, 117.0f, 123.0f}, {1.0f, 1.0f, 1.0f});
```

Call the `cv::resize` function to scale the image, as shown in the following figure.

Figure 48: **cv::resize Example**



```
// Resize it if its size is not match.
cv::Mat image;
auto input_tensor = task->getInputTensor();
CHECK_EQ(input_tensor.size() , 1) << " the dpu model must have only one input";
auto width = input_tensor[0].width;
auto height = input_tensor[0].height;
auto size = cv::Size(width, height);
if (size != input_image.size()) {
  cv::resize(input_image, image, size);
} else {
  image = input_image;
}
```

Using the Configuration File

The AMD Vitis™ AI Library provides a way to read model parameters with a configuration file. It facilitates the uniform configuration management of model parameters. The configuration file is located at `/usr/share/vitis_ai_library/models/[model_name]/[model_name].prototxt`.

```
model
{
  name:  "yolov3_voc"
  kernel {
     name:  "yolov3_voc"
     mean:  0.0
     mean:  0.0
     mean:  0.0
     scale:  0.00390625
     scale:  0.00390625
     scale:  0.00390625
  }
  model_type : YOLOv3
  yolo_v3_param {
    …
  }
  is_tf:  false
}
```

Table 69: **Compiling Model and Kernel Parameters**

Model/Kernel	Parameter Type	Description
model	name	Same as ${MODEL_NAME}.
	model_type	Type of model used. The following types are supported: • CLASSIFICATION • DENSE_BOX • YOLOv3 • SEGMENTATION • SSD • MULTI_TASK • TFREFINEDET • OPENPOSE • ROADLINE • POINTPILLARS_NUS • REFINEDET • POINTPILLARS • REID • MEDICALREFINEDET • FAIRMOT • HOURGLASS

Table 69: **Compiling Model and Kernel Parameters** *(cont'd)*

Model/Kernel	Parameter Type	Description
kernel	name	The result of your DNNC compile. This can have an extra postfix `_0`. Include the postfix with the name, for example, inception_v1_0.
	mean	Three lines corresponding to the mean-value of “BGR”, which are predefined in the model. It is listed in "BGR" order.
	scale	Three lines corresponding to the RGB-normalized scale. It is listed in "BGR" order. If the model had no scale in training stage, this value should be 1.
	is_tf	Boolean type. If your model is trained by TensorFlow, set the value to TRUE. It could be blank in the prototxt or set as FALSE, if the model is Caffe or PyTorch.

yolo_v3_param

```
model_type : YOLOv3
  yolo_v3_param {
    num_classes: 20
    anchorCnt: 3
    layer_name: "59"
    layer_name: "67"
    layer_name: "75"
    conf_threshold: 0.3
    nms_threshold: 0.45
    biases: 10
    biases: 13
    biases: 16
    biases: 30
    biases: 33
    biases: 23
    biases: 30
    biases: 61
    biases: 62
    biases: 45
    biases: 59
    biases: 119
    biases: 116
    biases: 90
    biases: 156
    biases: 198
    biases: 373
    biases: 326
    test_mAP: false
  }
```

The parameters for the YOLOv3 model are listed in the following table. You can modify them as per your requirement.

Table 70: **YOLOv3 Model Parameters**

Parameter Type	Description
num_classes	The number of the detection categories for this model.

Table 70: **YOLOv3 Model Parameters** *(cont'd)*

Parameter Type	Description
anchorCnt	The number of anchors for this model.
layer_name	The name of the output layer of the kernel. If your model has more than one output, use this parameter to ensure the required sequence. Ensure that the name is the same as the name in the kernel. (If you enter an invalid name, the model creator will use the kernel default order.)
conf_threshold	The threshold of the boxes' confidence, which can be modified to fit your practical application.
nms_threshold	The threshold of NMS.
biases	These parameters are the same as the model parameters. Write each bias in a separate line. (Biases amount) = anchorCnt * (output-node amount) * 2. Set correct lines in the prototxt.
test_mAP	If your model was trained with letterbox and you want to test its mAP, set this as TRUE. By default, it is set to FALSE for faster execution.

SSD_param

```
model_type : SSD
ssd_param :
{
    num_classes : 4
    nms_threshold : 0.4
    conf_threshold : 0.0
    conf_threshold : 0.6
    conf_threshold : 0.4
    conf_threshold : 0.3
    keep_top_k : 200
    top_k : 400
    prior_box_param {
    layer_width : 60,
    layer_height: 45,
    variances: 0.1
    variances: 0.1
    variances: 0.2
    variances: 0.2
    min_sizes: 21.0
    max_sizes: 45.0
    aspect_ratios: 2.0
    offset: 0.5
    step_width: 8.0
    step_height: 8.0
    flip: true
    clip: false
    }
}
```

The SSD parameters are listed in the following table. The parameters of the SSD-model include the threshold and PriorBox requirements. Refer to the `SSD deploy.prototxt` file for more information.

Table 71: **SSD Model Parameters**

Parameter Type	Description
num_classes	The actual number of detection categories for this model.

Table 71: **SSD Model Parameters** *(cont'd)*

Parameter Type	Description
anchorCnt	The number of anchors for this model.
conf_threshold	The threshold of the boxes' confidence. Each category can have a different threshold, but number must be equal to num_classes.
nms_threshold	The threshold of NMS.
biases	These parameters are same as the model parameters. Write each bias in a separate line. (Biases amount) = anchorCnt * (output-node amount) * 2. Set correct lines in the prototxt.
test_mAP	If your model was trained with letterbox and you want to test its mAP, set this as TRUE. By default, it is set to FALSE for faster execution.
keep_top_k	Each category of detection objects' top K boxes.
top_k	All the detection object's top K boxes, except the background (the first category)
prior_box_param	There is more than one PriorBox corresponding to different scales. You can find them in the original model (`deploy.prototxt`) These PriorBoxes should oppose each other.

Table 72: **PriorBox Parameters**

Parameter Type	Description
layer_width/layer_height	The input width/height of this layer. Such numbers can be computed from the net structure.
variances	These numbers are used for boxes regression. These should be filled as in the original model. There should be four variances.
min_sizes/max_size	Filled as the `deploy.prototxt`. Write each number on a separate line.
aspect_ratios	The ratio (each one should be written in a separate line). By default, the first ratio is 1.0. If you set a new number here, there will be two ratios created. One of the numbers is the value that you have set, and the other number is the reciprocal of the value that you have set. For example, this parameter has only one set element, "ratios: 2.0." The ratio vector has three numbers: 1.0, 2.0, and 0.5.
offset	Normally, the PriorBox is created by each central point of the feature map, so that the offset is 0.5.
step_width/step_height	Copy from the original file. If there are no such numbers there, you can use the following formula to compute them: $step\_width = img\_width \div layer\_width$ $step\_height = img\_height \div layer\_height$
offset	Normally, PriorBox is created by each central point of the feature map, so that the offset is 0.5.
flip	Control whether to rotate the PriorBox and change the ratio of length/width.
clip	Set as FALSE. If set to TRUE, the detection box coordinates will be [0, 1].

Example Code

The following is an example code.

```
Mat img = cv::imread(argv[1]);
auto yolo = vitis::ai::YOLOv3::create("yolov3_voc", true);
auto results = yolo->run(img);
for(auto &box : results.bboxes){
    int label = box.label;
    float xmin = box.x * img.cols + 1;
    float ymin = box.y * img.rows + 1;
    float xmax = xmin + box.width * img.cols;
    float ymax = ymin + box.height * img.rows;
    if(xmin < 0.) xmin = 1.;
    if(ymin < 0.) ymin = 1.;
    if(xmax > img.cols) xmax = img.cols;
    if(ymax > img.rows) ymax = img.rows;
    float confidence = box.score;
    cout << "RESULT: " << label << "\t" << xmin << "\t" << ymin <<
"\t"
            << xmax << "\t" << ymax << "\t" << confidence << "\n";
    rectangle(img, Point(xmin, ymin), Point(xmax, ymax), Scalar(0, 255, 0),
1, 1, 0);
}
imshow("", img);
waitKey(0);
```

To create the YOLOv3 object, use `create`.

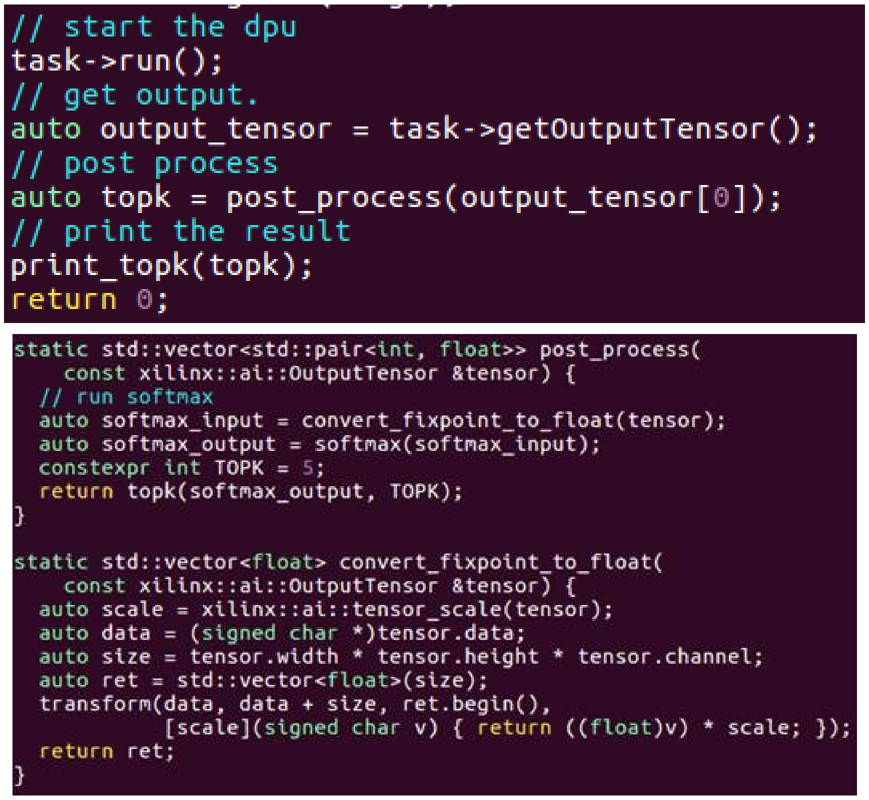
```
static std::unique_ptr<YOLOv3> create(const std::string& model_name, bool
need_mean_scale_process = true);
```

Note: The model_name is the same as the prototxt. For more details about the example, see `~/Vitis-AI/src/vai_library/yolov3/test/test_yolov3.cpp`.

Implementing User Post-Processing Code

You can also call your own post-processing functions. Take the `demo_yolov3.cpp` and `demo_classification.cpp` files as examples. Use `vitis::ai::DpuTask::create` or `vitis::ai::DpuRunner::create_dpu_runner` to create the task, and after the DPU processing is complete, you can invoke the post-processing function. The `post_process` function in the following figure shows an example of user post-processing code.

Figure 49: **User Post-Processing Code Example**



```
// start the dpu
task->run();
// get output.
auto output_tensor = task->getOutputTensor();
// post process
auto topk = post_process(output_tensor[0]);
// print the result
print_topk(topk);
return 0;
```

```
static std::vector<std::pair<int, float>> post_process(
    const xilinx::ai::OutputTensor &tensor) {
  // run softmax
  auto softmax_input = convert_fixpoint_to_float(tensor);
  auto softmax_output = softmax(softmax_input);
  constexpr int TOPK = 5;
  return topk(softmax_output, TOPK);
}

static std::vector<float> convert_fixpoint_to_float(
    const xilinx::ai::OutputTensor &tensor) {
  auto scale = xilinx::ai::tensor_scale(tensor);
  auto data = (signed char *)tensor.data;
  auto size = tensor.width * tensor.height * tensor.channel;
  auto ret = std::vector<float>(size);
  transform(data, data + size, ret.begin(),
            [scale](signed char v) { return ((float)v) * scale; });
  return ret;
}
```

Using the AI Library's Post-Processing Library

Each neural network has different post-processing methods. The `xnnpp` post-processing library is provided in the Vitis AI Library to facilitate user calls. It supports the following neural network post-processing.

- Classification
- Face detection
- Face landmark detection
- SSD detection
- Pose detection
- Semantic segmentation
- Road line detection
- YOLOv3 detection
- YOLOv2 detection

- Openpose detection
- RefineDet detection
- ReID detection
- Multi-task
- Multi-task V3
- Face recognition
- Plate detection
- Plate recognition
- Medical segmentation
- Medical detection
- Face quality
- Hourglass
- Retinaface
- Centerpoint
- Multitaskv3
- Pointpillars_nuscenes
- Rcan
- vehicleclassification
- ofa_yolo
- efficientdet_d2
- ocr
- textmountain
- YOLOx detection
- YOLOv6 detection

There are two ways to call `xnnpp`:

- Using an automatic call through `vitis::ai::<model>::create` to create the task such as `vitis::ai::YOLOv3::create("yolov3_bdd", true)`. After the <model> run is complete, `xnnpp` is automatically processed. You can modify the parameters through the model configuration file.
- Using a manual call through `vitis::ai::DpuTask::create` to create the task. Then, create the object of the post-process and run the post-process. Use the following steps. SSD post-processing is used as an example here.

1. Create a configuration and set the correlating data to control post-process.

   ```
   using DPU_conf = vitis::ai::proto::DpuModelParam;
   DPU_conf config;
   ```

2. If it is a Caffe model, set the "is_tf" as FALSE.

   ```
   config.set_is_tf(false);
   ```

3. Fill the other parameters.

   ```
   fillconfig(config);
   ```

4. Create an object of SSDPostProcess.

   ```
   auto input_tensor = task->getInputTensor();
   auto output_tensor = task->getOutputTensor();
   auto ssd = vitis::ai::SSDPostProcess::create(input_tensor,
   output_tensor,config);
   ```

5. Run the post-process.

   ```
   auto results = ssd->ssd_post_process();
   ```

***Note*:** For more details about the post processing examples, see the `~/Vitis-AI/examples/vai_library/samples/dpu_task/yolov3/demo_yolov3.cpp` and `~/Vitis-AI/src/vai_library/yolov3/test/test_yolov3.cpp` files in the host system.

Using the xdputil Tool

`xdputil` is designed for board development. It can be used for both Edge and Data Center targets. It is installed in the latest board image or Docker. The source code of `xdputil` is located in the `Vitis-AI/src/vai_library/usefultools` folder. It contains the following functions.

- **help:** It shows the usage of `xdputil`.

  ```
  xdputil --help
  ```

- **status:** It shows the status of the DPU.

  ```
  xdputil status
  ```

- **run:** Run the DPU with the input file. It can be used for DPU cross-checking.

  ```
  xdputil run <xmodel> [-i <subgraph_index>] <input_bin>

  xmodel: The model run on DPU
  -i : The subgraph_index of the model, the default value is 0
  input_bin: The input file for the model
  ```

Take `resnet50.xmodel` as an example.

```
root@xilinx-zcu104-2021_2:~# xdputil run /usr/share/vitis_ai_library/
models/resnet50/resnet50.xmodel input.bin
fillin data_fixed
dump output to 0.fc1000_fixed.bin
```

- **xmodel:** Check the xmodel information. You can convert the xmodel to png/svg/txt formats. Run the following command to show the usage of the `xmodel`.

```
root@xilinx-zcu102-20221:~# xdputil xmodel -h

usage: xdputil.py xmodel [-h] [-l] [--op [OP]] [-m] [-p [PNG]] [-s [SVG]]
[-S [SUBGRAPH_SVG]] [-t [TXT]] [-b [BINARY]] xmodel


xmodel


positional arguments:
xmodel xmodel file path


optional arguments:
  -h, --help show this help message and exit
  -l, --list show subgraph list
  --op [OP]             show op info
  -m, --meta_info show xcompiler version
  -p [PNG], --png [PNG]
        the output to png
  -s [SVG], --svg [SVG]
        the output svg path
  -S [SUBGRAPH_SVG], --subgraph_svg [SUBGRAPH_SVG]
        the output svg for subgraph level
  -t [TXT], --txt [TXT]
        when <txt> is missing, it dumps to standard output.
  -b [BINARY], --binary [BINARY]
        dump the binary data to the output directory, when  is missing,
it dumps to 'binary' directory
```

- **mem:** Read or write physical memory. The function is similar to `devmem`.

```
xdputil mem <-r|-w> <addr> <size> <output_file|input_file>
```

- **query:** Shows device information, including DPU, fingerprint, and Vitis AI version.

```
xdputil query
```

- **benchmark:** Performance of the test model. The value returned is in Frame Per Second (fps).

```
xdputil benchmark <xmodel> <-i subgraph_index> <num_of_threads>

-i : The subgraph_index of the model, index starts from 0, -1 means
running the whole graph.
```

Note: If the first subgraph is `USER` subgraph, `xdputil benchmark` will not work with subgraph_index of 0.

Take `resnet50.xmodel` as an example.

```
root@xilinx-zcu102-2021_2:~# xdputil benchmark /usr/share/
vitis_ai_library/models/resnet50/resnet50.xmodel -i -1 5
WARNING: Logging before InitGoogleLogging() is written to STDERR
I1229 23:39:07.248836 8713 test_dpu_runner_mt.cpp:473] shuffle results
for batch...
I1229 23:39:07.252218 8713 performance_test.hpp:73] 0% ...
I1229 23:39:13.252394 8713 performance_test.hpp:76] 10% ...
I1229 23:39:19.252584 8713 performance_test.hpp:76] 20% ...
I1229 23:39:25.252804 8713 performance_test.hpp:76] 30% ...
I1229 23:39:31.253026 8713 performance_test.hpp:76] 40% ...
I1229 23:39:37.253317 8713 performance_test.hpp:76] 50% ...
I1229 23:39:43.253564 8713 performance_test.hpp:76] 60% ...
I1229 23:39:49.253836 8713 performance_test.hpp:76] 70% ...
I1229 23:39:55.254051 8713 performance_test.hpp:76] 80% ...
I1229 23:40:01.254329 8713 performance_test.hpp:76] 90% ...
I1229 23:40:07.254683 8713 performance_test.hpp:76] 100% ...
I1229 23:40:07.254791 8713 performance_test.hpp:79] stop and waiting for
all threads terminated....
I1229 23:40:07.265725 8713 performance_test.hpp:85] thread-0 processes
2266 frames
I1229 23:40:07.265758 8713 performance_test.hpp:85] thread-1 processes
2072 frames
I1229 23:40:07.265779 8713 performance_test.hpp:85] thread-2 processes
2637 frames
I1229 23:40:07.278290 8713 performance_test.hpp:85] thread-3 processes
2280 frames
I1229 23:40:07.279388 8713 performance_test.hpp:85] thread-4 processes
2052 frames
I1229 23:40:07.279413 8713 performance_test.hpp:93] it takes 24599 us for
shutdown
I1229 23:40:07.279430 8713 performance_test.hpp:94] FPS= 188.365
number_of_frames= 11307 time= 60.0272 seconds.
I1229 23:40:07.279479 8713 performance_test.hpp:96] BYEBYE
Test PASS.
```

- **run_op:** Run the OP in the model.

```
xdputil run_op <xmodel> <op_name> [-r REF_DIR] [-d DUMP_DIR]
```

Run the following command to show the usage of `run_op`.

```
root@xilinx-zcu102-2021_2:~# xdputil run_op  -h
usage: xdputil.py run_op [-h] [-r REF_DIR] [-d DUMP_DIR] xmodel op_name

positional arguments:
  xmodel                xmodel file name
  op_name               op name, this op_name should be consistent with
the name in xmodel

optional arguments:
  -h, --help            show this help message and exit
  -r REF_DIR, --ref_dir REF_DIR
                        reference directory, this directory default as
"ref" should contain inputs tensor file like
```

```
                             <TENSOR_NAME>.bin
  -d DUMP_DIR, --dump_dir DUMP_DIR
                             dump directory, this directory default as "dump"
will be the dump destination of output tensor
                             file
```

- **comp_float:** Comparison between golden file and dump file, especially when they are float point numbers.

```
xdputil comp_float <golden_file> <dump_file> [-t threshold] [--verbose]
```

Note: `/usr/bin/python3 -m xdputil` is used instead of `xdputil` when `xdputil` is used in the Docker. For example, `/usr/bin/python3 -m xdputil query`.

Implementing and Registering Custom Operators

In this example, an XIR OP `add` is implemented. It adds two input tensors, assuming that both the tensors have the same shape.

To register a new XIR OP, refer to the *Vitis AI User Guide* (UG1414). This example assumes that the add OP is already registered in the Xmodel graph which means the model with `add` OP has been compiled successfully by Vitis AI xcompiler.

For a complete reference golden code, see Customized XIR OP example.

To implement an XIR OP, follow these steps:

1. Write a C++ class.
2. Write the constructor function.
3. Write the `calculate` function.
4. Register the implementation with the macro.
5. Build a shared library.
6. Deploy it.

Writing OP Implementation in C++

1. In `my_add_op.cpp`, create a C++ class. There are no requirements for naming the source file or class.

```
// in my_add_op.cpp
class MyAddOp {
};
```

2. Write the constructor function as shown in the following code snippet.

```
#include <vart/op_imp.h>

class MyAddOp {
    MyAddOp(const xir::Op* op1, xir::Attrs* attrs) : op{op1} {
      // op and attrs is not in use.
}
public:
    const xir::Op * const op;
};
```

Note: `MyAddOp` must have a public member variable named `op`. `op` is initialized with the first input argument of the constructor function, for example, `op1`. This is required for `DEF_XIR_OP_IMP`.

3. Write the member function, `calculate`, as shown in the following code snippet.

```
class MyAddOp {
  ...
  int calculate(vart::simple_tensor_buffer_t output,
                std::vector<vart::simple_tensor_buffer_t<float>> inputs)
{
    for (auto i = 0u; i < output.mem_size / sizeof(float); ++i) {
      output.data[i] = 0.0f;
      for (auto input : inputs) {
        output.data[i] = output.data[i] + input.data[i];
      }
    }
    return 0;
  }
...
}
```

4. Compile the source file.

```
% g++ -fPIC -std=c++17 -c -o  /tmp/my_add_op.o -Wall -Werror -I ~/.local/
Ubuntu.18.04.x86_64.Debug/include/ my_add_op.cpp
```

Note: Use C++ 17 or above. To build a shared library, enable `-fPIC`. It is assumed that the Vitis AI Library is installed at `~/.local/Ubuntu.18.04.x86_64.Debug`.

5. To link to a shared library, use the following code.

```
% mkdir -p /tmp/lib;
% g++ -Wl,--no-undefined -shared -o /tmp/lib/libvart_op_imp_add.so /tmp/
my_add_op.o -L ~/.local/Ubuntu.18.04.x86_64.Debug/lib -lglog -
lvitis_ai_library-runner_helper -lvart-runner -lxir
```

You can also use Makefile to compile and link the library. An example Makefile is shown in the following code snippet.

```
OUTPUT_DIR = $(HOME)/build/customer_op

all: $(OUTPUT_DIR) $(OUTPUT_DIR)/libvart_op_imp_add.so

$(OUTPUT_DIR):
    mkdir -p $@

$(OUTPUT_DIR)/my_add_op.o: my_add_op.cpp
    $(CXX) -std=c++17 -fPIC -c -o $@ -I. -I=/install/Debug/include -
```

```
Wall  -U_FORTIFY_SOURCE -D_FORTIFY_SOURCE=0 $<

$(OUTPUT_DIR)/libvart_op_imp_add.so:  $(OUTPUT_DIR)/my_add_op.o
    $(CXX) -Wl,--no-undefined -shared -o $@ $+ -L=/install/Debug/lib  -
lglog -lvitis_ai_library-runner_helper -lvart-runner -lxir
```

6. To test the op implementation, create a sample XIR graph first as below.

```
% ipython;
import xir
g = xir.Graph("simple_graph")
a = g.create_op("a", "data", {"shape": [1,2,2,4], "data_type":
"FLOAT32"});
b = g.create_op("b", "data", {"shape": [1,2,2,4], "data_type":
"FLOAT32"});
add = g.create_op("add_op", "add",  {"shape": [1,2,2,4], "data_type":
"FLOAT32"}, {"input": [a,b]})
root = g.get_root_subgraph()
root.create_child_subgraph()
user_subgraph = root.merge_children(set([g.get_leaf_subgraph(a),
g.get_leaf_subgraph(b)]))
cpu_subgraph = root.merge_children(set([g.get_leaf_subgraph(add)]))
user_subgraph.set_attr("device", "USER")
cpu_subgraph.set_attr("device", "CPU")
g.serialize("/tmp/add.xmodel")
```

RECOMMENDED: *Instead of writing complex Python codes, create an xmodel using the Xcompiler. For more information, refer to the Vitis AI User Guide (UG1414).*

7. Create a sample input file.

```
% cd /tmp
% mkdir -p ref
% ipython
import numpy as np
a = np.arange(1, 17, dtype=np.float32)
b = np.arange(1, 17, dtype=np.float32)
a.tofile("ref/a.bin")
b.tofile("ref/b.bin")
c = a + b
c.tofile("ref/c.bin")
```

```
% cd /tmp
% mkdir -p /tmp/dump
% env LD_LIBRARY_PATH=$HOME/.local/Ubuntu.18.04.x86_64.Debug/lib:/tmp/
lib $HOME/.local/Ubuntu.18.04.x86_64.Debug/share/vitis_ai_library/test/
cpu_task/test_op_imp --graph /tmp/add.xmodel --op "add_op"
```

***Note*:** Add `/tmp/lib` into the search path `LD_LIBRARY_PATH` so that the CPU runner can find the shared library you wrote.

IMPORTANT! *The name of the shared library must be* `libvart_op_imp_<YOUR_OP_TYPE>.so`*. The CPU runner uses this naming scheme to find the customized xir::Op implementation.*

You can also use `xdputil run_op` to verify the op:

```
root@xilinx-zcu102-2021_2:~/add_op# xdputil run_op add.xmodel add_op -r
ref -d dump
WARNING: Logging before InitGoogleLogging() is written to STDERR
I1202 09:32:41.497661  1208 test_op_run.cpp:79] try to test op: add_op
I1202 09:32:41.497745  1208 test_op_run.cpp:97]  input op: a tensor: a
I1202 09:32:41.497768  1208 test_op_run.cpp:97]  input op: b tensor: b
I1202 09:32:41.497865  1208 test_op_run.cpp:55] read ref/a.bin to
0xaaab17d605d0 size=64
I1202 09:32:41.497917  1208 test_op_run.cpp:55] read ref/b.bin to
0xaaab17c549b0 size=64
I1202 09:32:41.498561  1208 test_op_run.cpp:114] graph
name:simple_graphtesting op: {
    {args: input= TensorBuffer{@0xaaab17ba9b90,tensor=xir::Tensor{name =
a, type = FLOAT32, shape = {1, 2, 2,
4}},location=HOST_VIRT,data=[(Virt=0xaaab17d605d0, 64)]}
TensorBuffer{@0xaaab17e2a860,tensor=xir::Tensor{name = b, type =
FLOAT32, shape = {1, 2, 2,
4}},location=HOST_VIRT,data=[(Virt=0xaaab17c549b0, 64)]}}
{
I1202 09:32:41.499586  1208 test_op_run.cpp:68] write output to dump/
add_op.bin from 0xaaab17de7090 size=64
test pass
```

8. To verify that the `op` is implemented properly, compare it with the reference result.

```
% diff -u <(xxd ref/c.bin) <(xxd dump/add_op.bin)
% xxd ref/c.bin
% xxd dump/add_op.bin
```

Writing OP Implementation in Python

An `add` OP is used as an example in this section. For more Python examples, refer to TensorFlow2 and PyTorch examples.

1. Create a Python package and a module file.

```
% mkdir -p /tmp/demo_add_op/
% cd /tmp/demo_add_op/
% mkdir vart_op_imp/
% touch vart_op_imp/__init__.py
% touch vart_op_imp/add.py
```

Note:

- The folder name should be `vart_op_imp`.
- If you have more than one custom OP, create them separately and place them in the `vart_op_imp` folder.

2. Update `add.py` as shown in the following code snippet.

```
import numpy as np


class add:
    def __init__(self, op):
        pass
    def calculate(self, output, input):
        np_output = np.array(output, copy=False)
        L = len(input)
        if L == 0:
            return
        np_input_0 = np.array(input[0], copy=False)
        np.copyto(np_output, np_input_0)
        for i in range(1, L):
            np_output = np.add(np_output,
                               np.array(input[i], copy=False),
                               out=np_output)
```

You can also use the following simplified version.

```
import numpy as np


class add:
    def __init__(self, op):
        pass

    def calculate(self, output, input):
        np_output = np.array(output, copy=False)
        L = len(input)
        assert L == 2
        np_input_0 = np.array(input[0], copy=False)
        np_input_1 = np.array(input[1], copy=False)
        np_output = np.add(np_input_0, np_input_1, out=np_output)
```

3. Install the op as shown in the following code snippet.

```
% mkdir -p lib
% ln -sf ~/.local/Ubuntu.18.04.x86_64.Debug/lib/libvart_op_imp_python-
cpu-op.so lib/libvart_op_imp_add.so
% ls -l lib
% mkdir -p dump
% env LD_LIBRARY_PATH=$HOME/.local/Ubuntu.18.04.x86_64.Debug/lib:$PWD/
lib $HOME/.local/Ubuntu.18.04.x86_64.Debug/share/vitis_ai_library/test/
cpu_task/test_op_imp --graph /tmp/add.xmodel --op "add_op"
```

For Edge, you can execute the following command to install and test the op.

```
# ls -sf /usr/lib/libvart_op_imp_python-cpu-op.so /usr/lib/
libvart_op_imp_add.so
# cp -r vart_op_imp /usr/lib.python3.9/site-packages
# xdputil run_op add.xmodel add_op -r ref -d dump
```

Note: Before you execute the above commands, copy `add.xmodel`, the `ref` folder with the sample input files, and the `vart_op_imp` folder to the board.

Similar to the C++ interface, the Python module must have a class whose name is the same as the type of xir::Op. In this example, `add` is used. The class must have a constructor with a single argument `op` in addition to `self`. It is an XIR::Op. Refer to the XIR Python API for more information. Similarly, the class `add` must have a member function called `calculate`, in addition to the `self` argument. The first argument must be named `output`, and following argument names must comply with the XIR::OpDef associated with the XIR::Op, refer to XIR API for more detail.

***Note*:** A symbolic link to `libvart_op_imp_python-cpu-op.so` named `libvart_op_imp_add.so` is created. `libvart_op_imp_python-cpu-op.so` is a bridge between Python and C++. From the C++ side, it searches for `libvart_op_imp_add.so` and finds the `libvart_op_imp_python-cpu-op.so`. In `libvart_op_imp_python_cpu_op.so`, the Python module name `vart_op_imp.add` is imported and Python searches for the module as usual.

Application Demos

This chapter describes how to set up a test environment and run the application demos. There are more than two application demos provided within the AMD Vitis™ AI Library. You can find the demo source code here.

Demo Overview

There are six application demos provided within the Vitis AI Library. They use the Vitis AI Library to build their applications. The source code for the applications are stored in `Vitis-AI/examples/vai_library/apps`.

- Demo 1: segs_and_roadline_detect is a demo that includes MultiTask segmentation network processing, vehicle detection, and road line detection. It simultaneously performs 4-channel segmentation and vehicle detection and 1-channel road lane detection.
- Demo 2: seg_and_pose_detect is a demo that includes MultiTask segmentation network processing and pose detection. It simultaneously performs 1-channel segmentation process and 1-channel pose detection.
- Demo 3: multitask_v3_quad_windows is a demo running `multitask_v3` model and display it.
- Demo 4: vck190_4video is a demo running on VCK190 platform. It processes four channels of video input.
- Demo 5: vck190_4mipi is a demo running on VCK190 platform. It processes four channels of MIPI input.
- Demo 6: vek280_4video is a demo running on VEK280 platform. It processes four channels of video input.

***Note*:** For ZCU102/ZCU104 platform, to achieve the best performance, the demos use the Direct Render Manager (DRM) for video display. Log in the board using `ssh` or serial port and run the demo remotely. If you do not want to use DRM for video display, set `USE_DRM=0` in the compile option.

***Note*:** For `vck190_4mipi` and `vck190_4video`, refer to the Vitis AI Library Apps on GitHub.

Demo Platform and Setup

Demo Platform for ZCU102 Evaluation Kit

- **Hardware:**
  - 1 x ZCU102 Prod Silicon
  - 1 x Windows 7/10 laptop
  - 1 x 16 GB SD card
  - 1 x Ethernet cables
  - 1 x DP 1080P compatible monitor
  - 1 x DP cable
- **Software:**
  - ZCU102 board image
  - Vitis AI Library
  - Images and video files
  - Terminal software like MobaXterm, Putty

Demo Platform for VCK190 Evaluation Kit

- **Hardware:**
  - 1 x VCK190 Prod Silicon
  - 1 x Avnet Multi-Camera MIPI FMC Module
  - 1 x Windows 7/10 laptop
  - 2 x 16/32/64 GB GB SD card
  - 1 x Ethernet cables
  - 1 x HDMI™ 4K compatible monitor
  - 1 x HDMI cable
- **Software:**
  - ZU4 SC image, download it from here
  - VCK190 board image, download it from here
  - Vitis AI Library
  - Video files

- Terminal software like MobaXterm, Putty

Demo Platform for VEK280 Evaluation Kit

- **Hardware:**
  - 1 x VEK280 Silicon
  - 1 x Windows 7/10 laptop
  - 2 x 16/32/64 GB GB SD card
  - 1 x Ethernet cables
  - 1 x HDMI™ 4K compatible monitor
  - 1 x HDMI cable
- **Software:**
  - VEK280 board image, download it from here
  - Vitis AI Library
  - Video files
  - Terminal software like MobaXterm, Putty

DPU Configuration and Dev Tools Used for ZCU102 Evaluation Kit

- 3xB4096 @281 MHz
- Vivado 2022.2, Vitis AI Library v3.0

DPU Configuration and Dev Tools Used for VCK190 Evaluation Kit

- 3xC32B1
- Vivado 2022.1, Vitis AI Library v2.5

DPU Configuration and Dev Tools Used for VEK280 Evaluation Kit

- 1xC20B1
- Vivado 2023.1, Vitis AI Library v3.5

Demo Setup Illustration

Figure 50: **Demo Setup for the ZCU102 Evaluation Kit**

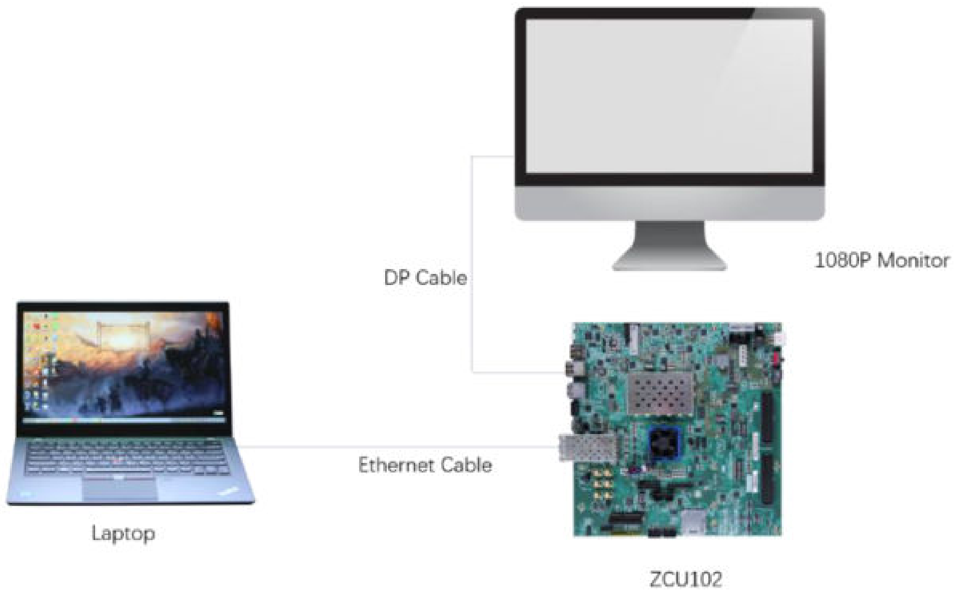
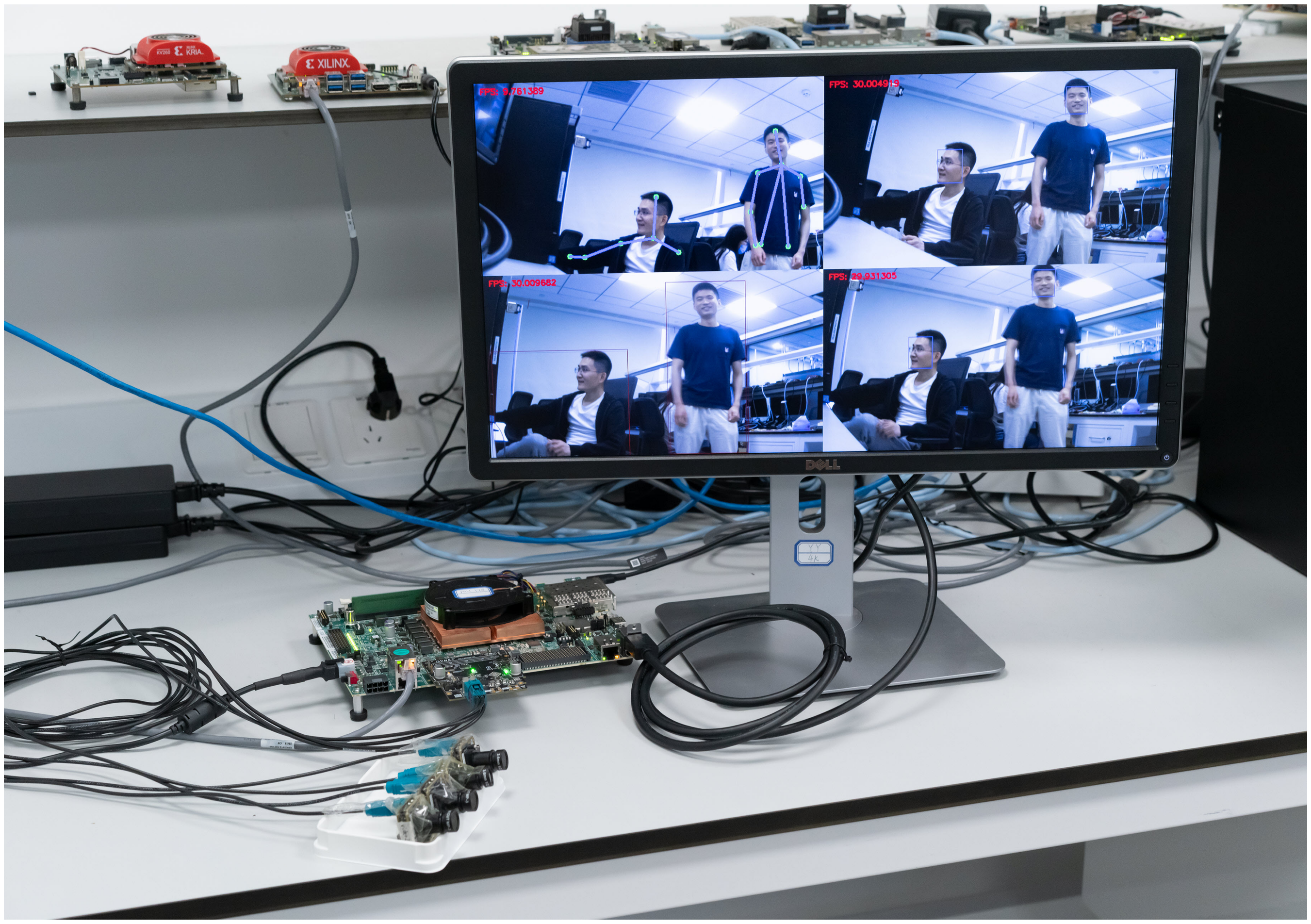


Figure 51: **Demo Setup for the VCK190 Evaluation Kit**



Demo 1: segs_and_roadline_detect

segs_and_roadline_detect is a demo that includes MultiTask segmentation network processing, vehicle detection, and road line detection.

Target Application

ADAS/AD

AI Model, Performance, and Power

- **FPN:** 512x288, 4ch, 20fp
- **VPGNET:** 640x480, 1ch, 56fps
- **20W @ ZU9EG:** N/A

Building and Running the Demo

Build the demo in the host and copy the program to the target board.

```
cd Vitis-AI/examples/vai_library/apps/segs_and_roadline_detect
bash -x build.sh
scp segs_and_roadline_detect_x segs_and_roadline_detect_drm
root@IP_OF_BOARD:~/
```

To use OpenCV display, run the following command:

```
./segs_and_roadline_detect_x seg_512_288.avi seg_512_288.avi
seg_512_288.aviseg_512_288.avi lane_640_480.avi -t 2 -t 2 -t 2 -t 2 -t 3
>/dev/null 2>&1
```

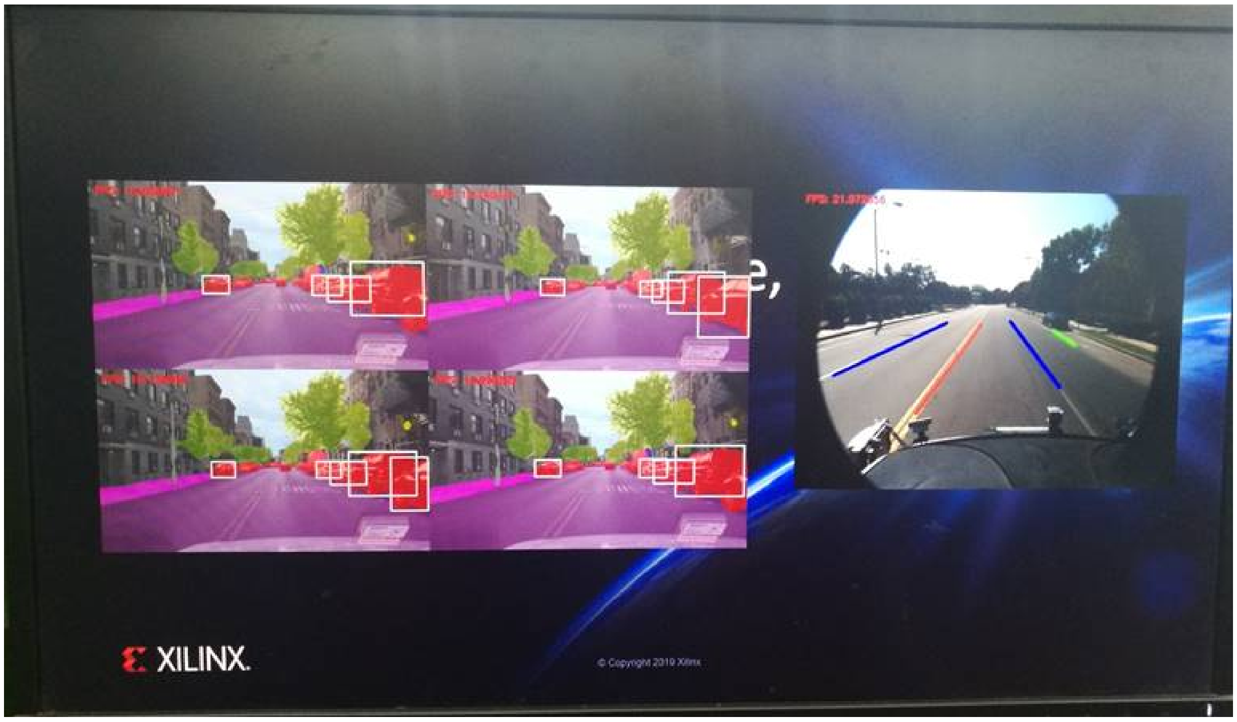
If you want to use DRM display, connect to the board using SSH and run the following command:

```
./segs_and_roadline_detect_drm seg_512_288.avi seg_512_288.avi
seg_512_288.avi
seg_512_288.avi lane_640_480.avi -t 2 -t 2 -t 2 -t 2 -t 3 >/dev/null 2>&1
```

***Note*:**

1. Download the video file package from here.
2. Due to limitations of the Docker environment, the MultiTask demos cannot run in DRM mode for Data Center targets.

Figure 52: **Segmentation and Roadline Detection Demo Picture**



Demo 2: seg_and_pose_detect

seg_and_pose_detect is a demo that includes MultiTask segmentation network processing and pose detection.

Target Application

- ADAS/AD
- Smartcity

AI Model, Performance, and Power

- **FPN:** seg_and_pose_detect

  960x540, 1ch, 30fps

- **Openpose:** 960x540, 1ch, 30fps
- **20W @ ZU9EG:** N/A

Building and Running the Demo

Build the demo in the host and copy the program to the target board:

```
cd Vitis-AI/examples/vai_library/apps/seg_and_pose_detect
bash -x build.sh
scp seg_and_pose_detect_x seg_and_pose_detect_drm root@IP_OF_BOARD:~/
```

To use OpenCV display, run the following command:

```
#./seg_and_pose_detect_x seg_960_540.avi pose_960_540.avi -t 4 -t 4 >/dev/
null 2>&1
```

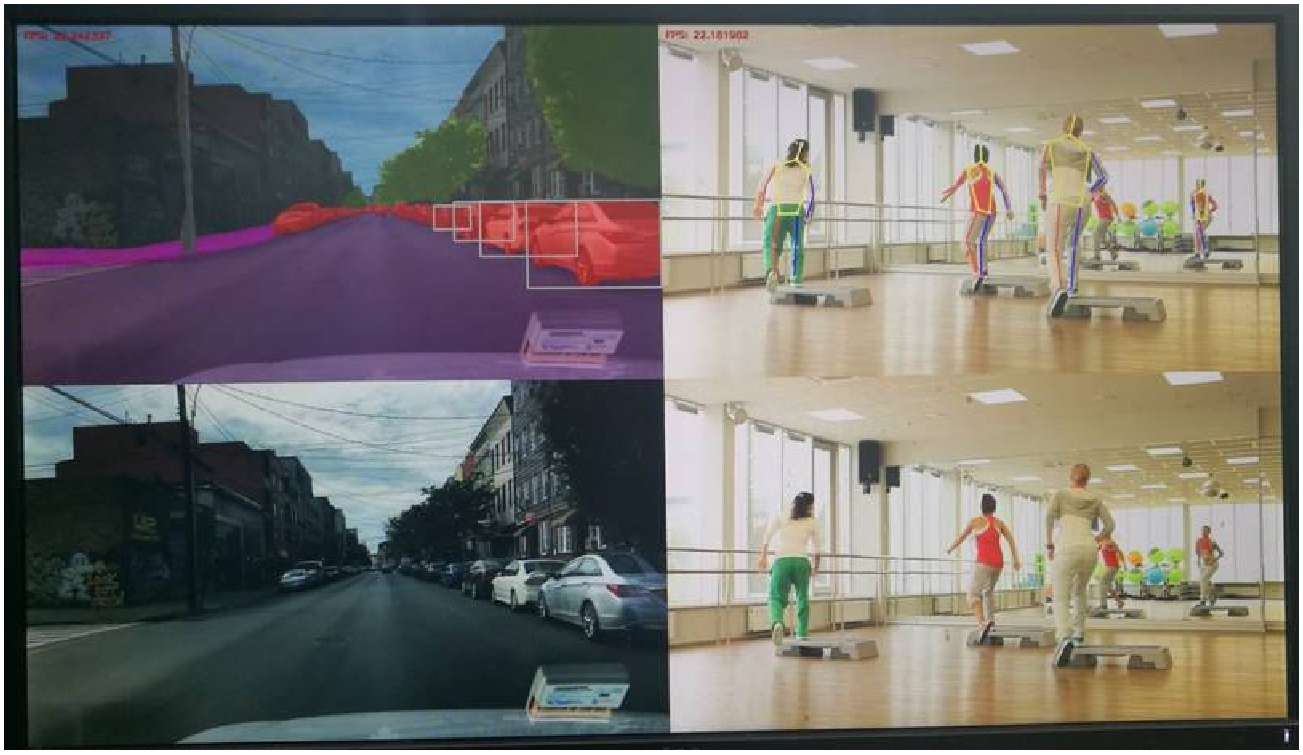
If you want to use DRM display, connect to the board using SSH and run the following command:

```
#./seg_and_pose_detect_drm seg_960_540.avi pose_960_540.avi -t 4 -t 4 >/dev/
null 2>&1
```

***Note*:**

1. Download the video file package from here.
2. Due to limitations of the Docker environment, the MultiTask demos cannot run in DRM mode for Data Center targets.

Figure 53: **Segmentation and Pose Detection Demo Picture**



Demo 3: multitask_v3_quad_windows

AI Model and Performance

- **multitask_v3:** 320x512, 1ch, 30fps

Building and Running the Demo

Build the demo in the host and copy the program to the target board.

```
cd Vitis-AI/examples/vai_library/apps/multitask_v3_quad_windows
bash -x build.sh
bash -x builddrm.sh
scp multitaskv3_quad_windows_drm multitaskv3_quad_windows_x
root@IP_OF_BOARD:~/
```

To use OpenCV display, run the following command:

```
#./multitaskv3_quad_windows_x d58cbda2-97976be7__640x360.avi -t 4 > /dev/
null 2>&1
```

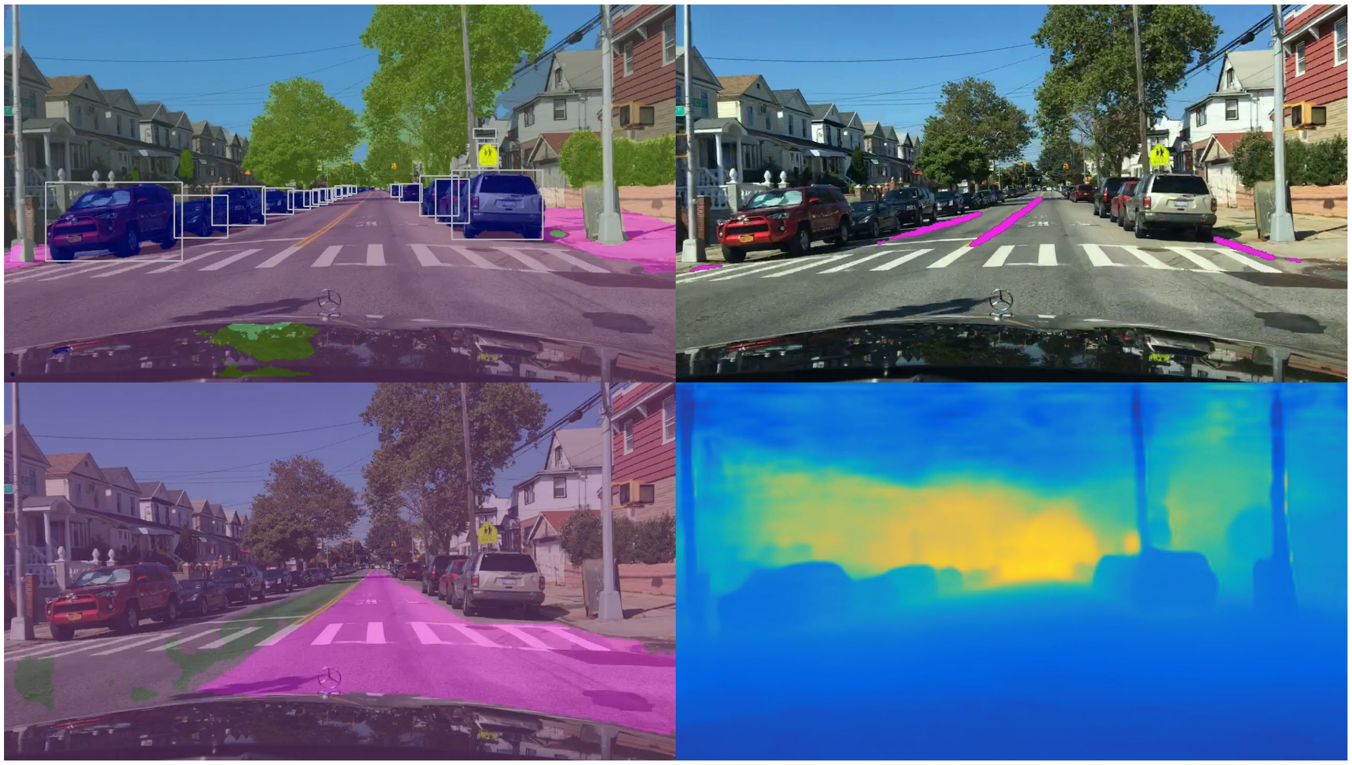
To use DRM display, connect to the board using SSH and run the following command:

```
#./multitaskv3_quad_windows_drm d58cbda2-97976be7__640x360.avi -t 4  > /dev/
null 2>&1
```

***Note*:**

1. Download the video file package from here.
2. Due to limitations of the Docker environment, the multitask_v3_quad_windows demos cannot run in DRM mode for Data Center targets.

Figure 54: **multitask_v3_quad_windows Demo Picture**



Demo 4: vck190_4video

AI Model and Performance

- **multi_task:** 512x288, 4ch, 25fps

Building and Running the Demo

***Note*:** This demo requires a special VCK190 image. It integrates the C32B1CU3 DPU and the Hardware Scaler IP. You can download the VCK190 image from here.

1. Build the demo in the host and copy the program to the target board.

```
cd Vitis-AI/examples/vai_library/apps/vck190_4video
bash build_4video.sh
scp -r vck190_4video root@IP_OF_BOARD:~/
```

2. Download the video file package from here, untar it, and find the `seg_512_288.avi` file in `apps/vek280_4video`.
3. Copy `seg_512_288.avi` file to the target.

```
scp seg_512_288.avi root@IP_OF_BOARD:~/vck190_4video
```

4. Run the demo.

```
# /etc/init.d/xserver-nodm stop
bash run_4video.sh
```

Demo 5: vck190_4mipi

AI Model and Performance

Table 73: **AI Model and Performance of vck190_4mipi Demo**

Channel	Model Name	Performance (fps)
0	sp_net	30
	ssd_pedestrian_pruned_0_97	
1	ssd_pedestrian_pruned_0_97	30
2	densebox_640_360	30
3	densebox_640_360	30

MIPI Camera Setup

1. Download ZU4 SC image from here and follow the instructions on the Update System Controller uSD Card page to update the ZU4 SC.
2. Download BoardUI tools from here.

   ***Note*:** Make sure you have accounts to download the above resources. You can use your AMD account or create a new one.

3. Refer to Board jumper and switch settings to set up the board.
4. Refer to Vadj Settings to set `Vadj`.

   ***Note*:** Make sure the Vadj 1.2V is set successfully.

Building and Running the Demo

***Note*:** This demo requires a special VCK190 image. It integrates the C32B1CU3 DPU and the Hardware Scaler IP. You can download the VCK190 image from here. For more details, refer to Demo Platform for VCK190 Evaluation Kit.

- Build the demo in the host and copy the program to the target board.

```
cd Vitis-AI/examples/vai_library/apps/vck190_4mipi
bash build_4mipi_spnet.sh
scp -r vck190_4mipi root@IP_OF_BOARD:~/
```

- Initialize the camera.

```
# /etc/init.d/xserver-nodm stop
# sh quad_640x360_bgr.sh
```

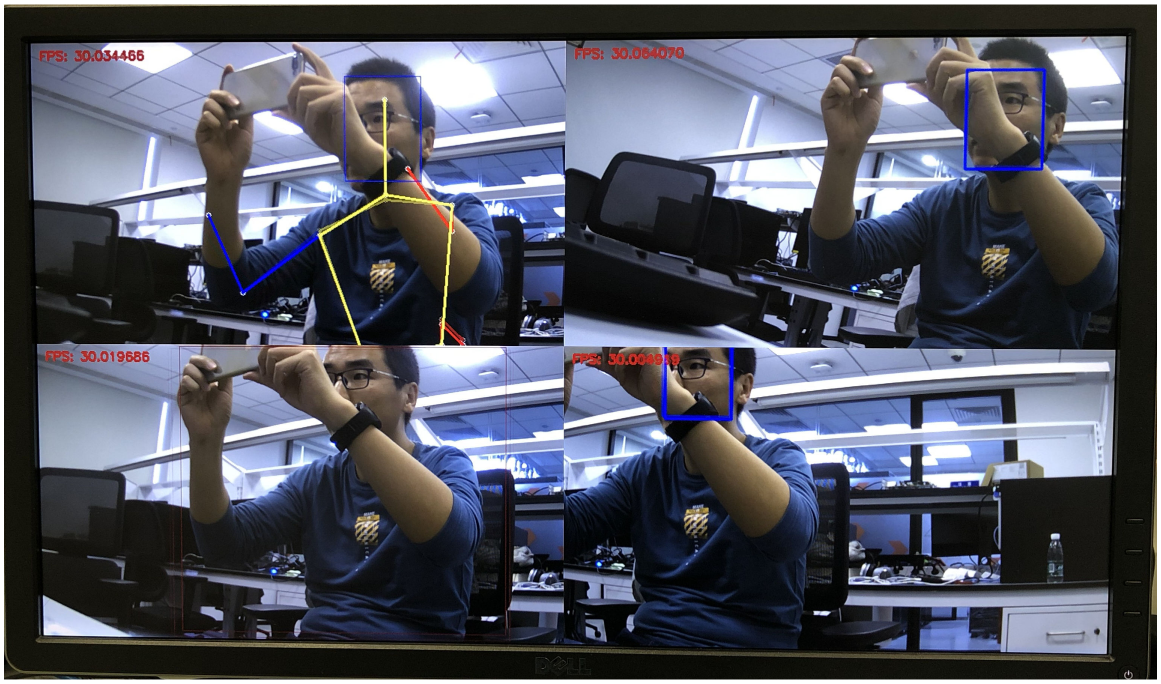
- Modify the camera ISP parameters.

```

v4l2-ctl -d /dev/v4l-subdev11 -c ar0231_green_balance=170
v4l2-ctl -d /dev/v4l-subdev12 -c ar0231_green_balance=170
v4l2-ctl -d /dev/v4l-subdev13 -c ar0231_green_balance=170
v4l2-ctl -d /dev/v4l-subdev14 -c ar0231_green_balance=170
v4l2-ctl -d /dev/v4l-subdev11 -c ar0231_blue_balance=500
v4l2-ctl -d /dev/v4l-subdev11 -c ar0231_red_balance=140
v4l2-ctl -d /dev/v4l-subdev11 -c ar0231_exposure=1000
v4l2-ctl -d /dev/v4l-subdev11 -c ar0231_digital_gain=800
v4l2-ctl -d /dev/v4l-subdev12 -c ar0231_blue_balance=500
v4l2-ctl -d /dev/v4l-subdev12 -c ar0231_red_balance=140
v4l2-ctl -d /dev/v4l-subdev12 -c ar0231_exposure=1000
v4l2-ctl -d /dev/v4l-subdev12 -c ar0231_digital_gain=800
v4l2-ctl -d /dev/v4l-subdev13 -c ar0231_blue_balance=500
v4l2-ctl -d /dev/v4l-subdev13 -c ar0231_red_balance=140
v4l2-ctl -d /dev/v4l-subdev13 -c ar0231_exposure=1000
v4l2-ctl -d /dev/v4l-subdev13 -c ar0231_digital_gain=800
v4l2-ctl -d /dev/v4l-subdev14 -c ar0231_blue_balance=500
v4l2-ctl -d /dev/v4l-subdev14 -c ar0231_red_balance=140
v4l2-ctl -d /dev/v4l-subdev14 -c ar0231_exposure=1000
v4l2-ctl -d /dev/v4l-subdev14 -c ar0231_digital_gain=800
```

- Run the demo.

```
sh run_4mipi_spnet.sh
```

Figure 55: **vck190_4mipi Demo Picture**



Demo 6: vek280_4video

AI Model and Performance

- **multi_task:** 512x288, 4ch, 25fps

Building and Running the Demo

***Note*:** This demo requires a special VEK280 image. It integrates the C20B1CU1 DPU and the Hardware Scaler IP. You can download the VEK280 image from here.

1. Build the demo in the host and copy the program to the target board.

```
cd Vitis-AI/examples/vai_library/apps/vek280_4video
bash build_4video.sh
scp -r vek280_4video root@IP_OF_BOARD:~/
```

2. Download the video file package from here, untar it, and find the `seg_512_288.avi` file in `apps/vek280_4video`.
3. Copy `seg_512_288.avi` file to the target.

```
scp seg_512_288.avi root@IP_OF_BOARD:~/vek280_4video
```

4. Run the demo.

```
# /etc/init.d/xserver-nodm stop
sh ./run_hdmi.sh
```

Chapter 6

Programming APIs

To use the library, you need to prepare the development board and cross-compilation environment. Pay attention to the header files, library files, and model library files.

Note: The files in the development environment must match the version provided in the AMD Vitis™ Unified Software Development Environment. These libraries can be executed on the ZCU102, ZCU104, VCK190 evaluation boards, and on the VCK5000 development card.

1. Select an image. For example, `cv::Mat`.
2. Call the `create` method provided by the corresponding library to get a class instance. If you set the `need_preprocess` variable to `false`, the model will not decrease its mean and scale.
3. Call the `getInputWidth()` and the `getInputHeight()` functions to get the network needed column and row values of the input image.
4. Resize image to `inputWidth x inputHeight`.
5. Call `run()` to get the result of the network.

For details about the Programming APIs, see Chapter 8: API Reference.

Also, for the Vitis AI APIs, see the *Vitis AI User Guide* (UG1414). You can download it from the website.

Performance

This chapter describes the performance of the Vitis AI Library on the following different evaluation boards and data center accelerator cards.

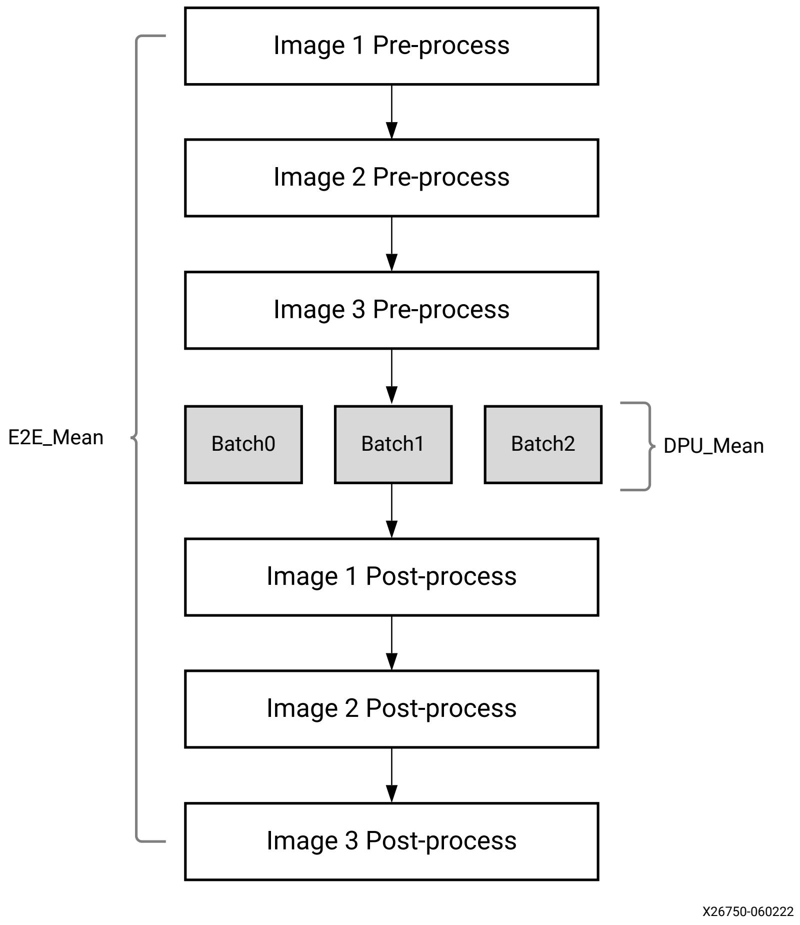
- VEK280 Evaluation Board
- V70 Versal Development Card

The following boards are not updated for Vitis AI 3.5. Refer to the performance data in the Vitis AI 3.0 version of this document for the performance metrics for these boards.

- ZCU102 Evaluation Kit
- ZCU104 Evaluation Board
- KV260 Vision AI Starter Kit
- VCK190 Evaluation Board
- VCK5000 Versal Development Card

For the one thread performance test, the performance is calculated using the formula: (1000 ms / E2E_mean time)*batch_number. Taking VCK190 as an example, if the batch_number is 3, the E2E_mean and DPU_mean are as shown in the following figure.

Figure 56: **One Thread Performance Test Example for batch_number = 3**



VEK280 Evaluation Board

VEK280 is a new evaluation platform for the Versal AI Edge device VE2802. VEK280 targets AI-ML applications with increased compute performance, lower latency, and higher levels of integration enabled by the Versal AI Edge family of devices. The primary focus of VEK280 is to enable solution demos to make it easier for customers to develop their own applications.

In Vitis AI 3.5, a C20B14CU1 DPU core is implemented. It delivers 179.2 TOPS INT8 peak performance for deep learning inference acceleration.

Refer to the following table for the throughput performance (in frames/sec or fps) for various neural network samples on VEK280 with AI Engines clocked at 1250 MHz and PL clocked at 300 MHz.

Table 74: **VEK280 Performance with Batch 14**

No	Neural Network	Input Size	GOPS	1-thread(fps)	Multi-thread(fps)
1	chen_color_resnet18_pt	224x224	3.627	2590.9	6067.8
2	efficientnet_lite_tf2	224x224	0.77	2228.0	5194.7
3	efficientNet-edgetpu-L_tf	300x300	19.36	629.5	961.1
4	efficientNet-edgetpu-M_tf	240x240	7.34	1476.2	3515.6
5	efficientNet-edgetpu-S_tf	224x224	4.72	1729.7	4588.8
6	face_mask_detection_pt	512x512	0.593	517.9	1043.9
7	fadnet_v2	576x960	412	9.5	18.4
8	fadnet_v2_pruned	576x960	201	9.7	19.1
9	HardNet_MSeg_pt	352x352	22.78	279.9	412.3
10	hfnet_tf	960x960	20.09	10.7	24.1
11	inception_v1_tf	224x224	3	1758.4	4611.7
12	inception_v3_pt	299x299	5.7	829.9	1573.0
13	inception_v3_tf	299x299	11.5	831.9	1571.5
14	inception_v3_tf2	299x299	11.5	891.3	1808.4
15	inception_v4_2016_09_09_tf	299x299	24.6	492.5	680.5
16	MLPerf_resnet50_v1.5_tf	224x224	8.19	1730.2	4563.7
17	mlperf_ssd_resnet34_tf	1200x1200	433	17.2	40.6
18	mobilenet_1_0_224_tf2	224x224	1.1	2376.4	5186.8
19	mobilenet_v1_0_25_128_tf	128x128	0.027	5149.5	10575
20	mobilenet_v1_1_0_224_tf	224x224	1.1	2375.6	5171.1
21	mobilenet_v2_1_0_224_tf	224x224	0.6	2292.4	5155.1
22	mobilenet_v2_1_4_224_tf	224x224	1.2	2074.9	5083.5
23	movenet_ntd_pt	192x192	0.5	240.8	422.9
24	ofa_depthwise_res50_pt	176x176	1.25	338.6	530.4
25	ofa_rcan_latency_pt	360x640	45.7	78.9	104.7
26	ofa_resnet50_0_9B_pt	160x160	1.8	2861.7	7949.4
27	ofa_yolo_pruned_0_30_pt	640x640	34.71	182.2	390.8
28	ofa_yolo_pruned_0_50_pt	640x640	24.62	202.6	413.9
29	ofa_yolo_pt	640x640	48.88	172.5	370.2
30	pointpillars_kitti_12000_pt	12000x100x4	10.8	58.8	71.7
31	rcan_pruned_tf	360x640	86.95	66.9	84.6
32	RefineDet-Medical_EDD_tf	320x320	9.8	648.5	1362.0
33	resnet_v1_101_tf	224x224	14.4	1448.0	3010.6
34	resnet_v1_152_tf	224x224	21.8	1207.3	2146.0
35	resnet_v1_50_tf	224x224	7	1795.4	4875.0
36	resnet_v2_101_tf	299x299	26.78	615.4	945.6
37	resnet_v2_152_tf	299x299	40.47	498.5	696.1
38	resnet_v2_50_tf	299x299	13.1	801.3	1471.3

Table 74: **VEK280 Performance with Batch 14** *(cont'd)*

No	Neural Network	Input Size	GOPS	1-thread(fps)	Multi-thread(fps)
39	resnet50_pt	224x224	4.1	1725.1	4558.2
40	resnet50_tf2	224x224	7.7	1749.1	4709.2
41	SESR_S_pt	360x640	7.48	398.9	627.5
42	squeezenet_pt	224x224	0.82	3929.0	9230.1
43	ssd_mobilenet_v1_coco_tf	300x300	2.5	906.5	1950.3
44	ssd_mobilenet_v2_coco_tf	300x300	3.8	821.679	1912.33
45	superpoint_tf	480x640	52.4	56.0723	124.5
46	unet2d_tf2	144x144	24.6	759.1	1608.5
47	vehicle_make_resnet18_pt	224x224	3.627	2585.1	6148.7
48	vehicle_type_resnet18_pt	224x224	3.627	2610.6	6294.0
49	vgg_16_tf	224x224	31	582.6	733.1
50	vgg_19_tf	224x224	39.3	572.7	718.4
51	yolov3_coco_416_tf2	416x416	65.9	271.0	532.8
52	yolov3_voc_tf	416x416	65.6	302.2	543.3
53	yolov4_csp_pt	640x640	121	78.9	117.4
54	yolov4_leaky_416_tf	416x416	60.3	197.7	365.7
55	yolov4_leaky_512_tf	512x512	91.2	139.8	240.2
56	yolov5_large_pt	640x640	109.6	135.2	268.9
57	yolov5_nano_pt	640x640	4.6	249.0	475.0
58	yolov5l_pt	640x640	109.6	94.4	163.7
59	yolov5m_pt	640x640	109.6	148.2	325.3
60	yolov5s6_pt	640x640	17	50.8	98.1
61	yolov6m_pt	640x640	82.2	39.7	51.6
62	yolov7_pt	640x640	104.8	101.8	161.7
63	yolov8m_pt	640x640	78.9	31.2	69.1
64	yolox_nano_pt	416x416x3	1	711.2	1404.8

V70 Versal Development Card

V70 is a new evaluation platform for the Versal AI data center application. It is based on the VC2802 device and is designed for high throughput AI inference and signal processing computer performance. The primary focus of the V70 card is to enable solution demos to make it easier for customers to develop their own applications.

In Vitis AI 3.5, a C20B14CU1 DPU core is implemented. It delivers 1.05G * 8192 * (20/16) * 14 = 150.5 TOPS INT8 peak performance for deep learning inference acceleration.

Refer to the following table for the throughput performance (in frames/sec or fps) for various neural network samples on the V70 card with AI Engines clocked at 1050 MHz and PL clocked at 250 MHz.

Table 75: **V70 Performance with Batch 14**

No	Neural Network	Input Size	GOPS	DPU Frequency (MHz)	Performance (fps) (Multiple thread)
1	chen_color_resnet18_pt	224x224	3.627	250	9679.80
2	efficientnet_lite_tf2	224x224	0.77	250	9842.18
3	efficientNet-edgetpu-L_tf	300x300	19.36	250	757.68
4	efficientNet-edgetpu-M_tf	240x240	7.34	250	2843.53
5	efficientNet-edgetpu-S_tf	224x224	4.72	250	3654.75
6	face_mask_detection_pt	512x512	0.593	250	2173.85
7	inception_v1_pruned_0_087_tf	224x224	2.73	250	3258.85
8	inception_v1_pruned_0_157_tf	224x224	2.52	250	3008.47
9	inception_v1_tf	224x224	3	250	3692.12
10	inception_v3_pruned_0_2_tf	299x299	9.1	250	1234.00
11	inception_v3_pruned_0_3_pt	299x299	8	250	1289.55
12	inception_v3_pruned_0_4_pt	299x299	6.8	250	1366.46
13	inception_v3_pruned_0_4_tf	299x299	6.9	250	1355.14
14	inception_v3_pruned_0_5_pt	299x299	5.7	250	1482.21
15	inception_v3_pruned_0_6_pt	299x299	4.5	250	1684.68
16	inception_v3_pt	299x299	5.7	250	1143.74
17	inception_v3_tf	299x299	11.5	250	1296.26
18	inception_v3_tf2	299x299	11.5	250	1503.39
19	inception_v4_2016_09_09_tf	299x299	24.6	250	403.08
20	inception_v4_pruned_0_2_tf	299x299	19.56	250	410.59
21	inception_v4_pruned_0_4_tf	299x299	14.79	250	441.69
22	MLPerf_resnet50_v1.5_tf	224x224	8.19	250	3792.32
23	mlperf_ssd_resnet34_tf	1200x1200	433	250	70.16
24	mobilenet_1_0_224_tf2	224x224	1.1	250	14585.10
25	mobilenet_v1_0_25_128_tf	128x128	0.027	250	63108.60
26	mobilenet_v1_1_0_224_pruned_0_11_tf	224x224	1.02	250	14905.70
27	mobilenet_v1_1_0_224_pruned_0_12_tf	224x224	1	250	14936.70
28	mobilenet_v1_1_0_224_tf	224x224	1.1	250	14581.60
29	mobilenet_v2_1_0_224_tf	224x224	0.6	250	11544.90
30	mobilenet_v2_1_4_224_tf	224x224	1.2	250	8081.16
31	movenet_ntd_pt	192x192	0.5	250	8326.82
32	ofa_depthwise_res50_pt	176x176	1.25	250	12925.70
33	ofa_rcan_latency_pt	360x640	45.7	250	53.00

Table 75: **V70 Performance with Batch 14** *(cont'd)*

No	Neural Network	Input Size	GOPS	DPU Frequency (MHz)	Performance (fps) (Multiple thread)
34	ofa_resnet50_0_9B_pt	160x160	0.9	250	7780.10
35	ofa_resnet50_pruned_0_45_pt	224x224	8.2	250	3165.36
36	ofa_resnet50_pruned_0_60_pt	224x224	6	250	3609.05
37	ofa_resnet50_pruned_0_74_pt	192x192	3.6	250	5265.23
38	ofa_yolo_pruned_0_30_pt	640x640	34.71	250	323.62
39	ofa_yolo_pruned_0_50_pt	640x640	24.62	250	456.91
40	ofa_yolo_pt	640x640	48.88	250	295.25
41	pointpillars_kitti_12000_pt	12000x100x4	10.8	250	187.11
42	rcan_pruned_tf	360x640	86.95	250	43.08
43	RefineDet-Medical_EDD_baseline_tf	320x320	81.28	250	339.95
44	RefineDet-Medical_EDD_pruned_0_5_tf	320x320	41.42	250	683.10
45	RefineDet-Medical_EDD_pruned_0_75_tf	320x320	20.54	250	1101.20
46	RefineDet-Medical_EDD_pruned_0_85_tf	320x320	12.32	250	1561.50
47	RefineDet-Medical_EDD_tf	320x320	9.8	250	1769.71
48	resnet_v1_101_pruned_0_346_tf	224x224	9.4	250	2966.77
49	resnet_v1_101_pruned_0_568_tf	224x224	6.21	250	3638.47
50	resnet_v1_101_tf	224x224	14.4	250	2542.30
51	resnet_v1_152_pruned_0_51_tf	224x224	10.68	250	2394.19
52	resnet_v1_152_pruned_0_60_tf	224x224	8.82	250	2614.17
53	resnet_v1_152_tf	224x224	21.8	250	1792.50
54	resnet_v1_50_pruned_0_38_tf	224x224	4.3	250	5038.76
55	resnet_v1_50_pruned_0_65_tf	224x224	2.45	250	6836.10
56	resnet_v1_50_tf	224x224	7	250	4329.41
57	resnet_v2_101_tf	299x299	26.78	250	760.86
58	resnet_v2_152_tf	299x299	40.47	250	564.92
59	resnet_v2_50_tf	299x299	13.1	250	1160.64
60	resnet50_pruned_0_3_pt	224x224	5.8	250	4132.65
61	resnet50_pruned_0_4_pt	224x224	4.9	250	4401.23
62	resnet50_pruned_0_5_pt	224x224	4.1	250	4671.20
63	resnet50_pruned_0_6_pt	224x224	3.3	250	5126.95
64	resnet50_pruned_0_7_pt	224x224	2.5	250	5842.50
65	resnet50_pt	224x224	4.1	250	3792.31
66	resnet50_tf2	224x224	7.7	250	3912.34
67	SESR_S_pt	360x640	7.48	250	298.31
68	squeezenet_pt	224x224	0.82	250	4500.98
69	ssd_mobilenet_v1_coco_tf	300x300	2.5	250	6100.36

Table 75: **V70 Performance with Batch 14** *(cont'd)*

No	Neural Network	Input Size	GOPS	DPU Frequency (MHz)	Performance (fps) (Multiple thread)
70	ssd_mobilenet_v2_coco_tf	300x300	3.8	250	2952.77
71	unet2d_tf2	144x144	24.6	250	1177.35
72	vehicle_make_resnet18_pt	224x224	3.627	250	9686.00
73	vehicle_type_resnet18_pt	224x224	3.627	250	9690.45
74	vgg_16_pruned_0_43_tf	224x224	17.67	250	1958.96
75	vgg_16_pruned_0_5_tf	224x224	15.64	250	2190.40
76	vgg_16_tf	224x224	31	250	619.22
77	vgg_19_pruned_0_24_tf	224x224	29.79	250	759.97
78	vgg_19_pruned_0_39_tf	224x224	23.78	250	1020.87
79	vgg_19_tf	224x224	39.3	250	691.80
80	yolov3_coco_416_tf2	416x416	65.9	250	414.54
81	yolov3_voc_tf	416x416	65.6	250	421.89
82	yolov4_csp_pt	640x640	121	250	88.08
83	yolov4_leaky_416_tf	416x416	60.3	250	232.00
84	yolov4_leaky_512_tf	512x512	91.2	250	96.77
85	yolov5_nano_pt	640x640	4.6	250	539.70
86	yolov5s6_pt	640x640	17	250	85.28
87	yolov6m_pt	640x640	82.2	250	279.67
88	yolov7_pt	640x641	104.8	250	78.08
89	yolov8m_pt	640x642	78.9	250	161.25
90	yolox_nano_pt	416x416	1	250	1253.45

Chapter 8

API Reference

vitis::ai::BCC

Base class for `BCC` (Bayesian crowd counting)

Input is an image (cv:Mat).

Output is a struct of detection results, named BCCResult.

Sample code :

```
Mat img = cv::imread("sample_BCC.jpg");
auto BCC = vitis::ai::BCC::create("bcc_pt",true);
auto result = BCC->run(img);
std::cout << result.count << "\n";
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::BCC` class:

Table 76: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< BCC >	create	const std::string & model_name bool need_preprocess
vitis::ai::BCCResult	run	const cv::Mat & img
std::vector< vitis::ai::BCCResult >	run	const std::vector< cv::Mat > & imgs
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void

Functions

create

Factory function to get an instance of derived classes of class `BCC` .

Prototype

```
std::unique_ptr<
                 BCC
               > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 77: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `BCC` class.

run

Function of get result of the `BCC` neural network.

Prototype

```
vitis::ai::BCCResult run(const cv::Mat &img)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 78: **run Arguments**

Type	Member	Description
const cv::Mat &	img	Input data of input image (cv::Mat).

Returns

BCCResult.

run

Function to get running results of the `BCC` neural network in batch mode.

Prototype

```
std::vector< vitis::ai::BCCResult > run(const std::vector< cv::Mat >
&imgs)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 79: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	imgs	Input data of input images (vector<cv::Mat>). The size of input images need equal to or less than batch size obtained by get_input_batch.

Returns

The vector of BCCResult.

getInputWidth

Function to get InputWidth of the `BCC` network (input image columns).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the `BCC` network.

getInputHeight

Function to get InputHeight of the `BCC` network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the `BCC` network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

vitis::ai::BEVdet

Base class for detecting objects in the input image(cv::Mat). Input is an image(cv::Mat). Output is the position of the objects in the input image. Sample code:

```
auto model = vitis::ai::BEVdet::create(argv[1], argv[2], argv[3]);
std::vector<std::string> names;
LoadImageNames(argv[4], names);
std::vector<cv::Mat> images;
for (auto&& i : names) {
  images.emplace_back(cv::imread(i));
}
std::vector<std::vector<char>> bins;
std::vector<std::string> bin_names;
LoadImageNames(argv[5], bin_names);
for (auto&& i : bin_names) {
  auto infile = std::ifstream(i, std::ios_base::binary);

bins.emplace_back(std::vector<char>(std::istreambuf_iterator<char>(infile),
                                    std::istreambuf_iterator<char>()));
}
std::vector<vitis::ai::CenterPointResult> res;

res = model->run(images, bins);
for (size_t i = 0; i < 32 && i < res.size(); i++) {
  const auto& r = res[i];
  cout << "label: " << r.label << " score: " << r.score
       << " bbox: " << r.bbox[0] << " " << r.bbox[1] << " " << r.bbox[2]
       << " " << r.bbox[3] << " " << r.bbox[4] << " " << r.bbox[5] << " "
       << r.bbox[6] << " " << r.bbox[7] << " " << r.bbox[8] << endl;
}
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::BEVdet` class:

Table 80: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `BEVdet` >	create	const std::string & model_name0 const std::string & model_name1 const std::string & model_name2
std::vector< `CenterPointResult` >	run	const std::vector< cv::Mat > & images const std::vector< std::vector< char > > & input_bins

Functions

create

Factory function to get an instance of derived classes of class `BEVdet` .

Prototype

```
std::unique_ptr<
                  BEVdet
                > create(const std::string &model_name0, const std::string
&model_name1, const std::string &model_name2);
```

Parameters

The following table lists the `create` function arguments.

Table 81: **create Arguments**

Type	Member	Description
const std::string &	model_name0	Model_0's name
const std::string &	model_name1	Model_1's name
const std::string &	model_name2	Model_2's name

Returns

An instance of `BEVdet` class.

run

Function to get running result of the `BEVdet` neural network.

Prototype

```
std::vector<
               CenterPointResult
             > run(const std::vector< cv::Mat > &images, const std::vector<
std::vector< char > > &input_bins)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 82: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std::vector<cv::Mat>).
const std::vector< std::vector< char > > &	input_bins	Input data of input bins (std::vector<std::vector<char>>).

Returns

A std::vector<CenterPointResult> data.

vitis::ai::C2D2_lite

Base class for detecting objects in the input image(cv::Mat). Input is an image(cv::Mat). Output is the position of the objects in the input image. Sample code:

```
std::vector<cv::Mat> images;
for (auto name : image_names) {
  images.push_back(cv::imread(name, cv::IMREAD_GRAYSCALE));
}
auto model = vitis::ai::C2D2_lite::create(C2D2_lite_0_pt, C2D2_lite_1_pt);
auto result = model->run(images);
std::cout << result;
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::C2D2_lite` class:

Table 83: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `C2D2_lite` >	create	const std::string & model_name0 const std::string & model_name1 bool need_preprocess

Table 83: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
float	run	const std::vector< cv::Mat > & image
std::vector< float >	run	const std::vector< std::vector< cv::Mat > > & images
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void

Functions

create

Factory function to get an instance of derived classes of class `C2D2_lite` .

Prototype

```
std::unique_ptr<
                C2D2_lite
              > create(const std::string &model_name0, const std::string
&model_name1, bool need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 84: **create Arguments**

Type	Member	Description
const std::string &	model_name0	Model0 name
const std::string &	model_name1	Model1 name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `C2D2_lite` class.

run

Function to get running result of the `C2D2_lite` neural network.

Prototype

```
float run(const std::vector< cv::Mat > &image)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 85: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	image	Input data of input image (std::vector<cv::Mat>).

Returns

A float data.

run

Function to get running result of the `C2D2_lite` neural network in batch mode.

Prototype

```
std::vector< float > run(const std::vector< std::vector< cv::Mat > >
&images)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 86: **run Arguments**

Type	Member	Description
const std::vector< std::vector< cv::Mat > > &	images	Input data of input images (std::vector<std::vector<cv::Mat>>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of float data.

getInputWidth

Function to get InputWidth of the `C2D2_lite` network (input image columns).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the `C2D2_lite` network

getInputHeight

Function to get InputHeight of the `C2D2_lite` network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the `C2D2_lite` network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

vitis::ai::CenterPoint

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::CenterPoint` class:

Table 87: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< CenterPoint >	create	void
int	getInputWidth	void

Table 87: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
int	getInputHeight	void
size_t	get_input_batch	void
std::vector< CenterPointResult >	run	const std::vector< float > & input
std::vector< std::vector< CenterPointResult > >	run	const std::vector< std::vector< float > > & inputs

Functions

create

Factory function to get an instance of derived classes of class CenterPoint.

value is true.

Prototype

```
std::unique_ptr< CenterPoint > create(const std::string &model_name_0,
const std::string &model_name_1);
```

Returns

An instance of CenterPoint class.

getInputWidth

Function to get InputWidth of the centerpoint network (input image columns).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the centerpoint network

getInputHeight

Function to get InputHeight of the centerpoint network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the centerpoint network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

run

Function to get result of the centerpoint network.

Prototype

```
std::vector<
                CenterPointResult
               > run(const std::vector< float > &input)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 88: **run Arguments**

Type	Member	Description
const std::vector< float > &	input	Input data of float vector.

Returns

vector of `CenterPointResult` .

run

Function to get result of the centerpoint network in batch mode.

Prototype

```
std::vector< std::vector<
               CenterPointResult
             > > run(const std::vector< std::vector< float > > &inputs)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 89: **run Arguments**

Type	Member	Description
const std::vector< std::vector< float > > &	inputs	vector of Input data of float vector.

Returns

vector of vector of `CenterPointResult` .

vitis::ai::Cflownet

Base class for `Cflownet` (production recognication)

Input is an image (cv:Mat).

Output is a struct of detected results, named CflownetResult.

Sample code :

```
Mat img = cv::imread("sample_Cflownet.jpg");
auto Cflownet = vitis::ai::Cflownet::create("bosch_fcnsemsegt",true);
auto result = Cflownet->run(img);
std::cout << result.width <<"\n";
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::Cflownet` class:

Table 90: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `Cflownet` >	create	const std::string & model_name bool need_preprocess
vitis::ai::CflownetResult	run	const float * p
std::vector< vitis::ai::CflownetResult >	run	const std::vector< const float * > ps
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void

Functions

create

Factory function to get an instance of derived classes of class `Cflownet` .

Prototype

```
std::unique_ptr<
              Cflownet
            > create(const std::string &model_name, bool
need_preprocess=false);
```

Parameters

The following table lists the `create` function arguments.

Table 91: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `Cflownet` class.

run

Function of get result of the `Cflownet` network.

Prototype

```
vitis::ai::CflownetResult run(const float *p)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 92: **run Arguments**

Type	Member	Description
const float *	p	const float pointer points to input data buffer.

Returns

CflownetResult.

run

Function to get running results of the `Cflownet` network in batch mode.

Prototype

```
std::vector< vitis::ai::CflownetResult > run(const std::vector< const float
* > ps)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 93: **run Arguments**

Type	Member	Description
const std::vector< const float * >	ps	const vector of float pointer points to input data buffer. The size of input images need equal to or less than batch size obtained by get_input_batch.

Returns

The vector of CflownetResult.

getInputWidth

Function to get InputWidth of the `Cflownet` network (input image columns).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the `Cflownet` network.

getInputHeight

Function to get InputHeight of the `Cflownet` network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the `Cflownet` network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

***Note*:** Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

vitis::ai::Classification

Base class for detecting objects in the input image (cv::Mat).

Input is an image (cv::Mat).

Output is index and score of objects in the input image.

Sample code:

```
auto image = cv::imread("sample_classification.jpg");
auto network = vitis::ai::Classification::create(
               "resnet50");
auto result = network->run(image);
for (const auto &r : result.scores) {
   auto score = r.score;
   auto index = result.lookup(r.index);
}
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::Classification` class:

Table 94: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `Classification` >	create	const std::string & model_name bool need_preprocess
`vitis::ai::ClassificationResult`	run	const cv::Mat & image
std::vector< `vitis::ai::ClassificationResult` >	run	const std::vector< cv::Mat > & images
std::vector< `vitis::ai::ClassificationResult` >	run	const std::vector< vart::xrt_bo_t > & input_bos

Functions

create

Factory function to get an instance of derived classes of class `Classification` .

Prototype

```
std::unique_ptr<
               Classification
             > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 95: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name.
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `Classification` class.

run

Function to get running results of the classification neural network.

Prototype

```


                vitis::ai::ClassificationResult
              run(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 96: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of input image (cv::Mat).

Returns

`ClassificationResult`.

run

Function to get running results of the classification neural network in batch mode.

Prototype

```
std::vector<
                vitis::ai::ClassificationResult
              > run(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 97: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of batch input images (vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `ClassificationResult` .

run

Function to get running results of the classification neural network in batch mode , used to receive user's xrt_bo to support zero copy.

Prototype

```
std::vector<
              vitis::ai::ClassificationResult
            > run(const std::vector< vart::xrt_bo_t > &input_bos)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 98: **run Arguments**

Type	Member	Description
const std::vector< vart::xrt_bo_t > &	input_bos	The vector of vart::xrt_bo_t.

Returns

The vector of ClassifcationResult.

vitis::ai::Clocs

Base class for clocs.

Input is points data and related params.

Output is a struct of detection results, named ClocsResult.

Sample code :

```
    ...
    std::string yolo_model_name = "clocs_yolox_pt";
    std::string pp_model_0 = "clocs_pointpillars_kitti_0_pt";
    std::string pp_model_1 = "clocs_pointpillars_kitti_1_pt";
    std::string fusion_model_name = "clocs_fusion_cnn_pt";

    auto clocs = vitis::ai::Clocs::create(yolo_model_name, pp_model_0,
pp_model_1, fusion_model_name, true);


    vector<ClocsInfo> batch_clocs_info(input_num);
    // see the test sample to read ClocsInfo
    //
    auto batch_ret = clocs->run(batch_clocs_info);

    ...
  please see the test sample for detail.
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::Clocs` class:

Table 99: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `Clocs` >	create	const std::string & yolo const std::string & pointpillars_0 const std::string & pointpillars_1 const std::string & fusionnet bool need_preprocess
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void
int	getPointsDim	void
ClocsResult	run	const `clocs::ClocsInfo` & input
ClocsResult	run	const std::vector< float > & detect2d_result const `clocs::ClocsInfo` & input
std::vector< ClocsResult >	run	const std::vector< `clocs::ClocsInfo` > & batch_inputs

Table 99: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
std::vector< ClocsResult >	run	const std::vector< std::vector< float > > & batch_detect2d_result const std::vector< `clocs::ClocsInfo` > & batch_inputs

Functions

create

Factory function to get an instance of derived classes of class `Clocs` .

Prototype

```
std::unique_ptr<
                Clocs
              > create(const std::string &yolo, const std::string
&pointpillars_0, const std::string &pointpillars_1, const std::string
&fusionnet, bool need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 100: **create Arguments**

Type	Member	Description
const std::string &	yolo	The yolo model name
const std::string &	pointpillars_0	The pointpillars 0 model name
const std::string &	pointpillars_1	The pointpillars 1 model name
const std::string &	fusionnet	The funsion model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of ClocsPointPillars class.

getInputWidth

Function to get input width of the first model of `Clocs` class.

Prototype

```
int getInputWidth() const =0;
```

Returns

Input width of the first model.

getInputHeight

Function to get input height of the first model of `Clocs` class.

Prototype

```
int getInputHeight() const =0;
```

Returns

Input height of the first model.

get_input_batch

Function to get the number of inputs processed by the DPU at one time.

Note: Batch size of different DPU core may be different, it depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

getPointsDim

Function to get the points dim of an input points. The dim depends on the last channel of the first model of `Clocs` .

Prototype

```
int getPointsDim() const =0;
```

Returns

The dim of points.

run

Function of get result of the `Clocs` class.

Prototype

```
ClocsResult run(const clocs::ClocsInfo &input)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 101: **run Arguments**

Type	Member	Description
const `clocs::ClocsInfo` &	input	`Clocs` input info.

Returns

ClocsResult.

run

Function of get result of the `Clocs` class with prepared 2d result, This api is only for debug.

Prototype

```
ClocsResult run(const std::vector< float > &detect2d_result, const
clocs::ClocsInfo &input)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 102: **run Arguments**

Type	Member	Description
const std::vector< float > &	detect2d_result	preloaded 2d result.
const `clocs::ClocsInfo` &	input	`Clocs` input info.

Returns

ClocsResult.

run

Function of get result of the `Clocs` class in batch mode.

Prototype

```
std::vector< ClocsResult > run(const std::vector< clocs::ClocsInfo >
&batch_inputs)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 103: **run Arguments**

Type	Member	Description
const std::vector< `clocs::ClocsInfo` > &	batch_inputs	`Clocs` input infos.

Returns

The vector of ClocsResult.

run

Function of get result of the `Clocs` class in batch mode. This api is only for debug.

Prototype

```
std::vector< ClocsResult > run(const std::vector< std::vector< float > >
&batch_detect2d_result, const std::vector< clocs::ClocsInfo >
&batch_inputs)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 104: **run Arguments**

Type	Member	Description
const std::vector< std::vector< float > > &	batch_detect2d_result	preloaded 2d results.
const std::vector< `clocs::ClocsInfo` > &	batch_inputs	`Clocs` input infos.

Returns

The vector of ClocsResult.

vitis::ai::Covid19Segmentation

Base class for `Covid19Segmentation`.

Declaration `Covid19Segmentation` Network Branch positive detection: label0-negative, label1-positive Branch Infected area detection: label0-negative, label1-consolidation, label2-GGO

Input is an image (cv:Mat).

Output is result of running the `Covid19Segmentation` network.

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::Covid19Segmentation` class:

Table 105: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `Covid19Segmentation` >	create	const std::string & model_name bool need_preprocess
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void
Covid19SegmentationResult	run_8UC1	const cv::Mat & image
std::vector< Covid19SegmentationResult >	run_8UC1	const std::vector< cv::Mat > & images
Covid19SegmentationResult	run_8UC3	const cv::Mat & image
std::vector< Covid19SegmentationResult >	run_8UC3	const std::vector< cv::Mat > & images

Functions

create

Factory function to get an instance of derived classes of class `Covid19Segmentation`.

Prototype

```
std::unique_ptr<
                Covid19Segmentation
              > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 106: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `Covid19Segmentation` class.

getInputWidth

Function to get InputWidth of the covid19segmentation network (input image columns).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the covid19segmentation network.

getInputHeight

Function to get InputHeight of the covid19segmentation network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the covid19segmentation network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

run_8UC1

Function to get running result of the covid19segmentation network.

Note: The type of CV_8UC1 of the covid19segmentation result.

Prototype

```
Covid19SegmentationResult run_8UC1(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run_8UC1` function arguments.

Table 107: **run_8UC1 Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of input image (cv::Mat).

Returns

Covid19segmentation output data.

run_8UC1

Function to get running results of the covid19segmentation neural network in batch mode.

Note: The type of CV_8UC1 of the covid19segmentation result.

Prototype

```
std::vector< Covid19SegmentationResult > run_8UC1(const std::vector<
cv::Mat > &images)=0;
```

Parameters

The following table lists the `run_8UC1` function arguments.

Table 108: **run_8UC1 Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of Covid19segmentationResult.

run_8UC3

Function to get running result of the covid19segmentation network.

***Note*:** The type of CV_8UC3 of the covid19segmentation result.

Prototype

```
Covid19SegmentationResult run_8UC3(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run_8UC3` function arguments.

Table 109: **run_8UC3 Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of input image (cv::Mat).

Returns

Covid19segmentation image and shape.

run_8UC3

Function to get running results of the covid19segmentation neural network in batch mode.

***Note*:** The type of CV_8UC3 of the Result's covid19segmentation.

Prototype

```
std::vector< Covid19SegmentationResult > run_8UC3(const std::vector<
cv::Mat > &images)=0;
```

Parameters

The following table lists the `run_8UC3` function arguments.

Table 110: **run_8UC3 Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of Covid19SegmentationResult.

vitis::ai::Covid19Segmentation8UC1

The Class of `Covid19Segmentation8UC1`, this class run function return a cv::Mat with the type is cv_8UC1.

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::Covid19Segmentation8UC1` class:

Table 111: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `Covid19Segmentation8UC1` >	create	const std::string & model_name bool need_preprocess
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void
Covid19SegmentationResult	run	const cv::Mat & image

Table 111: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
std::vector< Covid19SegmentationResult >	run	const std::vector< cv::Mat > & images

Functions

create

Factory function to get an instance of derived classes of class `Covid19Segmentation8UC1` .

Prototype

```
std::unique_ptr<
                Covid19Segmentation8UC1
              > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 112: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `Covid19Segmentation8UC1` class.

getInputWidth

Function to get InputWidth of the covid19segmentation network (input image columns).

Prototype

```
int getInputWidth() const;
```

Returns

InputWidth of the covid19segmentation network.

getInputHeight

Function to get InputHeight of the covid19segmentation network (input image rows).

Prototype

```
int getInputHeight() const;
```

Returns

InputHeight of the covid19segmentation network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: Different DPU core the batch size may be differnt. This depends on the IP used.

Prototype

```
size_t get_input_batch() const;
```

Returns

Batch size.

run

Function to get running result of the covid19segmentation network.

Note: The result cv::Mat of the type is CV_8UC1.

Prototype

```
Covid19SegmentationResult run(const cv::Mat &image);
```

Parameters

The following table lists the `run` function arguments.

Table 113: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of the image (cv::Mat)

Returns

The result of covid19segmentation network.

run

Function to get running results of the covid19segmentation neural network in batch mode.

Note: The type of CV_8UC1 of the Result's covid19segmentation.

Prototype

```
std::vector< Covid19SegmentationResult > run(const std::vector< cv::Mat >
&images);
```

Parameters

The following table lists the `run` function arguments.

Table 114: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of Covid19SegmentationResult.

vitis::ai::Covid19Segmentation8UC3

The Class of `Covid19Segmentation8UC3`, this class run function return a cv::Mat with the type is cv_8UC3.

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::Covid19Segmentation8UC3` class:

Table 115: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `Covid19Segmentation8UC3` >	create	const std::string & model_name bool need_preprocess

Table 115: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void
Covid19SegmentationResult	run	const cv::Mat & image
std::vector< Covid19SegmentationResult >	run	const std::vector< cv::Mat > & images

Functions

create

Factory function to get an instance of derived classes of class `Covid19Segmentation8UC3` .

Prototype

```
std::unique_ptr<
                Covid19Segmentation8UC3
              > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 116: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `Covid19Segmentation8UC3` class.

getInputWidth

Function to get InputWidth of the covid19segmentation network (input image columns).

Prototype

```
int getInputWidth() const;
```

Returns

InputWidth of the covid19segmentation network.

getInputHeight

Function to get InputWidth of the covid19segmentation network (input image rows).

Prototype

```
int getInputHeight() const;
```

Returns

InputWidth of the covid19segmentation network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const;
```

Returns

Batch size.

run

Function to get running result of the covid19segmentation network.

Note: The result cv::Mat of the type is CV_8UC3.

Prototype

```
Covid19SegmentationResult run(const cv::Mat &image);
```

Parameters

The following table lists the `run` function arguments.

Table 117: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of the image (cv::Mat)

Returns

Covid19SegmentationResult The result of covid19segmentation network.

run

Function to get running results of the covid19segmentation neural network in batch mode.

***Note*:** The type of CV_8UC3 of the Result's covid19segmentation.

Prototype

```
std::vector< Covid19SegmentationResult > run(const std::vector< cv::Mat >
&images);
```

Parameters

The following table lists the `run` function arguments.

Table 118: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of Covid19SegmentationResult.

vitis::ai::EfficientDetD2

Base class for detecting position of vehicle, pedestrian, and so on.

Input is an image (cv:Mat).

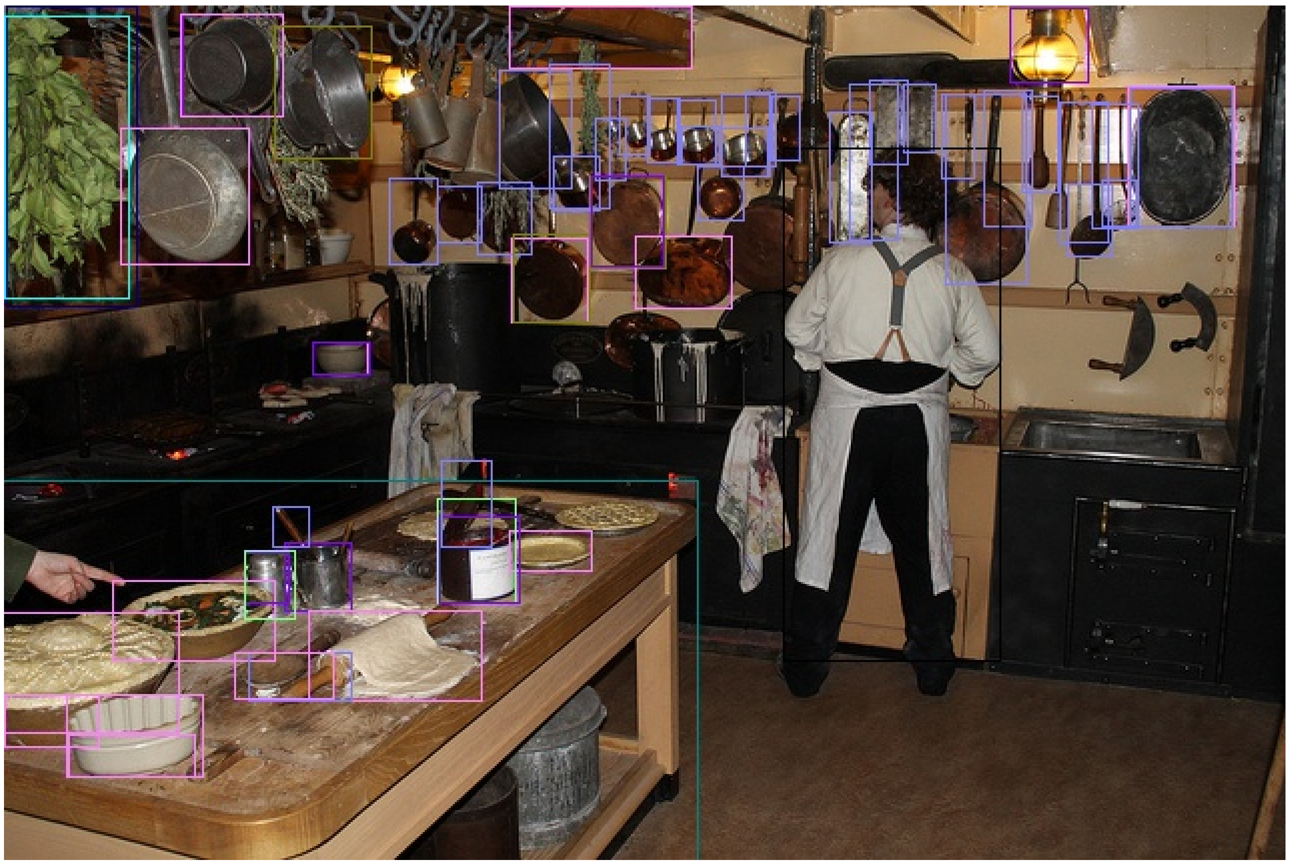
Output is a struct of detection results, named `EfficientDetD2Result`.

Sample code :

```
  Mat img = cv::imread("sample_efficientdet_d2.jpg");
  auto efficientdet_d2 =
vitis::ai::EfficientDetD2::create("efficientdet_d2_tf",true);
  auto results = efficientdet_d2->run(img);
  for(const auto &r : results.bboxes){
     auto label = r.label;
     auto x = r.x * img.cols;
     auto y = r.y * img.rows;
     auto width = r.width * img.cols;
     auto heigth = r.height * img.rows;
     auto score = r.score;
     std::cout << "RESULT: " << label << "\t" << x << "\t" << y << "\t" <<
width
        << "\t" << height << "\t" << score << std::endl;
  }
```

Display of the model results: width=\textwidth

Figure 57: **detection result**



Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::EfficientDetD2` class:

Table 119: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< EfficientDetD2 >	create	const std::string & model_name bool need_preprocess
std::unique_ptr< EfficientDetD2 >	create	const std::string & model_name xir::Attrs * attrs bool need_preprocess
vitis::ai::EfficientDetD2Result	run	const cv::Mat & image
std::vector< vitis::ai::EfficientDetD2Result >	run	const std::vector< cv::Mat > & images

Functions

create

Factory function to get an instance of derived classes of class `EfficientDetD2` .

Prototype

```
std::unique_ptr<
                  EfficientDetD2
                > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 120: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `EfficientDetD2` class.

create

Factory function to get an instance of derived classes of class `EfficientDetD2` .

Prototype

```
std::unique_ptr<
               EfficientDetD2
             > create(const std::string &model_name, xir::Attrs *attrs,
bool need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 121: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
xir::Attrs *	attrs	Xir attributes
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `EfficientDetD2` class.

run

Function to get running results of the `EfficientDetD2` neural network.

Prototype

```


               vitis::ai::EfficientDetD2Result
             run(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 122: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of input image (cv::Mat).

Returns

`EfficientDetD2Result`.

run

Function to get running results of the `EfficientDetD2` neural network in batch mode.

Prototype

```
std::vector<
                vitis::ai::EfficientDetD2Result
              > run(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 123: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (vector<cv::Mat>).The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `EfficientDetD2Result` .

vitis::ai::EfficientDetD2PostProcess

Class of the `EfficientDetD2` post-process. It initializes the parameters once instead of computing them each time the program executes.

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::EfficientDetD2PostProcess` class:

Table 124: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `EfficientDetD2PostProcess` >	create	const std::vector< vitis::ai::library::InputTensor > & input_tensors const std::vector< vitis::ai::library::OutputTensor > & output_tensors const vitis::ai::proto::DpuModelParam & config

Table 124: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
std::vector< EfficientDetD2Result >	postprocess	size_t batch_size const std::vector< float > & image_scales const std::vector< int > & swidths const std::vector< int > & sheights

Functions

create

Create an `EfficientDetD2PostProcess` object.

Prototype

```
std::unique_ptr<
               EfficientDetD2PostProcess
             > create(const std::vector< vitis::ai::library::InputTensor >
&input_tensors, const std::vector< vitis::ai::library::OutputTensor >
&output_tensors, const vitis::ai::proto::DpuModelParam &config);
```

Parameters

The following table lists the `create` function arguments.

Table 125: **create Arguments**

Type	Member	Description
const std::vector< vitis::ai::library::InputTensor > &	input_tensors	A vector of all input-tensors in the network. Usage: input_tensors[input_tensor_index].
const std::vector< vitis::ai::library::OutputTensor > &	output_tensors	A vector of all output-tensors in the network. Usage: output_tensors[output_index].
const vitis::ai::proto::DpuModelParam &	config	The DPU model configuration information.

Returns

An unique pointer of `EfficientDetD2PostProcess` .

postprocess

The batch mode post-processing function of the `EfficientDetD2` network.

Prototype

```
std::vector<
               EfficientDetD2Result
             > postprocess(size_t batch_size, const std::vector< int >
&swidths, const std::vector< int > &sheights, const std::vector< float >
&image_scales)=0;
```

Parameters

The following table lists the `postprocess` function arguments.

Table 126: **postprocess Arguments**

Type	Member	Description
size_t	batch_size	num of batch input
const std::vector< float > &	image_scales	image scale to fit the network input size
const std::vector< int > &	swidths	batch input image widths
const std::vector< int > &	sheights	batch input image heights

Returns

The vector of struct of `EfficientDetD2Result`.

vitis::ai::FaceDetect

Base class for detecting the position of faces in the input image (cv::Mat).

Input is an image (cv::Mat).

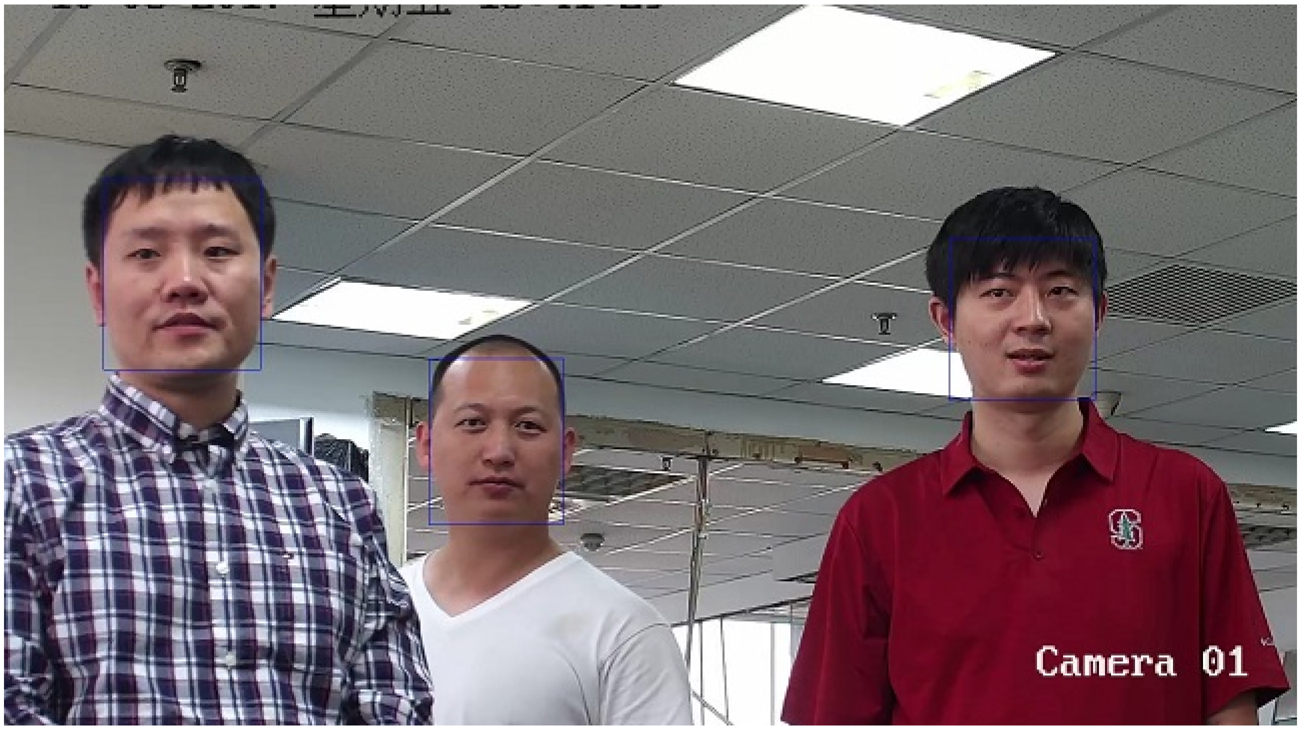
Output is a vector of position and score for faces in the input image.

Sample code:

```
auto image = cv::imread("sample_facedetect.jpg");
auto network = vitis::ai::FaceDetect::create(
               "densebox_640_360",
               true);
auto result = network->run(image);
for (const auto &r : result.rects) {
   auto score = r.score;
   auto x = r.x * image.cols;
   auto y = r.y * image.rows;
   auto width = r.width * image.cols;
   auto height = r.height * image.rows;
}
```

Display of the model results: width=\textwidth

Figure 58: **result image**



Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::FaceDetect` class:

Table 127: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `FaceDetect` >	create	const std::string & model_name bool need_preprocess
std::unique_ptr< `FaceDetect` >	create	const std::string & model_name xir::Attrs * attrs bool need_preprocess
float	getThreshold	void
void	setThreshold	float threshold
`FaceDetectResult`	run	const cv::Mat & img
std::vector< `FaceDetectResult` >	run	const std::vector< cv::Mat > & imgs

Table 127: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
std::vector< FaceDetectResult >	run	const std::vector< vart::xrt_bo_t > & input_bos

Functions

create

Factory function to get instance of derived classes of class FaceDetect .

Prototype

```
std::unique_ptr<
                FaceDetect
              > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 128: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of FaceDetect class.

create

Factory function to get instance of derived classes of class FaceDetect .

Prototype

```
std::unique_ptr<
                FaceDetect
              > create(const std::string &model_name, xir::Attrs *attrs,
bool need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 129: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
xir::Attrs *	attrs	Xir attributes
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `FaceDetect` class.

getThreshold

Function to get detect threshold.

Prototype

```
float getThreshold() const =0;
```

Returns

The detect threshold. The value ranges from 0 to 1.0f.

setThreshold

Function of update detect threshold.

***Note*:** The detection results will filter by detect threshold (score >= threshold).

Prototype

```
void setThreshold(float threshold)=0;
```

Parameters

The following table lists the `setThreshold` function arguments.

Table 130: **setThreshold Arguments**

Type	Member	Description
float	threshold	The detect threshold. The value ranges from 0 to 1.0f.

Returns

run

Function to get running result of the facedetect network.

Prototype

```
                    FaceDetectResult
                 run(const cv::Mat &img)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 131: **run Arguments**

Type	Member	Description
const cv::Mat &	img	Input Data ,input image (cv::Mat) need to be resized to InputWidth and InputHeight required by the network.

Returns

The detection result of the face detect network, filtered by score >= det_threshold

run

Function to get running results of the facedetect neural network in batch mode.

Prototype

```
std::vector<
                    FaceDetectResult
                 > run(const std::vector< cv::Mat > &imgs)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 132: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	imgs	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch. The input images need to be resized to InputWidth and InputHeight required by the network.

Returns

The vector of `FaceDetectResult` .

run

Function to get running results of the facedetect neural network in batch mode , used to receive user's xrt_bo to support zero copy.

Prototype

```
std::vector<
               FaceDetectResult
              > run(const std::vector< vart::xrt_bo_t > &input_bos)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 133: **run Arguments**

Type	Member	Description
const std::vector< vart::xrt_bo_t > &	input_bos	The vector of vart::xrt_bo_t.

Returns

The vector of `FaceDetectResult` .

vitis::ai::FaceFeature

Base class for getting the features of a face image (cv::Mat).

Input is a face image (cv::Mat).

Output is the features of a face in the input image.

Float sample code :

Note: Two interfaces are provided to get the float features or fixed features. They return `FaceFeatureFloatResult` or `FaceFeatureFixedResult`.

```
cv:Mat image = cv::imread("test_face.jpg");
auto network  = vitis::ai::FaceFeature::create("facerec_resnet20", true);
auto result = network->run(image);
auto features = result.feature;
```

Fixed sample code :

```
cv:Mat image = cv::imread("test_face.jpg");
auto network  = vitis::ai::FaceFeature::create("facerec_resnet20", true);
auto result = network->run_fixed(image);
auto features = result.feature;
```

Similarity calculation formula :

Calaculate the similarity of two images:

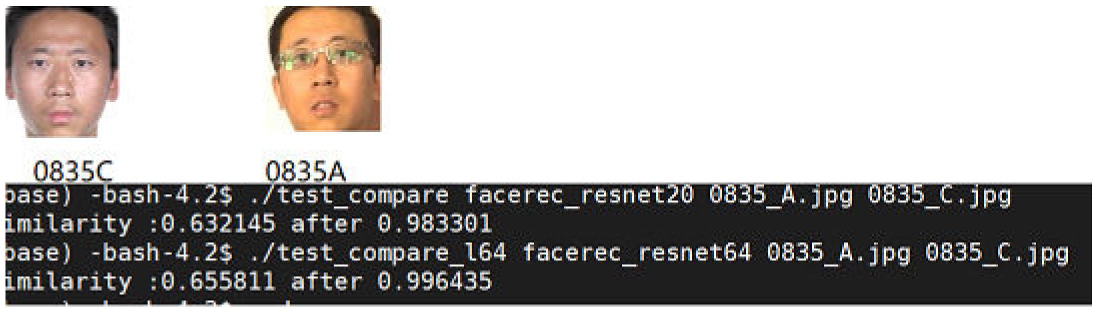
```
auto result_fixed = network->run_fixed(image);
auto result_fixed2 = network->run_fixed(image2);
auto similarity_original = feature_compare(result_fixed.feature->data(),
                                 result_fixed2.feature->data());
float similarity_mapped = score_map(similarity_original);
```

Fixed compare code :

```
  float feature_norm(const int8_t *feature) {
     int sum = 0;
     for (int i = 0; i < 512; ++i) {
         sum += feature[i] * feature[i];
     }
     return 1.f / sqrt(sum);
  }

 static float feature_dot(const int8_t *f1, const int8_t *f2) {
    int dot = 0;
    for (int i = 0; i < 512; ++i) {
       dot += f1[i] * f2[i];
    }
    return (float)dot;
 }

 float feature_compare(const int8_t *feature, const int8_t *feature_lib){
    float norm = feature_norm(feature);
    float feature_norm_lib = feature_norm(feature_lib);
    return feature_dot(feature, feature_lib) * norm * feature_norm_lib;
 }

 float score_map_l20(float score) { return 1.0 / (1 + exp(-12.4 * score
+ 3.763)); }

 float score_map_l64(float score) { return 1.0 / (1 + exp(-17.0836 * score
+ 5.5707)); }
```

Display of the compare result with a set of images: width=\textwidth

Figure 59: **facecompare result image**



```
base) -bash-4.2$ ./test_compare facerec_resnet20 0835_A.jpg 0835_C.jpg
imilarity :0.632145 after 0.983301
base) -bash-4.2$ ./test_compare_l64 facerec_resnet64 0835_A.jpg 0835_C.jpg
imilarity :0.655811 after 0.996435
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::FaceFeature` class:

Table 134: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `FaceFeature` >	create	const std::string & model_name bool need_preprocess
std::unique_ptr< `FaceFeature` >	create	const std::string & model_name xir::Attrs * attrs bool need_preprocess
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void
`FaceFeatureFloatResult`	run	const cv::Mat & img
`FaceFeatureFixedResult`	run_fixed	const cv::Mat & img
std::vector< `FaceFeatureFloatResult` >	run	const std::vector< cv::Mat > & imgs
std::vector< `FaceFeatureFixedResult` >	run_fixed	const std::vector< cv::Mat > & imgs

Functions

create

Factory function to get an instance of derived classes of class `FaceFeature` .

Prototype

```
std::unique_ptr<
                  FaceFeature
                > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 135: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not. Default value is true.

Returns

An instance of `FaceFeature` class.

create

Factory function to get an instance of derived classes of class `FaceFeature` .

Prototype

```
std::unique_ptr<
                  FaceFeature
                > create(const std::string &model_name, xir::Attrs *attrs,
bool need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 136: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
xir::Attrs *	attrs	Xir attributes

Table 136: **create Arguments** *(cont'd)*

Type	Member	Description
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `FaceFeature` class.

getInputWidth

Function to get InputWidth of the feature network (input image columns).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the feature network.

getInputHeight

Function to get InputHeight of the feature network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the feature network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

***Note*:** Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

run

Function of get running result of the feature network.

Prototype

```
FaceFeatureFloatResult
run(const cv::Mat &img)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 137: **run Arguments**

Type	Member	Description
const cv::Mat &	img	Input data for image (cv::Mat) detected by the facedetect network and then rotated and aligned.

Returns

`FaceFeatureFloatResult`

run_fixed

Function of get running result of the feature network.

Prototype

```
FaceFeatureFixedResult
run_fixed(const cv::Mat &img)=0;
```

Parameters

The following table lists the `run_fixed` function arguments.

Table 138: **run_fixed Arguments**

Type	Member	Description
const cv::Mat &	img	Input data for image (cv::Mat) detected by the facedetect network and then rotated and aligned.

Returns

`FaceFeatureFixedResult`

run

Function of get running result of the feature network in batch mode.

Prototype

```
std::vector<
                FaceFeatureFloatResult
            > run(const std::vector< cv::Mat > &imgs)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 139: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	imgs	Input data of batch input images (vector<cv::Mat>) detected by the facedetect network and then rotated and aligned. The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `FaceFeatureFloatResult` .

run_fixed

Function of get running result of the feature network in batch mode.

Prototype

```
std::vector<
                FaceFeatureFixedResult
            > run_fixed(const std::vector< cv::Mat > &imgs)=0;
```

Parameters

The following table lists the `run_fixed` function arguments.

Table 140: **run_fixed Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	imgs	Input data of batch input images (vector<cv::Mat>) detected by the facedetect network and then rotated and aligned. The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `FaceFeatureFixedResult` .

vitis::ai::FaceLandmark

Base class for detecting five key points, and the score from a face image (cv::Mat).

Input a face image (cv::Mat).

Output score, five key points of the face.

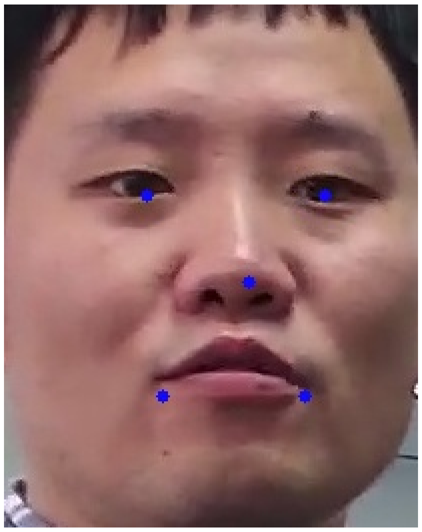
Sample code:

***Note*:** Usually the input image contains only one face. When it contains multiple faces, the function returns the highest score.

```
cv:Mat image = cv::imread("sample_facelandmark.jpg");
auto landmark  = vitis::ai::FaceLandmark::create("face_landmark");
auto result = landmark->run(image);
float score = result.score;
auto points = result.points;
for(int i = 0; i< 5 ; ++i){
    auto x = points[i].frist  * image.cols;
    auto y = points[i].second * image.rows;
}
```

Display of the model results:

Figure 60: **result image**



Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::FaceLandmark` class:

Table 141: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< FaceLandmark >	create	const std::string & model_name bool need_preprocess
std::unique_ptr< FaceLandmark >	create	const std::string & model_name xir::Attrs * attrs bool need_preprocess
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void
FaceLandmarkResult	run	const cv::Mat & input_image
std::vector< FaceLandmarkResult >	run	const std::vector< cv::Mat > & input_images

Functions

create

Factory function to get an instance of derived classes of class `FaceLandmark` .

Prototype

```
std::unique_ptr<
                FaceLandmark
              > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 142: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `FaceLandmark` class.

create

Factory function to get an instance of derived classes of class `FaceLandmark` .

Prototype

```
std::unique_ptr<
                FaceLandmark
              > create(const std::string &model_name, xir::Attrs *attrs,
bool need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 143: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
xir::Attrs *	attrs	Xir attributes
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `FaceLandmark` class.

getInputWidth

Function to get InputWidth of the landmark network (input image columns).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the face landmark network.

getInputHeight

Function to get InputHeight of the landmark network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the face landmark network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

run

Function to get running result of the face landmark network.

Set data of a face(e.g. data of cv::Mat) and get the five key points.

Prototype

```
                      FaceLandmarkResult
                    run(const cv::Mat &input_image)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 144: **run Arguments**

Type	Member	Description
const cv::Mat &	input_image	Input data of input image (cv::Mat) of detected by the facedetect network and resized as inputwidth and inputheight.

Returns

The struct of `FaceLandmarkResult`

run

Function to get running results of the face landmark neural network in batch mode.

Prototype

```
std::vector<
                FaceLandmarkResult
              > run(const std::vector< cv::Mat > &input_images)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 145: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	input_images	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch. The input images need to be resized to InputWidth and InputHeight required by the network.

Returns

The vector of `FaceLandmarkResult` .

vitis::ai::FaceQuality5pt

Base class for evaluating the quality and five key points coordinate of a face image (cv::Mat).

Input is a face image (cv::Mat).

Output is the quality and five key points coordinate of a face in the input image.

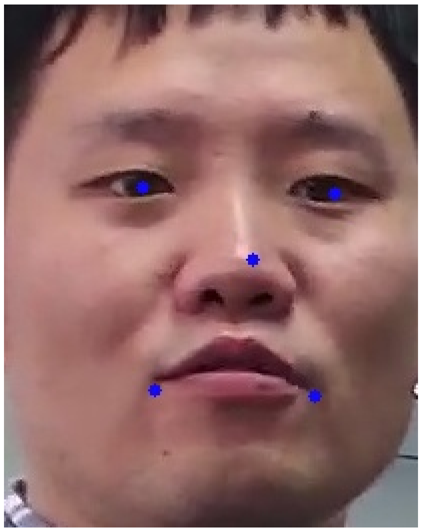
Sample code : Display of the `FaceQuality5pt` model results: width=\textwidth

```
cv:Mat image = cv::imread("sample_facequality5pt.jpg");
auto network =
      vitis::ai::FaceQuality5pt::create("face-quality", true);
auto result = network->run(image);
auto quality = result.score;
auto points = result.points;
for(int i = 0; i< 5 ; ++i){
    auto x = points[i].frist  * image.cols;
    auto y = points[j].second * image.rows;
}
```

Note: Default mode is day, if day night switch network is used and the background of the input image is night, please use API setMode

```
network->setMode(vitis::ai::FaceQuality5pt::Mode::NIGHT);
```

Figure 61: **result image**



Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::FaceQuality5pt` class:

Table 146: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `FaceQuality5pt` >	create	const std::string & model_name bool need_preprocess
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void
`Mode`	getMode	void
void	setMode	`Mode` mode
`FaceQuality5ptResult`	run	const cv::Mat & img

Table 146: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
std::vector< FaceQuality5ptResult >	run	const std::vector< cv::Mat > & images

Functions

create

Factory function to get an instance of derived classes of class `FaceQuality5pt` .

Prototype

```
std::unique_ptr<
                FaceQuality5pt
              > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 147: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `FaceQuality5pt` class.

getInputWidth

Function to get InputWidth of the facequality5pt network (input image columns).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the facequality5pt network.

getInputHeight

Function to get InputHeight of the facequality5pt network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of facequality5pt network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

getMode

Function to get Mode.

Prototype

```
Mode
getMode()=0;
```

setMode

Function to set Mode.

Prototype

```
void setMode(Mode mode)=0;
```

Parameters

The following table lists the `setMode` function arguments.

Table 148: **setMode Arguments**

Type	Member	Description
Mode	mode	Type::Mode

Returns

mode

run

Function of get running result of the facequality5pt network.

Prototype

```
                     FaceQuality5ptResult
                   run(const cv::Mat &img)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 149: **run Arguments**

Type	Member	Description
const cv::Mat &	img	Input data of input image (cv::Mat) of detected counterpart and resized to InputWidth and InputHeight required by the network.

Returns

The result of the facequality5pt network.

run

Function of get running results of the facequality5pt network in batch mode.

Prototype

```
std::vector<
                     FaceQuality5ptResult
                   > run(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 150: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std::vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch. The input images need to be resized to InputWidth and InputHeight required by the network.

Returns

The vector of the `FaceQuality5ptResult` .

Enumerations

Enumeration Mode

Scene of sending image.

Table 151: **Enumeration Mode Values**

Value	Description
DAY	Use DAY when the background of the image is daytime.
NIGHT	Use NIGHT when the background of the image is night.

vitis::ai::FairMot

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::FairMot` class:

Table 152: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< FairMot >	create	const std::string & model_name bool need_preprocess
`FairMotResult`	run	const cv::Mat & image
std::vector< `FairMotResult` >	run	const std::vector< cv::Mat > & images

Functions

create

Factory function to get an instance of derived classes of class FairMot.

Prototype

```
std::unique_ptr< FairMot > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 153: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of FairMot class.

run

Function to get running result of the FAIRMOT neural network.

Prototype

```
                FairMotResult
              run(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 154: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of input image (cv::Mat).

Returns

FairMotResult.

run

Function to get running result of the FAIRMOT neural network in batch mode.

Prototype

```
std::vector<
              FairMotResult
            > run(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 155: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (vector<cv::Mat>).

Returns

vector of FairMotResult.

vitis::ai::GraphRunner

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::GraphRunner` class:

Table 156: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< vart::RunnerExt >	create_graph_runner	const xir::Graph * graph xir::Attrs * attrs

Functions

create_graph_runner

Factory fucntion to create an instance of runner by graph and attributes.

Usage:

```
auto graph = xir::Graph::deserialize(xmodel_file);
auto attrs = xir::Attrs::create();
auto runner = vitis::ai::GraphRunner::create_graph_runner(graph.get(),
attrs.get());
auto input_tensor_buffers = runner->get_inputs();
```

Graph runner Example

Sample code:

```
// The way to create graph runner and the APIs usage of runner are shown
below.
auto graph = xir::Graph::deserialize(xmodel_file);
auto attrs = xir::Attrs::create();
auto runner = vitis::ai::GraphRunner::create_graph_runner(graph.get(),
attrs.get());
// get input and output tensor buffers
auto input_tensor_buffers = runner->get_inputs();
auto output_tensor_buffers = runner->get_outputs();
// sync input tensor buffers
for (auto& input : input_tensor_buffers) { input->sync_for_write(0, input-
>get_tensor()->get_data_size() / input->get_tensor()->get_shape()[0]);
}
// run graph runner
auto v = runner->execute_async(input_tensor_buffers, output_tensor_buffers);
auto status = runner->wait((int)v.first, 1000000000);
// sync output tensor buffers
for (auto& output : output_tensor_buffers) { output->sync_for_read(0,
output->get_tensor()->get_data_size() / output->get_tensor()->get_shape()
[0]);
}
```

Prototype

```
std::unique_ptr< vart::RunnerExt > create_graph_runner(const xir::Graph
*graph, xir::Attrs *attrs);
```

Parameters

The following table lists the `create_graph_runner` function arguments.

Table 157: **create_graph_runner Arguments**

Type	Member	Description
const xir::Graph *	graph	XIR Graph

Table 157: **create_graph_runner Arguments** *(cont'd)*

Type	Member	Description
xir::Attrs *	attrs	XIR attrs object, this object is shared among all runners on the same graph.

Returns

An instance of runner.

vitis::ai::Hourglass

`Hourglass` model, input size is 256x256.

Base class for detecting poses of people.

Input is an image (cv:Mat).

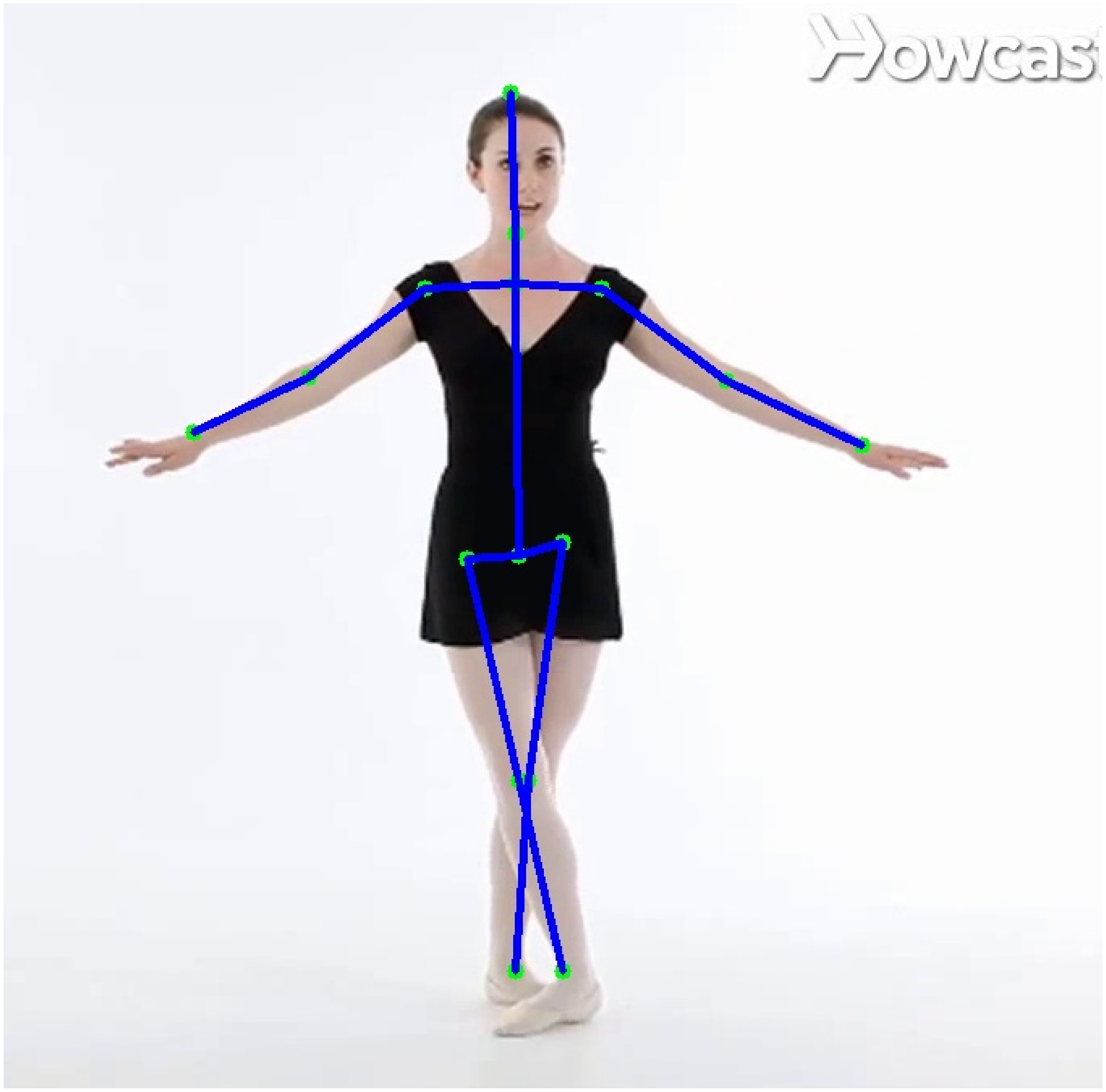
Output is `HourglassResult`.

Sample code:

```
auto image = cv::imread(argv[2]);
if (image.empty()) {
  std::cerr << "cannot load " << argv[2] << std::endl;
  abort();
}
auto det = vitis::ai::Hourglass::create(argv[1]);
vector<vector<int>> limbSeq = {{0, 1},  {1, 2},   {2, 6},  {3, 6},  {3, 4},
{4, 5},
                                {6, 7},   {7, 8},   {8, 9}, {7, 12},
                                {12, 11}, {11, 10}, {7, 13}, {13, 14}, {14,
15}};

auto results = det->run(image.clone());
for (size_t i = 0; i < results.poses.size(); ++i) {
  cout<< results.poses[i].point<<endl;
  if (results.poses[i].type == 1) {
    cv::circle(image, results.poses[i].point, 5, cv::Scalar(0, 255, 0),
               -1);
  }
}
for (size_t i = 0; i < limbSeq.size(); ++i) {
  Result a = results.poses[limbSeq[i][0]];
  Result b = results.poses[limbSeq[i][1]];
  if (a.type == 1 && b.type == 1) {
    cv::line(image, a.point, b.point, cv::Scalar(255, 0, 0), 3, 4);
  }
}
```

Display of the hourglass model results:

Figure 62: **hourglass result image**



Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::Hourglass` class:

Table 158: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `Hourglass` >	create	const std::string & model_name bool need_preprocess
`HourglassResult`	run	const cv::Mat & image
std::vector< `HourglassResult` >	run	const std::vector< cv::Mat > & images
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void

Functions

create

Factory function to get an instance of derived classes of class `Hourglass` .

Prototype

```
std::unique_ptr<
                Hourglass
              > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 159: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `Hourglass` class.

run

Function to get running result of the hourglass neural network.

Prototype

```
           HourglassResult
          run(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 160: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of input image (cv::Mat).

Returns

`HourglassResult`.

run

Function to get running results of the hourglass neural network in batch mode.

Prototype

```
std::vector<
           HourglassResult
          > run(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 161: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of batch input images (vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `HourglassResult`.

getInputWidth

Function to get InputWidth of the hourglass network (input image columns).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the hourglass network

getInputHeight

Function to get InputHeight of the hourglass network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the hourglass network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

vitis::ai::MedicalDetection

Base class for detecting five objects of Endoscopy Disease Detection and `Segmentation` database (EDD2020) .

Input is an image (cv:Mat).

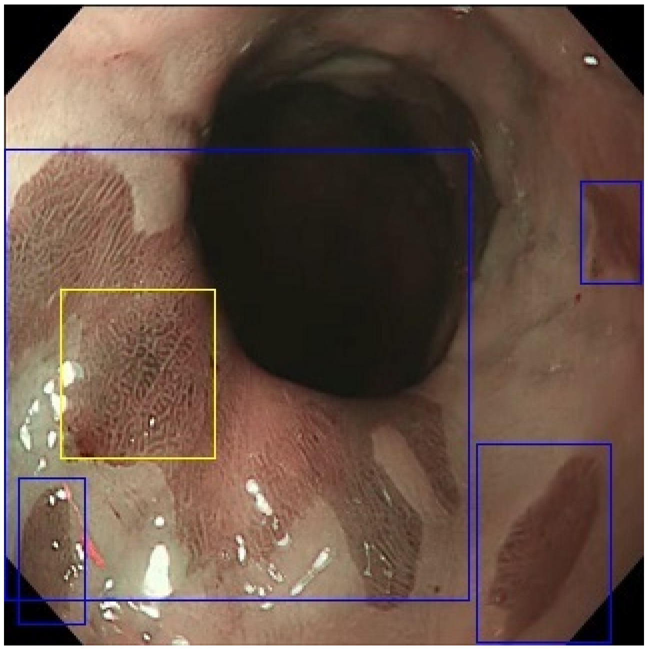
Output is a struct of detection results, named `MedicalDetectionResult`.

Sample code :

```
Mat img = cv::imread("sample_medicaldetection.jpg");
auto medicaldetection = vitis::ai::MedicalDetection::create("RefineDet-
Medical_EDD_tf",true);
auto results = medicaldetection->run(img);
for(const auto &r : results.bboxes){
   auto label = r.label;
   auto x = r.x * img.cols;
   auto y = r.y * img.rows;
   auto width = r.width * img.cols;
   auto height = r.height * img.rows;
   auto score = r.score;
   std::cout << "RESULT: " << label << "\t" << x << "\t" << y << "\t" <<
width
      << "\t" << height << "\t" << score << std::endl;
}
```

Display of the model results:

Figure 63: **detection result**



Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::MedicalDetection` class:

Table 162: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< MedicalDetection >	create	const std::string & model_name bool need_preprocess
vitis::ai::MedicalDetectionResult	run	const cv::Mat & img
std::vector< vitis::ai::MedicalDetectionResult >	run	const std::vector< cv::Mat > & imgs
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void

Functions

create

Factory function to get an instance of derived classes of class `MedicalDetection`.

Prototype

```
std::unique_ptr<
                MedicalDetection
              > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 163: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `MedicalDetection` class.

run

Function of get result of the `MedicalDetection` neural network.

Prototype

```
              vitis::ai::MedicalDetectionResult
            run(const cv::Mat &img)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 164: **run Arguments**

Type	Member	Description
const cv::Mat &	img	Input data of input image (cv::Mat).

Returns

`MedicalDetectionResult`.

run

Function to get running results of the `MedicalDetection` neural network in batch mode.

Prototype

```
std::vector<
              vitis::ai::MedicalDetectionResult
            > run(const std::vector< cv::Mat > &imgs)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 165: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	imgs	Input data of input images (vector<cv::Mat>).The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `MedicalDetectionResult`.

getInputWidth

Function to get InputWidth of the `MedicalDetection` network (input image cols).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the `MedicalDetection` network.

getInputHeight

Function to get InputHeight of the `MedicalDetection` network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the `MedicalDetection` network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

vitis::ai::MedicalDetectionPostProcess

Class of the `MedicalDetection` post-process. It initializes the parameters once instead of computing them every time when the program execute.

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::MedicalDetectionPostProcess` class:

Table 166: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `MedicalDetectionPostProcess` >	create	const std::vector< vitis::ai::library::InputTensor > & input_tensors const std::vector< vitis::ai::library::OutputTensor > & output_tensors const vitis::ai::proto::DpuModelParam & config
`MedicalDetectionResult`	medicaldetection_post_process	void
std::vector< `MedicalDetectionResult` >	medicaldetection_post_process	void

Functions

create

Create an `MedicalDetectionPostProcess` object.

Prototype

```
std::unique_ptr<
               MedicalDetectionPostProcess
             > create(const std::vector< vitis::ai::library::InputTensor >
&input_tensors, const std::vector< vitis::ai::library::OutputTensor >
&output_tensors, const vitis::ai::proto::DpuModelParam &config, int
&real_batch_size);
```

Parameters

The following table lists the `create` function arguments.

Table 167: **create Arguments**

Type	Member	Description
const std::vector< vitis::ai::library::InputTensor > &	input_tensors	A vector of all input-tensors in the network. Usage: input_tensors[input_tensor_index].
const std::vector< vitis::ai::library::OutputTensor > &	output_tensors	A vector of all output-tensors in the network. Usage: output_tensors[output_index].

Table 167: **create Arguments** *(cont'd)*

Type	Member	Description
const vitis::ai::proto::DpuModel Param &	config	The dpu model configuration information.

Returns

An unique pointer of `MedicalDetectionPostProcess` .

medicaldetection_post_process

The post-processing function of the `MedicalDetection` network.

Prototype

```
          MedicalDetectionResult
         medicaldetection_post_process(unsigned int idx)=0;
```

Returns

Struct of `MedicalDetectionResult` .

medicaldetection_post_process

The batch mode post-processing function of the `MedicalDetection` network.

Prototype

```
std::vector<
          MedicalDetectionResult
         > medicaldetection_post_process()=0;
```

Returns

The vector of struct of `MedicalDetectionResult` .

vitis::ai::MedicalSegcell

Base class for segmenting nuclei from images of cells.

Input is an image (cv:Mat).

Output is a struct of detection results, named `MedicalSegcellResult`.

Sample code :

```
Mat img = cv::imread("sample_medicalsegcell.jpg");
auto medicalsegcell =
vitis::ai::MedicalSegcell::create("medical_seg_cell_tf2",true);
auto results = medicalsegcell->run(img);
// results is structure holding cv::Mat.
// please check test samples for detail usage.
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::MedicalSegcell` class:

Table 168: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `MedicalSegcell` >	create	const std::string & model_name bool need_preprocess
`vitis::ai::MedicalSegcellResult`	run	const cv::Mat & img
std::vector< `vitis::ai::MedicalSegcellResult` >	run	const std::vector< cv::Mat > & imgs
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void

Functions

create

Factory function to get an instance of derived classes of class `MedicalSegcell` .

Prototype

```
std::unique_ptr<
                 MedicalSegcell
               > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 169: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `MedicalSegcell` class.

run

Function of get result of the `MedicalSegcell` neural network.

Prototype

```
                  vitis::ai::MedicalSegcellResult
                run(const cv::Mat &img)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 170: **run Arguments**

Type	Member	Description
const cv::Mat &	img	Input data of input image (cv::Mat).

Returns

`MedicalSegcellResult` .

run

Function to get running results of the `MedicalSegcell` neural network in batch mode.

Prototype

```
std::vector<
                  vitis::ai::MedicalSegcellResult
                > run(const std::vector< cv::Mat > &imgs)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 171: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	imgs	Input data of input images (vector<cv::Mat>).The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `MedicalSegcellResult` .

getInputWidth

Function to get InputWidth of the `MedicalSegcell` network (input image columns).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the `MedicalSegcell` network.

getInputHeight

Function to get InputHeight of the `MedicalSegcell` network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the `MedicalSegcell` network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

***Note*:** Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

vitis::ai::MedicalSegmentation

Base class for segment five objects of Endoscopy Disease Detection and `Segmentation` database (EDD2020).

Input is an image (cv:Mat).

Output is a struct of detection results, named `MedicalSegmentationResult`.

Sample code :

```
Mat img = cv::imread("sample_medicalsegmentation.jpg");
auto medicalsegmentation =
vitis::ai::MedicalSegmentation::create("FPN_Res18_Medical_segmentation",true
);
auto results = medicalsegmentation->run(img);
// results is  std::vector<cv::Mat>(5) for 5 classes.
// please check test samples for detail usage.
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::MedicalSegmentation` class:

Table 172: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `MedicalSegmentation` >	create	const std::string & model_name bool need_preprocess
`vitis::ai::MedicalSegmentationResult`	run	const cv::Mat & img
std::vector< `vitis::ai::MedicalSegmentationResult` >	run	const std::vector< cv::Mat > & imgs
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void

Functions

create

Factory function to get an instance of derived classes of class `MedicalSegmentation`.

Prototype

```
std::unique_ptr<
                MedicalSegmentation
              > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 173: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `MedicalSegmentation` class.

run

Function of get result of the `MedicalSegmentation` neural network.

Prototype

```

                vitis::ai::MedicalSegmentationResult
              run(const cv::Mat &img)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 174: **run Arguments**

Type	Member	Description
const cv::Mat &	img	Input data of input image (cv::Mat).

Returns

MedicalSegmentationResult.

run

Function to get running results of the MedicalSegmentation neural network in batch mode.

Prototype

```
std::vector<
              vitis::ai::MedicalSegmentationResult
            > run(const std::vector< cv::Mat > &imgs)=0;
```

Parameters

The following table lists the run function arguments.

Table 175: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	imgs	Input data of input images (vector<cv::Mat>).The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of MedicalSegmentationResult.

getInputWidth

Function to get InputWidth of the MedicalSegmentation network (input image columns).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the MedicalSegmentation network.

getInputHeight

Function to get InputHeight of the MedicalSegmentation network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the `MedicalSegmentation` network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

vitis::ai::MedicalSegmentationPostProcess

Class of the `MedicalSegmentation` post-process. It will initialize the parameters once instead of computing them every time when the program executes.

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::MedicalSegmentationPostProcess` class:

Table 176: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `MedicalSegmentationPostProcess` >	create	const std::vector< vitis::ai::library::InputTensor > & input_tensors const std::vector< vitis::ai::library::OutputTensor > & output_tensors const vitis::ai::proto::DpuModelParam & config
`MedicalSegmentationResult`	medicalsegmentation_post_process	void
std::vector< `MedicalSegmentationResult` >	medicalsegmentation_post_process	void

Functions

create

Create an `MedicalSegmentationPostProcess` object.

Prototype

```
std::unique_ptr<
              MedicalSegmentationPostProcess
             > create(const std::vector< vitis::ai::library::InputTensor >
&input_tensors, const std::vector< vitis::ai::library::OutputTensor >
&output_tensors, const vitis::ai::proto::DpuModelParam &config, int
&real_batch_size);
```

Parameters

The following table lists the `create` function arguments.

Table 177: **create Arguments**

Type	Member	Description
const std::vector< vitis::ai::library::InputTensor > &	input_tensors	A vector of all input-tensors in the network. Usage: input_tensors[input_tensor_index].
const std::vector< vitis::ai::library::OutputTensor > &	output_tensors	A vector of all output-tensors in the network. Usage: output_tensors[output_index].
const vitis::ai::proto::DpuModelParam &	config	The DPU model configuration information.

Returns

A unique pointer of `MedicalSegmentationPostProcess`.

medicalsegmentation_post_process

The post-processing function of the `MedicalSegmentation` network.

Prototype

```

              MedicalSegmentationResult
             medicalsegmentation_post_process(unsigned int idx)=0;
```

Returns

Struct of `MedicalSegmentationResult`.

medicalsegmentation_post_process

The batch mode post-processing function of the `MedicalSegmentation` network.

Prototype

```
std::vector<
                    MedicalSegmentationResult
                  > medicalsegmentation_post_process()=0;
```

Returns

The vector of struct of `MedicalSegmentationResult` .

vitis::ai::Monodepth2

Base class for `Monodepth2` (production segmentation)

Input is an image (cv:Mat).

Output is a struct of segmentation results, named Monodepth2Result.

Sample code :

```
Mat img = cv::imread("sample_monodepth2.jpg");
auto Monodepth2 = vitis::ai::Monodepth2::create("monodepth2_pt",true);
auto result = Monodepth2->run(img);
// result is structure holding the mat.
std::cout << result.mat.cols <<"\n";
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::Monodepth2` class:

Table 178: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `Monodepth2` >	create	const std::string & model_name bool need_preprocess
vitis::ai::Monodepth2Result	run	const cv::Mat & img
std::vector< vitis::ai::Monodepth2Result >	run	const std::vector< cv::Mat > & imgs

Table 178: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void

Functions

create

Factory function to get an instance of derived classes of class `Monodepth2` .

Prototype

```
std::unique_ptr<
                 Monodepth2
               > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 179: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `Monodepth2` class.

run

Function of get result of the `Monodepth2` network.

Prototype

```
vitis::ai::Monodepth2Result run(const cv::Mat &img)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 180: **run Arguments**

Type	Member	Description
const cv::Mat &	img	Input data of input image (cv::Mat).

Returns

Monodepth2Result.

run

Function to get running results of the `Monodepth2` network in batch mode.

Prototype

```
std::vector< vitis::ai::Monodepth2Result > run(const std::vector< cv::Mat >
&imgs)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 181: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	imgs	Input data of input images (vector<cv::Mat>). The size of input images need equal to or less than batch size obtained by get_input_batch.

Returns

The vector of Monodepth2Result.

getInputWidth

Function to get InputWidth of the `Monodepth2` network (input image columns).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the `Monodepth2` network.

getInputHeight

Function to get InputHeight of the `Monodepth2` network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the `Monodepth2` network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

vitis::ai::Movenet

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::Movenet` class:

Table 182: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< Movenet >	create	const std::string & model_name bool need_preprocess
`MovenetResult`	run	const cv::Mat & image
std::vector< `MovenetResult` >	run	const std::vector< cv::Mat > & images

Table 182: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void

Functions

create

Factory function to get an instance of derived classes of class Movenet.

Prototype

```
std::unique_ptr< Movenet > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 183: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of Movenet class.

run

Function to get running result of the movenet neural network.

Prototype

```
                MovenetResult
              run(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 184: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of input image (cv::Mat).

Returns

`MovenetResult` .

run

Function to get running results of the movenet neural network in batch mode.

Prototype

```
std::vector<
               MovenetResult
             > run(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 185: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of batch input images (vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `MovenetResult` .

getInputWidth

Function to get InputWidth of the movenet network (input image columns).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the movenet network

getInputHeight

Function to get InputHeight of the movenet network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the movenet network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

vitis::ai::MultiTask

Base class for ADAS MuiltTask from an image (cv::Mat).

Input an image (cv::Mat).

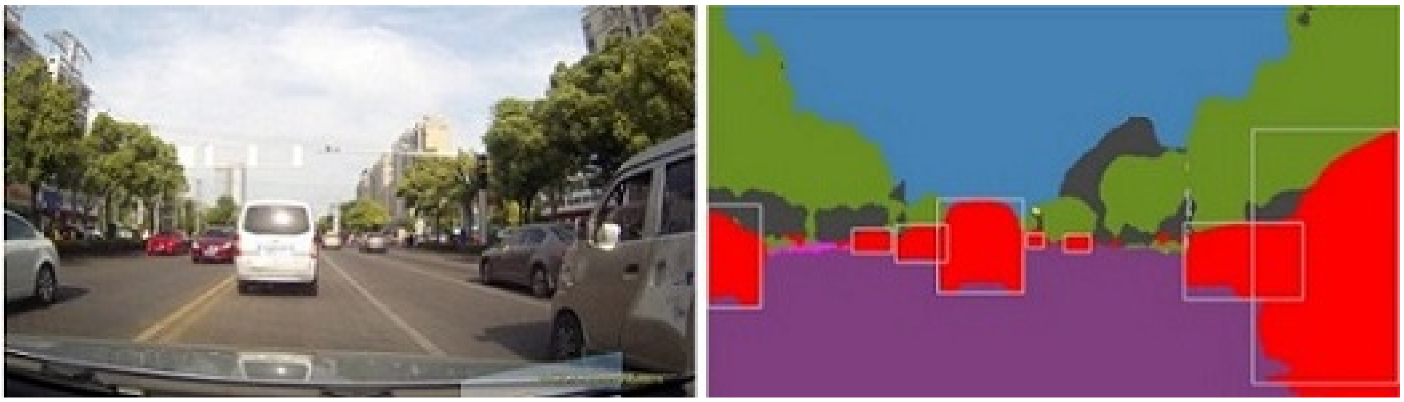
Output is a struct of `MultiTaskResult` includes segmentation results, detection results and vehicle towards;

Sample code:

```
auto det = vitis::ai::MultiTask::create("multi_task");
auto image = cv::imread("sample_multitask.jpg");
auto result = det->run_8UC3(image);
cv::imwrite("sample_multitask_result.jpg",result.segmentation);
```

Display of the model results:

Figure 64: **result image**



Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::MultiTask` class:

Table 186: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `MultiTask` >	create	const std::string & model_name bool need_preprocess
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void
`MultiTaskResult`	run_8UC1	const cv::Mat & image
std::vector< `MultiTaskResult` >	run_8UC1	const std::vector< cv::Mat > & images
`MultiTaskResult`	run_8UC3	const cv::Mat & image
std::vector< `MultiTaskResult` >	run_8UC3	const std::vector< cv::Mat > & images

Functions

create

Factory function to get an instance of derived classes of class Multitask.

Prototype

```
std::unique_ptr<
                MultiTask
              > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 187: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of Multitask class.

getInputWidth

Function to get InputWidth of the multitask network (input image columns).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the multitask network.

getInputHeight

Function to get InputHeight of the multitask network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the multitask network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: For different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

run_8UC1

Function to get running result from the `MultiTask` network.

Note: The type is CV_8UC1 of the `MultiTaskResult.segmentation.`

Prototype

```
                    MultiTaskResult
                 run_8UC1(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run_8UC1` function arguments.

Table 188: **run_8UC1 Arguments**

Type	Member	Description
const cv::Mat &	image	Input image

Returns

The struct of `MultiTaskResult`

run_8UC1

Function to get running results of the `MultiTask` neural network in batch mode.

Note: The type is CV_8UC1 of the `MultiTaskResult.segmentation`.

Prototype

```
std::vector<
                MultiTaskResult
              > run_8UC1(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run_8UC1` function arguments.

Table 189: **run_8UC1 Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `MultiTaskResult` .

run_8UC3

Function to get running result from the `MultiTask` network.

Note: The type is CV_8UC3 of the `MultiTaskResult.segmentation`.

Prototype

```


                MultiTaskResult
              run_8UC3(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run_8UC3` function arguments.

Table 190: **run_8UC3 Arguments**

Type	Member	Description
const cv::Mat &	image	Input image;

Returns

The struct of `MultiTaskResult`

run_8UC3

Function to get running results of the `MultiTask` neural network in batch mode.

Note: The type is CV_8UC3 of the `MultiTaskResult.segmentation`.

Prototype

```
std::vector<
               MultiTaskResult
             > run_8UC3(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run_8UC3` function arguments.

Table 191: **run_8UC3 Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `MultiTaskResult` .

vitis::ai::MultiTask8UC1

Base class for ADAS MuiltTask8UC1 from an image (cv::Mat).

Input is an image (cv::Mat).

Output is struct `MultiTaskResult` includes segmentation results, detection results and vehicle towards; The result cv::Mat type is CV_8UC1

Sample code:

```
auto det = vitis::ai::MultiTask8UC1::create(vitis::ai::MULTITASK);
auto image = cv::imread("sample_multitask.jpg");
auto result = det->run(image);
cv::imwrite("res.jpg",result.segmentation);
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::MultiTask8UC1` class:

Table 192: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< MultiTask8UC1 >	create	const std::string & model_name bool need_preprocess
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void
MultiTaskResult	run	const cv::Mat & image
std::vector< MultiTaskResult >	run	const std::vector< cv::Mat > & images

Functions

create

Factory function to get an instance of derived classes of class `MultiTask8UC1` .

Prototype

```
std::unique_ptr<
                  MultiTask8UC1
                > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 193: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `MultiTask8UC1` class.

getInputWidth

Function to get InputWidth of the multitask network (input image columns).

Prototype

```
int getInputWidth() const;
```

Returns

InputWidth of the multitask network.

getInputHeight

Function to get InputHeight of the multitask network (input image rows).

Prototype

```
int getInputHeight() const;
```

Returns

InputHeight of the multitask network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: For different DPU core the batch size may be differnt. This depends on the IP used.

Prototype

```
size_t get_input_batch() const;
```

Returns

Batch size.

run

Function to get running result from the `MultiTask` network.

Note: The type is CV_8UC1 of the `MultiTaskResult.segmentation`.

Prototype

```
                MultiTaskResult
              run(const cv::Mat &image);
```

Parameters

The following table lists the `run` function arguments.

Table 194: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input image

Returns

The struct of `MultiTaskResult`

run

Function to get running results of the `MultiTask` neural network in batch mode.

Note: The type of the `MultiTaskResult.segmentation` is CV_8UC1 .

Prototype

```
std::vector<
                MultiTaskResult
              > run(const std::vector< cv::Mat > &images);
```

Parameters

The following table lists the `run` function arguments.

Table 195: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `MultiTaskResult` .

vitis::ai::MultiTask8UC3

Base class for ADAS MuiltTask8UC3 from an image (cv::Mat).

Input is an image (cv::Mat).

Output is struct `MultiTaskResult` includes segmentation results, detection results and vehicle orientation; The result cv::Mat type is CV_8UC3

Sample code:

```
auto det = vitis::ai::MultiTask8UC3::create(vitis::ai::MULITASK);
auto image = cv::imread("sample_multitask.jpg");
auto result = det->run(image);
cv::imwrite("res.jpg",result.segmentation);
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::MultiTask8UC3` class:

Table 196: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `MultiTask8UC3` >	create	const std::string & model_name bool need_preprocess
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void
`MultiTaskResult`	run	const cv::Mat & image
std::vector< `MultiTaskResult` >	run	const std::vector< cv::Mat > & images

Functions

create

Factory function to get an instance of derived classes of class `MultiTask8UC3` .

Prototype

```
std::unique_ptr<
                MultiTask8UC3
              > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 197: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `MultiTask8UC3` class.

getInputWidth

Function to get InputWidth of the multitask network (input image columns).

Prototype

```
int getInputWidth() const;
```

Returns

InputWidth of the multitask network.

getInputHeight

Function to get InputHeight of the multitask network (input image rows).

Prototype

```
int getInputHeight() const;
```

Returns

InputHeight of the multitask network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: For different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const;
```

Returns

Batch size.

run

Function to get running result from the `MultiTask` network.

Note: The type is CV_8UC3 of the `MultiTaskResult.segmentation`.

Prototype

```
                    MultiTaskResult
                  run(const cv::Mat &image);
```

Parameters

The following table lists the `run` function arguments.

Table 198: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input image

Returns

The struct of `MultiTaskResult`

run

Function to get running results of the `MultiTask` neural network in batch mode.

Note: The type is CV_8UC3 of the `MultiTaskResult.segmentation`.

Prototype

```
std::vector<
                MultiTaskResult
             > run(const std::vector< cv::Mat > &images);
```

Parameters

The following table lists the `run` function arguments.

Table 199: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `MultiTaskResult` .

vitis::ai::MultiTaskPostProcess

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::MultiTaskPostProcess` class:

Table 200: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< MultiTaskPostProcess >	create	const std::vector< std::vector< vitis::ai::library::InputTensor > > & input_tensors const std::vector< std::vector< vitis::ai::library::OutputTensor > > & output_tensors const vitis::ai::proto::DpuModelParam & config
std::vector< `MultiTaskResult` >	post_process_seg	void
std::vector< `MultiTaskResult` >	post_process_seg_visualization	void

Functions

create

Factory function to get an instance of derived classes of MultiTaskPostProcess.

Prototype

```
std::unique_ptr< MultiTaskPostProcess > create(const std::vector<
std::vector< vitis::ai::library::InputTensor > > &input_tensors, const
std::vector< std::vector< vitis::ai::library::OutputTensor > >
&output_tensors, const vitis::ai::proto::DpuModelParam &config);
```

Parameters

The following table lists the `create` function arguments.

Table 201: **create Arguments**

Type	Member	Description
const std::vector< std::vector< vitis::ai::library::InputTensor > > &	input_tensors	A vector of all input-tensors in the network. Usage: input_tensors[kernel_index][input_tensor_index].
const std::vector< std::vector< vitis::ai::library::OutputTensor > > &	output_tensors	A vector of all output-tensors in the network. Usage: output_tensors[kernel_index][output_index].
const vitis::ai::proto::DpuModelParam &	config	The dpu model configuration information.

Returns

Struct of `MultiTaskResult`.

post_process_seg

The post-processing function of the multitask which stored the original segmentation classes.

Prototype

```
std::vector<
               MultiTaskResult
             > post_process_seg(size_t batch_size)=0;
```

Returns

Struct of `SegmentationResult`.

post_process_seg_visualization

The post-processing function of the multitask which return a result include segmentation image mapped to color.

Prototype

```
std::vector<
               MultiTaskResult
             > post_process_seg_visualization(size_t batch_size)=0;
```

Returns

Struct of `SegmentationResult` .

vitis::ai::MultiTaskv3

Base class for ADAS MuiltTask from an image (cv::Mat).

Input an image (cv::Mat).

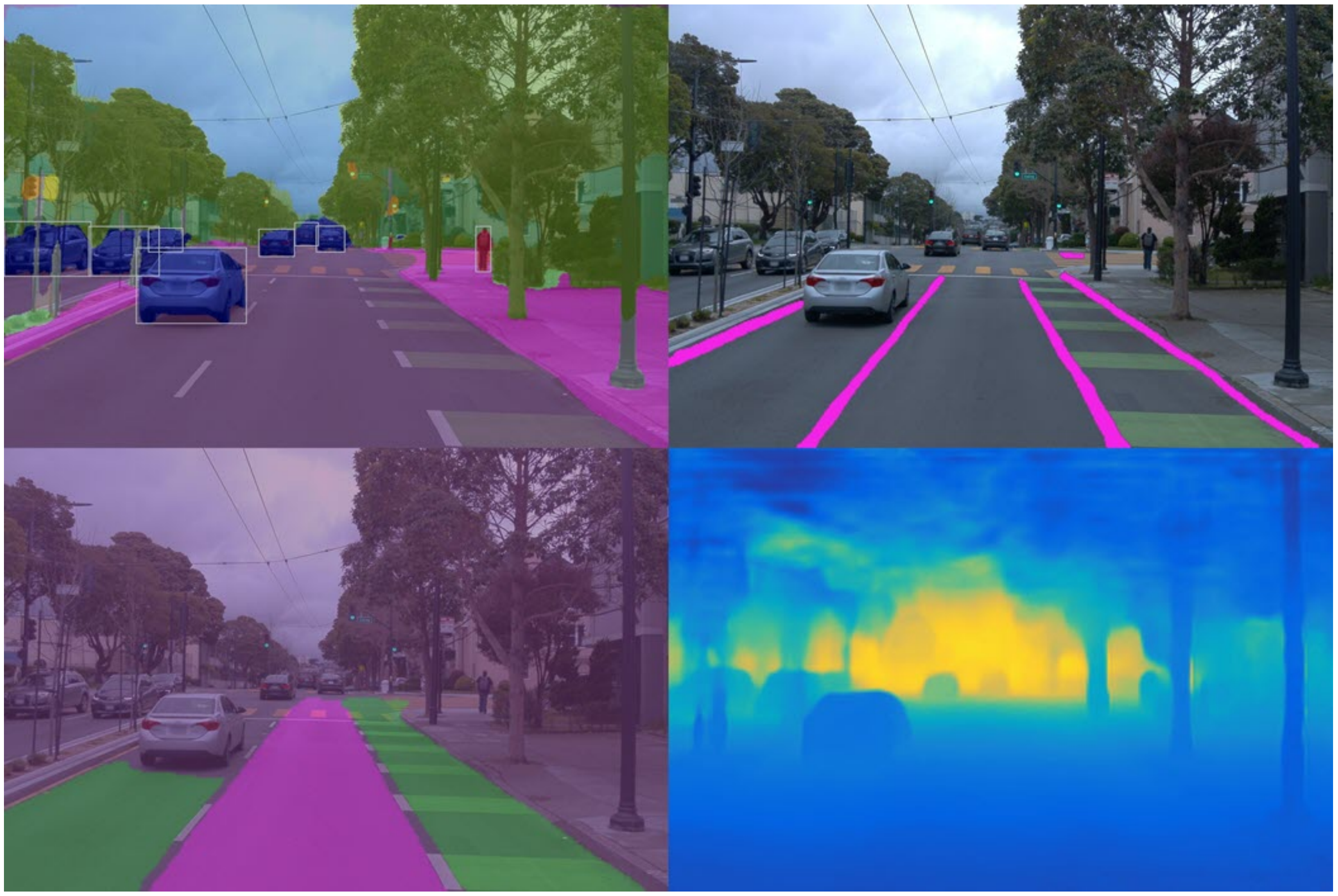
Output is a struct of `MultiTaskv3Result` including segmentation results, detection results and vehicle towards;

Sample code:

```
auto det = vitis::ai::MultiTaskv3::create("multi_task");
auto image = cv::imread("sample_multitaskv3.jpg");
auto result = det->run_8UC3(image);
cv::imwrite("sample_multitaskv3_result.jpg",result.segmentation);
cv::imwrite("sample_multitaskv3_result.jpg",result.depth);
```

Display of the model results:

Figure 65: **result image**



Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::MultiTaskv3` class:

Table 202: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `MultiTaskv3` >	create	const std::string & model_name bool need_preprocess
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void
`MultiTaskv3Result`	run_8UC1	const cv::Mat & image

Table 202: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
std::vector< MultiTaskv3Result >	run_8UC1	const std::vector< cv::Mat > & images
MultiTaskv3Result	run_8UC3	const cv::Mat & image
std::vector< MultiTaskv3Result >	run_8UC3	const std::vector< cv::Mat > & images

Functions

create

Factory function to get an instance of derived classes of class Multitaskv3.

Prototype

```
std::unique_ptr<
                 MultiTaskv3
               > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 203: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of Multitaskv3 class.

getInputWidth

Function to get InputWidth of the multitaskv3 network (input image columns).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the multitaskv3 network.

getInputHeight

Function to get InputHeight of the multitaskv3 network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the multitaskv3 network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

run_8UC1

Function of get running result from the `MultiTaskv3` network.

Note: The type is CV_8UC1 of the `MultiTaskv3Result.segmentation` and all cv::Mat output.

Prototype

```
                          MultiTaskv3Result
                        run_8UC1(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run_8UC1` function arguments.

Table 204: **run_8UC1 Arguments**

Type	Member	Description
const cv::Mat &	image	Input image

Returns

The struct of `MultiTaskv3Result`

run_8UC1

Function to get running results of the `MultiTaskv3` neural network in batch mode.

Note: The type is CV_8UC1 of all cv::Mat output.

Prototype

```
std::vector<
                MultiTaskv3Result
              > run_8UC1(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run_8UC1` function arguments.

Table 205: **run_8UC1 Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `MultiTaskv3Result` .

run_8UC3

Function to get running result from the `MultiTaskv3` network.

Note: The type is CV_8UC3 of all cv::Mat result except depth estimation.

Prototype

```

                MultiTaskv3Result
              run_8UC3(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run_8UC3` function arguments.

Table 206: **run_8UC3 Arguments**

Type	Member	Description
const cv::Mat &	image	Input image;

Returns

The struct of `MultiTaskv3Result`

run_8UC3

Function to get running results of the `MultiTaskv3` neural network in batch mode.

***Note*:** The type is CV_8UC3 of all cv::Mat result except depth estimation.

Prototype

```
std::vector<
                MultiTaskv3Result
              > run_8UC3(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run_8UC3` function arguments.

Table 207: **run_8UC3 Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `MultiTaskv3Result` .

vitis::ai::MultiTaskv38UC1

Base class for ADAS MuiltTask8UC1 from an image (cv::Mat).

Input is an image (cv::Mat).

Output is struct MultiTaskv3Result including segmentation results, detection results and vehicle towards; The result cv::Mat type is CV_8UC1

Sample code:

```
auto det = vitis::ai::MultiTaskv38UC1::create(vitis::ai::MULTITASKv3);
auto image = cv::imread("sample_multitaskv3.jpg");
auto result = det->run(image);
cv::imwrite("res.jpg",result.segmentation);
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::MultiTaskv38UC1` class:

Table 208: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< MultiTaskv38UC1 >	create	const std::string & model_name bool need_preprocess
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void
MultiTaskv3Result	run	const cv::Mat & image
std::vector< MultiTaskv3Result >	run	const std::vector< cv::Mat > & images

Functions

create

Factory function to get an instance of derived classes of class MultiTaskv38UC1 .

Prototype

```
std::unique_ptr<
               MultiTaskv38UC1
             > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 209: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `MultiTaskv38UC1` class.

getInputWidth

Function to get InputWidth of the multitaskv3 network (input image columns).

Prototype

```
int getInputWidth() const;
```

Returns

InputWidth of the multitaskv3 network.

getInputHeight

Function to get InputHeight of the multitaskv3 network (input image rows).

Prototype

```
int getInputHeight() const;
```

Returns

InputHeight of the multitaskv3 network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

***Note*:** Different DPU core the batch size may be differnt. This depends on the IP used.

Prototype

```
size_t get_input_batch() const;
```

Returns

Batch size.

run

Function of get running result from the MultiTaskv3 network.

Note: The type is CV_8UC1 of the `MultiTaskv3Result.segmentation`.

Prototype

```
                    MultiTaskv3Result
                  run(const cv::Mat &image);
```

Parameters

The following table lists the `run` function arguments.

Table 210: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input image

Returns

The struct of `MultiTaskv3Result`

run

Function to get running results of the MultiTaskv3 neural network in batch mode.

Note: The type is CV_8UC1 of the `MultiTaskv3Result.segmentation`.

Prototype

```
std::vector<
                    MultiTaskv3Result
                  > run(const std::vector< cv::Mat > &images);
```

Parameters

The following table lists the `run` function arguments.

Table 211: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `MultiTaskv3Result` .

vitis::ai::MultiTaskv38UC3

Base class for ADAS MuiltTask8UC3 from an image (cv::Mat).

Input is an image (cv::Mat).

Output is struct `MultiTaskv3Result` including segmentation results, detection results and vehicle orientation; The result cv::Mat type is CV_8UC3(except depth estimation)

Sample code:

```
auto det = vitis::ai::MultiTaskv38UC3::create(vitis::ai::MULITASK);
auto image = cv::imread("sample_multitaskv3.jpg");
auto result = det->run(image);
cv::imwrite("res.jpg",result.segmentation);
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::MultiTaskv38UC3` class:

Table 212: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `MultiTaskv38UC3` >	create	const std::string & model_name bool need_preprocess
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void
`MultiTaskv3Result`	run	const cv::Mat & image

Table 212: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
std::vector< MultiTaskv3Result >	run	const std::vector< cv::Mat > & images

Functions

create

Factory function to get an instance of derived classes of class `MultiTaskv38UC3` .

Prototype

```
std::unique_ptr<
                MultiTaskv38UC3
              > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 213: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `MultiTaskv38UC3` class.

getInputWidth

Function to get InputWidth of the multitaskv3 network (input image columns).

Prototype

```
int getInputWidth() const;
```

Returns

InputWidth of the multitaskv3 network.

getInputHeight

Function to get InputHeight of the multitaskv3 network (input image rows).

Prototype

```
int getInputHeight() const;
```

Returns

InputHeight of the multitaskv3 network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const;
```

Returns

Batch size.

run

Function of get running result from the `MultiTaskv3` network.

Note: The type is CV_8UC3 of the `MultiTaskv3Result.segmentation`.

Prototype

```
                    MultiTaskv3Result
                   run(const cv::Mat &image);
```

Parameters

The following table lists the `run` function arguments.

Table 214: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input image

Returns

The struct of `MultiTaskv3Result`

run

Function to get running results of the `MultiTaskv3` neural network in batch mode.

Note: The type is CV_8UC3 of the `MultiTaskv3Result.segmentation`.

Prototype

```
std::vector<
            MultiTaskv3Result
           > run(const std::vector< cv::Mat > &images);
```

Parameters

The following table lists the `run` function arguments.

Table 215: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `MultiTaskv3Result` .

vitis::ai::MultiTaskv3PostProcess

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::MultiTaskv3PostProcess` class:

Table 216: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< MultiTaskv3PostProcess >	create	const std::vector< std::vector< vitis::ai::library::InputTensor > > & input_tensors const std::vector< std::vector< vitis::ai::library::OutputTensor > > & output_tensors const vitis::ai::proto::DpuModelParam & config
std::vector< MultiTaskv3Result >	post_process	void
std::vector< MultiTaskv3Result >	post_process_visualization	void

Functions

create

Factory function to get an instance of derived classes of MultiTaskv3PostProcess.

Prototype

```
std::unique_ptr< MultiTaskv3PostProcess > create(const std::vector<
std::vector< vitis::ai::library::InputTensor > > &input_tensors, const
std::vector< std::vector< vitis::ai::library::OutputTensor > >
&output_tensors, const vitis::ai::proto::DpuModelParam &config);
```

Parameters

The following table lists the `create` function arguments.

Table 217: **create Arguments**

Type	Member	Description
const std::vector< std::vector< vitis::ai::library::InputTensor > > &	input_tensors	A vector of all input-tensors in the network. Usage: input_tensors[kernel_index][input_tensor_index].
const std::vector< std::vector< vitis::ai::library::OutputTensor > > &	output_tensors	A vector of all output-tensors in the network. Usage: output_tensors[kernel_index][output_index].
const vitis::ai::proto::DpuModelParam &	config	The dpu model configuration information.

Returns

Struct of `MultiTaskv3Result` .

post_process

The post-processing function of the multitask which stored the original multitaskv3 classes.

Prototype

```
std::vector<
              MultiTaskv3Result
            > post_process(size_t batch_size)=0;
```

Returns

Struct of Multitaskv3Result.

post_process_visualization

The post-processing function of the multitask which return a result include multitaskv3 image mapped to color.

Prototype

```
std::vector<
              MultiTaskv3Result
            > post_process_visualization(size_t batch_size)=0;
```

Returns

Struct of Multitaskv3Result.

vitis::ai::OCR

Base class for ocr.

Input is an image (cv:Mat).

Output is a struct of detection results, named OCRResult.

Sample code :

```
Mat img = cv::imread("sample_ocr.jpg");
auto ocr = vitis::ai::OCR::create("ocr_pt",true);
auto results = ocr->run(img);
// please check test samples for detail usage.
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::OCR` class:

Table 218: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< OCR >	create	const std::string & model_name bool need_preprocess
vitis::ai::OCRResult	run	const cv::Mat & img
std::vector< vitis::ai::OCRResult >	run	const std::vector< cv::Mat > & imgs
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void

Functions

create

Factory function to get an instance of derived classes of class `OCR` .

Prototype

```
std::unique_ptr<
                  OCR
                > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 219: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `OCR` class.

run

Function of get result of the `OCR` neural network.

Prototype

```
vitis::ai::OCRResult run(const cv::Mat &img)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 220: **run Arguments**

Type	Member	Description
const cv::Mat &	img	Input data of input image (cv::Mat).

Returns

OCRResult.

run

Function to get running results of the `OCR` neural network in batch mode.

Prototype

```
std::vector< vitis::ai::OCRResult > run(const std::vector< cv::Mat >
&imgs)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 221: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	imgs	Input data of input images (vector<cv::Mat>).The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of OCRResult.

getInputWidth

Function to get InputWidth of the `OCR` network (input image columns).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the `OCR` network.

getInputHeight

Function to get InputHeight of the `OCR` network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the `OCR` network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

vitis::ai::OCRPost

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::OCRPost` class:

Table 222: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< OCRPost >	create	const std::vector< vitis::ai::library::InputTensor > & input_tensors const std::vector< vitis::ai::library::OutputTensor > & output_tensors const std::string & cfgpath int batch_size int & real_batch_size std::vector< int > & target_h8 std::vector< int > & target_w8 std::vector< float > & ratioh std::vector< float > & ratiow std::vector< cv::Mat > & oriimg
OCRResult	process	int idx
std::vector< OCRResult >	process	void

Functions

create

Create an OCRPost object.

Prototype

```
std::unique_ptr< OCRPost > create(const std::vector<
vitis::ai::library::InputTensor > &input_tensors, const std::vector<
vitis::ai::library::OutputTensor > &output_tensors, const std::string
&cfgpath, int batch_size, int &real_batch_size, std::vector< int >
&target_h8, std::vector< int > &target_w8, std::vector< float > &ratioh,
std::vector< float > &ratiow, std::vector< cv::Mat > &oriimg);
```

Parameters

The following table lists the `create` function arguments.

Table 223: **create Arguments**

Type	Member	Description
const std::vector< vitis::ai::library::InputTensor > &	input_tensors	A vector of all input-tensors in the network. Usage: input_tensors[input_tensor_index].
const std::vector< vitis::ai::library::OutputTensor > &	output_tensors	A vector of all output-tensors in the network. Usage: output_tensors[output_index].

Table 223: **create Arguments** *(cont'd)*

Type	Member	Description
const std::string &	cfgpath	configuration file path (*_officialcfg.prototxt)
int	batch_size	the model batch information
int &	real_batch_size	the real batch information of the model
std::vector< int > &	target_h8	inner data structure
std::vector< int > &	target_w8	inner data structure
std::vector< float > &	ratioh	inner data structure for height ratio
std::vector< float > &	ratiow	inner data structure for width ratio
std::vector< cv::Mat > &	oriimg	original image

Returns

An unique pointer of OCRPostProcess.

process

Post-process the ocr result.

Prototype

```
OCRResult process(int idx)=0;
```

Parameters

The following table lists the `process` function arguments.

Table 224: **process Arguments**

Type	Member	Description
int	idx	batch index.

Returns

OCRResult.

process

Post-process the ocr result.

Prototype

```
std::vector< OCRResult > process()=0;
```

Returns

vector of OCRResult.

vitis::ai::OFAYOLO

Base class for detecting objects in the input image (cv::Mat).

Input is an image (cv::Mat).

Output is the position of the pedestrians in the input image.

Sample code:

```
auto yolo = vitis::ai::OFAYOLO::create("ofa_yolo_pt", true);

Mat img = cv::imread("sample_ofa_yolo.jpg");
auto results = yolo->run(img);

for (auto& box : results.bboxes) {
  int label = box.label;
  float xmin = box.x * img.cols + 1;
  float ymin = box.y * img.rows + 1;
  float xmax = xmin + box.width * img.cols;
  float ymax = ymin + box.height * img.rows;
  if (xmin < 0.) xmin = 1.;
  if (ymin < 0.) ymin = 1.;
  if (xmax > img.cols) xmax = img.cols;
  if (ymax > img.rows) ymax = img.rows;
  float confidence = box.score;

  cout << "RESULT: " << label << "\t" << xmin << "\t" << ymin << "\t" <<
xmax
       << "\t" << ymax << "\t" << confidence << "\n";
  rectangle(img, Point(xmin, ymin), Point(xmax, ymax), Scalar(0, 255, 0), 1,
            1, 0);
}
imwrite("result.jpg", img);
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::OFAYOLO` class:

Table 225: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `OFAYOLO` >	create	const std::string & model_name bool need_preprocess
`OFAYOLOResult`	run	const cv::Mat & image

Table 225: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
std::vector< OFAYOLOResult >	run	const std::vector< cv::Mat > & images

Functions

create

Factory function to get an instance of derived classes of class OFAYOLO .

Prototype

```
std::unique_ptr<
                OFAYOLO
              > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 226: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of OFAYOLO class.

run

Function to get running result of the OFA_YOLO neural network.

Prototype

```

                OFAYOLOResult
              run(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 227: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of input image (cv::Mat).

Returns

`OFAYOLOResult` .

run

Function to get running result of the OFA_YOLO neural network in batch mode.

Prototype

```
std::vector<
               OFAYOLOResult
             > run(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 228: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `OFAYOLOResult` .

vitis::ai::OpenPose

openpose model, input size is 368x368.

Base class for detecting poses of people.

Input is an image (cv:Mat).

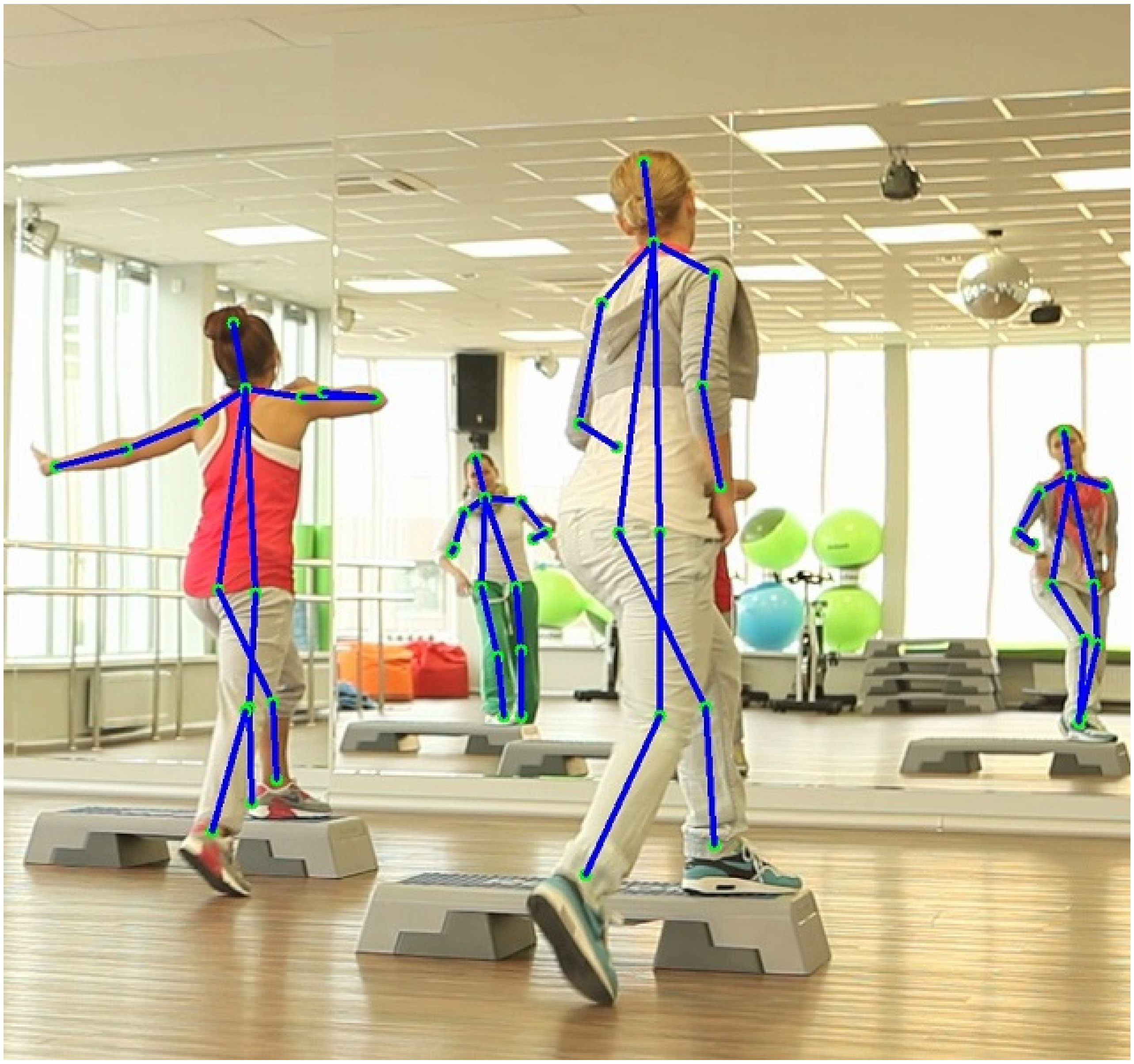
Output is a `OpenPoseResult`.

Sample code :

```
 auto image = cv::imread("sample_openpose.jpg");
 if (image.empty()) {
   std::cerr << "cannot load image" << std::endl;
   abort();
 }
 auto det = vitis::ai::OpenPose::create("openpose_pruned_0_3");
 int width = det->getInputWidth();
 int height = det->getInputHeight();
 vector<vector<int>> limbSeq = {{0,1}, {1,2}, {2,3}, {3,4}, {1,5}, {5,6},
{6,7}, {1,8}, \ {8,9}, {9,10}, {1,11}, {11,12}, {12,13}}; float scale_x =
float(image.cols) / float(width); float scale_y = float(image.rows) /
float(height); auto results = det->run(image); for(size_t k = 1; k <
results.poses.size(); ++k){ for(size_t i = 0; i < results.poses[k].size();
++i){ if(results.poses[k][i].type == 1){ results.poses[k][i].point.x *=
scale_x; results.poses[k][i].point.y *= scale_y; cv::circle(image,
results.poses[k][i].point, 5, cv::Scalar(0, 255, 0), -1);
       }
   }
   for(size_t i = 0; i < limbSeq.size(); ++i){
       Result a = results.poses[k][limbSeq[i][0]];
       Result b = results.poses[k][limbSeq[i][1]];
       if(a.type == 1 && b.type == 1){
           cv::line(image, a.point, b.point, cv::Scalar(255, 0, 0), 3, 4);
       }
   }
 }
```

Display of the openpose model results: width=\textwidth

Figure 66: **openpose result image**



Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::OpenPose` class:

Table 229: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `OpenPose` >	create	const std::string & model_name bool need_preprocess

Table 229: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
OpenPoseResult	run	const cv::Mat & image
std::vector< OpenPoseResult >	run	const std::vector< cv::Mat > & images
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void

Functions

create

Factory function to get an instance of derived classes of class `OpenPose` .

Prototype

```
std::unique_ptr<
                  OpenPose
                > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 230: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `OpenPose` class.

run

Function to get running result of the openpose neural network.

Prototype

```
                OpenPoseResult
              run(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 231: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of input image (cv::Mat).

Returns

`OpenPoseResult` .

run

Function to get running results of the openpose neural network in batch mode.

Prototype

```
std::vector<
                OpenPoseResult
              > run(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 232: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of batch input images (vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `OpenPoseResult` .

getInputWidth

Function to get InputWidth of the openpose network (input image columns).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the openpose network

getInputHeight

Function to get InputHeight of the openpose network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the openpose network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

vitis::ai::PlateDetect

Base class for detecting the position of plate in a vehicle image (cv::Mat).

Input is a vehicle image (cv::Mat).

Output is position and score of plate in the input image.

Sample code:

```
cv::Mat image = cv::imread("car.jpg");
auto network = vitis::ai::PlateDetect::create(true);
auto r = network->run(image);
auto score = r.box.score.
auto x = r.box.x * image.cols;
auto y = r.box.y * image.rows;
auto witdh = r.box.width * image.cols;
auto height = r.box.height * image.rows;
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::PlateDetect` class:

Table 233: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< PlateDetect >	create	const std::string & model_name bool need_mean_scale_process
std::unique_ptr< PlateDetect >	create	const std::string & model_name xir::Attrs * attrs bool need_mean_scale_process
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void
PlateDetectResult	run	const cv::Mat & image
std::vector< PlateDetectResult >	run	const std::vector< cv::Mat > & images

Functions

create

Factory function to get an instance of derived classes of class platedetect.

Prototype

```
std::unique_ptr<
                PlateDetect
              > create(const std::string &model_name, bool
need_mean_scale_process=true);
```

Parameters

The following table lists the `create` function arguments.

Table 234: **create Arguments**

Type	Member	Description
const std::string &	model_name	the model name of the created model
bool	need_mean_scale_proces s	Normalize with mean/scale or not, true by default.

Returns

An instance of the `PlateDetect` class.

create

Factory function to get an instance of derived classes of class platedetect.

Prototype

```
std::unique_ptr<
                PlateDetect
              > create(const std::string &model_name, xir::Attrs *attrs,
bool need_mean_scale_process=true);
```

Parameters

The following table lists the `create` function arguments.

Table 235: **create Arguments**

Type	Member	Description
const std::string &	model_name	the model name of the created model
xir::Attrs *	attrs	xir::Attrs pointer points to the provided attributes
bool	need_mean_scale_proces s	Normalize with mean/scale or not, true by default.

Returns

An instance of the `PlateDetect` class.

getInputWidth

Function to get InputWidth of the platedetect network (input image columns).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the platedetect network.

getInputHeight

Function to get InputHeight of the platedetect network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the platedetect network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

run

Function of get running result of the platedetect network.

Prototype

```
              PlateDetectResult
             run(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 236: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of input image (cv::Mat) of detected counterpart and resized as inputwidth an outputheight.

Returns

Plate position and plate score.

run

Function to get running results of the platedetect neural network in batch mode.

Prototype

```
std::vector<
                PlateDetectResult
              > run(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 237: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch. The input images need to be resized to InputWidth and InputHeight required by the network.

Returns

The vector of PLateDetectResult.

vitis::ai::PlateNum

Base class for recognizing plate from an image (cv::Mat).

Input is a plate image (cv::Mat).

Output is the number and color of plate in the input image.

sample code:

Note: Only China plate Only edge platform supported @endnote

```
cv::Mat image = cv::imread("plate.jpg");
auto network = vitis::ai::PlateNum::create(true);
auto r = network->run(image);
auto plate_number = r.plate_number;
auto plate_color = r.plate_color;
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::PlateNum` class:

Table 238: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `PlateNum` >	create	const std::string & model_name bool need_mean_scale_process
std::unique_ptr< `PlateNum` >	create	const std::string & model_name xir::Attrs * attrs bool need_mean_scale_process
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void
`PlateNumResult`	run	const cv::Mat & img
std::vector< `PlateNumResult` >	run	const std::vector< cv::Mat > & imgs

Functions

create

Factory function to get an instance of derived classes of class `PlateNum` .

Prototype

```
std::unique_ptr<
               PlateNum
             > create(const std::string &model_name, bool
need_mean_scale_process=true);
```

Parameters

The following table lists the `create` function arguments.

Table 239: **create Arguments**

Type	Member	Description
const std::string &	model_name	the model name of the created model
bool	need_mean_scale_proces s	normalize with mean/scale or not, true by default.

Returns

An instance of `PlateNum` class.

create

Factory function to get an instance of derived classes of class `PlateNum` .

Prototype

```
std::unique_ptr<
                 PlateNum
                > create(const std::string &model_name, xir::Attrs *attrs,
bool need_mean_scale_process=true);
```

Parameters

The following table lists the `create` function arguments.

Table 240: **create Arguments**

Type	Member	Description
const std::string &	model_name	the model name of the created model
xir::Attrs *	attrs	xir::Attrs pointer points to the provided attributes
bool	need_mean_scale_proces s	normalize with mean/scale or not, true by default.

Returns

An instance of `PlateNum` class.

getInputWidth

Function to get InputWidth of the platenum network (input image columns).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the platenum network.

getInputHeight

Function to get InputHeight of the platenum network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the platenum network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

run

Function of get running result of platenum network.

Prototype

```
                              PlateNumResult
                          run(const cv::Mat &img)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 241: **run Arguments**

Type	Member	Description
const cv::Mat &	img	Input data of input image (cv::Mat) and resized as InputWidth and IntputHeight.

Returns

The plate number and plate color.

run

Function to get running results of the platenum neural network in batch mode.

Prototype

```
std::vector<
               PlateNumResult
             > run(const std::vector< cv::Mat > &imgs)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 242: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	imgs	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch. The input images need to be resized to InputWidth and InputHeight required by the network.

Returns

The vector of PLateNumResult.

vitis::ai::PointPainting

Base class for pointpating.

Input is points data and related params.

Output is a struct of detection results, named PointPaintingResult.

Sample code :

```
  ...
  std::string anno_file_name = "./sample_pointpainting.info";
  PointsInfo points_info;
  std::vector<cv::Mat> images;
  read_inno_file_pointpainting(anno_file_name, points_info, 5,
points_info.sweep_infos, 16, images);
  std::string seg_model = "semanticfpn_nuimage_576_320_pt";
  std::string model_0 = "pointpainting_nuscenes_40000_64_0_pt";
  std::string model_1 = "pointpainting_nuscenes_40000_64_1_pt";
  auto pointpainting = vitis::ai::PointPainting::create(
       seg_model, model_0, model_1);
  auto ret = pointpainting->run(images, points_info);
  ...
please see the test sample for detail.
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::PointPainting` class:

Table 243: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `PointPainting` >	create	const std::string & seg_model_name const std::string & pp_model_name_0 const std::string & pp_model_name_1
int	getInputWidth	void
int	getInputHeight	void
size_t	get_pointpillars_batch	void
size_t	get_segmentation_batch	void
`PointPaintingResult`	run	const std::vector< cv::Mat > & input_images const `vitis::ai::pointpillars_nus::PointsInfo` & points_info
std::vector< `PointPaintingResult` >	run	const std::vector< std::vector< cv::Mat > > & batch_input_images const std::vector< `vitis::ai::pointpillars_nus::PointsInfo` > & batch_points_info
std::vector< cv::Mat >	runSegmentation	std::vector< cv::Mat > batch_images

Table 243: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
std::vector< float >	fusion	const std::vector< cv::Mat > & seg_images const `vitis::ai::pointpillars_nus::PointsInfo` & points_info
`vitis::ai::pointpillars_nus::PointsInfo`	runSegmentationFusion	const std::vector< cv::Mat > & input_images const `vitis::ai::pointpillars_nus::PointsInfo` & points
`PointPaintingResult`	runPointPillars	const `vitis::ai::pointpillars_nus::PointsInfo` & points_info
std::vector< `PointPaintingResult` >	runPointPillars	const std::vector< `vitis::ai::pointpillars_nus::PointsInfo` > & batch_points_info

Functions

create

Factory function to get an instance of derived classes of class `PointPainting`.

Prototype

```
std::unique_ptr<
                PointPainting
              > create(const std::string &seg_model_name, const std::string
&pp_model_name_0, const std::string &pp_model_name_1, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 244: **create Arguments**

Type	Member	Description
const std::string &	seg_model_name	`Segmentation` model name
const std::string &	pp_model_name_0	The first pointpillars nuscenes model name
const std::string &	pp_model_name_1	The second pointpillars nuscenes model name

Returns

An instance of `PointPainting` class.

getInputWidth

Function to get input width of the first model of pointpainting (segmentation model).

Prototype

```
int getInputWidth() const =0;
```

Returns

Input width of the first model (segmentation model).

getInputHeight

Function to get input height of the first model of pointpainting (segmentation model).

Prototype

```
int getInputHeight() const =0;
```

Returns

Input height of the first model (segmentation model).

get_pointpillars_batch

Function to get the number of pointpillars inputs processed by the DPU at one time.

Note: Batch size of different DPU core may be different, it depends on the IP used. For pointpainting class, segmentation model and pointpillars models may be running on different DPU cores.

Prototype

```
size_t get_pointpillars_batch() const =0;
```

Returns

Batch size of pointpillars model.

get_segmentation_batch

Function to get the number of segmentation inputs processed by the DPU at one time.

Note: Batch size of different DPU core may be different, it depends on the IP used. For pointpainting class, segmentation model and pointpillars models may be running on different DPU cores.

Prototype

```
size_t get_segmentation_batch() const =0;
```

Returns

Batch size of segmentation model.

run

Function of get result of the pointpainting full flow.

Prototype

```
               PointPaintingResult
            run(const std::vector< cv::Mat > &input_images, const
vitis::ai::pointpillars_nus::PointsInfo &points_info)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 245: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	input_images	Images from different cameras for segmentation .
const `vitis::ai::pointpillars_nus::PointsInfo` &	points_info	points data and camera related params.

Returns

PointPaintingResult.

run

Function of get result of the pointpainting full flow in batch mode.

Prototype

```
std::vector<
              PointPaintingResult
             > run(const std::vector< std::vector< cv::Mat > >
&batch_input_images, const std::vector<
vitis::ai::pointpillars_nus::PointsInfo > &batch_points_info)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 246: **run Arguments**

Type	Member	Description
const std::vector< std::vector< cv::Mat > > &	batch_input_images	Batch input of images from different cameras for segmentation. The size should be equal to the result of get_pointpillars_batch.
const std::vector< `vitis::ai::pointpillars_nus::PointsInfo` > &	batch_points_info	Batch input of points datas and camera related params.The size should be equal to the result of get_pointpillars_batch.

Returns

A Vector of PointPaintingResult.

runSegmentation

Function of get result of the segmentation in batch mode.

Prototype

```
std::vector< cv::Mat > runSegmentation(std::vector< cv::Mat >
batch_images)=0;
```

Parameters

The following table lists the `runSegmentation` function arguments.

Table 247: **runSegmentation Arguments**

Type	Member	Description
std::vector< cv::Mat >	batch_images	Batch input of images from different cameras for segmentation. The size should be equal to the result of get_segmentation_batch.

Returns

A Vector of segmentation result(cv::Mat).

fusion

Function of get result points fusion.

Prototype

```
std::vector< float > fusion(const std::vector< cv::Mat > &seg_images, const
vitis::ai::pointpillars_nus::PointsInfo &points_info)=0;
```

Parameters

The following table lists the `fusion` function arguments.

Table 248: **fusion Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	seg_images	`Segmentation` result images.
const `vitis::ai::pointpillars_nus::PointsInfo` &	points_info	Points data and camera related params.

Returns

Points data after fusion.

runSegmentationFusion

Function of get result of segmentation and points fusion.

Prototype

```
                    vitis::ai::pointpillars_nus::PointsInfo
                 runSegmentationFusion(const std::vector< cv::Mat >
&input_images, const vitis::ai::pointpillars_nus::PointsInfo &points)=0;
```

Parameters

The following table lists the `runSegmentationFusion` function arguments.

Table 249: **runSegmentationFusion Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	input_images	Images for segmentation

Table 249: **runSegmentationFusion Arguments** *(cont'd)*

Type	Member	Description
const `vitis::ai::pointpillars_nus::PointsInfo` &	points	Points data and camera related params.

Returns

an instance of PointsInfo with points data result

runPointPillars

Function of get result of pointpillars nuscenes neural network.

Prototype

```


                PointPaintingResult
              runPointPillars(const vitis::ai::pointpillars_nus::PointsInfo
&points_info)=0;
```

Parameters

The following table lists the `runPointPillars` function arguments.

Table 250: **runPointPillars Arguments**

Type	Member	Description
const `vitis::ai::pointpillars_nus::PointsInfo` &	points_info	Points data and camera related params.

Returns

PointPaintingResult(same as PointPillarsNuscenesResult).

runPointPillars

Function of get result of pointpillars nuscenes neural network in batch mode.

Prototype

```
std::vector<
                PointPaintingResult
              > runPointPillars(const std::vector<
vitis::ai::pointpillars_nus::PointsInfo > &batch_points_info)=0;
```

Parameters

The following table lists the `runPointPillars` function arguments.

Table 251: **runPointPillars Arguments**

Type	Member	Description
const std::vector< `vitis::ai::pointpillars_nus::PointsInfo` > &	batch_points_info	A batch of Points data and camera related params.

Returns

A Vector of PointPaintingResult(same as PointPillarsNuscenesResult).

vitis::ai::PointPillars

Base class for pointpillars .

Input is points data.

Output is a struct of detection results, named `PointPillarsResult`.

Sample code :

```
    ...
    auto net =
vitis::ai::PointPillars::create("pointpillars_kitti_12000_0_pt",
"pointpillars_kitti_12000_1_pt", true); V1F PointCloud ; int len =
getfloatfilelen( lidar_path); PointCloud.resize( len );
    myreadfile(PointCloud.data(), len, lidar_path);
    auto res = net->run(PointCloud);
    ...
  please see the test sample for detail.
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::PointPillars` class:

Table 252: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `PointPillars` >	create	const std::string & model_name const std::string & model_name1
`vitis::ai::PointPillarsResult`	run	const V1F & v1f

Table 252: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
std::vector< vitis::ai::PointPillarsResult >	run	const V2F & v2f
vitis::ai::PointPillarsResult	run	const float * points
std::vector< vitis::ai::PointPillarsResult >	run	const std::vector< const float * > & vpoints
size_t	get_input_batch	void
void	do_pointpillar_display	PointPillarsResult & res int flag DISPLAY_PARAM & dispp cv::Mat & rgb_map cv::Mat & bev_map int width int height ANNORET & annoret

Functions

create

Factory function to get an instance of derived classes of class `PointPillars`.

Prototype

```
std::unique_ptr<
                PointPillars
              > create(const std::string &model_name, const std::string
&model_name1);
```

Parameters

The following table lists the `create` function arguments.

Table 253: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name for PointNet
const std::string &	model_name1	Model name for RPN

Returns

An instance of `PointPillars` class.

run

Function of get result of the `PointPillars` neural network.

Prototype

```
                vitis::ai::PointPillarsResult
              run(const V1F &v1f)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 254: **run Arguments**

Type	Member	Description
const V1F &	v1f	point data in vector<float>

Returns

`PointPillarsResult`.

run

Function of get result of the `PointPillars` neural network in batch mode.

Prototype

```
std::vector<
                vitis::ai::PointPillarsResult
              > run(const V2F &v2f)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 255: **run Arguments**

Type	Member	Description
const V2F &	v2f	vector of point data in vector<float>

Returns

vector of `PointPillarsResult`.

run

Function of get result of the `PointPillars` neural network.

Prototype

```
                  vitis::ai::PointPillarsResult
                run(const float *points, int len)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 256: **run Arguments**

Type	Member	Description
const float *	points	point data refered by float* @len: length of the points data (float data length, not byte data length)

Returns

`PointPillarsResult`.

run

Function of get result of the `PointPillars` neural network in batch mode.

Prototype

```
std::vector<
                  vitis::ai::PointPillarsResult
                > run(const std::vector< const float * > &vpoints, const
std::vector< int > &vlen)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 257: **run Arguments**

Type	Member	Description
const std::vector< const float * > &	vpoints	vector of point data refered by float* @vlen: vector of length of the points data (float data length, not byte data length)

Returns

vector of PointPillarsResult .

get_input_batch

Function to get input batch of the PointPillars network.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Input batch of the PointPillars network.

do_pointpillar_display

Function to produce the visible result from PointPillarsResult after calling run() . This is a helper function which can be ignored if you wants to process the PointPillarsResult using another method.

Prototype

```
void do_pointpillar_display(PointPillarsResult &res, int flag,
DISPLAY_PARAM &dispp, cv::Mat &rgb_map, cv::Mat &bev_map, int width, int
height, ANNORET &annoret)=0;
```

Parameters

The following table lists the do_pointpillar_display function arguments.

Table 258: **do_pointpillar_display Arguments**

Type	Member	Description
PointPillarsResult &	res	[input] PointPillarsResult from run() .
int	flag	[input] which visible result to produce. can be assigned to below values: E_BEV : only produce BEV picture E_RGB : only produce RGB picture E_BEV|E_RGB : produce both pictures
DISPLAY_PARAM &	dispp	[input] display parameter for the Points data. Refer to the readme in the overview for more detail.
cv::Mat &	rgb_map	[input|output] : original rgb picture for drawing detect result. It can be blank (cv::Mat{}), if only BEV is required
cv::Mat &	bev_map	[input|output] original bev picture for drawing detect result. It can be blank (cv::Mat{}), if only RGB required
int	width	[input] original rgb picture width.
int	height	[input] original rgb picture height.
ANNORET &	annoret	[output] return the annoret variable for accuracy calculation.

vitis::ai::PointPillarsNuscenes

Base class for pointpillars_nuscenes .

Input is points data and related params.

Output is a struct of detection results, named `PointPillarsNuscenesResult`.

Sample code :

```
    ...
    std::string anno_file_name = "./sample_pointpillars_nus.info";
    PointsInfo points_info;
    std::string model_0 = "pointpillars_nuscenes_40000_64_0_pt";
    std::string model_1 = "pointpillars_nuscenes_40000_64_1_pt";
    auto pointpillars = vitis::ai::PointPillarsNuscenes::create(
         model_0, model_1);
    auto points_dim = pointpillars->getPointsDim();
    read_inno_file_pp_nus(anno_file_name, points_info, points_dim,
points_info.sweep_infos);

    auto ret = pointpillars->run(points_info);

    ...
  please see the test sample for detail.
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::PointPillarsNuscenes` class:

Table 259: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `PointPillarsNuscenes` >	create	const std::string & model_name_0 const std::string & model_name_1 bool need_preprocess
std::unique_ptr< `PointPillarsNuscenes` >	create	const std::string & model_name_0 const std::string & model_name_1 xir::Attrs * attrs bool need_preprocess
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void

Table 259: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
int	getPointsDim	void
std::vector< float >	sweepsFusionFilter	const `vitis::ai::pointpillars_nus::PointsInfo` & input
std::vector< std::vector< float > >	sweepsFusionFilter	const std::vector< `vitis::ai::pointpillars_nus::PointsInfo` > & batch_input
`PointPillarsNuscenesResult`	run	const std::vector< float > & input_points
std::vector< `PointPillarsNuscenesResult` >	run	const std::vector< std::vector< float > > & batch_points
`PointPillarsNuscenesResult`	run	const `vitis::ai::pointpillars_nus::PointsInfo` & input
std::vector< `PointPillarsNuscenesResult` >	run	const std::vector< `vitis::ai::pointpillars_nus::PointsInfo` > & batch_input

Functions

create

Factory function to get an instance of derived classes of class `PointPillarsNuscenes` .

Prototype

```
std::unique_ptr<
                PointPillarsNuscenes
              > create(const std::string &model_name_0, const std::string
&model_name_1, bool need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 260: **create Arguments**

Type	Member	Description
const std::string &	model_name_0	The first model name

Table 260: **create Arguments** *(cont'd)*

Type	Member	Description
const std::string &	model_name_1	The second model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `PointPillarsNuscenes` class.

create

Factory function to get an instance of derived classes of class `PointPillarsNuscenes`.

Prototype

```
std::unique_ptr<
                PointPillarsNuscenes
              > create(const std::string &model_name_0, const std::string
&model_name_1, xir::Attrs *attrs, bool need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 261: **create Arguments**

Type	Member	Description
const std::string &	model_name_0	The first model name
const std::string &	model_name_1	The second model name
xir::Attrs *	attrs	XIR attributes, used to bind different models to the same dpu core
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `PointPillarsNuscenes` class.

getInputWidth

Function to get input width of the first model of `PointPillarsNuscenes` class.

Prototype

```
int getInputWidth() const =0;
```

Returns

Input width of the first model.

getInputHeight

Function to get input height of the first model of `PointPillarsNuscenes` class.

Prototype

```
int getInputHeight() const =0;
```

Returns

Input height of the first model.

get_input_batch

Function to get the number of inputs processed by the DPU at one time.

Note: Batch size of different DPU core may be different, it depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

getPointsDim

Function to get the points dim of an input points. The dim depends on the last channel of the first model of `PointPillarsNuscenes` network.

Prototype

```
int getPointsDim() const =0;
```

Returns

The dim of points.

sweepsFusionFilter

Function to get filtered sweeps points data in batch mode.

Prototype

```
std::vector< float > sweepsFusionFilter(const
vitis::ai::pointpillars_nus::PointsInfo &input)=0;
```

Parameters

The following table lists the `sweepsFusionFilter` function arguments.

Table 262: **sweepsFusionFilter Arguments**

Type	Member	Description
const `vitis::ai::pointpillars_nus::PointsInfo` &	input	An object of structure PointsInfo, include points data and other params.

Returns

The vector of filtered points.

sweepsFusionFilter

Function to get filtered sweep points data.

Prototype

```
std::vector< std::vector< float > > sweepsFusionFilter(const std::vector<
vitis::ai::pointpillars_nus::PointsInfo > &batch_input)=0;
```

Parameters

The following table lists the `sweepsFusionFilter` function arguments.

Table 263: **sweepsFusionFilter Arguments**

Type	Member	Description
const std::vector< `vitis::ai::pointpillars_nus::PointsInfo` > &	batch_input	Vector of PointsInfo, the size of batch_input should be equal to batch num.

Returns

Filtered points.

run

Function of get result of the `PointPillarsNuscenes` neural network.

Prototype

```
               PointPillarsNuscenesResult
             run(const std::vector< float > &input_points)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 264: **run Arguments**

Type	Member	Description
const std::vector< float > &	input_points	Filtered points data.

Returns

`PointPillarsNuscenesResult`.

run

Function of get result of the `PointPillarsNuscenes` neural network in batch mode.

Prototype

```
std::vector<
               PointPillarsNuscenesResult
             > run(const std::vector< std::vector< float > >
&batch_points)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 265: **run Arguments**

Type	Member	Description
const std::vector< std::vector< float > > &	batch_points	Filtered points data in batch mode.

Returns

The vector of `PointPillarsNuscenesResult`.

run

Function of get result of the `PointPillarsNuscenes` neural network.

Prototype

```
                PointPillarsNuscenesResult
              run(const vitis::ai::pointpillars_nus::PointsInfo &input)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 266: **run Arguments**

Type	Member	Description
const `vitis::ai::pointpillars_nus::PointsInfo` &	input	An object of structure PointsInfo, include points data and other params.

Returns

`PointPillarsNuscenesResult`.

run

Function of get result of the `PointPillarsNuscenes` neural network.

Prototype

```
std::vector<
                PointPillarsNuscenesResult
              > run(const std::vector<
vitis::ai::pointpillars_nus::PointsInfo > &batch_input)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 267: **run Arguments**

Type	Member	Description
const std::vector< `vitis::ai::pointpillars_nus::PointsInfo` > &	batch_input	Vector of PointsInfo, the size of batch_input should be equal to batch num.

Returns

The vector of `PointPillarsNuscenesResult`.

vitis::ai::PointPillarsNuscenesPostProcess

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::PointPillarsNuscenesPostProcess` class:

Table 268: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< PointPillarsNuscenesPostProcess >	create	const std::vector< vitis::ai::library::InputTensor > & input_tensors const std::vector< vitis::ai::library::OutputTensor > & output_tensors const vitis::ai::proto::DpuModelParam & config
std::vector< `PointPillarsNuscenesResult` >	postprocess	void

Functions

create

Create an PointPillarsNuscenesPostProcess object.

Prototype

```
std::unique_ptr< PointPillarsNuscenesPostProcess > create(const
std::vector< vitis::ai::library::InputTensor > &input_tensors, const
std::vector< vitis::ai::library::OutputTensor > &output_tensors, const
vitis::ai::proto::DpuModelParam &config);
```

Parameters

The following table lists the `create` function arguments.

Table 269: **create Arguments**

Type	Member	Description
const std::vector< vitis::ai::library::InputTensor > &	input_tensors	A vector of all input-tensors in the network. Usage: input_tensors[input_tensor_index].
const std::vector< vitis::ai::library::OutputTensor > &	output_tensors	A vector of all output-tensors in the network. Usage: output_tensors[output_index].

Table 269: **create Arguments** *(cont'd)*

Type	Member	Description
const vitis::ai::proto::DpuModelParam &	config	The DPU model configuration information.

Returns

An unique pointer of PointPillarsNuscenesPostProcess.

postprocess

The batch mode post-processing function of the `PointPillarsNuscenes` network.

Prototype

```
std::vector<
               PointPillarsNuscenesResult
             > postprocess(size_t batch_size)=0;
```

Returns

The vector of struct of `PointPillarsNuscenesResult` .

vitis::ai::PolypSegmentation

Base class for detecting objects in the input image(cv::Mat). Input is an image(cv::Mat). Output is the position of the objects in the input image. Sample code:

```
 auto img = cv::imread("sample_yolov2.jpg");
 auto model = vitis::ai::PolypSegmentation::create("yolov2_voc");
 auto result = model->run(img);
 for (const auto &bbox : result.bboxes) {
   int label = bbox.label;
   float xmin = bbox.x * img.cols + 1;
   float ymin = bbox.y * img.rows + 1;
   float xmax = xmin + bbox.width * img.cols;
   float ymax = ymin + bbox.height * img.rows;
   if (xmax > img.cols)
     xmax = img.cols;
   if (ymax > img.rows)
     ymax = img.rows;
   float confidence = bbox.score;

   cout << "RESULT: " << label << "\t" << xmin << "\t" << ymin << "\t" <<
xmax
```

```
        << "\t" << ymax << "\t" << confidence << "\n";
    rectangle(img, Point(xmin, ymin), Point(xmax, ymax), Scalar(0, 255, 0),
1,
                1, 0);
}
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::PolypSegmentation` class:

Table 270: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< PolypSegmentation >	create	const std::string & model_name bool need_preprocess
SegmentationResult	run	const cv::Mat & image
std::vector< SegmentationResult >	run	const std::vector< cv::Mat > & images

Functions

create

Factory function to get an instance of derived classes of class `PolypSegmentation`.

Prototype

```
std::unique_ptr<
                  PolypSegmentation
                > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 271: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `PolypSegmentation` class.

run

Function to get running result of the `PolypSegmentation` neural network.

Prototype

```
                    SegmentationResult
                 run(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 272: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of input image (cv::Mat).

Returns

A Struct of `SegmentationResult` .

run

Function to get running result of the `PolypSegmentation` neural network in batch mode.

Prototype

```
std::vector<
                    SegmentationResult
                 > run(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 273: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `SegmentationResult` .

vitis::ai::PoseDetect

Base class for detecting a pose from an input image (cv::Mat).

Input an image (cv::Mat).

Note: Support detect a single pose.

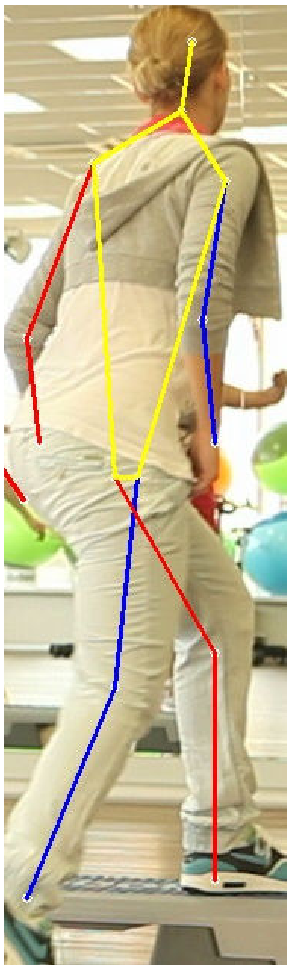
Output is a struct of `PoseDetectResult`, include 14 point.

Sample code:

```
auto det = vitis::ai::PoseDetect::create("sp_net");
auto image = cv::imread("sample_posedetect.jpg");
auto results = det->run(image);
for(auto result: results.pose14pt) {
    std::cout << result << std::endl;
}
```

Display of the posedetect model results:

Figure 67: **pose detect image**



Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::PoseDetect` class:

Table 274: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `PoseDetect` >	create	const std::string & model_name bool need_preprocess
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void

Table 274: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
`PoseDetectResult`	run	const cv::Mat & image
std::vector< `PoseDetectResult` >	run	const std::vector< cv::Mat > & images

Functions

create

Factory function to get an instance of derived classes of class `PoseDetect` .

Prototype

```
std::unique_ptr<
                PoseDetect
              > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 275: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name .
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `PoseDetect` class.

getInputWidth

Function to get InputWidth of the `PoseDetect` network (input image columns).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the `PoseDetect` network.

getInputHeight

Function to get InputHeight of the `PoseDetect` network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the `PoseDetect` network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

run

Function to get running results of the posedetect neural network.

Prototype

```
                        PoseDetectResult
                      run(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 276: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of input image (cv::Mat).

Returns

PoseDetectResult.

run

Function to get running results of the posedetect neural network in batch mode.

Prototype

```
std::vector<
               PoseDetectResult
             > run(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 277: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of PoseDetectResult.

vitis::ai::Rcan

Base class for detecting rcan from an image (cv::Mat).

Input is an image (cv::Mat).

Output is the enlarged image.

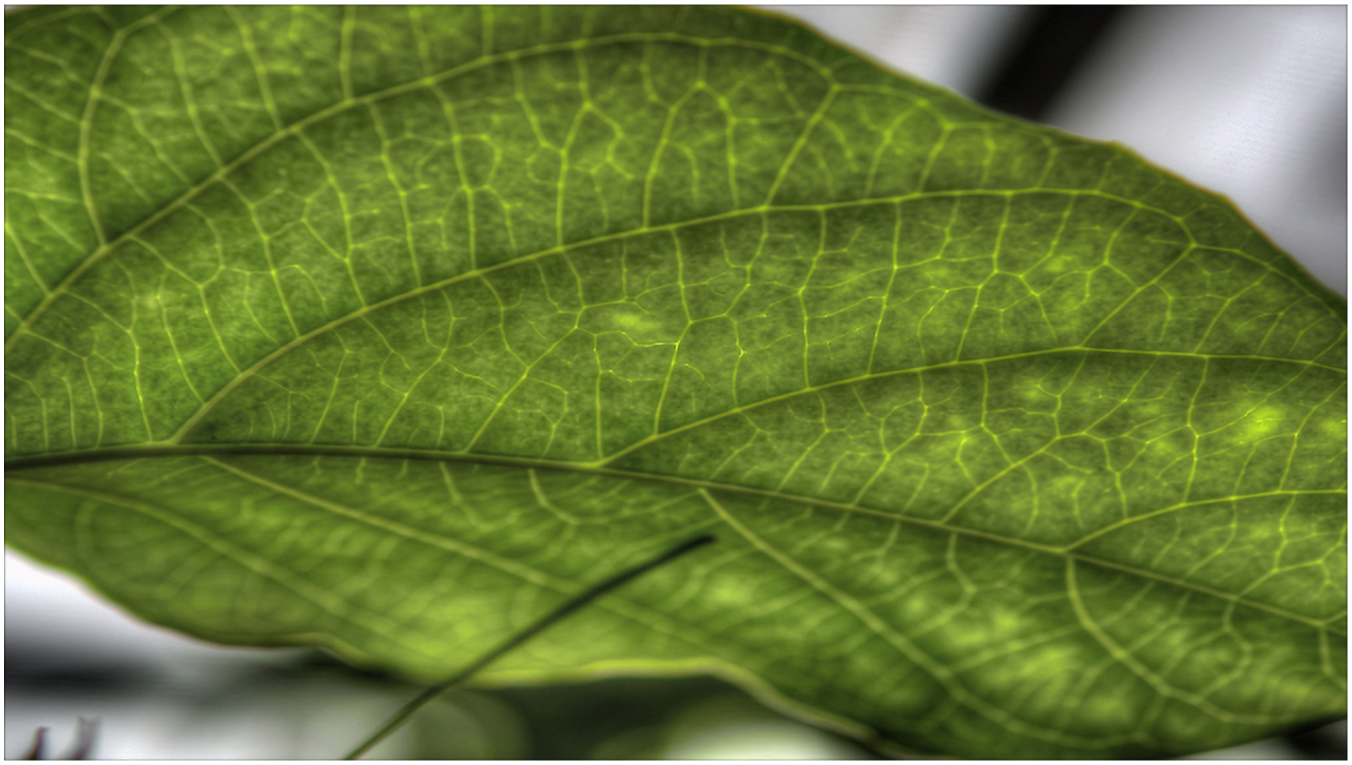
Sample code:

***Note*:** The input image size is 640x360

```
if (argc < 2) {
  cerr << "usage: " << argv[0] << "  modelname  image_file_url " << endl;
  abort();
}
Mat input_img = imread(argv[2]);
if (input_img.empty()) {
  cerr << "can't load image! " << argv[2] << endl;
  return -1;
}
auto det = vitis::ai::Rcan::create(argv[1]);
Mat ret_img = det->run(input_img).feat;
imwrite("sample_rcan_result.png", ret_img);
```

Display of the model results:

Figure 68: **result image**



Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::Rcan` class:

Table 278: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `Rcan` >	create	const std::string & model_name bool need_preprocess
`RcanResult`	run	const cv::Mat & image
std::vector< `RcanResult` >	run	const std::vector< cv::Mat > & images

Table 278: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
std::vector< RcanResult >	run	const std::vector< vart::xrt_bo_t > & input_bos

Functions

create

Factory function to get an instance of derived classes of class Rcan .

Prototype

```
std::unique_ptr<
                    Rcan
                > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the create function arguments.

Table 279: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of Rcan class.

run

Function to get running result of the RCAN neural network.

Prototype

```


                    RcanResult
                run(const cv::Mat &image)=0;
```

Parameters

The following table lists the run function arguments.

Table 280: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of input image (cv::Mat).

Returns

`RcanResult` .

run

Function to get running result of the RCAN neural network in batch mode.

Prototype

```
std::vector<
              RcanResult
             > run(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 281: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (vector<cv::Mat>).

Returns

vector of `RcanResult` .

run

Function to get running results of the rcan neural network in batch mode , used to receive user's xrt_bo to support zero copy.

Prototype

```
std::vector<
              RcanResult
             > run(const std::vector< vart::xrt_bo_t > &input_bos)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 282: **run Arguments**

Type	Member	Description
const std::vector< vart::xrt_bo_t > &	input_bos	The vector of vart::xrt_bo_t.

Returns

The vector of `RcanResult`.

vitis::ai::RefineDet

Base class for detecting pedestrians in the input image (cv::Mat).

Input is an image (cv::Mat).

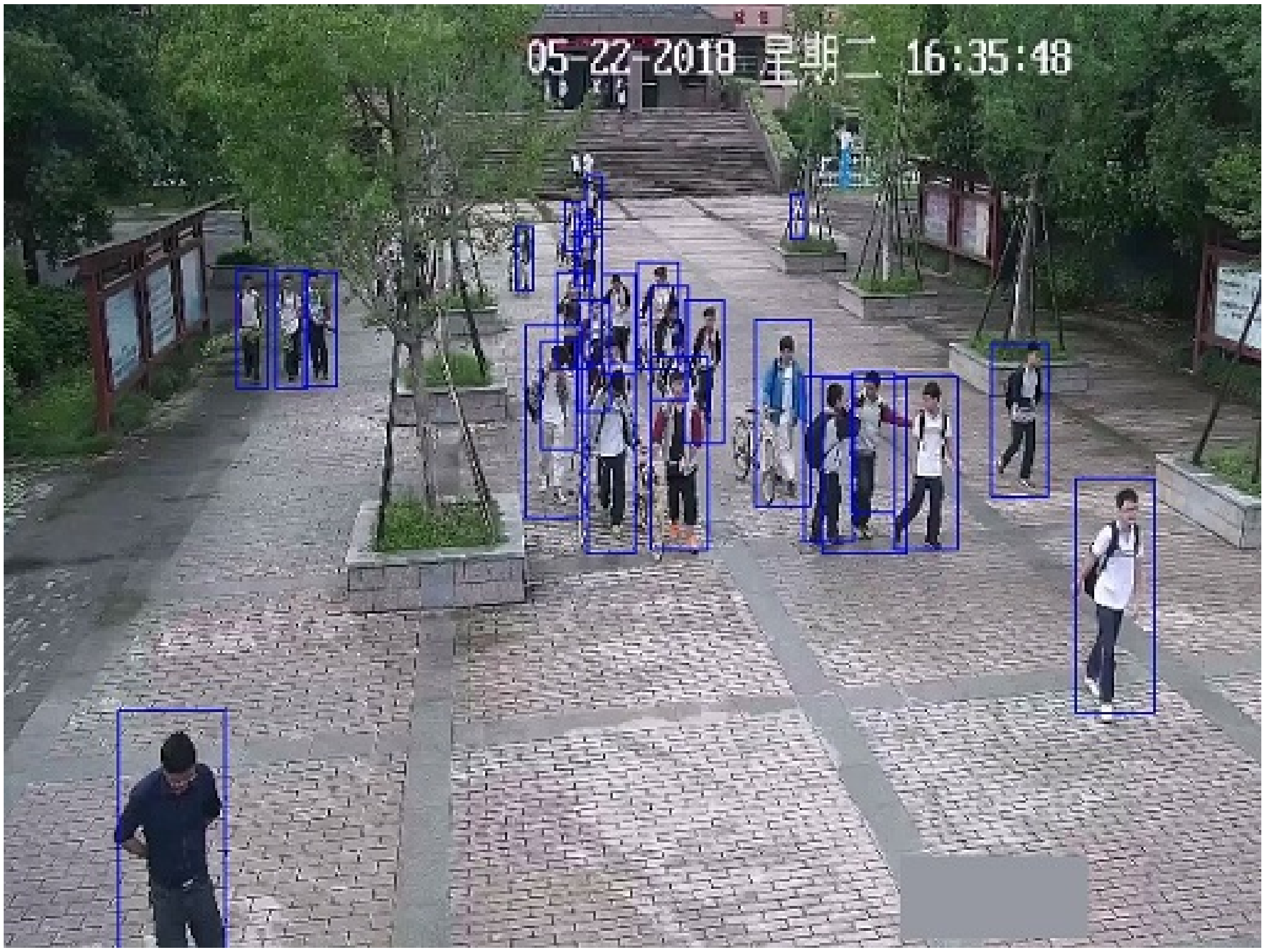
Output is the position and score of the pedestrians in the input image.

Sample code:

```
auto det = vitis::ai::RefineDet::create("refinedet_pruned_0_8");
auto image = cv::imread("sample_refinedet.jpg");
cout << "load image" << endl;
if (image.empty()) {
  cerr << "cannot load " << argv[1] << endl;
  abort();
}

auto results = det->run(image);

auto img = image.clone();
for (auto &box : results.bboxes) {
    float x = box.x * (img.cols);
    float y = box.y * (img.rows);
    int xmin = x;
    int ymin = y;
    int xmax = x + (box.width) * (img.cols);
    int ymax = y + (box.height) * (img.rows);
    float score = box.score;
    xmin = std::min(std::max(xmin, 0), img.cols);
    xmax = std::min(std::max(xmax, 0), img.cols);
    ymin = std::min(std::max(ymin, 0), img.rows);
    ymax = std::min(std::max(ymax, 0), img.rows);

    cv::rectangle(img, cv::Point(xmin, ymin), cv::Point(xmax, ymax),
                    cv::Scalar(0, 255, 0), 1, 1, 0);
}
auto out = "sample_refinedet_result.jpg";
LOG(INFO) << "write result to " << out;
cv::imwrite(out, img);
```

Display of the model results:

Figure 69: **result image**



Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::RefineDet` class:

Table 283: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `RefineDet` >	create	const std::string & model_name bool need_preprocess
`RefineDetResult`	run	const cv::Mat & image
std::vector< `RefineDetResult` >	run	const std::vector< cv::Mat > & images

Table 283: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
std::vector< vitis::ai::RefineDetResult >	run	const std::vector< vart::xrt_bo_t > & input_bos

Functions

create

Factory function to get an instance of derived classes of class `RefineDet` .

Prototype

```
std::unique_ptr<
                 RefineDet
               > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 284: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `RefineDet` class.

run

Function to get running result of the `RefineDet` neural network.

Prototype

```
                 RefineDetResult
               run(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 285: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of input image (cv::Mat).

Returns

A Struct of `RefineDetResult` .

run

Function to get running result of the `RefineDet` neural network in batch mode.

Prototype

```
std::vector<
               RefineDetResult
             > run(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 286: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (vector<cv::Mat>).

Returns

vector of Struct of `RefineDetResult` .

run

Function to get running results of the `RefineDet` neural network in batch mode, used to receive user's xrt_bo to support zero copy.

Prototype

```
std::vector<
               vitis::ai::RefineDetResult
             > run(const std::vector< vart::xrt_bo_t > &input_bos)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 287: **run Arguments**

Type	Member	Description
const std::vector< vart::xrt_bo_t > &	input_bos	The vector of vart::xrt_bo_t.

Returns

The vector of `RefineDetResult` .

vitis::ai::RefineDetPostProcess

Class of the refinedet post-process. It initializes the parameters once instead of computing them each time the program executes.

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::RefineDetPostProcess` class:

Table 288: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `RefineDetPostProcess` >	create	const std::vector< vitis::ai::library::InputTensor > & input_tensors const std::vector< vitis::ai::library::OutputTensor > & output_tensors const vitis::ai::proto::DpuModelParam & config
std::vector< `RefineDetResult` >	refine_det_post_process	void

Functions

create

Create an `RefineDetPostProcess` object.

Prototype

```
std::unique_ptr<
                RefineDetPostProcess
              > create(const std::vector< vitis::ai::library::InputTensor >
&input_tensors, const std::vector< vitis::ai::library::OutputTensor >
&output_tensors, const vitis::ai::proto::DpuModelParam &config);
```

Parameters

The following table lists the `create` function arguments.

Table 289: **create Arguments**

Type	Member	Description
const std::vector< vitis::ai::library::InputTensor > &	input_tensors	A vector of all input-tensors in the network. Usage: input_tensors[input_tensor_index].
const std::vector< vitis::ai::library::OutputTensor > &	output_tensors	A vector of all output-tensors in the network. Usage: output_tensors[output_index].
const vitis::ai::proto::DpuModelParam &	config	The DPU model configuration information.

Returns

A unique pointer of `RefineDetPostProcess` .

refine_det_post_process

Run batch mode of refinedet post-process.

Prototype

```
std::vector<
                RefineDetResult
              > refine_det_post_process(size_t batch_size)=0;
```

Returns

The vector of struct of `RefineDetResult` .

vitis::ai::Reid

Base class for getting feat from an image (cv::Mat).

Input is an image (cv::Mat).

Output image reid feat.

Sample code:

```
if(argc < 3){
    cerr<<"need two images"<<endl;
    return -1;
}
Mat imgx = imread(argv[1]);
if(imgx.empty()){
    cerr<<"can't load image! "<<argv[1]<<endl;
    return -1;
}
Mat imgy = imread(argv[2]);
if(imgy.empty()){
    cerr<<"can't load image! "<<argv[2]<<endl;
    return -1;
}
auto det = vitis::ai::Reid::create("reid");
Mat featx = det->run(imgx).feat;
Mat featy = det->run(imgy).feat;
double dismat= cosine_distance(featx, featy);
printf("dismat : %.3lf \n", dismat);
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::Reid` class:

Table 290: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `Reid` >	create	const std::string & model_name bool need_preprocess
`ReidResult`	run	const cv::Mat & image
std::vector< `ReidResult` >	run	const std::vector< cv::Mat > & images
std::vector< `ReidResult` >	run	const std::vector< vart::xrt_bo_t > & input_bos

Functions

create

Factory function to get an instance of derived classes of class `Reid`.

Prototype

```
std::unique_ptr<
                Reid
              > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 291: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `Reid` class.

run

Function to get running result of the ReID neural network.

Prototype

```


                ReidResult
              run(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 292: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of input image (cv::Mat).

Returns

`ReidResult` .

run

Function to get running result of the ReID neural network in batch mode.

Prototype

```
std::vector<
               ReidResult
             > run(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 293: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (vector<cv::Mat>).

Returns

vector of `ReidResult` .

run

Function to get running results of the reid neural network in batch mode , used to receive user's xrt_bo to support zero copy.

Prototype

```
std::vector<
               ReidResult
             > run(const std::vector< vart::xrt_bo_t > &input_bos)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 294: **run Arguments**

Type	Member	Description
const std::vector< vart::xrt_bo_t > &	input_bos	The vector of vart::xrt_bo_t.

Returns

The vector of `ReidResult` .

vitis::ai::RetinaFace

Base class for detecting the position,score and landmark of faces in the input image (cv::Mat).

Input is an image (cv::Mat).

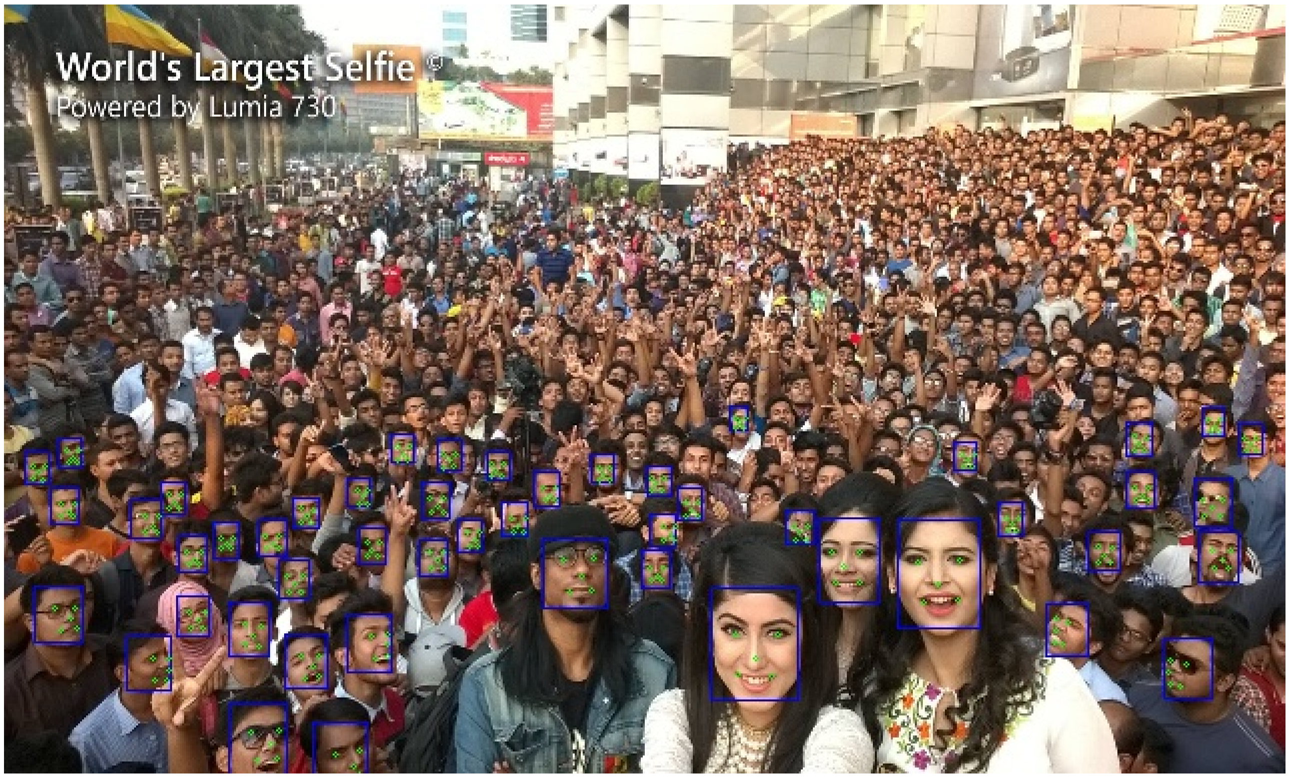
Output is a vector of position and score for faces in the input image.

Sample code:

```
auto image = cv::imread("sample_retinaface.jpg");
auto network = vitis::ai::RetinaFace::create(
               "retinaface",
               true);
auto result = network->run(image);
for (auto i = 0u; i < result.bboxes.size(); ++i) {
   auto score = result.bboxes[i].score;
   auto x = result.bboxes[i].x * image.cols;
   auto y = result.bboxes[i].y * image.rows;
   auto width = result.bboxes[i].width * image.cols;
   auto height = result.bboxes[i].height * image.rows;
   auto landmark = results.landmarks[i];
   for (auto j = 0; j < 5; ++j) {
     auto px = landmark[j].first * image.cols;
     auto py = landmark[j].second * image.rows;
   }
}
```

Display of the model results: width=\textwidth

Figure 70: **result image**



Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::RetinaFace` class:

Table 295: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `RetinaFace` >	create	const std::string & model_name bool need_preprocess
std::unique_ptr< `RetinaFace` >	create	const std::string & model_name xir::Attrs * attrs bool need_preprocess
`RetinaFaceResult`	run	const cv::Mat & img
std::vector< `RetinaFaceResult` >	run	const std::vector< cv::Mat > & imgs
std::vector< `RetinaFaceResult` >	run	const std::vector< vart::xrt_bo_t > & input_bos

Functions

create

Factory function to get an instance of derived classes of class `RetinaFace`.

Prototype

```
std::unique_ptr<
                  RetinaFace
                > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 296: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `RetinaFace` class.

create

Factory function to get an instance of derived classes of class `RetinaFace`.

Prototype

```
std::unique_ptr<
                  RetinaFace
                > create(const std::string &model_name, xir::Attrs *attrs,
bool need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 297: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
xir::Attrs *	attrs	Xir attributes

Table 297: **create Arguments** *(cont'd)*

Type	Member	Description
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `RetinaFace` class.

run

Function to get running result of the retinaface network.

Prototype

```
                RetinaFaceResult
              run(const cv::Mat &img)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 298: **run Arguments**

Type	Member	Description
const cv::Mat &	img	Input Data ,input image (cv::Mat) need to be resized to InputWidth and InputHeight required by the network.

Returns

The detection result of the face detect network , filtered by score >= det_threshold

run

Function to get running results of the retinaface neural network in batch mode.

Prototype

```
std::vector<
                RetinaFaceResult
              > run(const std::vector< cv::Mat > &imgs)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 299: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	imgs	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch. The input images need to be resized to InputWidth and InputHeight required by the network.

Returns

The vector of `RetinaFaceResult` .

run

Function to get running results of the retina neural network in batch mode , used to receive user's xrt_bo to support zero copy.

Prototype

```
std::vector<
               RetinaFaceResult
             > run(const std::vector< vart::xrt_bo_t > &input_bos)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 300: **run Arguments**

Type	Member	Description
const std::vector< vart::xrt_bo_t > &	input_bos	The vector of vart::xrt_bo_t.

Returns

The vector of RetinaFacesResult.

vitis::ai::RetinaFacePostProcess

Class of the retinaface post-process. It initializes the parameters once instead of computing them each time the program executes.

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::RetinaFacePostProcess` class:

Table 301: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `RetinaFacePostProcess` >	create	const std::vector< vitis::ai::library::InputTensor > & input_tensors const std::vector< vitis::ai::library::OutputTensor > & output_tensors const vitis::ai::proto::DpuModelParam & config
std::vector< `RetinaFaceResult` >	retinaface_post_process	void

Functions

create

Create a `RetinaFacePostProcess` object.

Prototype

```
std::unique_ptr<
               RetinaFacePostProcess
             > create(const std::vector< vitis::ai::library::InputTensor >
&input_tensors, const std::vector< vitis::ai::library::OutputTensor >
&output_tensors, const vitis::ai::proto::DpuModelParam &config);
```

Parameters

The following table lists the `create` function arguments.

Table 302: **create Arguments**

Type	Member	Description
const std::vector< vitis::ai::library::InputTensor > &	input_tensors	A vector of all input-tensors in the network. Usage: input_tensors[input_tensor_index].
const std::vector< vitis::ai::library::OutputTensor > &	output_tensors	A vector of all output-tensors in the network. Usage: output_tensors[output_index].
const vitis::ai::proto::DpuModelParam &	config	The DPU model configuration information.

Returns

A unique pointer of RetinaFacePostProcess .

retinaface_post_process

The batch mode post-processing function of the retinaface network.

Prototype

```
std::vector<
                RetinaFaceResult
              > retinaface_post_process(size_t batch_size)=0;
```

Returns

The vector of struct of RetinaFaceResult .

vitis::ai::RGBDsegmentation

Base class for RGBDsegmentation.

Input is a pair images which are RGB image (cv::Mat) and HHA map generated with depth map (cv::Mat).

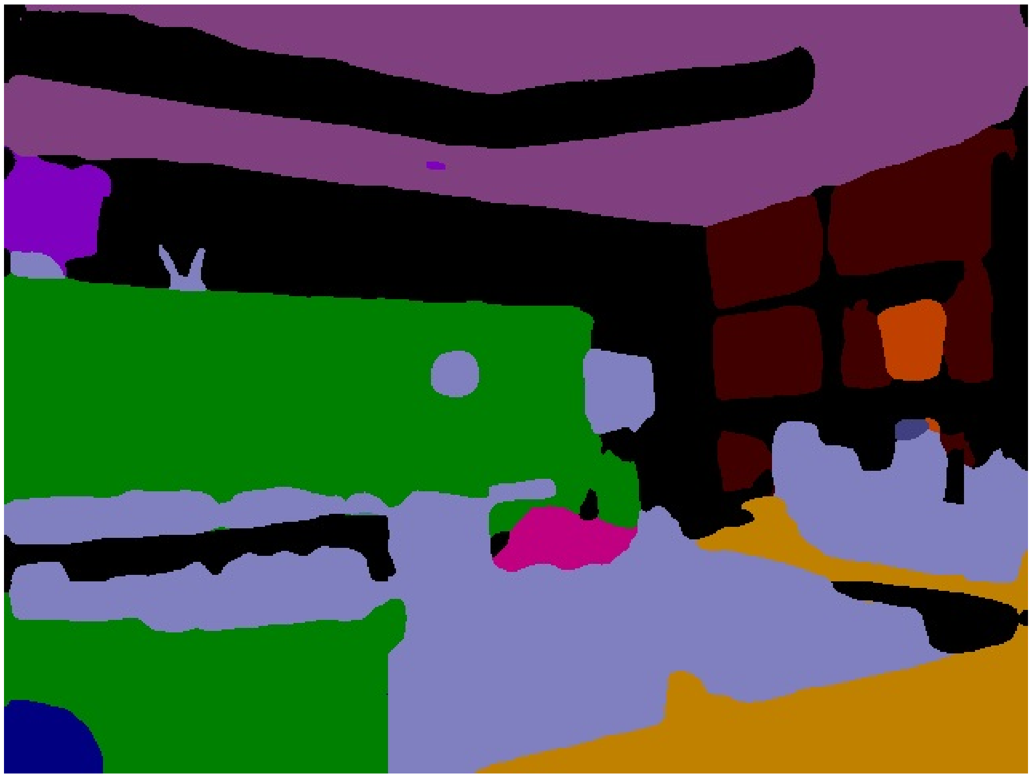
Output is a heatmap where each pixels is predicted with a semantic category, like chair, bed, usual object in indoor.

Sample code:

```
Mat img_bgr = cv::imread("sample_rgbdsegmentation_bgr.jpg");
Mat img_hha = cv::imread("sample_rgbdsegmentation_hha.jpg");

auto segmentation = vitis::ai::RGBDsegmentation::create("SA-Gate_pt", true);

auto result = segmentation->run(img_bgr, img_hha);

imwrite("result.jpg", result.segmentation);
```

Display of the model results: width=\textwidth

Figure 71: **out image**



Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::RGBDsegmentation` class:

Table 303: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `RGBDsegmentation` >	create	const std::string & model_name bool need_preprocess
`SegmentationResult`	run	const cv::Mat & image_bgr const cv::Mat & image_hha

Functions

create

Factory function to get an instance of derived classes of class `RGBDsegmentation`.

Prototype

```
std::unique_ptr<
                RGBDsegmentation
              > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 304: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `RGBDsegmentation` class.

run

Function to get running result of the `RGBDsegmentation` neural network.

Prototype

```


                SegmentationResult
              run(const cv::Mat &image_bgr, const cv::Mat &image_hha)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 305: **run Arguments**

Type	Member	Description
const cv::Mat &	image_bgr	Input data of input image (cv::Mat).
const cv::Mat &	image_hha	Input data of input image_hha (cv::Mat).

Returns

`SegmentationResult`.

vitis::ai::RoadLine

Base class for detecting lanedetect from an image (cv::Mat).

Input is an image (cv::Mat).

Output road line type and points marked road line.

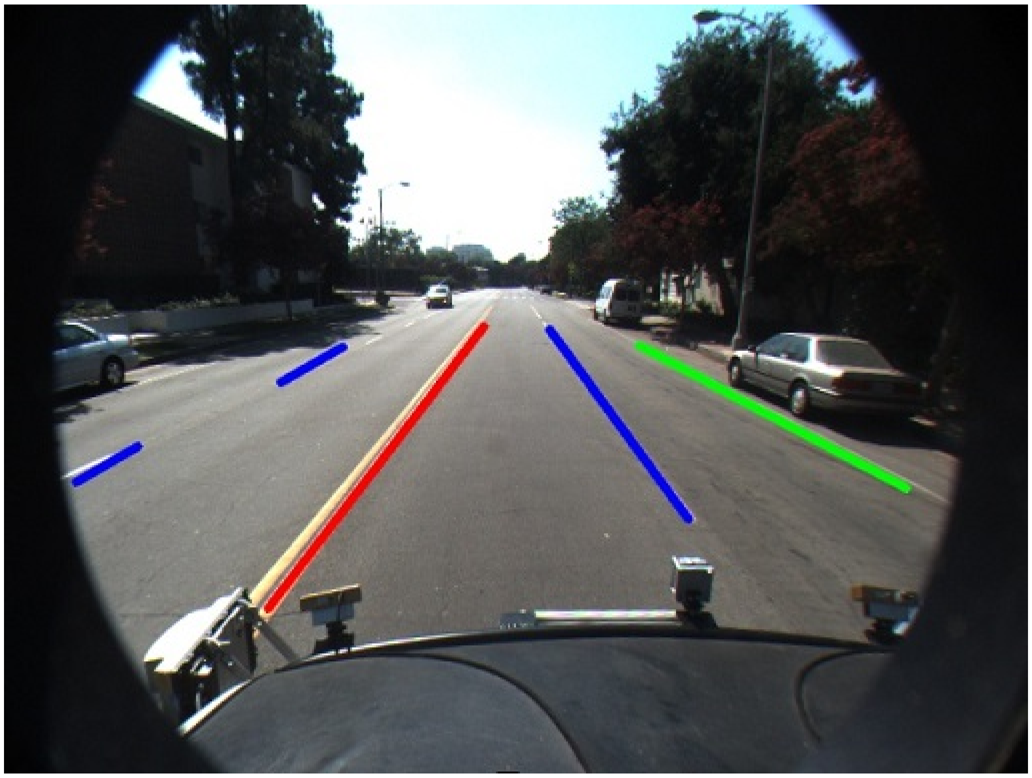
Sample code:

***Note*:** The input image size is 640x480

```
 auto det = vitis::ai::RoadLine::create("vpgnet_pruned_0_99");
 auto image = cv::imread("sample_lanedetect.jpg");
 //    Mat image;
 //    resize(img, image, Size(640, 480));
 if (image.empty()) {
   cerr << "cannot load " << argv[1] << endl;
   abort();
 }

 vector<int> color1 = {0, 255, 0, 0, 100, 255};
 vector<int> color2 = {0, 0, 255, 0, 100, 255};
 vector<int> color3 = {0, 0, 0, 255, 100, 255};

 RoadLineResult results = det->run(image);
 for (auto &line : results.lines) {
   vector<Point> points_poly = line.points_cluster;
   // for (auto &p : points_poly) {
   //  std::cout << p.x << " " << (int)p.y << std::endl;
   //}
   int type = line.type < 5 ? line.type : 5;
   if (type == 2 && points_poly[0].x < image.rows * 0.5)
     continue;
   cv::polylines(image, points_poly, false,
                 Scalar(color1[type], color2[type], color3[type]), 3,
cv::LINE_AA, 0);
 }
```

Display of the model results:

Figure 72: **result image**



Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::RoadLine` class:

Table 306: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `RoadLine` >	create	const std::string & model_name bool need_preprocess
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void
`RoadLineResult`	run	const cv::Mat & image
std::vector< `RoadLineResult` >	run	const std::vector< cv::Mat > & images

Functions

create

Factory function to get an instance of derived classes of class `RoadLine` .

Prototype

```
std::unique_ptr<
                  RoadLine
                > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 307: **create Arguments**

Type	Member	Description
const std::string &	model_name	String of model name
bool	need_preprocess	normalize with mean/scale or not, default value is true.

Returns

An instance of `RoadLine` class.

getInputWidth

Function to get InputWidth of the lanedetect network (input image columns).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the lanedetect network.

getInputHeight

Function to get InputHeight of the lanedetect network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the lanedetect network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

run

Function to get running result of the `RoadLine` network.

Prototype

```
                    RoadLineResult
                  run(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 308: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input data , input image (cv::Mat) need to resized as 640x480.

Returns

The struct of `RoadLineResult`

run

Function to get running result of the `RoadLine` network in batch mode.

Prototype

```
std::vector<
                RoadLineResult
              > run(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 309: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (vector<cv::Mat>).The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `RoadLineResult`

vitis::ai::RoadLinePostProcess

Class of the roadline post-process. It will initializes the parameters once instead of computing them each time the program executes.

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::RoadLinePostProcess` class:

Table 310: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `RoadLinePostProcess` >	create	const std::vector< vitis::ai::library::InputTensor > & input_tensors const std::vector< vitis::ai::library::OutputTensor > & output_tensors const vitis::ai::proto::DpuModelParam & config
std::vector< `RoadLineResult` >	road_line_post_process	void

Functions

create

Create an RoadLinePostProcess object.

Prototype

```
std::unique_ptr<
                 RoadLinePostProcess
               > create(const std::vector< vitis::ai::library::InputTensor >
&input_tensors, const std::vector< vitis::ai::library::OutputTensor >
&output_tensors, const vitis::ai::proto::DpuModelParam &config);
```

Parameters

The following table lists the `create` function arguments.

Table 311: **create Arguments**

Type	Member	Description
const std::vector< vitis::ai::library::InputTensor > &	input_tensors	A vector of all input-tensors in the network. Usage: input_tensors[input_tensor_index].
const std::vector< vitis::ai::library::OutputTensor > &	output_tensors	A vector of all output-tensors in the network. Usage: output_tensors[output_index].
const vitis::ai::proto::DpuModelParam &	config	The DPU model configuration information.

Returns

A unique pointer of RoadLinePostProcess.

road_line_post_process

Run roadline post-process in batch mode.

Prototype

```
std::vector<
                 RoadLineResult
               > road_line_post_process(const std::vector< int > &inWidth,
const std::vector< int > &inHeight, size_t batch_size)=0;
```

Returns

The vector of struct of RoadLineResult.

vitis::ai::Segmentation

Base class for `Segmentation`.

Declaration `Segmentation` Network number of segmentation classes label 0 name: "unlabeled" label 1 name: "ego vehicle" label 2 name: "rectification border" label 3 name: "out of roi" label 4 name: "static" label 5 name: "dynamic" label 6 name: "ground" label 7 name: "road" label 8 name: "sidewalk" label 9 name: "parking" label 10 name: "rail track" label 11 name: "building" label 12 name: "wall" label 13 name: "fence" label 14 name: "guard rail" label 15 name: "bridge" label 16 name: "tunnel" label 17 name: "pole"

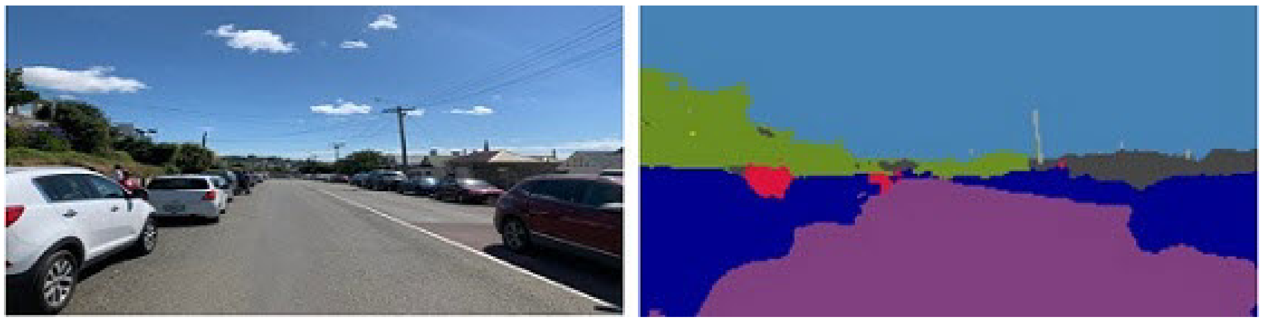
Input is an image (cv:Mat).

Output is result of running the `Segmentation` network.

Sample code :

```
   auto det =vitis::ai::Segmentation::create("fpn", true);

   auto img= cv::imread("sample_segmentation.jpg");
   int width = det->getInputWidth();
   int height = det->getInputHeight();
   cv::Mat image;
   cv::resize(img, image, cv::Size(width, height), 0, 0,
              cv::INTER_LINEAR);
   auto result = det->run_8UC1(image);
   for (auto y = 0; y < result.segmentation.rows; y++) {
     for (auto x = 0; x < result.segmentation.cols; x++) {
           result.segmentation.at<uchar>(y,x) *= 10;
       }
   }
   cv::imwrite("segres.jpg",result.segmentation);

   auto resultshow = det->run_8UC3(image);
   resize(resultshow.segmentation, resultshow.segmentation,
cv::Size(resultshow.cols * 2, resultshow.rows * 2));
   cv::imwrite("sample_segmentation_result.jpg",resultshow.segmentation);
```

Figure 73: **segmentation Visualization Result Image**



Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::Segmentation` class:

Table 312: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `Segmentation` >	create	const std::string & model_name bool need_preprocess
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void
`SegmentationResult`	run_8UC1	const cv::Mat & image
std::vector< `SegmentationResult` >	run_8UC1	const std::vector< cv::Mat > & images
`SegmentationResult`	run_8UC3	const cv::Mat & image
std::vector< `SegmentationResult` >	run_8UC3	const std::vector< cv::Mat > & images

Functions

create

Factory function to get an instance of derived classes of class `Segmentation` .

Prototype

```
std::unique_ptr<
                 Segmentation
               > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 313: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `Segmentation` class.

getInputWidth

Function to get InputWidth of the segmentation network (input image columns).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the segmentation network.

getInputHeight

Function to get InputHeight of the segmentation network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the segmentation network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

run_8UC1

Function to get running result of the segmentation network.

Note: The type of CV_8UC1 of the segmentation result.

Prototype

```
                SegmentationResult
              run_8UC1(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run_8UC1` function arguments.

Table 314: **run_8UC1 Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of input image (cv::Mat).

Returns

A result that includes segmentation output data.

run_8UC1

Function to get running results of the segmentation neural network in batch mode.

Note: The type of CV_8UC1 of the segmentation result.

Prototype

```
std::vector<
                SegmentationResult
              > run_8UC1(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run_8UC1` function arguments.

Table 315: **run_8UC1 Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of SegmentationResult .

run_8UC3

Function to get running result of the segmentation network.

Note: The type of CV_8UC3 of the segmentation result.

Prototype

```
                              SegmentationResult
                          run_8UC3(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run_8UC3` function arguments.

Table 316: **run_8UC3 Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of input image (cv::Mat).

Returns

A result that include segmentation image and shape;.

run_8UC3

Function to get running results of the segmentation neural network in batch mode.

Note: The type of CV_8UC3 of the segmentation result.

Prototype

```
std::vector<
                              SegmentationResult
                          > run_8UC3(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run_8UC3` function arguments.

Table 317: **run_8UC3 Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `SegmentationResult`.

vitis::ai::Segmentation3D

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::Segmentation3D` class:

Table 318: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< Segmentation3D >	create	bool need_mean_scale_process
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void
`Segmentation3DResult`	run	std::vector< std::vector< float > > & array
std::vector< `Segmentation3DResult` >	run	std::vector< std::vector< std::vector< float > > > & arrays

Functions

create

Factory function to get an instance of derived classes of class 3Dsegmentation.

Prototype

```
std::unique_ptr< Segmentation3D > create(const std::string &model_name,
bool need_mean_scale_process=false);
```

Parameters

The following table lists the `create` function arguments.

Table 319: **create Arguments**

Type	Member	Description
bool	need_mean_scale_proces s	Normalize with mean/scale or not, true by default.

Returns

An instance of the 3Dsegmentation class.

getInputWidth

Function to get InputWidth of the 3D segmentation network.

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the 3D segmentation network.

getInputHeight

Function to get InputHeight of the 3D segmentation network.

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the 3D segmentation network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

***Note*:** Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

run

Function of get running result of the 3D segmentation network.

Prototype

```
                          Segmentation3DResult
                       run(std::vector< std::vector< float > > &array)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 320: **run Arguments**

Type	Member	Description
std::vector< std::vector< float > > &	array	Input data of 3D object data in vector<float> mode.

Returns

`Segmentation3DResult`.

run

Function of get running result of the 3D segmentation network in batch mode.

Prototype

```
std::vector<
                          Segmentation3DResult
                       > run(std::vector< std::vector< std::vector< float > > >
&arrays)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 321: **run Arguments**

Type	Member	Description
std::vector< std::vector< std::vector< float > > > &	arrays	A vector of Input data of 3D object data in vector<float> mode.

Returns

A vector of `Segmentation3DResult` .

vitis::ai::Segmentation8UC1

The Class of `Segmentation8UC1`. This class run function `run(const cv::Mat& image)` return a cv::Mat with the type is cv_8UC1. Sample code :

```
   auto det =
vitis::ai::Segmentation8UC1::create(vitis::ai::SEGMENTATION_FPN);
  auto img = cv::imread("sample_segmentation.jpg");
   int width = det->getInputWidth();
   int height = det->getInputHeight();
   cv::Mat image;
   cv::resize(img, image, cv::Size(width, height), 0, 0,
              cv::INTER_LINEAR);
   auto result = det->run(image);
   for (auto y = 0; y < result.segmentation.rows; y++) {
     for (auto x = 0; x < result.segmentation.cols; x++) {
           result.segmentation.at<uchar>(y,x) *= 10;
       }
   }
   cv::imwrite("segres.jpg",result.segmentation);
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::Segmentation8UC1` class:

Table 322: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `Segmentation8UC1` >	create	const std::string & model_name bool need_preprocess
int	getInputWidth	void
int	getInputHeight	void

Table 322: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
size_t	get_input_batch	void
SegmentationResult	run	const cv::Mat & image
std::vector< SegmentationResult >	run	const std::vector< cv::Mat > & images

Functions

create

Factory function to get an instance of derived classes of class Segmentation8UC1 .

Prototype

```
std::unique_ptr<
                  Segmentation8UC1
                > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the create function arguments.

Table 323: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of Segmentation8UC1 class.

getInputWidth

Function to get InputWidth of the segmentation network (input image columns).

Prototype

```
int getInputWidth() const;
```

Returns

InputWidth of the segmentation network.

getInputHeight

Function to get InputHeight of the segmentation network (input image rows).

Prototype

```
int getInputHeight() const;
```

Returns

InputHeight of the segmentation network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const;
```

Returns

Batch size.

run

Function to get running result of the segmentation network.

Note: The result cv::Mat of the type is CV_8UC1.

Prototype

```
                    SegmentationResult
                  run(const cv::Mat &image);
```

Parameters

The following table lists the `run` function arguments.

Table 324: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of the image (cv::Mat)

Returns

A Struct of `SegmentationResult` ,the result of segmentation network.

run

Function to get running results of the segmentation neural network in batch mode.

Note: The type of CV_8UC1 of the Result's segmentation.

Prototype

```
std::vector<
               SegmentationResult
             > run(const std::vector< cv::Mat > &images);
```

Parameters

The following table lists the `run` function arguments.

Table 325: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `SegmentationResult` .

vitis::ai::Segmentation8UC3

The Class of `Segmentation8UC3`, this class run function `run(const cv::Mat& image)` return a cv::Mat with the type is cv_8UC3 Sample code :

```
   auto det =
vitis::ai::Segmentation8UC3::create(vitis::ai::SEGMENTATION_FPN);
  auto img = cv::imread("sample_segmentation.jpg");

   int width = det->getInputWidth();
   int height = det->getInputHeight();
```

```
    cv::Mat image;
    cv::resize(img, image, cv::Size(width, height), 0, 0,
               cv::INTER_LINEAR);
    auto result = det->run(image);
    cv::imwrite("segres.jpg",result.segmentation);
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::Segmentation8UC3` class:

Table 326: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `Segmentation8UC3` >	create	const std::string & model_name bool need_preprocess
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void
`SegmentationResult`	run	const cv::Mat & image
std::vector< `SegmentationResult` >	run	const std::vector< cv::Mat > & images

Functions

create

Factory function to get an instance of derived classes of class `Segmentation8UC3` .

Prototype

```
std::unique_ptr<
                  Segmentation8UC3
                > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 327: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `Segmentation8UC3` class.

getInputWidth

Function to get InputWidth of the segmentation network (input image columns).

Prototype

```
int getInputWidth() const;
```

Returns

InputWidth of the segmentation network.

getInputHeight

Function to get InputWidth of the segmentation network (input image rows).

Prototype

```
int getInputHeight() const;
```

Returns

InputWidth of the segmentation network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const;
```

Returns

Batch size.

run

Function to get running result of the segmentation network.

Note: The result cv::Mat of the type is CV_8UC3.

Prototype

```
                SegmentationResult
              run(const cv::Mat &image);
```

Parameters

The following table lists the `run` function arguments.

Table 328: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of the image (cv::Mat)

Returns

`SegmentationResult` , the result of the segmentation network.

run

Function to get running results of the segmentation neural network in batch mode.

Note: The type of CV_8UC3 of the segmentation result.

Prototype

```
std::vector<
                SegmentationResult
              > run(const std::vector< cv::Mat > &images);
```

Parameters

The following table lists the `run` function arguments.

Table 329: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `SegmentationResult` .

vitis::ai::Solo

Base class for SOLO semantic segmentation from an image (cv::Mat).

Input is an image (cv::Mat).

Output is the enlarged image.

Sample code:

***Note*:** The input image size is 640x360

```
if (argc < 2) {
  cerr << "usage: " << argv[0] << "  modelname  image_file_url " << endl;
  abort();
}
Mat input_img = imread(argv[2]);
if (input_img.empty()) {
  cerr << "can't load image! " << argv[2] << endl;
  return -1;
}
auto det = vitis::ai::Solo::create(argv[1]);
Mat ret_img = det->run(input_img).feat;
imwrite("sample_solo_result.png", ret_img);
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::Solo` class:

Table 330: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `Solo` >	create	const std::string & model_name bool need_preprocess
`SoloResult`	run	const cv::Mat & image
std::vector< `SoloResult` >	run	const std::vector< cv::Mat > & images

Functions

create

Factory function to get an instance of derived classes of class `Solo` .

Prototype

```
std::unique_ptr<
               Solo
            > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 331: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `Solo` class.

run

Function to get running result of the SOLO neural network.

Prototype

```


               SoloResult
             run(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 332: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of input image (cv::Mat).

Returns

SoloResult .

run

Function to get running result of the SOLO neural network in batch mode.

Prototype

```
std::vector<
                SoloResult
              > run(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 333: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (vector<cv::Mat>).

Returns

vector of SoloResult .

vitis::ai::SSD

Base class for detecting position of vehicle, pedestrian, and so on.

Input is an image (cv:Mat).

Output is a struct of detection results, named SSDResult.

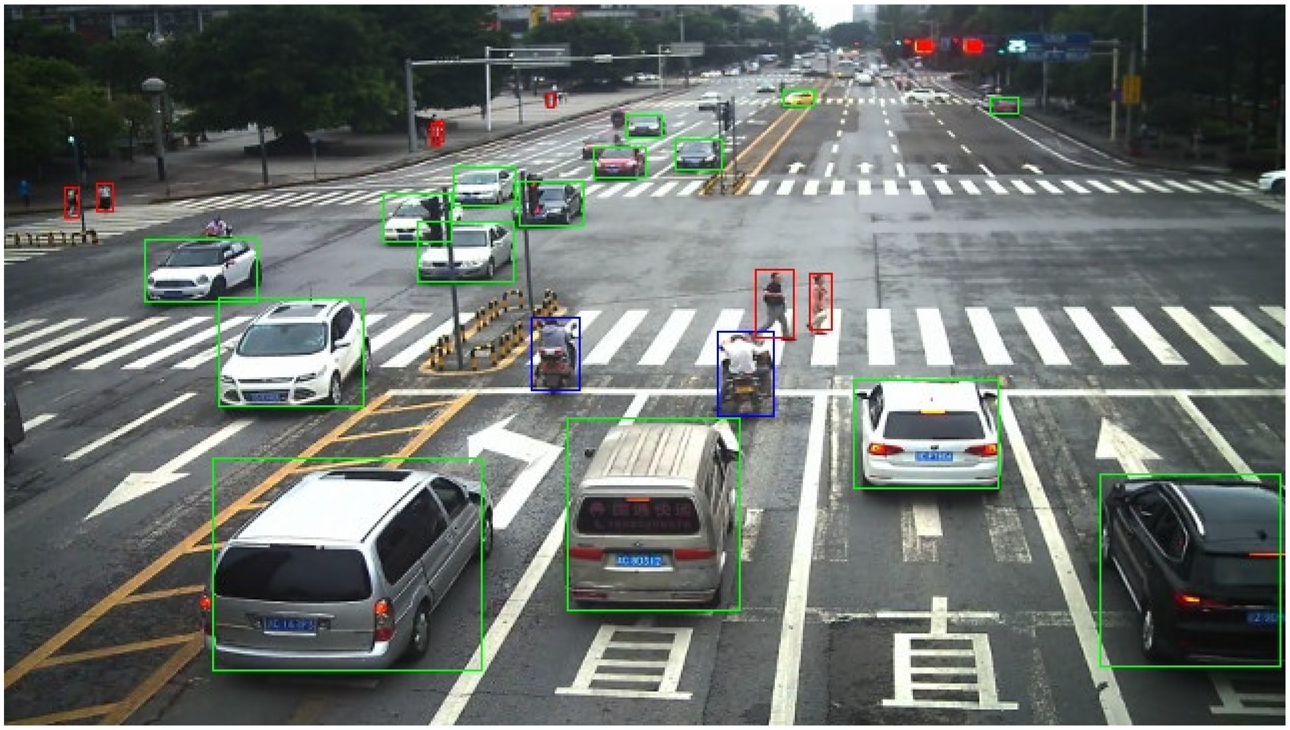
Sample code :

```
  Mat img = cv::imread("sample_ssd.jpg");
  auto ssd = vitis::ai::SSD::create("ssd_traffic_pruned_0_9",true);
  auto results = ssd->run(img);
  for(const auto &r : results.bboxes){
     auto label = r.label;
     auto x = r.x * img.cols;
     auto y = r.y * img.rows;
     auto width = r.width * img.cols;
     auto heigth = r.height * img.rows;
```

```
      auto score = r.score;
      std::cout << "RESULT: " << label << "\t" << x << "\t" << y << "\t" <<
width
         << "\t" << height << "\t" << score << std::endl;
   }
```

Display of the model results:

Figure 74: **detection result**



Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::SSD` class:

Table 334: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `SSD` >	create	const std::string & model_name bool need_preprocess
std::unique_ptr< `SSD` >	create	const std::string & model_name xir::Attrs * attrs bool need_preprocess
`vitis::ai::SSDResult`	run	const cv::Mat & image

Table 334: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
std::vector< vitis::ai::SSDResult >	run	const std::vector< cv::Mat > & images
std::vector< vitis::ai::SSDResult >	run	const std::vector< vart::xrt_bo_t > & input_bos

Functions

create

Factory function to get an instance of derived classes of class SSD .

Prototype

```
std::unique_ptr<
                SSD
              > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 335: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of SSD class.

create

Factory function to get an instance of derived classes of class SSD .

Prototype

```
std::unique_ptr<
                SSD
              > create(const std::string &model_name, xir::Attrs *attrs,
bool need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 336: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
xir::Attrs *	attrs	Xir attributes
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `SSD` class.

run

Function to get running results of the `SSD` neural network.

Prototype

```
                    vitis::ai::SSDResult
                 run(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 337: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of input image (cv::Mat).

Returns

`SSDResult`.

run

Function to get running results of the `SSD` neural network in batch mode.

Prototype

```
std::vector<
                    vitis::ai::SSDResult
                 > run(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 338: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (vector<cv::Mat>).The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `SSDResult`.

run

Function to get running results of the `SSD` neural network in batch mode, used to receive user's xrt_bo to support zero copy.

Prototype

```
std::vector<
                vitis::ai::SSDResult
              > run(const std::vector< vart::xrt_bo_t > &input_bos)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 339: **run Arguments**

Type	Member	Description
const std::vector< vart::xrt_bo_t > &	input_bos	The vector of vart::xrt_bo_t.

Returns

The vector of `SSDResult`.

vitis::ai::SSDPostProcess

Class of the `SSD` post-process. It initializes the parameters once instead of computing them each time the program executes.

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::SSDPostProcess` class:

Table 340: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `SSDPostProcess` >	create	const std::vector< vitis::ai::library::InputTensor > & input_tensors const std::vector< vitis::ai::library::OutputTensor > & output_tensors const vitis::ai::proto::DpuModelParam & config
std::vector< `SSDResult` >	ssd_post_process	void

Functions

create

Create an `SSDPostProcess` object.

Prototype

```
std::unique_ptr<
                SSDPostProcess
             > create(const std::vector< vitis::ai::library::InputTensor >
&input_tensors, const std::vector< vitis::ai::library::OutputTensor >
&output_tensors, const vitis::ai::proto::DpuModelParam &config);
```

Parameters

The following table lists the `create` function arguments.

Table 341: **create Arguments**

Type	Member	Description
const std::vector< vitis::ai::library::InputTensor > &	input_tensors	A vector of all input-tensors in the network. Usage: input_tensors[input_tensor_index].
const std::vector< vitis::ai::library::OutputTensor > &	output_tensors	A vector of all output-tensors in the network. Usage: output_tensors[output_index].
const vitis::ai::proto::DpuModelParam &	config	The DPU model configuration information.

Returns

An unique pointer of SSDPostProcess .

ssd_post_process

The batch mode post-processing function of the SSD network.

Prototype

```
std::vector<
                SSDResult
              > ssd_post_process(size_t batch_size)=0;
```

Returns

The vector of struct of SSDResult .

vitis::ai::TextMountain

Base class for TextMountain (text detection)

Input is an image (cv:Mat).

Output is a struct of classification results, named TextMountainResult.

Sample code :

```
Mat img = cv::imread("sample_TextMountain.jpg");
auto TextMountain = vitis::ai::TextMountain::create("textmountain_pt",true);
auto result = TextMountain->run(img);
// result is structure holding the text information
std::cout << result.res.size() <<"\n";
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::TextMountain` class:

Table 342: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< TextMountain >	create	const std::string & model_name bool need_preprocess
vitis::ai::TextMountainResult	run	const cv::Mat & img

Table 342: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
std::vector< vitis::ai::TextMountainResult >	run	const std::vector< cv::Mat > & imgs
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void

Functions

create

Factory function to get an instance of derived classes of class `TextMountain` .

Prototype

```
std::unique_ptr<
                 TextMountain
               > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 343: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `TextMountain` class.

run

Function of get result of the `TextMountain` network.

Prototype

```
vitis::ai::TextMountainResult run(const cv::Mat &img)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 344: **run Arguments**

Type	Member	Description
const cv::Mat &	img	Input data of input image (cv::Mat).

Returns

TextMountainResult.

run

Function to get running results of the `TextMountain` network in batch mode.

Prototype

```
std::vector< vitis::ai::TextMountainResult > run(const std::vector< cv::Mat
> &imgs)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 345: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	imgs	Input data of input images (vector<cv::Mat>). The size of input images need equal to or less than batch size obtained by get_input_batch.

Returns

The vector of TextMountainResult.

getInputWidth

Function to get InputWidth of the `TextMountain` network (input image columns).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the `TextMountain` network.

getInputHeight

Function to get InputHeight of the `TextMountain` network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the `TextMountain` network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

***Note*:** Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

vitis::ai::TextMountainPost

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::TextMountainPost` class:

Table 346: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< TextMountainPost >	create	const std::vector< vitis::ai::library::InputTensor > & input_tensors const std::vector< vitis::ai::library::OutputTensor > & output_tensors int batch_size int & real_batch_size float * scale_h float * scale_w
TextMountainResult	process	int idx
std::vector< TextMountainResult >	process	void

Functions

create

Create an TextMountainPost object.

Prototype

```
std::unique_ptr< TextMountainPost > create(const std::vector<
vitis::ai::library::InputTensor > &input_tensors, const std::vector<
vitis::ai::library::OutputTensor > &output_tensors, int batch_size, int
&real_batch_size, float *scale_h, float *scale_w);
```

Parameters

The following table lists the `create` function arguments.

Table 347: **create Arguments**

Type	Member	Description
const std::vector< vitis::ai::library::InputTensor > &	input_tensors	A vector of all input-tensors in the network. Usage: input_tensors[input_tensor_index].
const std::vector< vitis::ai::library::OutputTensor > &	output_tensors	A vector of all output-tensors in the network. Usage: output_tensors[output_index].
int	batch_size	the model batch information
int &	real_batch_size	the real batch information of the model
float *	scale_h	the array to hold the height scale for each input img
float *	scale_w	the array to hold the width scale for each input img

Returns

An unique pointer of TextMountainPost

process

Post-process the textmountain result.

Prototype

```
TextMountainResult process(int idx)=0;
```

Parameters

The following table lists the `process` function arguments.

Table 348: **process Arguments**

Type	Member	Description
int	idx	batch index.

Returns

TextMountainResult.

process

Post-process the textmountain result.

Prototype

```
std::vector< TextMountainResult > process()=0;
```

Returns

vector of TextMountainResult.

vitis::ai::TFRefineDetPostProcess

Class of the tfrefinedet post-process. It initializes the parameters once instead of computing them each time the program executes.

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::TFRefineDetPostProcess` class:

Table 349: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `TFRefineDetPostProcess` >	create	const std::vector< vitis::ai::library::InputTensor > & input_tensors const std::vector< vitis::ai::library::OutputTensor > & output_tensors const vitis::ai::proto::DpuModelParam & config
std::vector< `RefineDetResult` >	tfrefinedet_post_process	void

Functions

create

Create an `TFRefineDetPostProcess` object.

Prototype

```
std::unique_ptr<
                TFRefineDetPostProcess
              > create(const std::vector< vitis::ai::library::InputTensor >
&input_tensors, const std::vector< vitis::ai::library::OutputTensor >
&output_tensors, const vitis::ai::proto::DpuModelParam &config);
```

Parameters

The following table lists the `create` function arguments.

Table 350: **create Arguments**

Type	Member	Description
const std::vector< vitis::ai::library::InputTensor > &	input_tensors	A vector of all input-tensors in the network. Usage: input_tensors[input_tensor_index].
const std::vector< vitis::ai::library::OutputTensor > &	output_tensors	A vector of all output-tensors in the network. Usage: output_tensors[output_index].
const vitis::ai::proto::DpuModelParam &	config	The DPU model configuration information.

Returns

A unique pointer of `TFRefineDetPostProcess` .

tfrefinedet_post_process

Run batch mode of tfrefinedet post-process.

Prototype

```
std::vector<
               RefineDetResult
             > tfrefinedet_post_process(size_t batch_size)=0;
```

Returns

The vector of struct of `RefineDetResult` .

vitis::ai::TFSSD

Base class for detecting 90 objects of the COCO dataset.

Input is an image (cv:Mat).

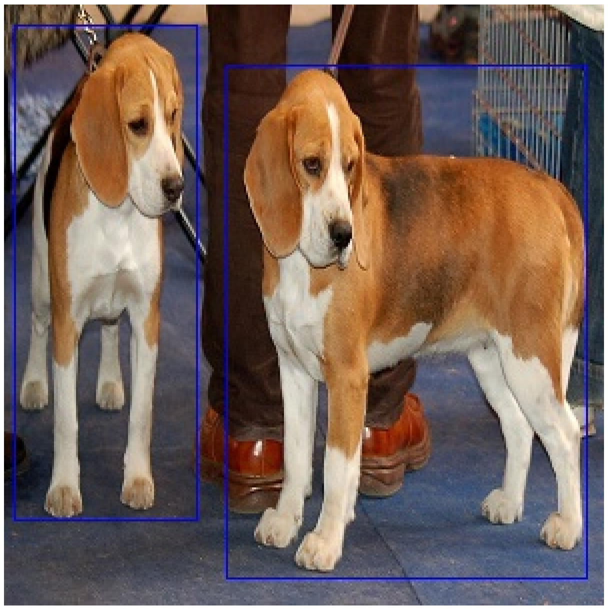
Output is a struct of detection results, named `TFSSDResult`.

Sample code :

```
  Mat img = cv::imread("sample_tfssd.jpg");
  auto tfssd = vitis::ai::TFSSD::create("ssd_resnet_50_fpn_coco_tf",true);
  auto results = tfssd->run(img);
  for(const auto &r : results.bboxes){
     auto label = r.label;
     auto x = r.x * img.cols;
     auto y = r.y * img.rows;
     auto width = r.width * img.cols;
     auto height = r.height * img.rows;
     auto score = r.score;
     std::cout << "RESULT: " << label << "\t" << x << "\t" << y << "\t" <<
width
        << "\t" << height << "\t" << score << std::endl;
  }
```

Display of the model results:

Figure 75: **detection result**



Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::TFSSD` class:

Table 351: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `TFSSD` >	create	const std::string & model_name bool need_preprocess
`vitis::ai::TFSSDResult`	run	const cv::Mat & img
std::vector< `vitis::ai::TFSSDResult` >	run	const std::vector< cv::Mat > & imgs
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void

Functions

create

Factory function to get an instance of derived classes of class `SSD` .

Prototype

```
std::unique_ptr<
                TFSSD
              > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 352: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `TFSSD` class.

run

Function of get result of the ssd neural network.

Prototype

```

                vitis::ai::TFSSDResult
              run(const cv::Mat &img)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 353: **run Arguments**

Type	Member	Description
const cv::Mat &	img	Input data of input image (cv::Mat).

Returns

TFSSDResult.

run

Function to get running results of the SSD neural network in batch mode.

Prototype

```
std::vector<
                vitis::ai::TFSSDResult
              > run(const std::vector< cv::Mat > &imgs)=0;
```

Parameters

The following table lists the run function arguments.

Table 354: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	imgs	Input data of input images (vector<cv::Mat>).The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of TFSSDResult.

getInputWidth

Function to get InputWidth of the SSD network (input image columns).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the TFSSD network.

getInputHeight

Function to get InputHeight of the SSD network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the `TFSSD` network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

***Note*:** Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

vitis::ai::TFSSDPostProcess

Class of the `TFSSD` post-process. It initializes the parameters once instead of computing them each time the program executes.

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::TFSSDPostProcess` class:

Table 355: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `TFSSDPostProcess` >	create	const std::vector< vitis::ai::library::InputTensor > & input_tensors const std::vector< vitis::ai::library::OutputTensor > & output_tensors const vitis::ai::proto::DpuModelParam & config
`TFSSDResult`	ssd_post_process	void
std::vector< `TFSSDResult` >	ssd_post_process	void

Functions

create

Create an TFSSDPostProcess object.

Prototype

```
std::unique_ptr<
                TFSSDPostProcess
             > create(const std::string &model_name, const std::vector<
vitis::ai::library::InputTensor > &input_tensors, const std::vector<
vitis::ai::library::OutputTensor > &output_tensors, const
vitis::ai::proto::DpuModelParam &config, const std::string &dirname, int
&real_batch_size);
```

Parameters

The following table lists the `create` function arguments.

Table 356: **create Arguments**

Type	Member	Description
const std::vector< vitis::ai::library::InputTensor > &	input_tensors	A vector of all input-tensors in the network. Usage: input_tensors[input_tensor_index].
const std::vector< vitis::ai::library::OutputTensor > &	output_tensors	A vector of all output-tensors in the network. Usage: output_tensors[output_index].
const vitis::ai::proto::DpuModelParam &	config	The DPU model configuration information.

Returns

A unique pointer of TFSSDPostProcess .

ssd_post_process

The post-processing function of the TFSSD network.

Prototype

```
                TFSSDResult
             ssd_post_process(unsigned int idx)=0;
```

Returns

Struct of TFSSDResult .

ssd_post_process

The batch mode post-processing function of the TFSSD network.

Prototype

```
std::vector<
               TFSSDResult
             > ssd_post_process()=0;
```

Returns

The vector of struct of TFSSDResult .

vitis::ai::UltraFast

Base class for detecting traffic lane for CULane dataset.

Input is an image (cv:Mat).

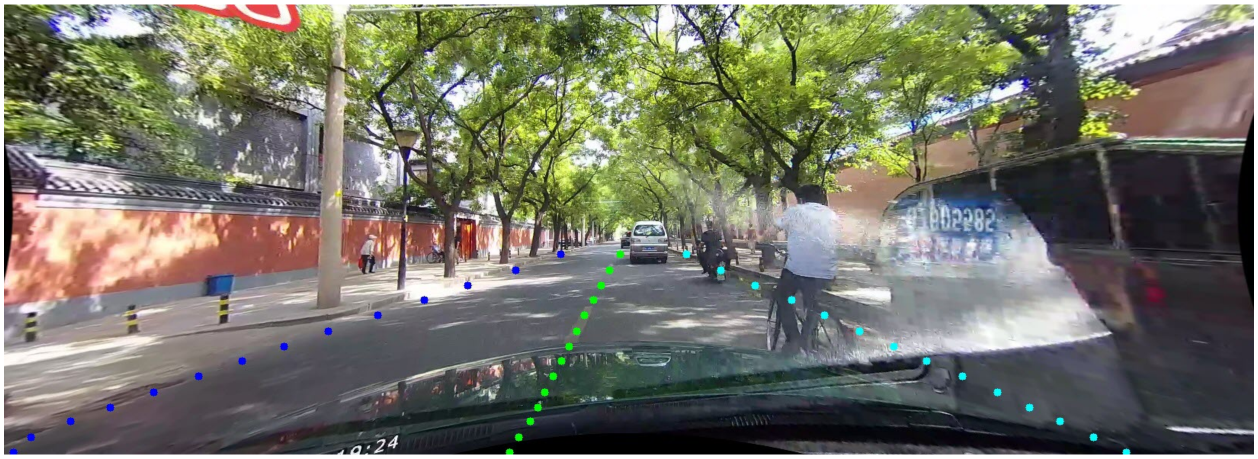
Output is a struct of detection results, named UltraFastResult.

Sample code :

```
Mat img = cv::imread("sample_ultrafast.jpg");
auto ultrafast = vitis::ai::UltraFast::create("ultrafast_pt",true);
auto results = ultrafast->run(img);
for(const auto &lanes : results.lanes){
  std::cout <<"lane:\n";
  for(auto &v: lanes) {
     std::cout << "    ( " << v.first << ", " << v.second << " )\n";
  }
}
```

Display of the model results:

Figure 76: **detection result**



Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::UltraFast` class:

Table 357: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `UltraFast` >	create	const std::string & model_name bool need_preprocess
`vitis::ai::UltraFastResult`	run	const cv::Mat & img
std::vector< `vitis::ai::UltraFastResult` >	run	const std::vector< cv::Mat > & imgs
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void

Functions

create

Factory function to get an instance of derived classes of class `UltraFast` .

Prototype

```
std::unique_ptr<
                UltraFast
              > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 358: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `UltraFast` class.

run

Function of get result of the `UltraFast` neural network.

Prototype

```


                vitis::ai::UltraFastResult
              run(const cv::Mat &img)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 359: **run Arguments**

Type	Member	Description
const cv::Mat &	img	Input data of input image (cv::Mat).

Returns

`UltraFastResult`.

run

Function to get running results of the `UltraFast` neural network in batch mode.

Prototype

```
std::vector<
               vitis::ai::UltraFastResult
             > run(const std::vector< cv::Mat > &imgs)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 360: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	imgs	Input data of input images (vector<cv::Mat>).The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `UltraFastResult` .

getInputWidth

Function to get InputWidth of the `UltraFast` network (input image cols).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the `UltraFast` network.

getInputHeight

Function to get InputHeight of the `UltraFast` network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the `UltraFast` network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

***Note*:** Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

vitis::ai::UltraFastPost

Class of the `UltraFast` post-process.

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::UltraFastPost` class:

Table 361: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `UltraFastPost` >	create	const std::vector< vitis::ai::library::InputTensor > & input_tensors const std::vector< vitis::ai::library::OutputTensor > & output_tensors const vitis::ai::proto::DpuModelParam & config int batch_size int & real_batch_size std::vector< cv::Size > & pic_size
`UltraFastResult`	post_process	unsigned int idx
std::vector< `UltraFastResult` >	post_process	void

Functions

create

Create an `UltraFastPost` object.

Prototype

```
std::unique_ptr<
               UltraFastPost
             > create(const std::vector< vitis::ai::library::InputTensor >
&input_tensors, const std::vector< vitis::ai::library::OutputTensor >
&output_tensors, const vitis::ai::proto::DpuModelParam &config, int
batch_size, int &real_batch_size, std::vector< cv::Size > &pic_size);
```

Parameters

The following table lists the `create` function arguments.

Table 362: **create Arguments**

Type	Member	Description
const std::vector< vitis::ai::library::InputTensor > &	input_tensors	A vector of all input-tensors in the network. Usage: input_tensors[input_tensor_index].
const std::vector< vitis::ai::library::OutputTensor > &	output_tensors	A vector of all output-tensors in the network. Usage: output_tensors[output_index].
const vitis::ai::proto::DpuModelParam &	config	The DPU model configuration information.
int	batch_size	the model batch information
int &	real_batch_size	the real batch information of the model
std::vector< cv::Size > &	pic_size	vector holding the size information of input pics

Returns

A unique pointer of `UltraFastPost`.

post_process

Post-process the `UltraFast` result.

Prototype

```


               UltraFastResult
             post_process(unsigned int idx)=0;
```

Parameters

The following table lists the `post_process` function arguments.

Table 363: **post_process Arguments**

Type	Member	Description
unsigned int	idx	batch index.

Returns

UltraFastResult .

post_process

Post-process the UltraFast result.

Prototype

```
std::vector<
               UltraFastResult
             > post_process()=0;
```

Returns

vector of UltraFastResult .

vitis::ai::Unet2D

Base class for Unet2D.

Input is an 4 channel binary data: NxNx4

Output is a struct of segmentation results, named Unet2DResult.

Sample code:

```
std::vector<float> vf = get_binary_data();
auto Unet2D = vitis::ai::Unet2D::create("unet2d_tf");
auto result = Unet2D->run(vf.data(), vf.size());
std::cout << result.data.size() << "\n";
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::Unet2D` class:

Table 364: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< Unet2D >	create	const std::string & model_name bool need_preprocess
vitis::ai::Unet2DResult	run	float * img int len
vitis::ai::Unet2DResult	run	const std::vector< float > & img
std::vector< vitis::ai::Unet2DResult >	run	const std::vector< float * > & imgs int len
std::vector< vitis::ai::Unet2DResult >	run	const std::vector< std::vector< float > > & imgs
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void

Functions

create

Factory function to get an instance of derived classes of class `Unet2D` .

Prototype

```
std::unique_ptr<
                  Unet2D
                > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 365: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name

Table 365: **create Arguments** *(cont'd)*

Type	Member	Description
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `Unet2D` class.

run

Function of get result of the `Unet2D` neural network.

Prototype

```
vitis::ai::Unet2DResult run(float *img, int len)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 366: **run Arguments**

Type	Member	Description
float *	img	pointer to Input data (binary data of 4 channels).
int	len	length of Input data.

Returns

Unet2DResult.

run

Function of get result of the `Unet2D` neural network.

Prototype

```
vitis::ai::Unet2DResult run(const std::vector< float > &img)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 367: **run Arguments**

Type	Member	Description
const std::vector< float > &	img	vector holding the Input data (binary data of 4 channels).

Returns

Unet2DResult.

run

Function to get running results of the `Unet2D` neural network in batch mode.

Prototype

```
std::vector< vitis::ai::Unet2DResult > run(const std::vector< float * >
&imgs, int len)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 368: **run Arguments**

Type	Member	Description
const std::vector< float * > &	imgs	vector of Input data of input images (vector<float*>). The size of input images need equal to or less than batch size obtained by get_input_batch. If it is greater than batch, the extra part is ignored.
int	len	length of Input data: all input data should be same size.

Returns

The vector of Unet2DResult.

run

Function to get running results of the `Unet2D` neural network in batch mode.

Prototype

```
std::vector< vitis::ai::Unet2DResult > run(const std::vector< std::vector<
float > > &imgs)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 369: **run Arguments**

Type	Member	Description
const std::vector< std::vector< float > > &	imgs	Input data of input images (vector<vector<float>>). The size of input images need equal to or less than batch size obtained by get_input_batch. If it is greater than batch, the extra part is ignored.

Returns

The vector of Unet2DResult.

getInputWidth

Function to get InputWidth of the `Unet2D` network (input image columns).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the `Unet2D` network.

getInputHeight

Function to get InputHeight of the `Unet2D` network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the `Unet2D` network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

vitis::ai::VehicleClassification

Base class for detecting objects in the input image (cv::Mat).

Input is an image (cv::Mat).

Output is index and score of objects in the input image.

Sample code:

```
auto image = cv::imread("sample_vehicleclassification.jpg");
auto network = vitis::ai::VehicleClassification::create(
                  "vehicle_type_resnet18_pt");
auto result = network->run(image);
for (const auto &r : result.scores) {
   auto score = r.score;
   auto index = result.lookup(r.index);
}
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::VehicleClassification` class:

Table 370: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `VehicleClassification` >	create	const std::string & model_name bool need_preprocess
`vitis::ai::VehicleClassificationResult`	run	const cv::Mat & image
std::vector< `vitis::ai::VehicleClassificationResult` >	run	const std::vector< cv::Mat > & images

Functions

create

Factory function to get an instance of derived classes of class `VehicleClassification`.

Prototype

```
std::unique_ptr<
              VehicleClassification
            > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 371: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name.
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `VehicleClassification` class.

run

Function to get running results of the vehicleclassification neural network.

Prototype

```


              vitis::ai::VehicleClassificationResult
            run(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 372: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of input image (cv::Mat).

Returns

`VehicleClassificationResult`.

run

Function to get running results of the vehicleclassification neural network in batch mode.

Prototype

```
std::vector<
                vitis::ai::VehicleClassificationResult
              > run(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 373: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of batch input images (vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `VehicleClassificationResult` .

vitis::ai::YOLOv2

Base class for detecting objects in the input image(cv::Mat). Input is an image(cv::Mat). Output is the position of the objects in the input image. Sample code:

```
 auto img = cv::imread("sample_yolov2.jpg");
 auto model = vitis::ai::YOLOv2::create("yolov2_voc");
 auto result = model->run(img);
 for (const auto &bbox : result.bboxes) {
   int label = bbox.label;
   float xmin = bbox.x * img.cols + 1;
   float ymin = bbox.y * img.rows + 1;
   float xmax = xmin + bbox.width * img.cols;
   float ymax = ymin + bbox.height * img.rows;
   if (xmax > img.cols)
     xmax = img.cols;
   if (ymax > img.rows)
     ymax = img.rows;
   float confidence = bbox.score;

   cout << "RESULT: " << label << "\t" << xmin << "\t" << ymin << "\t" <<
xmax
       << "\t" << ymax << "\t" << confidence << "\n";
   rectangle(img, Point(xmin, ymin), Point(xmax, ymax), Scalar(0, 255, 0),
1,
            1, 0);
}
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::YOLOv2` class:

Table 374: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< YOLOv2 >	create	const std::string & model_name bool need_preprocess
YOLOv2Result	run	const cv::Mat & image
std::vector< YOLOv2Result >	run	const std::vector< cv::Mat > & images
int	getInputWidth	void
int	getInputHeight	void
size_t	get_input_batch	void

Functions

create

Factory function to get an instance of derived classes of class `YOLOv2` .

Prototype

```
std::unique_ptr<
                  YOLOv2
                > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 375: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `YOLOv2` class.

run

Function to get running result of the YOLOv2 neural network.

Prototype

```
                YOLOv2Result
              run(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 376: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of input image (cv::Mat).

Returns

A Struct of YOLOv2Result .

run

Function to get running result of the YOLOv2 neural network in batch mode.

Prototype

```
std::vector<
                YOLOv2Result
              > run(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 377: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of YOLOv2Result .

getInputWidth

Function to get InputWidth of the `YOLOv2` network (input image columns).

Prototype

```
int getInputWidth() const =0;
```

Returns

InputWidth of the `YOLOv2` network

getInputHeight

Function to get InputHeight of the `YOLOv2` network (input image rows).

Prototype

```
int getInputHeight() const =0;
```

Returns

InputHeight of the `YOLOv2` network.

get_input_batch

Function to get the number of images processed by the DPU at one time.

Note: Different DPU core the batch size may be different. This depends on the IP used.

Prototype

```
size_t get_input_batch() const =0;
```

Returns

Batch size.

vitis::ai::YOLOv3

Base class for detecting objects in the input image (cv::Mat).

Input is an image (cv::Mat).

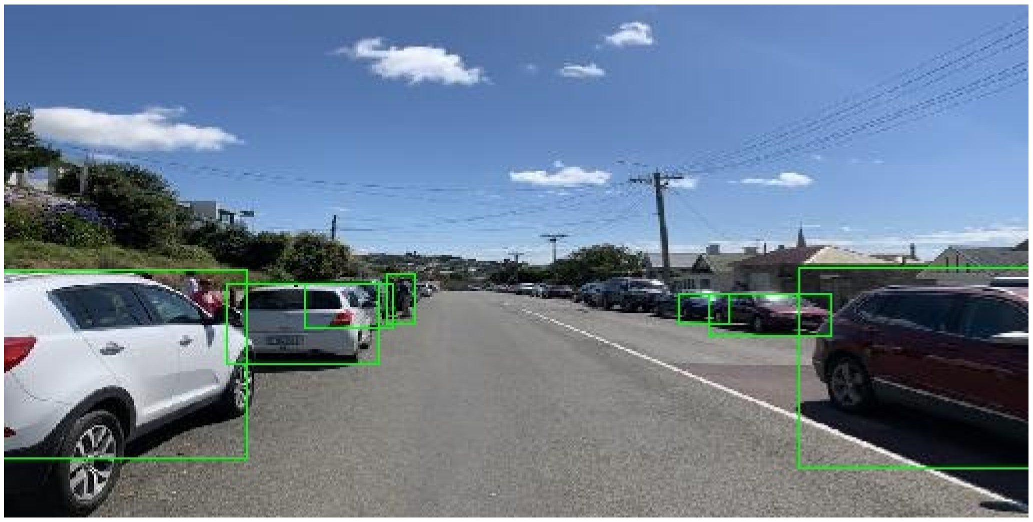
Output is the position of the pedestrians in the input image.

Sample code:

```
   auto yolo =
vitis::ai::YOLOv3::create("yolov3_adas_pruned_0_9", true);
   Mat img = cv::imread("sample_yolov3.jpg");

   auto results = yolo->run(img);

   for(auto &box : results.bboxes){
     int label = box.label;
     float xmin = box.x * img.cols + 1;
     float ymin = box.y * img.rows + 1;
     float xmax = xmin + box.width * img.cols;
     float ymax = ymin + box.height * img.rows;
     if(xmin < 0.) xmin = 1.;
     if(ymin < 0.) ymin = 1.;
     if(xmax > img.cols) xmax = img.cols;
     if(ymax > img.rows) ymax = img.rows;
     float confidence = box.score;

     cout << "RESULT: " << label << "\t" << xmin << "\t" << ymin << "\t"
          << xmax << "\t" << ymax << "\t" << confidence << "\n";
     if (label == 0) {
       rectangle(img, Point(xmin, ymin), Point(xmax, ymax), Scalar(0, 255,
0),
                 1, 1, 0);
     } else if (label == 1) {
       rectangle(img, Point(xmin, ymin), Point(xmax, ymax), Scalar(255, 0,
0),
                 1, 1, 0);
     } else if (label == 2) {
       rectangle(img, Point(xmin, ymin), Point(xmax, ymax), Scalar(0, 0,
255),
                 1, 1, 0);
     } else if (label == 3) {
       rectangle(img, Point(xmin, ymin), Point(xmax, ymax),
                 Scalar(0, 255, 255), 1, 1, 0);
     }

   }
   imwrite("sample_yolov3_result.jpg", img);
```

Display of the model results:

Figure 77: **out image**



Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::YOLOv3` class:

Table 378: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `YOLOv3` >	create	const std::string & model_name bool need_preprocess
`YOLOv3Result`	run	const cv::Mat & image
std::vector< `YOLOv3Result` >	run	const std::vector< cv::Mat > & images
std::vector< `YOLOv3Result` >	run	const std::vector< vart::xrt_bo_t > & input_bos

Functions

create

Factory function to get an instance of derived classes of class `YOLOv3` .

Prototype

```
std::unique_ptr<
                YOLOv3
              > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 379: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `YOLOv3` class.

run

Function to get running result of the `YOLOv3` neural network.

Prototype

```


                YOLOv3Result
              run(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 380: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of input image (cv::Mat).

Returns

`YOLOv3Result` .

run

Function to get running result of the `YOLOv3` neural network in batch mode.

Prototype

```
std::vector<
            YOLOv3Result
          > run(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 381: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `YOLOv3Result` .

run

Function to get running result of the `YOLOv3` neural network in batch mode, used to receive user's xrt_bo to support zero copy.

Prototype

```
std::vector<
            YOLOv3Result
          > run(const std::vector< vart::xrt_bo_t > &input_bos)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 382: **run Arguments**

Type	Member	Description
const std::vector< vart::xrt_bo_t > &	input_bos	The vector of vart::xrt_bo_t.

Returns

The vector of `YOLOv3Result` .

vitis::ai::YOLOv7

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::YOLOv7` class:

Table 383: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< YOLOv7 >	create	const std::string & model_name bool need_preprocess
`YOLOv7Result`	run	const cv::Mat & image
std::vector< `YOLOv7Result` >	run	const std::vector< cv::Mat > & images

Functions

create

Factory function to get an instance of derived classes of class YOLOv7.

Prototype

```
std::unique_ptr< YOLOv7 > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 384: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of YOLOv7 class.

run

Function to get running result of the YOLOv7 neural network.

Prototype

```
          YOLOv7Result
        run(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 385: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of input image (cv::Mat).

Returns

`YOLOv7Result`.

run

Function to get running result of the YOLOv7 neural network in batch mode.

Prototype

```
std::vector<
          YOLOv7Result
        > run(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 386: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `YOLOv7Result`.

vitis::ai::YOLOv8

Base class for detecting objects in the input image (cv::Mat).

Input is an image (cv::Mat).

Output is the position of the pedestrians in the input image.

Sample code:

```
static cv::Scalar getColor(int label) {
  return cv::Scalar(label * 2, 255 - label * 2, label + 50);
}

auto yolo = vitis::ai::YOLOv8::create("yolov8", true);

Mat img = cv::imread("sample_yolov8.jpg");
auto results = yolov8->run(img);
for (const auto& result : results.bboxes) {
  int label = result.label;
  auto& box = result.box;
  LOG_IF(INFO, is_jpeg) << "RESULT: " << label << "\t" << std::fixed
                        << std::setprecision(2) << box[0] << "\t" << box[1]
                        << "\t" << box[2] << "\t" << box[3] << "\t"
                        << std::setprecision(6) << result.score << "\n";

  cv::rectangle(image, cv::Point(box[0], box[1]), cv::Point(box[2], box[3]),
                getColor(label), 1, 1, 0);
}
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::YOLOv8` class:

Table 387: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `YOLOv8` >	create	const std::string & model_name bool need_preprocess
std::unique_ptr< `YOLOv8` >	create	const std::string & model_name xir::Attrs * attrs bool need_preprocess
`YOLOv8Result`	run	const cv::Mat & image
std::vector< `YOLOv8Result` >	run	const std::vector< cv::Mat > & images

Functions

create

Factory function to get an instance of derived classes of class YOLOv8 .

Prototype

```
std::unique_ptr<
                  YOLOv8
                > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 388: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of YOLOv8 class.

create

Factory function to get an instance of derived classes of class YOLOv8 .

Prototype

```
std::unique_ptr<
                  YOLOv8
                > create(const std::string &model_name, xir::Attrs *attrs,
bool need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 389: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
xir::Attrs *	attrs	XIR attributes, used to bind different models to the same dpu core

Table 389: **create Arguments** *(cont'd)*

Type	Member	Description
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `YOLOv8` class.

run

Function to get running result of the `YOLOv8` neural network.

Prototype

```
                    YOLOv8Result
                  run(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 390: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of input image (cv::Mat).

Returns

`YOLOv8Result` .

run

Function to get running result of the `YOLOv8` neural network in batch mode.

Prototype

```
std::vector<
                    YOLOv8Result
                  > run(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 391: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `YOLOv8Result` .

vitis::ai::YOLOvX

Base class for detecting objects in the input image (cv::Mat).

Input is an image (cv::Mat).

Output is the position of the pedestrians in the input image.

Sample code:

```
auto yolo = vitis::ai::YOLOvX::create("yolovx_pt", true);

Mat img = cv::imread("sample_yolovx.jpg");
auto results = yolo->run(img);

for (auto& result : results.bboxes) {
  int label = result.label;
  auto& box = result.box;

  cout << "RESULT: " << label << "\t" << std::fixed << std::setprecision(2)
       << box[0] << "\t" << box[1] << "\t" << box[2] << "\t" << box[3] <<
"\t"
       << std::setprecision(6) << result.score << "\n";
  rectangle(img, Point(box[0], box[1]), Point(box[2], box[3]),
            Scalar(0, 255, 0), 1, 1, 0);
}
imwrite("result.jpg", img);
```

Quick Function Reference

The following table lists all the functions defined in the `vitis::ai::YOLOvX` class:

Table 392: **Quick Function Reference**

Type	Member	Arguments
std::unique_ptr< `YOLOvX` >	create	const std::string & model_name bool need_preprocess

Table 392: **Quick Function Reference** *(cont'd)*

Type	Member	Arguments
std::unique_ptr< YOLOvX >	create	const std::string & model_name xir::Attrs * attrs bool need_preprocess
YOLOvXResult	run	const cv::Mat & image
std::vector< YOLOvXResult >	run	const std::vector< cv::Mat > & images

Functions

create

Factory function to get an instance of derived classes of class `YOLOvX` .

Prototype

```
std::unique_ptr<
                YOLOvX
              > create(const std::string &model_name, bool
need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 393: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `YOLOvX` class.

create

Factory function to get an instance of derived classes of class `YOLOvX` .

Prototype

```
std::unique_ptr<
                YOLOvX
             > create(const std::string &model_name, xir::Attrs *attrs,
bool need_preprocess=true);
```

Parameters

The following table lists the `create` function arguments.

Table 394: **create Arguments**

Type	Member	Description
const std::string &	model_name	Model name
xir::Attrs *	attrs	XIR attributes, used to bind different models to the same dpu core
bool	need_preprocess	Normalize with mean/scale or not, default value is true.

Returns

An instance of `YOLOvX` class.

run

Function to get running result of the `YOLOvX` neural network.

Prototype

```


                YOLOvXResult
             run(const cv::Mat &image)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 395: **run Arguments**

Type	Member	Description
const cv::Mat &	image	Input data of input image (cv::Mat).

Returns

`YOLOvXResult`.

run

Function to get running result of the `YOLOvX` neural network in batch mode.

Prototype

```
std::vector<
               YOLOvXResult
             > run(const std::vector< cv::Mat > &images)=0;
```

Parameters

The following table lists the `run` function arguments.

Table 396: **run Arguments**

Type	Member	Description
const std::vector< cv::Mat > &	images	Input data of input images (std:vector<cv::Mat>). The size of input images equals batch size obtained by get_input_batch.

Returns

The vector of `YOLOvXResult` .

Data Structures

vitis::ai::ANNORET

Struct of the result returned by the pointpillars neural network in the annotation mode. It is mainly used for accuracy test or bev image drawing.

Declaration

```
typedef struct
{
  std::vector< std::string > name;
  V1I label;
  V1F truncated;
  V1I occluded;
  V1F alpha;
  V2I bbox;
  V2F dimensions;
  V2F location;
  V1F rotation_y;
  V1F score;
  V2F box3d_camera;
  V2F box3d_lidar;
  void clear();

} vitis::ai::ANNORET;
```

Table 397: **Structure vitis::ai::ANNORET Member Description**

Member	Description
name	Name of detected result in vector: such as Car Cylist Pedestrian.
label	Label of detected result.
truncated	Truncated information.
occluded	Occluded information.
alpha	Alpha information.
bbox	bbox information.
dimensions	Dimensions information.
location	Location information.
rotation_y	rotation_y information.
score	Score information.
box3d_camera	box3d_camera information.
box3d_lidar	box3d_lidar information.
clear	Inner function to clear all fields.

vitis::ai::BCCResult

Declaration

```
typedef struct
{
  int width;
  int height;
  int count;

} vitis::ai::BCCResult;
```

Table 398: **Structure vitis::ai::BCCResult Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
count	Count of crowd.

vitis::ai::BoundingBox

Base class for detecting persons and feats from an image (cv::Mat).

Input is an image (cv::Mat).

Output is the enlarged image.

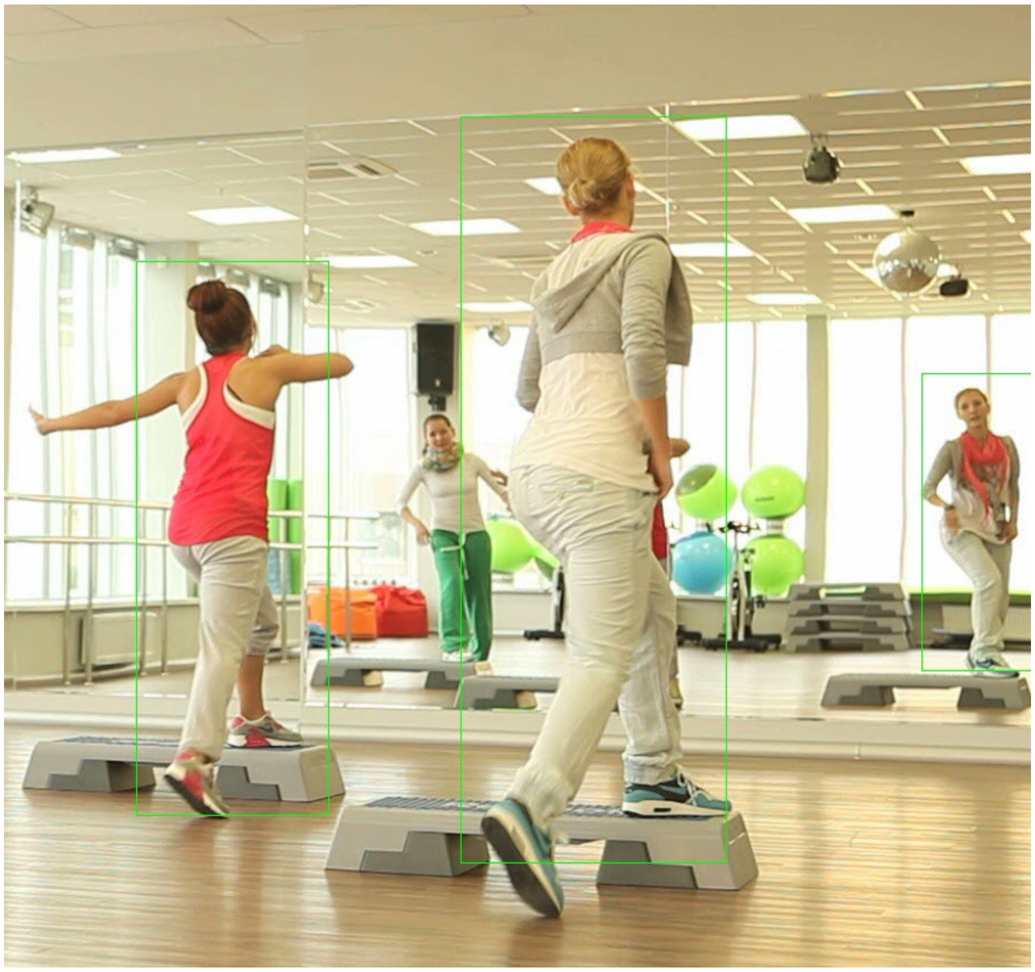
Sample code:

***Note*:** The input image size is 640x480

```
auto image_file = string(argv[2]);
Mat input_img = imread(image_file);
if (input_img.empty()) {
  cerr << "can't load image! " << argv[2] << endl;
  return -1;
}
auto det = vitis::ai::FairMot::create(argv[1]);
auto result = det->run(input_img);
auto feats = result.feats;
auto bboxes = result.bboxes;
auto img = input_img.clone();
for (auto i = 0u; i < bboxes.size(); ++i) {
  auto box = bboxes[i];
  float x = box.x * (img.cols);
  float y = box.y * (img.rows);
  int xmin = x;
  int ymin = y;
  int xmax = x + (box.width) * (img.cols);
  int ymax = y + (box.height) * (img.rows);
  float score = box.score;
  xmin = std::min(std::max(xmin, 0), img.cols);
  xmax = std::min(std::max(xmax, 0), img.cols);
  ymin = std::min(std::max(ymin, 0), img.rows);
  ymax = std::min(std::max(ymax, 0), img.rows);

  LOG(INFO) << "RESULT " << box.label << " :\t" << xmin << "\t" << ymin
            << "\t" << xmax << "\t" << ymax << "\t" << score << "\n";
  LOG(INFO) << "feat size: " << feats[i].size()
            << " First 5 digits: " << feats[i].data[0] + 0.0f << " "
            << feats[i].data[1] + 0.0f << " " << feats[i].data[2] + 0.0f
            << " " << feats[i].data[3] + 0.0f << " "
            << feats[i].data[4] + 0.0f << endl;
  cv::rectangle(img, cv::Point(xmin, ymin), cv::Point(xmax, ymax),
                cv::Scalar(0, 255, 0), 1, 1, 0);
}
auto out = image_file.substr(0, image_file.size() - 4) + "_out.jpg";
LOG(INFO) << "write result to " << out;
cv::imwrite(out, img);
```

Display of the model results:

Figure 78: **result image**



Struct of an object coordinates and confidence.

Declaration

```
typedef struct
{
  float x;
  float y;
  float width;
  float height;
  int label;
  float score;

} vitis::ai::BoundingBox;
```

Table 399: **Structure vitis::ai::BoundingBox Member Description**

Member	Description
x	x-coordinate. x is normalized relative to the input image columns. Range from 0 to 1.

Table 399: **Structure vitis::ai::BoundingBox Member Description** *(cont'd)*

Member	Description
y	y-coordinate. y is normalized relative to the input image rows. Range from 0 to 1.
width	Body width. Width is normalized relative to the input image columns, Range from 0 to 1.
height	Body height. Heigth is normalized relative to the input image rows, Range from 0 to 1.
label	Body detection label. The value ranges from 0 to 21.
score	Body detection confidence. The value ranges from 0 to 1.

vitis::ai::CenterPointResult

Struct of the result with the centerpoint network.

Declaration

```
typedef struct
{
  std::vector< float > bbox;
  float score;
  int label;
  float bbox[9];
  uint32_t label;

} vitis::ai::CenterPointResult;
```

Table 400: **Structure vitis::ai::CenterPointResult Member Description**

Member	Description
bbox	Bounding box 3d: {x, y, z, x_size, y_size, z_size, yaw}.
score	Score.
label	`Classification`.
bbox	Bounding box 3d: {x, y, z, x_size, y_size, z_size, yaw,vel1,vel2}.
label	the class label

vitis::ai::CflownetResult

Declaration

```
typedef struct
{
  int width;
  int height;
  std::vector< float > data;

} vitis::ai::CflownetResult;
```

Table 401: **Structure vitis::ai::CflownetResult Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
data	(128x128)

vitis::ai::ClassificationResult

Struct of the result with the classification network.

Declaration

```
typedef struct
{
  int width;
  int height;
  std::vector< Score > scores;
  int type;
  const char * lookup(int index);

} vitis::ai::ClassificationResult;
```

Table 402: **Structure vitis::ai::ClassificationResult Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
scores	A vector of object width confidence in the first k; k defaults to 5 and can be modified through the model configuration file.
type	`Classification` label type.
lookup	The classification corresponding by index.

vitis::ai::ClassificationResult::Score

Struct of index and confidence for an object.

Declaration

```
typedef struct
{
  int index;
  float score;

} vitis::ai::ClassificationResult::Score;
```

Table 403: **Structure vitis::ai::ClassificationResult::Score Member Description**

Member	Description
index	The index of the result in the ImageNet.
score	Confidence of this category.

vitis::ai::clocs::ClocsInfo

Structure to describe clocs input information.

Declaration

```
typedef struct
{
  std::vector< float > calib_P2;
  std::vector< float > calib_Trv2c;
  std::vector< float > calib_rect;
  std::vector< float > points;
  cv::Mat image;

} vitis::ai::clocs::ClocsInfo;
```

Table 404: **Structure vitis::ai::clocs::ClocsInfo Member Description**

Member	Description
calib_P2	P2 size: 16.
calib_Trv2c	Tr_velo_to_cam size: 16.
calib_rect	R0_rect size: 16.
points	3D Lidar Points.
image	2D Image

vitis::ai::ClocsResult

Declaration

```
typedef struct
{
  std::vector< PPBbox > bboxes;

} vitis::ai::ClocsResult;
```

Table 405: **Structure vitis::ai::ClocsResult Member Description**

Member	Description
bboxes	All bounding boxes.

vitis::ai::ClocsResult::PPBbox

Declaration

```
typedef struct
{
  float score;
  std::vector< float > bbox;
  uint32_t label;

} vitis::ai::ClocsResult::PPBbox;
```

Table 406: **Structure vitis::ai::ClocsResult::PPBbox Member Description**

Member	Description
score	Confidence.
bbox	3D lidar bounding box: x, y, z, x-size, y-size, z-size, yaw.
label	`Classification`, for `Clocs`, only one class: Car.

vitis::ai::Covid19SegmentationResult

Declaration

```
typedef struct
{
  int width;
  int height;
  cv::Mat positive_classification;
  cv::Mat infected_area_classification;

} vitis::ai::Covid19SegmentationResult;
```

Table 407: **Structure vitis::ai::Covid19SegmentationResult Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
positive_classification	Positive detection result. The cv::Mat type is CV_8UC1 or CV_8UC3.
infected_area_classification	Infected area detection result. The cv::Mat type is CV_8UC1 or CV_8UC3.

vitis::ai::DISPLAY_PARAM

Four data structure getting from the calibration information. It is mainly used for accuracy test or bev image drawing. See detail in the overview/samples/pointpillars/readme for more information.

Declaration

```
typedef struct
{
  V2F P2;
  V2F rect;
  V2F Trv2c;
  V2F p2rect;

} vitis::ai::DISPLAY_PARAM;
```

Table 408: **Structure vitis::ai::DISPLAY_PARAM Member Description**

Member	Description
P2	P2 information.
rect	rect information
Trv2c	Trv2c information.
p2rect	p2rect information

vitis::ai::EfficientDetD2Result

Struct of the result returned by the `EfficientDetD2` neural network.

Declaration

```
typedef struct
{
  int width;
  int height;
  std::vector< BoundingBox > bboxes;

} vitis::ai::EfficientDetD2Result;
```

Table 409: **Structure vitis::ai::EfficientDetD2Result Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
bboxes	All objects, a vector of `BoundingBox`.

vitis::ai::EfficientDetD2Result::BoundingBox

Struct of an object coordinate, confidence and classification.

Declaration

```
typedef struct
{
  int label;
  float score;
  float x;
  float y;
  float width;
  float height;

} vitis::ai::EfficientDetD2Result::BoundingBox;
```

Table 410: **Structure vitis::ai::EfficientDetD2Result::BoundingBox Member Description**

Member	Description
label	`Classification`.
score	Confidence.
x	x-coordinate. x is normalized relative to the input image columns. Range from 0 to 1.
y	y-coordinate. y is normalized relative to the input image rows. Range from 0 to 1.
width	Width. Width is normalized relative to the input image columns, Range from 0 to 1.
height	Height. Heigth is normalized relative to the input image rows, Range from 0 to 1.

vitis::ai::FaceDetectResult

Struct of the result with the facedetect network.

Declaration

```
typedef struct
{
  int width;
  int height;
  std::vector< BoundingBox > rects;

} vitis::ai::FaceDetectResult;
```

Table 411: **Structure vitis::ai::FaceDetectResult Member Description**

Member	Description
width	Width of an input image.
height	Height of an input image.
rects	All faces, filtered by confidence >= detect threshold.

vitis::ai::FaceDetectResult::BoundingBox

The coordinate and confidence of a face.

Declaration

```
typedef struct
{
  float x;
  float y;
  float width;
  float height;
  float score;

} vitis::ai::FaceDetectResult::BoundingBox;
```

Table 412: **Structure vitis::ai::FaceDetectResult::BoundingBox Member Description**

Member	Description
x	x-coordinate. x is normalized relative to the input image columns. Range from 0 to 1.
y	y-coordinate. y is normalized relative to the input image rows. Range from 0 to 1.
width	face width. Width is normalized relative to the input image columns, Range from 0 to 1.
height	face height. Heigth is normalized relative to the input image rows, Range from 0 to 1.
score	face confidence, the value range from 0 to 1.

vitis::ai::FaceFeatureFixedResult

The result of `FaceFeature`. It is a 512 dimensions vector, fix point values.

Declaration

```
typedef struct
{
  std::array< int8_t, 512 > vector_t;
  int width;
  int height;
  float scale;
  std::unique_ptr< vector_t > feature;

} vitis::ai::FaceFeatureFixedResult;
```

Table 413: **Structure vitis::ai::FaceFeatureFixedResult Member Description**

Member	Description
vector_t	The 512 dimensions vector, in fix point format.
width	Width of an input image.
height	Height of an input image.

Table 413: **Structure vitis::ai::FaceFeatureFixedResult Member Description** *(cont'd)*

Member	Description
scale	The fix point.
feature	A vector of 512 fixed values.

vitis::ai::FaceFeatureFloatResult

The result of FaceFeature. It is a 512 dimensions vector, float value.

Declaration

```
typedef struct
{
  std::array< float, 512 > vector_t;
  int width;
  int height;
  std::unique_ptr< vector_t > feature;

} vitis::ai::FaceFeatureFloatResult;
```

Table 414: **Structure vitis::ai::FaceFeatureFloatResult Member Description**

Member	Description
vector_t	The 512 dimensions vector.
width	Width of an input image.
height	Height of an input image.
feature	A vector of 512 float values.

vitis::ai::FaceLandmarkResult

Struct of the result returned by the facelandmark network.

Declaration

```
typedef struct
{
  std::array< std::pair< float, float >, 5 > points;

} vitis::ai::FaceLandmarkResult;
```

Table 415: **Structure vitis::ai::FaceLandmarkResult Member Description**

Member	Description
points	Five key points coordinate. This array of <x,y> has five elements, x / y is normalized relative to width / height, the value range from 0 to 1.

vitis::ai::FaceQuality5ptResult

Struct of result returned by the facequality5pt network.

Declaration

```
typedef struct
{
  int width;
  int height;
  float score;
  std::array< std::pair< float, float >, 5 > points;

} vitis::ai::FaceQuality5ptResult;
```

Table 416: **Structure vitis::ai::FaceQuality5ptResult Member Description**

Member	Description
width	Width of a input image.
height	Height of a input image.
score	The quality of face. The value range is from 0 to 1. If the option "original_quality" in the model prototxt is false, it is a normal mode. If the option "original_quality" is true, the quality score can be larger than 1, this is a special mode only for accuracy test.
points	Five key points coordinate. An array of <x,y> has five elements where x and y are normalized relative to input image columns and rows. The value range is from 0 to 1.

vitis::ai::FairMotResult

Result with the `Rcan` network.

Declaration

```
typedef struct
{
  int width;
  int height;
  std::vector< cv::Mat > feats;
  std::vector< BoundingBox > bboxes;

} vitis::ai::FairMotResult;
```

Table 417: **Structure vitis::ai::FairMotResult Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
feats	The vector of reid feat.
bboxes	The vector of `BoundingBox`.

vitis::ai::HourglassResult

Result with the openpose network.

Declaration

```
typedef struct
{
  int width;
  int height;
  std::vector< PosePoint > poses;

} vitis::ai::HourglassResult;
```

Table 418: **Structure vitis::ai::HourglassResult Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
poses	A vector of pose, pose is represented by a vector of `PosePoint`. Joint points are arranged in order 0: head, 1: neck, 2: L_shoulder, 3:L_elbow, 4: L_wrist, 5: R_shoulder, 6: R_elbow, 7: R_wrist, 8: L_hip, 9:L_knee, 10: L_ankle, 11: R_hip, 12: R_knee, 13: R_ankle

vitis::ai::HourglassResult::PosePoint

Struct of a coordinate point and the point type.

Declaration

```
typedef struct
{
  int type;
  cv::Point2f point;

} vitis::ai::HourglassResult::PosePoint;
```

Table 419: **Structure vitis::ai::HourglassResult::PosePoint Member Description**

Member	Description
type	Point type. • 1 : "valid" • 3 : "invalid"
point	Coordinate point.

vitis::ai::MedicalDetectionResult

Struct of the result returned by the medical refinedet network.

Declaration

```
typedef struct
{
  int width;
  int height;
  std::vector< BoundingBox > bboxes;

} vitis::ai::MedicalDetectionResult;
```

Table 420: **Structure vitis::ai::MedicalDetectionResult Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
bboxes	All objects, a vector of `BoundingBox`.

vitis::ai::MedicalDetectionResult::BoundingBox

Struct of an object coordinate ,confidence and classification.

Declaration

```
typedef struct
{
  int label;
  float score;
  float x;
  float y;
  float width;
  float height;

} vitis::ai::MedicalDetectionResult::BoundingBox;
```

Table 421: **Structure vitis::ai::MedicalDetectionResult::BoundingBox Member Description**

Member	Description
label	`Classification`.
score	Confidence.
x	x-coordinate. x is normalized relative to the input image columns. The range of values is from 0 to 1.
y	y-coordinate. y is normalized relative to the input image rows. The range of values is from 0 to 1.
width	width. width is normalized relative to the input image columns. The value range from 0 to 1.
height	height, height is normalized relative to the input image rows. The value range from 0 to 1.

vitis::ai::MedicalSegcellResult

Struct of the result returned by the segmentation neural network.

Declaration

```
typedef struct
{
  int width;
  int height;
  cv::Mat segmentation;

} vitis::ai::MedicalSegcellResult;
```

Table 422: **Structure vitis::ai::MedicalSegcellResult Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
segmentation	`Segmentation` result in cv::Mat mode.

vitis::ai::MedicalSegmentationResult

Struct of the result returned by the `MedicalSegmentation` neural network.

Declaration

```
typedef struct
{
  int width;
  int height;
  std::vector< cv::Mat > segmentation;

} vitis::ai::MedicalSegmentationResult;
```

Table 423: **Structure vitis::ai::MedicalSegmentationResult Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
segmentation	A vector of cv::Mat (segmentation result).

vitis::ai::Monodepth2Result

Declaration

```
typedef struct
{
  int width;
  int height;
  cv::Mat mat;

} vitis::ai::Monodepth2Result;
```

Table 424: **Structure vitis::ai::Monodepth2Result Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
mat	cv::Mat of returned pic

vitis::ai::MovenetResult

Movenet model, input size is 192x192.

Base class for detecting poses of people.

Input is an image (cv:Mat).

Output is `MovenetResult`.

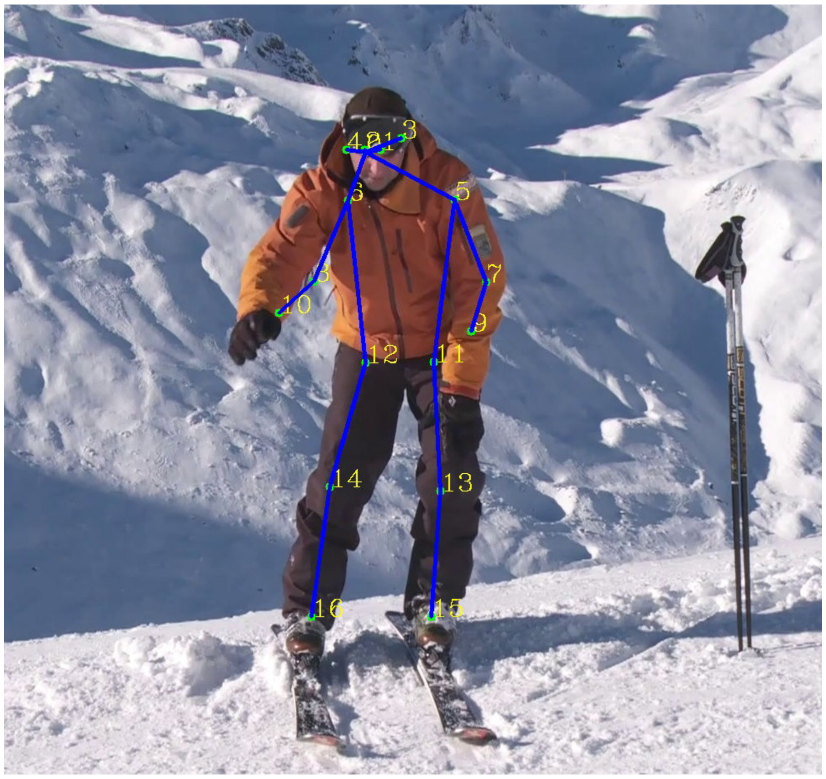
Sample code:

```
auto image = cv::imread(argv[2]);
if (image.empty()) {
  std::cerr << "cannot load " << argv[2] << std::endl;
  abort();
}
auto det = vitis::ai::Movenet::create(argv[1]);
vector<vector<int>> limbSeq = {{0, 1}, {0, 2},{0, 3},{0, 4},{0, 5},{0, 6},
                               {5, 7},  {7, 9},  {6, 8}, {8, 10},
                                {5, 11},   {6, 12},  {11, 13}, {13, 15},
                                {12, 14}, {14, 16}};

auto results = det->run(image.clone());
for (size_t i = 0; i < results.poses.size(); ++i) {
  cout<< results.poses[i]<<endl;
  if (results.poses[i].y >0 && results.poses[i].x > 0) {
    cv::putText(image, to_string(i),results.poses[i],
    cv::FONT_HERSHEY_COMPLEX, 1, cv::Scalar(0, 255, 255), 1, 1, 0);
    cv::circle(image, results.poses[i], 5, cv::Scalar(0, 255, 0),
               -1);
  }
}
for (size_t i = 0; i < limbSeq.size(); ++i) {
  auto a = results.poses[limbSeq[i][0]];
```

```
    auto b = results.poses[limbSeq[i][1]];
    if (a.x >0  && b.x > 0) {
      cv::line(image, a, b, cv::Scalar(255, 0, 0), 3, 4);
    }
}
```

Display of the movenet model results: width=400px

Figure 79: **movenet result image**



Declaration

```
typedef struct
{
    int width;
    int height;
    std::vector< cv::Point2f > poses;

} vitis::ai::MovenetResult;
```

Table 425: **Structure vitis::ai::MovenetResult Member Description**

Member	Description
width	Width of input image.
height	Height of input image.

Table 425: **Structure vitis::ai::MovenetResult Member Description** *(cont'd)*

Member	Description
poses	A vector of pose, pose is represented by a vector of Point. Joint points are arranged in order 0: 'nose', 1: 'left_eye', 2: 'right_eye', 3: 'left_ear', 4: 'right_ear', 5: 'left_shoulder', 6: 'right_shoulder', 7: 'left_elbow', 8: 'right_elbow', 9: 'left_wrist', 10 : 'right_wrist', 11: 'left_hip', 12: 'right_hip', 13: 'left_knee', 14: 'right_knee', 15: 'left_ankle', 16: 'right_ankle']

vitis::ai::MultiTaskResult

Struct of the result returned by the `MultiTask` network.

Declaration

```
typedef struct
{
  int width;
  int height;
  std::vector< VehicleResult > vehicle;
  cv::Mat segmentation;

} vitis::ai::MultiTaskResult;
```

Table 426: **Structure vitis::ai::MultiTaskResult Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
vehicle	Detection result of `SSD` task.
segmentation	`Segmentation` result to visualize, cv::Mat type is CV_8UC1 or CV_8UC3.

vitis::ai::MultiTaskv3Result

Struct of the result returned by the `MultiTaskv3` network.

Declaration

```
typedef struct
{
  int width;
  int height;
  std::vector< Vehiclev3Result > vehicle;
  cv::Mat segmentation;
  cv::Mat lane;
  cv::Mat drivable;
  cv::Mat depth;

} vitis::ai::MultiTaskv3Result;
```

Table 427: **Structure vitis::ai::MultiTaskv3Result Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
vehicle	Detection result of SSD task.
segmentation	Segmentation result to visualize, cv::Mat type is CV_8UC1 or CV_8UC3.
lane	Lane segmentation.
drivable	Drivable area.
depth	Depth estimation.

vitis::ai::OCRResult

Declaration

```
typedef struct
{
  int width;
  int height;
  std::vector< std::string > words;
  std::vector< std::vector< cv::Point > > box;

} vitis::ai::OCRResult;
```

Table 428: **Structure vitis::ai::OCRResult Member Description**

Member	Description
width	width of network input.
height	Height of network input.
words	vector of recognized words in input pic
box	vector of box information of the recognized words in input pic

vitis::ai::OFAYOLOResult

Struct of the result returned by the OFA_YOLO neural network.

Declaration

```
typedef struct
{
  int width;
  int height;
  std::vector< BoundingBox > bboxes;

} vitis::ai::OFAYOLOResult;
```

Table 429: **Structure vitis::ai::OFAYOLOResult Member Description**

Member	Description
width	Width of input image.
height	Height of output image.
bboxes	All objects, The vector of `BoundingBox`.

vitis::ai::OFAYOLOResult::BoundingBox

Struct of detection result with an object.

Declaration

```
typedef struct
{
  int label;
  float score;
  float x;
  float y;
  float width;
  float height;

} vitis::ai::OFAYOLOResult::BoundingBox;
```

Table 430: **Structure vitis::ai::OFAYOLOResult::BoundingBox Member Description**

Member	Description
label	`Classification`.
score	Confidence. The value ranges from 0 to 1.
x	x-coordinate. x is normalized relative to the input image columns. Range from 0 to 1.
y	y-coordinate. y is normalized relative to the input image rows. Range from 0 to 1.
width	Width. Width is normalized relative to the input image columns, Range from 0 to 1.
height	Height. Heigth is normalized relative to the input image rows, Range from 0 to 1.

vitis::ai::OpenPoseResult

Result with the openpose network.

Declaration

```
typedef struct
{
  int width;
  int height;
  std::vector< std::vector< PosePoint > > poses;

} vitis::ai::OpenPoseResult;
```

Table 431: **Structure vitis::ai::OpenPoseResult Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
poses	A vector of pose. Pose is represented by a vector of `PosePoint`. Joint points are arranged in order 0: head, 1: neck, 2: L_shoulder, 3:L_elbow, 4: L_wrist, 5: R_shoulder, 6: R_elbow, 7: R_wrist, 8: L_hip, 9:L_knee, 10: L_ankle, 11: R_hip, 12: R_knee, 13: R_ankle

vitis::ai::OpenPoseResult::PosePoint

Struct of a coordinate point and the point type.

Declaration

```
typedef struct
{
  int type;
  cv::Point2f point;

} vitis::ai::OpenPoseResult::PosePoint;
```

Table 432: **Structure vitis::ai::OpenPoseResult::PosePoint Member Description**

Member	Description
type	Point type. • 1 : "valid" • 3 : "invalid"
point	Coordinate point.

vitis::ai::PlateDetectResult

Struct of the result returned by the platedetect network.

Declaration

```
typedef struct
{
  int width;
  int height;
  BoundingBox box;
  Point top_left;
  Point top_right;
  Point bottom_left;
  Point bottom_right;

} vitis::ai::PlateDetectResult;
```

Table 433: **Structure vitis::ai::PlateDetectResult Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
box	The position of plate.
top_left	The top_left point.
top_right	The top_right point.
bottom_left	The bottom_left point.
bottom_right	The bottom_right point.

vitis::ai::PlateDetectResult::BoundingBox

Declaration

```
typedef struct
{
  float score;
  float x;
  float y;
  float width;
  float height;

} vitis::ai::PlateDetectResult::BoundingBox;
```

Table 434: **Structure vitis::ai::PlateDetectResult::BoundingBox Member Description**

Member	Description
score	Plate confidence, the value ranges from 0 to 1.
x	x-coordinate. x is normalized relative to the input image columns. Range from 0 to 1.
y	y-coordinate. y is normalized relative to the input image rows. Range from 0 to 1.
width	Plate width. Width is normalized relative to the input image columns, Range from 0 to 1.
height	Plate height. Heigth is normalized relative to the input image rows, Range from 0 to 1.

vitis::ai::PlateDetectResult::Point

Plate coordinate point.

Declaration

```
typedef struct
{
  float x;
  float y;

} vitis::ai::PlateDetectResult::Point;
```

Table 435: **Structure vitis::ai::PlateDetectResult::Point Member Description**

Member	Description
x	x-coordinate, the value ranges from 0 to 1.
y	y-coordinate, the value ranges from 0 to 1.

vitis::ai::PlateNumResult

Struct of the result of the platenum network.

Declaration

```
typedef struct
{
  int width;
  int height;
  std::string plate_number;
  std::string plate_color;

} vitis::ai::PlateNumResult;
```

Table 436: **Structure vitis::ai::PlateNumResult Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
plate_number	The plate number.
plate_color	The plate color, Blue / Yellow.

vitis::ai::pointpillars_nus::CamInfo

Camera information of input points.

Declaration

```
typedef struct
{
  uint64_t timestamp;
  std::array< float, 3 > s2l_t;
  std::array< float, 9 > s2l_r;
  std::array< float, 9 > cam_intr;

} vitis::ai::pointpillars_nus::CamInfo;
```

Table 437: **Structure vitis::ai::pointpillars_nus::CamInfo Member Description**

Member	Description
timestamp	Timestamp of input points.
s2l_t	Sensor to lidar translation params.
s2l_r	Sensor to lidar rotation params.
cam_intr	Camera intrinsic params.

vitis::ai::pointpillars_nus::Points

Structure to describe input points data.

Declaration

```
typedef struct
{
  int dim;
  std::shared_ptr< std::vector< float > > points;

} vitis::ai::pointpillars_nus::Points;
```

Table 438: **Structure vitis::ai::pointpillars_nus::Points Member Description**

Member	Description
dim	`Points` dim.
points	`Points` data.

vitis::ai::pointpillars_nus::PointsInfo

Structure to describe points information.

Declaration

```
typedef struct
{
  std::vector< CamInfo > cam_info;
  Points points;
  uint64_t timestamp;
  std::vector< SweepInfo > sweep_infos;

} vitis::ai::pointpillars_nus::PointsInfo;
```

Table 439: **Structure vitis::ai::pointpillars_nus::PointsInfo Member Description**

Member	Description
cam_info	Camera information.
points	Points.
timestamp	Timestamp of points.
sweep_infos	Sweeps information.

vitis::ai::pointpillars_nus::SweepInfo

Structure to describe sweeps.

Declaration

```
typedef struct
{
  CamInfo cam_info;
  Points points;

} vitis::ai::pointpillars_nus::SweepInfo;
```

Table 440: **Structure vitis::ai::pointpillars_nus::SweepInfo Member Description**

Member	Description
cam_info	Camera information.
points	Points.

vitis::ai::PointPillarsNuscenesResult

Struct of the result returned by the PointPillarsNuscenes network.

Declaration

```
typedef struct
{
  std::vector< PPBbox > bboxes;

} vitis::ai::PointPillarsNuscenesResult;
```

Table 441: **Structure vitis::ai::PointPillarsNuscenesResult Member Description**

Member	Description
bboxes	All bounding boxes.

vitis::ai::PointPillarsResult

Struct of the final result returned by the pointpillars neural network encapsulated with width/ height information.

Declaration

```
typedef struct
{
  int width;
  int height;
  PPResult ppresult;

} vitis::ai::PointPillarsResult;
```

Table 442: **Structure vitis::ai::PointPillarsResult Member Description**

Member	Description
width	Width of network input.
height	Height of network input.
ppresult	Final result returned by the pointpillars neural network.

vitis::ai::PoseDetectResult

Struct of the result returned by the posedetect network.

Declaration

```
typedef struct
{
  cv::Point2f Point;
  int width;
  int height;
  Pose14Pt pose14pt;

} vitis::ai::PoseDetectResult;
```

Table 443: **Structure vitis::ai::PoseDetectResult Member Description**

Member	Description
Point	A coordinate points.
width	Width of input image.
height	Height of input image.
pose14pt	The pose of input image.

vitis::ai::PoseDetectResult::Pose14Pt

Data structure for a pose. Represented by 14 coordinate points.

Declaration

```
typedef struct
{
  Point right_shoulder;
  Point right_elbow;
  Point right_wrist;
  Point left_shoulder;
  Point left_elbow;
  Point left_wrist;
  Point right_hip;
  Point right_knee;
  Point right_ankle;
  Point left_hip;
  Point left_knee;
  Point left_ankle;
  Point head;
  Point neck;

} vitis::ai::PoseDetectResult::Pose14Pt;
```

Table 444: **Structure vitis::ai::PoseDetectResult::Pose14Pt Member Description**

Member	Description
right_shoulder	R_shoulder coordinate.
right_elbow	R_elbow coordinate.
right_wrist	R_wrist coordinate.
left_shoulder	L_shoulder coordinate.
left_elbow	L_elbow coordinate.
left_wrist	L_wrist coordinate.
right_hip	R_hip coordinate.
right_knee	R_knee coordinate.
right_ankle	R_ankle coordinate.
left_hip	L_hip coordinate.
left_knee	L_knee coordinate.
left_ankle	L_ankle coordinate.
head	Head coordinate.
neck	Neck coordinate.

vitis::ai::PPBbox

Struct of an object coordinate, confidence and classification.

Declaration

```
typedef struct
{
  float score;
  std::vector< float > bbox;
  uint32_t label;

} vitis::ai::PPBbox;
```

Table 445: **Structure vitis::ai::PPBbox Member Description**

Member	Description
score	Confidence.
bbox	Bounding box: x, y, z, x-size, y-size, z-size, yaw, custom value and so on.
label	`Classification`.

vitis::ai::PPResult

Struct of the final result returned by the pointpillars neural network.

Declaration

```
typedef struct
{
  V2F final_box_preds;
  V1F final_scores;
  V1I label_preds;

} vitis::ai::PPResult;
```

Table 446: **Structure vitis::ai::PPResult Member Description**

Member	Description
final_box_preds	Final box predicted.
final_scores	Final scores predicted.
label_preds	Final label predicted.

vitis::ai::RcanResult

Result with the `Rcan` network.

Declaration

```
typedef struct
{
  int width;
  int height;
  cv::Mat feat;

} vitis::ai::RcanResult;
```

Table 447: **Structure vitis::ai::RcanResult Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
feat	Double size of input image.

vitis::ai::RefineDetResult

Struct of the result with the refinedet network.

Declaration

```
typedef struct
{
  int width;
  int height;
  std::vector< BoundingBox > bboxes;

} vitis::ai::RefineDetResult;
```

Table 448: **Structure vitis::ai::RefineDetResult Member Description**

Member	Description
width	Width of the input image.
height	Height of the input image.
bboxes	The vector of `BoundingBox`.

vitis::ai::RefineDetResult::BoundingBox

Struct of an object coordinates and confidence.

Declaration

```
typedef struct
{
  float x;
  float y;
  float width;
  float height;
  int label;
  float score;

} vitis::ai::RefineDetResult::BoundingBox;
```

Table 449: **Structure vitis::ai::RefineDetResult::BoundingBox Member Description**

Member	Description
x	x-coordinate. x is normalized relative to the input image columns. Range from 0 to 1.
y	y-coordinate. y is normalized relative to the input image rows. Range from 0 to 1.
width	Body width. Width is normalized relative to the input image columns, Range from 0 to 1.
height	Body height. Heigth is normalized relative to the input image rows, Range from 0 to 1.
label	Body detection label. The value ranges from 0 to 21.
score	Body detection confidence. The value ranges from 0 to 1.

vitis::ai::ReidResult

Result with the ReID network.

Declaration

```
typedef struct
{
  int width;
  int height;
  cv::Mat feat;

} vitis::ai::ReidResult;
```

Table 450: **Structure vitis::ai::ReidResult Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
feat	The feature of input image.

vitis::ai::RetinaFaceResult

Struct of the result with the retinaface network.

Declaration

```
typedef struct
{
  int width;
  int height;
  std::vector< BoundingBox > bboxes;
  std::vector< std::array< std::pair< float, float >, 5 > > landmarks;

} vitis::ai::RetinaFaceResult;
```

Table 451: **Structure vitis::ai::RetinaFaceResult Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
bboxes	All faces, filtered by confidence >= detect threshold.
landmarks	Landmarks.

vitis::ai::RetinaFaceResult::BoundingBox

The coordinate and confidence of a face.

Declaration

```
typedef struct
{
  float x;
  float y;
  float width;
  float height;
  float score;

} vitis::ai::RetinaFaceResult::BoundingBox;
```

Table 452: **Structure vitis::ai::RetinaFaceResult::BoundingBox Member Description**

Member	Description
x	x-coordinate. x is normalized relative to the input image columns. Range from 0 to 1.
y	y-coordinate. y is normalized relative to the input image rows. Range from 0 to 1.
width	Face width. Width is normalized relative to the input image columns, Range from 0 to 1.
height	Face height. Heigth is normalized relative to the input image rows, Range from 0 to 1.
score	Face confidence. The value ranges from 0 to 1.

vitis::ai::RoadLineResult

Struct of the result returned by the roadline network.

Declaration

```
typedef struct
{
  int width;
  int height;
  std::vector< Line > lines;

} vitis::ai::RoadLineResult;
```

Table 453: **Structure vitis::ai::RoadLineResult Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
lines	The vector of line.

vitis::ai::RoadLineResult::Line

Struct of the result returned by the roadline network.

Declaration

```
typedef struct
{
  int type;
  std::vector< cv::Point > points_cluster;

} vitis::ai::RoadLineResult::Line;
```

Table 454: **Structure vitis::ai::RoadLineResult::Line Member Description**

Member	Description
type	Road line type, the value range from 0 to 3. • 0 : background • 1 : white dotted line • 2 : white solid line • 3 : yollow line
points_cluster	Point clusters, make line from these.

vitis::ai::Segmentation3DResult

Base class for segmentation 3D object data in the vector<float> mode.

Declaration

```
typedef struct
{
  int width;
  int height;
  std::vector< int > array;

} vitis::ai::Segmentation3DResult;
```

Table 455: **Structure vitis::ai::Segmentation3DResult Member Description**

Member	Description
width	Width of the network model.
height	Height of the network model.
array	Input 3D object data.

vitis::ai::SegmentationResult

Struct of the result returned by the segmentation network.

FPN Num of segmentation classes

- 0 : "unlabeled"
- 1 : "ego vehicle"
- 2 : "rectification border"
- 3 : "out of roi"
- 4 : "static"
- 5 : "dynamic"
- 6 : "ground"
- 7 : "road"
- 8 : "sidewalk"
- 9 : "parking"
- 10 : "rail track"
- 11 : "building"
- 12 : "wall"
- 13 : "fence"

- 14 : "guard rail"
- 15 : "bridge"
- 16 : "tunnel"
- 17 : "pole"
- 18 : "polegroup"

Declaration

```
typedef struct
{
  int width;
  int height;
  cv::Mat segmentation;

} vitis::ai::SegmentationResult;
```

Table 456: **Structure vitis::ai::SegmentationResult Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
segmentation	`Segmentation` result. The cv::Mat type is CV_8UC1 or CV_8UC3.

vitis::ai::SoloResult

Result with the `Solo` network.

Declaration

```
typedef struct
{
  int width;
  int height;
  Ndarray< int > seg_masks;
  Ndarray< int > cate_labels;
  Ndarray< float > cate_scores;

} vitis::ai::SoloResult;
```

Table 457: **Structure vitis::ai::SoloResult Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
seg_masks	Double size of input image.
cate_labels	the labels
cate_scores	the scores

vitis::ai::SSDResult

Struct of the result returned by the SSD neural network.

Declaration

```
typedef struct
{
  int width;
  int height;
  std::vector< BoundingBox > bboxes;

} vitis::ai::SSDResult;
```

Table 458: **Structure vitis::ai::SSDResult Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
bboxes	All objects, a vector of BoundingBox.

vitis::ai::SSDResult::BoundingBox

Struct of an object coordinate, confidence and classification.

Declaration

```
typedef struct
{
  int label;
  float score;
  float x;
  float y;
  float width;
  float height;

} vitis::ai::SSDResult::BoundingBox;
```

Table 459: **Structure vitis::ai::SSDResult::BoundingBox Member Description**

Member	Description
label	Classification.
score	Confidence.
x	x-coordinate. x is normalized relative to the input image columns. Range from 0 to 1.
y	y-coordinate. y is normalized relative to the input image rows. Range from 0 to 1.
width	Width. Width is normalized relative to the input image columns, Range from 0 to 1.

Table 459: **Structure vitis::ai::SSDResult::BoundingBox Member Description** *(cont'd)*

Member	Description
height	Height. Heigth is normalized relative to the input image rows, Range from 0 to 1.

vitis::ai::TextMountainResult

Declaration

```
typedef struct
{
  int width;
  int height;
  std::vector< tmitem > res;

} vitis::ai::TextMountainResult;
```

Table 460: **Structure vitis::ai::TextMountainResult Member Description**

Member	Description
width	width of network input.
height	height of network input.
res	vector to hold the detected result

vitis::ai::TextMountainResult::tmitem

Struct to hold each textmountain detected result.

Declaration

```
typedef struct
{
  arr4_point2d box;
  float score;
   tmitem(arr4_point2d &inbox, float inscore);

} vitis::ai::TextMountainResult::tmitem;
```

Table 461: **Structure vitis::ai::TextMountainResult::tmitem Member Description**

Member	Description
box	4 Point2f to hold the box coordinate. sequence is clock-wise
score	scores for each box
tmitem	construct function;

vitis::ai::TFSSDResult

Struct of the result returned by the `TFSSD` neural network.

Declaration

```
typedef struct
{
  int width;
  int height;
  std::vector< BoundingBox > bboxes;

} vitis::ai::TFSSDResult;
```

Table 462: **Structure vitis::ai::TFSSDResult Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
bboxes	All objects, a vector of `BoundingBox`.

vitis::ai::TFSSDResult::BoundingBox

Struct of an object coordinate, confidence, classification.

Declaration

```
typedef struct
{
  int label;
  float score;
  float x;
  float y;
  float width;
  float height;

} vitis::ai::TFSSDResult::BoundingBox;
```

Table 463: **Structure vitis::ai::TFSSDResult::BoundingBox Member Description**

Member	Description
label	`Classification`.
score	Confidence.
x	x-coordinate. x is normalized relative to the input image columns. Range from 0 to 1.
y	y-coordinate. y is normalized relative to the input image rows. Range from 0 to 1.
width	Width. Width is normalized relative to the input image columns, Range from 0 to 1.

Table 463: **Structure vitis::ai::TFSSDResult::BoundingBox Member Description** *(cont'd)*

Member	Description
height	Height. Heigth is normalized relative to the input image rows, Range from 0 to 1.

vitis::ai::UltraFastResult

Struct of the result returned by the ultrafast neural network.

Declaration

```
typedef struct
{
  int width;
  int height;
  std::vector< std::vector< std::pair< float, float > > > lanes;

} vitis::ai::UltraFastResult;
```

Table 464: **Structure vitis::ai::UltraFastResult Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
lanes	vector of lanes information. each lane is a vector holding pair structure.

vitis::ai::Unet2DResult

Declaration

```
typedef struct
{
  int width;
  int height;
  std::vector< float > data;

} vitis::ai::Unet2DResult;
```

Table 465: **Structure vitis::ai::Unet2DResult Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
data	4 channels out data. Format: HWC

vitis::ai::VehicleClassificationResult

Struct of the result with the vehicleclassification network.

Declaration

```
typedef struct
{
  int width;
  int height;
  std::vector< Score > scores;
  int type;
  const char * lookup(int index);

} vitis::ai::VehicleClassificationResult;
```

Table 466: **Structure vitis::ai::VehicleClassificationResult Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
scores	A vector of object width confidence in the first k; k defaults to 5 and can be modified through the model configuration file.
type	`VehicleClassification` label type.
lookup	The vehicleclassification corresponding by index.

vitis::ai::VehicleClassificationResult::Score

Struct of index and confidence for an object.

Declaration

```
typedef struct
{
  int index;
  float score;

} vitis::ai::VehicleClassificationResult::Score;
```

Table 467: **Structure vitis::ai::VehicleClassificationResult::Score Member Description**

Member	Description
index	The index of the result in the ImageNet.
score	Confidence of this category.

vitis::ai::VehicleResult

A struct to define detection result of `MultiTask`.

Declaration

```
typedef struct
{
  int label;
  float score;
  float x;
  float y;
  float width;
  float height;
  float angle;

} vitis::ai::VehicleResult;
```

Table 468: **Structure vitis::ai::VehicleResult Member Description**

Member	Description
label	number of classes • 0 : "background" • 1 : "person" • 2 : "car" • 3 : "truck" • 4 : "bus" • 5 : "bike" • 6 : "sign" • 7 : "light"
score	Confidence of this target.
x	x-coordinate. x is normalized relative to the input image columns. Range from 0 to 1.
y	y-coordinate. y is normalized relative to the input image rows. Range from 0 to 1.
width	Width. Width is normalized relative to the input image columns, Range from 0 to 1.
height	Height. Heigth is normalized relative to the input image rows, Range from 0 to 1.
angle	The angle between the target vehicle and ourself.

vitis::ai::Vehiclev3Result

A struct to define detection result of `MultiTaskv3`.

Declaration

```
typedef struct
{
  int label;
  float score;
  float x;
  float y;
  float width;
  float height;
  float angle;

} vitis::ai::Vehiclev3Result;
```

Table 469: **Structure vitis::ai::Vehiclev3Result Member Description**

Member	Description
label	number of classes • 0 : "car" • 1 : "sign" • 2 : "person"
score	Confidence of this target.
x	x-coordinate. x is normalized relative to the input image columns. Range from 0 to 1.
y	y-coordinate. y is normalized relative to the input image rows. Range from 0 to 1.
width	Width. Width is normalized relative to the input image columns, Range from 0 to 1.
height	Height. Heigth is normalized relative to the input image rows, Range from 0 to 1.
angle	The angle between the target vehicle and ourself.

vitis::ai::YOLOv2Result

Struct of the result returned by the `YOLOv2` network.

Declaration

```
typedef struct
{
  int width;
  int height;
  std::vector< BoundingBox > bboxes;

} vitis::ai::YOLOv2Result;
```

Table 470: **Structure vitis::ai::YOLOv2Result Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
bboxes	All objects.

vitis::ai::YOLOv2Result::BoundingBox

Struct of an object coordinate, confidence, and classification.

Declaration

```
typedef struct
{
  int label;
  float score;
  float x;
  float y;
  float width;
  float height;

} vitis::ai::YOLOv2Result::BoundingBox;
```

Table 471: **Structure vitis::ai::YOLOv2Result::BoundingBox Member Description**

Member	Description
label	`Classification`.
score	Confidence. The value ranges from 0 to 1.
x	x-coordinate. x is normalized relative to the input image columns. Range from 0 to 1.
y	y-coordinate. y is normalized relative to the input image rows. Range from 0 to 1.
width	Width. Width is normalized relative to the input image columns, Range from 0 to 1.
height	Height. Heigth is normalized relative to the input image rows, Range from 0 to 1.

vitis::ai::YOLOv3Result

Struct of the result returned by the `YOLOv3` neural network.

Note: VOC dataset category:string label[20] = {"aeroplane", "bicycle", "bird", "boat", "bottle", "bus","car", "cat", "chair", "cow", "diningtable", "dog", "horse", "motorbike","person", "pottedplant", "sheep", "sofa", "train", "tvmonitor"};

Note: ADAS dataset category : string label[3] = {"car", "person", "cycle"};

Declaration

```
typedef struct
{
  int width;
  int height;
  std::vector< BoundingBox > bboxes;

} vitis::ai::YOLOv3Result;
```

Table 472: **Structure vitis::ai::YOLOv3Result Member Description**

Member	Description
width	Width of input image.
height	Height of output image.
bboxes	All objects, The vector of `BoundingBox`.

vitis::ai::YOLOv3Result::BoundingBox

Struct of detection result with an object.

Declaration

```
typedef struct
{
  int label;
  float score;
  float x;
  float y;
  float width;
  float height;

} vitis::ai::YOLOv3Result::BoundingBox;
```

Table 473: **Structure vitis::ai::YOLOv3Result::BoundingBox Member Description**

Member	Description
label	`Classification`.
score	Confidence. The value ranges from 0 to 1.
x	x-coordinate. x is normalized relative to the input image columns. Range from 0 to 1.
y	y-coordinate. y is normalized relative to the input image rows. Range from 0 to 1.
width	Width. Width is normalized relative to the input image columns, Range from 0 to 1.
height	Height. Heigth is normalized relative to the input image rows, Range from 0 to 1.

vitis::ai::YOLOv7Result

Struct of the result returned by the YOLOv7 neural network.

Declaration

```
typedef struct
{
  int width;
  int height;
  std::vector< BoundingBox > bboxes;

} vitis::ai::YOLOv7Result;
```

Table 474: **Structure vitis::ai::YOLOv7Result Member Description**

Member	Description
width	Width of input image.
height	Height of output image.
bboxes	All objects, The vector of `BoundingBox`.

vitis::ai::YOLOv7Result::BoundingBox

Struct of detection result with an object.

Declaration

```
typedef struct
{
  int label;
  float score;
  float x;
  float y;
  float width;
  float height;

} vitis::ai::YOLOv7Result::BoundingBox;
```

Table 475: **Structure vitis::ai::YOLOv7Result::BoundingBox Member Description**

Member	Description
label	`Classification`.
score	Confidence. The value ranges from 0 to 1.
x	x-coordinate. x is normalized relative to the input image columns. Range from 0 to 1.
y	y-coordinate. y is normalized relative to the input image rows. Range from 0 to 1.
width	Width. Width is normalized relative to the input image columns, Range from 0 to 1.
height	Height. Heigth is normalized relative to the input image rows, Range from 0 to 1.

vitis::ai::YOLOv8Result

Struct of the result returned by the `YOLOv8` neural network.

Declaration

```
typedef struct
{
  std::vector< BoundingBox > bboxes;

} vitis::ai::YOLOv8Result;
```

Table 476: **Structure vitis::ai::YOLOv8Result Member Description**

Member	Description
bboxes	All objects, The vector of `BoundingBox`.

vitis::ai::YOLOv8Result::BoundingBox

Struct of detection result with an object.

Declaration

```
typedef struct
{
  int label;
  float score;
  std::vector< float > box;

} vitis::ai::YOLOv8Result::BoundingBox;
```

Table 477: **Structure vitis::ai::YOLOv8Result::BoundingBox Member Description**

Member	Description
label	`Classification`.
score	Confidence. The value ranges from 0 to 1.
box	(x0,y0,x1,y1). x0, x1 Range from 0 to the input image columns. y0,y1. Range from 0 to the input image rows.

vitis::ai::YOLOvXResult

Struct of the result returned by the `YOLOvX` neural network.

Declaration

```
typedef struct
{
  std::vector< BoundingBox > bboxes;

} vitis::ai::YOLOvXResult;
```

Table 478: **Structure vitis::ai::YOLOvXResult Member Description**

Member	Description
bboxes	All objects, The vector of `BoundingBox`.

vitis::ai::YOLOvXResult::BoundingBox

Struct of detection result with an object.

Declaration

```
typedef struct
{
  int label;
  float score;
  std::vector< float > box;

} vitis::ai::YOLOvXResult::BoundingBox;
```

Table 479: **Structure vitis::ai::YOLOvXResult::BoundingBox Member Description**

Member	Description
label	`Classification`.
score	Confidence. The value ranges from 0 to 1.
box	(x0,y0,x1,y1). x0, x1 Range from 0 to the input image columns. y0,y1. Range from 0 to the input image rows.

xilinx::ai::FaceQualityResult

The result of the facequality network. It is a single float value.

Declaration

```
typedef struct
{
  int width;
  int height;
  float value;

} xilinx::ai::FaceQualityResult;
```

Table 480: **Structure xilinx::ai::FaceQualityResult Member Description**

Member	Description
width	Width of input image.
height	Height of input image.
value	Quality value ranges from 0.0 to 1.0.

Appendix A

Additional Resources and Legal Notices

Finding Additional Documentation

Documentation Portal

The AMD Adaptive Computing Documentation Portal is an online tool that provides robust search and navigation for documentation using your web browser. To access the Documentation Portal, go to https://docs.xilinx.com.

Documentation Navigator

Documentation Navigator (DocNav) is an installed tool that provides access to AMD Adaptive Computing documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the AMD Vivado™ IDE, select **Help → Documentation and Tutorials**.
- On Windows, click the **Start** button and select **Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

***Note*:** For more information on DocNav, refer to the *Documentation Navigator User Guide* (UG968).

Design Hubs

AMD Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- Go to the Design Hubs webpage.

Support Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Support.

References

These documents provide supplemental material useful with this guide:

1. *Vitis AI User Guide* (UG1414)
2. *Vitis AI Optimizer User Guide* (UG1333)

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
06/29/2023 Version 3.5	
Chapter 1: Introduction	Added Vitis AI Library v3.5 Release Notes section.
Chapter 2: Installation	Updated the entire chapter.
Chapter 3: Libraries and Samples	Added three new libraries including YOLOv7 Detection, YOLOv8 Detection and 2DUnet.
Chapter 4: Programming Examples	Updated Developing with the ONNX Runtime chapter.
Chapter 7: Performance	Updated the performance data for all the platforms.
Chapter 8: API Reference	Updated the API References.
01/12/2023 Version 3.0	
Chapter 1: Introduction	Added Vitis AI 3.0 release notes.
Chapter 2: Installation	Update the entire chapter.
Chapter 3: Libraries and Samples	Added six new libraries including Monodepth2, BEVDet Detection, cFlownet, and YOLOv6 Detection.
Chapter 4: Programming Examples	Added Developing with the ONNX Runtime Engine API.
Chapter 7: Performance	Updated the performance data for all the platforms.
Chapter 8: API Reference	Updated the API References.
06/15/2022 Version 2.5	
Chapter 1: Introduction	Added Vitis AI Library 2.5 Release Notes section.
Chapter 2: Installation	Update the entire chapter.
Chapter 3: Libraries and Samples	Added five new libraries including OCR, Textmountain Detection, Vehicle Classification, OFA_YOLO Detection, EfficientDet_D2 and Movenet Detection.

Section	Revision Summary
Chapter 4: Programming Examples	Added Developing with Vitis AI API_1 Updated the Implementing and Registering Custom Operators section. Updated the Using the xdputil Tool section.
Chapter 5: Application Demos	Update the whole chapter
Chapter 7: Performance	Updated the performance data for all the platforms.
Chapter 8: API Reference	Updated the API References.
01/20/2022 Version 2.0	
Chapter 1: Introduction	Added Vitis AI Library 2.0 Release Notes section. Updated the block diagram in the Overview section.
Chapter 2: Installation	Update the whole chapter.
Chapter 3: Libraries and Samples	Added six new libraries including YOLOX Detection, Polyp Segmentation, UltraFast Road Line Detection, CLOCs, SOLO and FairMot. Added Model Accuracy Test.
Chapter 4: Programming Examples	Added the Implementing and Registering Custom Operators section. Updated the Using the xdputil Tool section.
Chapter 7: Performance	Updated the performance data for all the platforms.
Chapter 8: API Reference	Updated the API References.
07/22/2021 Version 1.4	
Chapter 1: Introduction	Added Vitis AI Library 1.4 Release Notes section. Updated the block diagram in the Overview section.
Chapter 2: Installation	Updated the installation instructions.
Chapter 3: Libraries and Samples	Added eight new libraries including PointPainting, PointPillars_nuscenes, MultiTask V3, and SA-Gate Segmentation.
Chapter 4: Programming Examples	Added the Developing with Vitis AI API_3 (Graph Runner) section. Added the Using the xdputil Tool section.
Chapter 7: Performance	Updated the performance data for all the platforms.
Chapter 8: API Reference	Updated the API References.
02/03/2021 Version 1.3	
Entire document	Updated links. Updated the performance data in the Chapter 7: Performance chapter.
12/17/2020 Version 1.3	
Chapter 1: Introduction	Added the AMD Vitis™ AI Library 1.3 Release Notes. Updated the block diagram in the Overview section
Chapter 2: Installation	Update the whole chapter.
Chapter 3: Libraries and Samples	Added eight new libraries including Retinaface, Face Quality, Hourglass Pose Detection, and PointPillars.
Chapter 7: Performance	Updated the performance data for all the platform.
Chapter 8: API Reference	Updated the API References.
07/21/2020 Version 1.2	
Entire document	Minor changes

Section	Revision Summary
07/13/2020 Version 1.2	
Chapter 1: Introduction	Added Vitis AI Library 1.2 Release Notes. Updated the block diagram in the Overview section
Chapter 2: Installation	Update the whole chapter.
Chapter 3: Libraries and Samples	Added manipulation methods for multiple elf models. Added Face Recognition, Plate Detection, Plate Recognition, and Medical Segmentation.
Chapter 6: Programming APIs	Updated this chapter and all the APIs are introduced as appendices in this document.
Chapter 7: Performance	Added the performance data of U280 and U50LV Updated the performance data of U50, ZCU102, and ZCU104
03/23/2020 Version 1.1	
Chapter 1: Introduction	Added Vitis AI Library 1.1 Release Notes.
Chapter 2: Installation	Added content for data center operation. Updated the Setting Up the Host section.
Chapter 7: Performance	Added the performance data of U50
04/29/2019 Version 1.0	
Initial release.	N/A

Please Read: Important Legal Notices

Copyright