DXNN Runtime (DX-RT) User Manual

v2.9.5

DEEPX.ai

Table of contents

1. DXNN Runtime Overview 4
   1.1 DEEPX SDK Architecture 4
   1.2 Inference Flow of DX-RT 5
2. Installation on Linux 8
   2.1 System Requirements 8
   2.2 Build Environment Setup 9
   2.3 Source File Structure 10
   2.4 Framework Build on Linux 11
   2.5 Linux Device Driver Installation 14
   2.6 Python Package Installation 20
   2.7 Service Registration 20
   2.8 Sanity Check 21
3. Installation on Windows 23
   3.1 System Requirements 23
   3.2 Execute Installer 23
   3.3 File Structure 24
   3.4 Running Examples 25
   3.5 Python Package Installation 25
4. Model Inference 27
   4.1 Model File Format 27
   4.2 Inference Workflow 27
   4.3 Multiple Device Inference 30
   4.4 Data Format of Device Tensor 30
   4.5 Profile Application 31
   4.6 How To Create an Application Using DX-RT 33
   4.7 (Optional) Improving CPU Task Throughput with DXRT_DYNAMIC_CPU_THREAD 34
5. Command Line Interface 36
   5.1 Parse Model 36
   5.2 Run Model 36
   5.3 DX-RT CLI Tool (Firmware Interface) 37

6. Tutorials 39
6.1 C++ Tutorials 39
6.2 Python Tutorials 62
7. API Reference 72
7.1 C++ API Reference 72
7.2 Python API Reference 82
8. Change Log 89
8.1 v2.9.5 (May 2025) 89
8.2 v2.8.2 (April 2025) 90

1. DXNN Runtime Overview

This chapter provides an overview of the DEEPX SDK architecture and explains each core component and its role in the AI development workflow.

1.1 DEEPX SDK Architecture

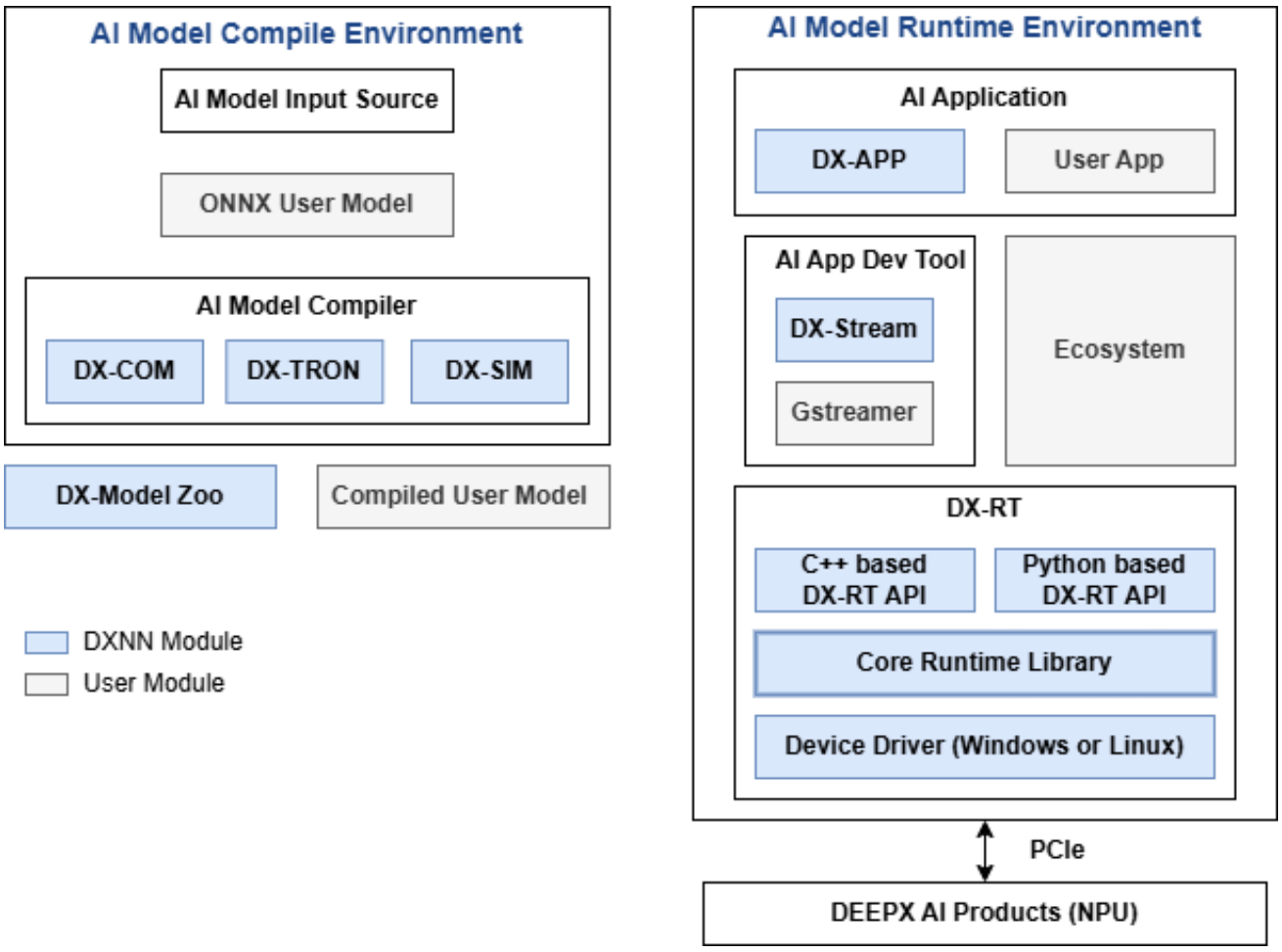

Figure. DEEPX SDK Architecture

DEEPX SDK is an all-in-one software development platform that streamlines the process of compiling, optimizing, simulating, and deploying AI inference applications on DEEPX NPUs (Neural Processing Units). It provides a complete toolchain, from AI model creation to runtime deployment, optimized for edge and embedded systems, enabling developers to build high-performance AI applications with minimal effort.

DX-COM is the compiler in the DEEPX SDK that converts a pre-trained ONNX model and its associated configuration JSON file into a hardware-optimized .dxnn binary for DEEPX NPUs. The ONNX file contains the model structure and weights, while the JSON file defines pre/post-processing settings and compilation parameters. DX-COM provides a fully compiled .dxnn file, optimized for low-latency and high-efficient inference on DEEPX NPU.

DX-RT is the runtime software responsible for executing ,dxnn models on DEEPX NPU hardware. DX-RT directly interacts with the DEEPX NPU through firmware and device drivers, using PCIe interface for high-speed data transfer between the host and the NPU, and provides C/C++ and Python APIs for application-level inference control. DX-RT offers a complete runtime environment, including model loading, I/O buffer management, inference execution, and real-time hardware monitoring.

DX ModelZoo is a curated collection of pre-trained neural network models optimized for DEEPX NPU, designed to simplify AI development for DEEPX users. It includes pre-trained ONNX models, configuration JSON files, and pre-compiled DXNN binaries, allowing developers to rapidly test and deploy applications. DX ModelZoo also provides benchmark tools for comparing the performance of quantized INT8 models on DEEPX NPU with full-precision FP32 models on CPU or GPU.

DX-STREAM is a custom GStreamer plugin that enables real-time streaming data integration into AI inference applications on DEEPX NPU. It provides a modular pipeline framework with configurable elements for preprocessing, inference, and postprocessing, tailored to vision AI work. DX-Stream allows developers to build flexible, high-performance applications for use cases such as video analytics, smart cameras, and edge AI systems.

DX-APP is a sample application that demonstrates how to run compiled models on actual DEEPX NPU using DX-RT. It includes ready-to-use code for common vision tasks such as object detection, face recognition, and image classification. DX-APP helps developers quickly set up the runtime environment and serves as a template for building and customizing their own AI applications.

1.2 Inference Flow of DX-RT

Here is the inference flow of **DX-RT**.

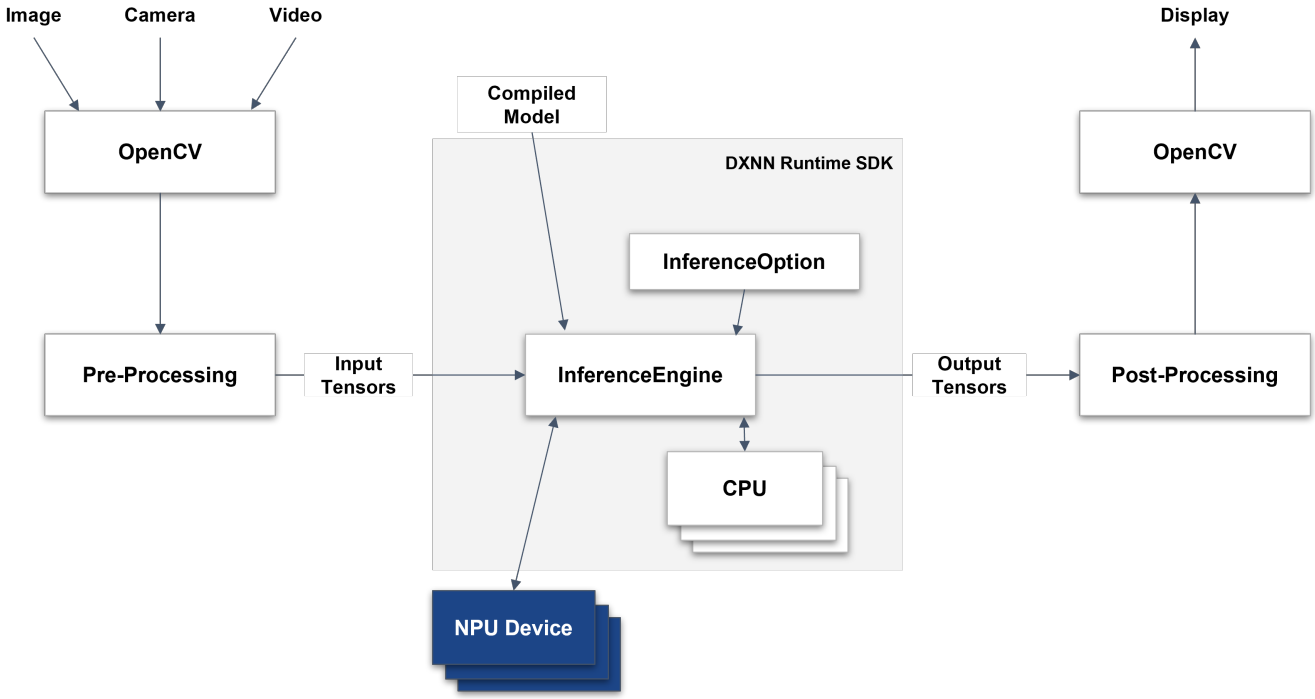

Figure. Inference Flow of DXNN Runtime

This figure illustrates the inference workflow of the DXNN Runtime SDK, which integrates OpenCV-based input/output handling with efficient NPU-accelerated model execution.

Input & Pre-Processing

Input data—such as images, camera feeds, or video—is captured using OpenCV. The data is then passed through a Pre-Processing module, which transforms it into \input tensors suitable for the model.

Feeding Input to the Inference Engine

The pre-processed input tensors are fed into the InferenceEngine along with the compiled model (.dxnn). Before execution, you **must** configure the InferenceOption, which specifies the target device and available resources.

Model Execution

The InferenceEngine is the core component of the DXNN Runtime SDK. It:
- Initializes and controls the NPU device - Manages memory for input/output tensors - Schedules inference tasks across NPU and CPU, optimizing their interaction for real-time performance

Post-Processing & Display

The output tensors are processed to a Post-Processing stage, typically involving OpenCV for decoding, formatting, or visualization. Finally, the results are displayed or forwarded to the next processing step.

2. Installation on Linux

This chapter describes the system requirements, source file structure, and the installation instructions for setting up **DX-RT** on a Linux-based host system.

After you check the system requirements, follow these instructions.

- System Requirement Check
- Build Environment Setup
- Source File Structure Check
- Framework Build
- Linux Device Driver Installation
- Python Package Installation
- Service Registration

2.1 System Requirements

This section describes the hardware and software requirements for running **DX-RT** on Linux.

Hardware and Software Requirements

- **CPU:** aarch64, x86_64, riscv64
- **RAM:** 8GB RAM (16GB RAM or higher is recommended)
- **Storage:** 4GB or higher available disk space
- **OS:** Ubuntu 20.04 / 22.04 / 24.04 (x64 / aarch64 / riscv64)
- **Hardware:** The system **must** support connection to an **M1 M.2** module with the **M.2 interface** on the host PC.

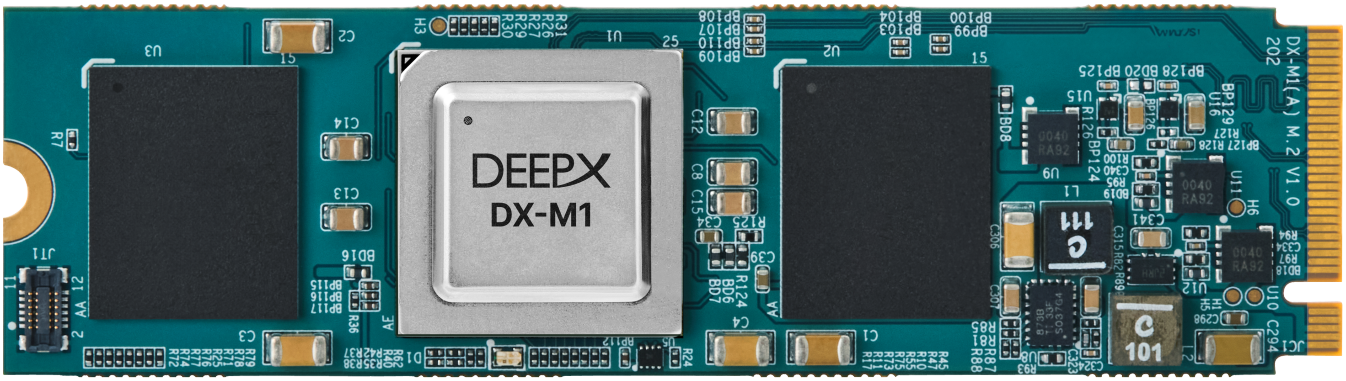

Figure. DX-M1 M.2 Module

2.2 Build Environment Setup

DEEPX provides an installation shell script to set up the **DX-RT** build environment. You can install the entire toolchain installation or perform a partial installation as necessary.

DX-RT supports the Target OS of **Ubuntu 18.04, Ubuntu 20.04, Ubuntu 22.04,** and **Ubuntu 24.04**.

Installation of DX-RT
To install the full **DX-RT** toolchain, use the following commands.

```
$ cd dx_rt
$ ./install.sh --all
```

Here are the available `install.sh` options.

```
./install.sh [ options ]
  --help            Shows help message
  --arch [x86_64, aarch64]
                    Sets target CPU architecture
  --dep             Installs build dependencies : cmake, gcc, ninja, etc..
  --onnxruntime     (Optional) Installs onnxruntime library
  --all             Installs architecture + dependency + onnxruntime library
```

Installation with ONNX Runtime
Use the ONNX Runtime option if you need to offload certain neural network (NN) operations to the CPU that are **not** supported by the NPU.

We recommend using ONNX Runtime linux x64 version more than v1.20.1.

```
https://github.com/microsoft/onnxruntime/releases/download/v1.20.1/onnxruntime-linux-
x64-1.20.1.tgz
$ sudo tar -xvzf onnxruntime-linux-x64-1.20.1.tgz -C /usr/local --strip-components=1
$ sudo ldconfig
```

To install the ONNX Runtime library, run the following command.

```
./install.sh --onnxruntime
```

Installation for a Specific CPU Architecture
The **DX-RT** targets the **x86_64** architecture. If you're compiling for another architecture (e.g., **aarch64**), specify it using the `--arch` option.

```
./install.sh --arch aarch64 --onnxruntime
```

2.3 Source File Structure

The **DX-RT** source directory is organized as follows. You can install the full toolchain using the `install.sh`, and the build and library using `build.sh`.

```
├── assets
├── bin
├── cli
├── build.sh
├── build_x86_64
├── build_aarch64
├── cmake
├── docs
├── examples
├── extern
├── install.sh
├── lib
├── python_package
├── sample
```

```
├── service
└── tool
```

- `assets` : Images for documentation
- `bin` : Compiled binary executables
- `cli` : Command-line application source code
- `build.sh` : Shell script for building the framework
- `build_arch` : Build outputs for aarch64 architecture
- `cmake` : CMake scripts for build configuration
- `docs` : Markdown documents
- `examples` : Inference example files
- `extern` : Third-party libraries
- `install.sh` : Shell script for toolchain installation
- `lib` : DX-RT library sources
- `python_package` : Python modules for DX-RT
- `sample` : Sample input files for demo apps
- `service` : Service unit files for runtime management
- `tool` : Profiler result visualization tools

2.4 Framework Build on Linux

After compiling the **DX-RT** environment setup, you can build the framework using the provided `build.sh` shell script.

2.4.1 Framework Source Build

DEEPX supports the default target CPU architecture as **x86_64, aarch64**.

The build script also supports options for build cleaning, specifying build type, and installing libraries to the system paths.

Build Instructions

To build the **DX-RT** framework, run the following command.

```
$ cd dx_rt
$ ./build.sh
```

Here are the `build.sh` options and their descriptions.

```
./build.sh [ options ]
    --help       Shows help message
    --clean      Cleans previous build artifacts
    --verbose    Shows full build commands during execution
    --type [Release, Debug, RelWithDebInfo]
                 Specifies the cmake build type
    --arch [x86_64, aarch64]
                 Sets target CPU architecture
    --install <path>
                 Sets the installation path for built libraries
    --uninstall  Removes installed DX-RT files
    --clang      Compiles using clang
```

Example. Build with `clean` Option

To clean existing build files before rebuilding.

```
$ ./build.sh --clean
```

Example. Build with `library` Option

To install build library files to `/usr/local`.

```
# default path is /usr/local
$ ./build.sh --install /usr/local
```

2.4.2 Options for Build Target

DX-RT supports configuration build targets, allowing you to enable or disable option features such as ONNX Runtime, Python API, multi-process service support, and shared library builds.

You can configure these options by editing the following file: `cmake/dxrt.cfg.cmake`

Here are the available options for building targets.

```
option(USE_ORT "Use ONNX Runtime" OFF)
option(USE_PYTHON "Use Python" OFF)
```

```
option(USE_SERVICE "Use Service" OFF)
option(USE_SHARED_DXRT_LIB "Build for DX-RT Shared Library" ON)
```

- `USE_ORT` : Enables ONNX Runtime for NN (neural network) operations that NPU does **not** support
- `USE_PYTHON` : Enables Python API support
- `USE_SERVICE` : Enables service for multi-process support
- `USE_SHARED_DXRT_LIB` : Builds **DX-RT** as shared library (default: ON)

2.4.3 Build Guide for Cross-compile

Setup Files for Cross-compile DEEPX supports cross-compilation for the following default target CPU Architecture: **x86_64, aarch64**.
DEEPX supports the default target CPU architecture as **x86_64**.

Toolchain Configuration
To cross-compile for a specific target, configure the toolchain file.
`cmake/toolchain.<CMAKE_SYSTEM_PROCESSOR>.cmake`

Example
To cross-compile files for **aarch64**.

```
SET(CMAKE_C_COMPILER /usr/bin/aarch64-linux-gnu-gcc )
SET(CMAKE_CXX_COMPILER /usr/bin/aarch64-linux-gnu-g++ )
SET(CMAKE_LINKER /usr/bin/aarch64-linux-gnu-ld )
SET(CMAKE_NM /usr/bin/aarch64-linux-gnu-nm )
SET(CMAKE_OBJCOPY /usr/bin/aarch64-linux-gnu-objcopy )
SET(CMAKE_OBJDUMP /usr/bin/aarch64-linux-gnu-objdump )
SET(CMAKE_RANLIB /usr/bin/aarch64-linux-gnu-ranlib )
```

Non Cross-compile Case (Build on Host)
To build and install **DX-RT** on the host system, run the following command.

```
./build.sh --install /usr/local
```

Recommended install path: `/usr/local` (commonly included in OS search paths)

Cross-compile Case (Build for Target Architecture)
Cross-compile for a specific architecture, run the following command.

```
./build.sh --arch <target_cpu>
```

Here are the `build.sh` options and their descriptions.

```
./build.sh [ options ]
   --help   Shows help message
   --clean  Cleans previous build artifacts
   --verbose    Shows full build commands during execution
   --type   Specifies the cmake build type : [ Release, Debug, RelWithDebInfo ]
   --arch   Sets target CPU architecture : [ x86_64, aarch64, riscv64 ]
   --install    Installs build libraries
   --uninstall  Removes installed DX-RT files
```

Here are the examples of cross-compile cases.

```
./build.sh --arch aarch64
./build.sh --arch x86_64
```

Output Directory

After a successful build, output binaries is located under `<build directory> /bin/`

```
<build directory>/bin/
 ├── dxrtd
 ├── dxrt-cli
 ├── parse_model
 ├── run_model
 └── examples
```

2.5 Linux Device Driver Installation

After building the **DX-RT** framework, you can install the Linux device driver for **M1 AI Accelerator** (NPU).

2.5.1 Prerequisites

Before installing the Linux device driver, you should check that the accelerator device is properly recognized by the system.

To check PCIe device recognition, run the following command.

```
$ lspci -vn | grep 1ff4
0b:00.0 1200: 1ff4:0000
```

Note. If there is no output, the PCIe link is **not** properly connected. Please check the physical connection and system BIOS settings.

Optional. Display the DEEPX name in `lspci`.
If you want to display the DEEPX name in `lspci`, you can modify the PCI DB. (Only for Ubuntu)

To display the DeepX device name, run the following command.

```
$ sudo update-pciids
$ lspci
...
0b:00.0 Processing accelerators: DEEPX Co., Ltd. DX_M1
```

2.5.2 Linux Device Driver Structure

The **DX-RT** Linux device driver source is structured to support flexible builds across devices, architectures, and modules. The directory layout is as follows.

```
- .gitmodules

- [modules]
    |
    - device.mk
    - kbuild
    - Makefile
    - build.sh
    - [rt]
        - Kbuild
    - [pci_deepx] : submodule
        - Kbuild

- [utils] : submodule
```

- `device.mk` : Device configuration file
- `kbuild` : Top-level build rules
- `Makefile` : Build entry point
- `build.sh` : Build automation script
- `rt` : Runtime driver source (`dxrt_driver.ko`)
- `pci_deepx` : PCIe DMA driver (`submodule, dx_dma.ko`)
- `utils` : Supporting utilities (`submodule`)

Here are the descriptions of the key components.

`device.mk`

Defines supported device configuration.

To build for a specific device, run the following command.

```
$ make DEVICE=[device]
```

For example, in the case of a device like **M1**, you should select a submodule, such as PCIe, that has a dependency on **M1**.

```
$ make DEVICE=m1 PCIE=[deepx]
```

`kbuild`

Linux kernel build configuration file for each module directory. It instructs the kernel build system on how to compile driver modules.

`build.sh`

Shell script to streamline the build process. It runs the Makefile with common options.

Here are the options for `build.sh`.

```
Usage:
Usage:
   build.sh <options>

options:
   -d, --device   [device]      select target device: m1
   -m, --module   [module]      select PCIe module: deepx
   -k, --kernel   [kernel dir]  'KERNEL_DIR=[kernel dir]', The directory where the
                                kernel source is located
                                default: /lib/modules/6.5.0-18-generic/build)
   -a, --arch     [arch]        set 'ARCH=[arch]' Target CPU architecture for
                                cross-compilation, default: x86_64
   -t, --compiler [cross tool]  'CROSS_COMPILE=[cross tool]' cross compiler binary,
                                e.g aarch64-linux-gnu-
   -i, --install  [install dir] 'INSTALL_MOD_PATH=[install dir]', module install
                                directory install to:
                                [install dir]/lib/modules/[KERNELRELEASE]/extra/
   -c, --command  [command]     clean | install | uninstall
                                - uninstall: Remove the module files installed
                                on the host PC.
   -j, --jops     [jobs]        set build jobs
   -f, --debug    [debug]       set debug feature [debugfs | log | all]
   -v, --verbose                build verbose (V=1)
   -h, --help                   show this help
```

The build process generates the following kernel modules.

- `modules/rt` -> `dxrt_driver.ko`
  : a core runtime driver for **M1 NPU** devices. This is responsible for system-level communication, memory control, and device command execution.
- `modules/pci_deepx` -> `dx_dma.ko`
  : PCIe DMA (Direct Memory Access) kernel module for high-speed data transfer between host and the **M1** device. This enables efficient data movement with minimal CPU overhead, ideal for real-time and data intensive AI workloads.

2.5.3 Linux Device Driver Build

After completing the environment setup of the DXNN Linux Device Driver, you can build the kernel modules using either the make(Makefile) or `build.sh` script. Both methods are supported by DEEPX.

Option 1. Build Using `Makefile`

`build`

```
e.g $ cd modules
e.g $ make DEVICE=m1 PCIE=deepx
```

`clean`

```
e.g $ cd modules
e.g $ make DEVICE=m1 PCIE=deepx clean
```

`install`
Installs the driver to: `/lib/modules/$(KERNELRELEASE)/extra/`

```
e.g $ cd modules
e.g $ make DEVICE=m1 PCIE=deepx install
```

Option 2. Build Using `build.sh`

Use this method if your system supports self-compiling kernel modules (`.ko` files).

`build`

```
e.g $ ./build.sh -d m1 -m deepx
(Default device: m1, PCI3 module: deepx)
```

`clean`

```
e.g $ ./build.sh -c clean
```

`install`
Installs the driver to: `/lib/modules/$(KERNELRELEASE)/extra/`

```
e.g $ sudo ./build.sh -c install
```

2.5.4 Auto-Loading Modules at Boot Time

DEEPX allows kernel modules to be automatically loaded at system boot, either through manual setup or using the `build.sh` script.

Manual Installation Method

Step 1. Install Kernel Modules
Installs modules to: `/lib/modules/$(KERNELRELEASE)/extra/`

```
make DEVICE=m1 PCIE=deepx install
```

Step 2. Update Module Dependencies
Updates: `/lib/modules/$(KERNELRELEASE)/modules.dep`

```
$ sudo depmod -A
```

Step 3. Add Module Confiduration
Copy the preconfigured module config file.

```
$ sudo cp modules/dx_dma.conf /etc/modprobe.d/
```

This ensures the modules (`dx_dma`) are auto-loaded on boot.

Step 4. Test with modprobe
To verify the correct installation.

```
$ sudo modprobe dx_dma
$ lsmod
  dxrt_driver            40960  0
  dx_dma                176128  1 dxrt_driver
```

Automated Installation Using `build.sh`

The `build.sh` script automates installation and setup, including dependency updates and modprobe configuration.

Run the following command

```
$ sudo ./build.sh -d m1 -m deepx -c install
- DEVICE        : m1
- PCIE          : deepx
- MODULE CONF   : /.../rt_npu_linux_driver/modules/dx_dma.conf
- ARCH (HOST)   : x86_64
- KERNEL        : /lib/modules/5.15.0-102-generic/build
- INSTALL       : /lib/modules/5.15.0-102-generic/extra/

 *** Build : install ***
$ make DEVICE=m1 PCIE=deepx install

make -C /lib/modules/5.15.0-102-generic/build M=/home/jhk/deepx/dxrt/module/
rt_npu_linux_driver/modules  modules_install
 ....
 - SUCCESS

 *** Update : /lib/modules/5.15.0-102-generic/modules.dep ***
 $ depmod -A
 $ cp /home/jhk/deepx/dxrt/module/rt_npu_linux_driver/modules/dx_dma.conf /etc/
modprobe.d/
```

Uninstalling Modules

To completely remove the installed modules and configs.

```
$ ./build.sh -d m1 -m deepx -c uninstall
- DEVICE        : m1
- PCIE          : deepx
- MODULE CONF   : /.../rt_npu_linux_driver/modules/dx_dma.conf
- ARCH (HOST)   : x86_64
- KERNEL        : /lib/modules/5.15.0-102-generic/build
- INSTALL       : /lib/modules/5.15.0-102-generic/extra/

 *** Remove : /lib/modules/5.15.0-102-generic/extra ***
 $ rm -rf /lib/modules/5.15.0-102-generic/extra/pci_deepx
 $ rm -rf /lib/modules/5.15.0-102-generic/extra/rt

 *** Remove : /etc/modprobe.d ***
 $ rm /etc/modprobe.d/dx_dma.conf

 *** Update : /lib/modules/5.15.0-102-generic/modules.dep ***
 $ depmod
```

2.6 Python Package Installation

DEEPX provides a Python package for **DX-RT**, available under the module name dx-engine. It supports Python 3.8 or later and allows you to interface with **DX-RT** in Python-based applications.

Installation Steps

1. Navigate to the python_package directory.

```
$ cd python_package
```

2. Install the package

```
$ pip install .
```

3. Verify the installation

```
$ pip list | grep dx
dx-engine          1.0.0
```

For details on using DX-RT with Python, refer to **Section 6.2 Python in 6. Programming Guide**.

2.7 Service Registration

DX-RT supports multi-process operation through the background service (`dxrtd daemon`). To enable the multi-process feature, you **must** build the Runtime with Service support and the service must be registered in the system below.

Note.

- **DX-RT must** be built with `USE_SERVICE=ON` . (default setting)
- **DX-RT must** be registered and managed as a system service using `systemd` .

Registering and Running the DX-RT Service

1. Modify the service unit file.

Ensure the ExecStart path is correctly configured.

```
$ vi ./service/dxrt.service
```

2. Copy the service file to the system folder.

```
$ sudo cp ./service/dxrt.service /etc/systemd/system
```

3. Start the service.

```
$ sudo systemctl start dxrt.service
```

Service Management Commands

```
$ sudo systemctl stop dxrt.service          # Stop the service
$ sudo systemctl status dxrt.service        # Check service status
$ sudo systemctl restart dxrt.service       # Restart the service
$ sudo systemctl enable dxrt.service        # Enable on boot
$ sudo systemctl disable dxrt.service       # Disable on boot
$ sudo journalctl -u dxrt.service           # View service logs
```

2.8 Sanity Check

The Sanity Check is a script used to quickly verify that a driver has been installed correctly and that the device is recognized properly.

```
$ sudo ./SanityCheck.sh
================================================================================
==== Sanity Check Date : DATE ====
Log file location : .../dx_rt/dx_report/sanity/result/sanity_check_result_[date]_[hh/mm/
ss].log

==== PCI Link-up Check ====
[OK] Vendor ID 1ff4 is present in the PCI devices.(num=2)
==== Device File Check ====
[OK] /dev/dxrt0 exists.
[OK] /dev/dxrt0 is a character device.
[OK] /dev/dxrt0 has correct permissions (0666).
[OK] /dev/dxrt1 exists.
[OK] /dev/dxrt1 is a character device.
[OK] /dev/dxrt1 has correct permissions (0666).
==== Kernel Module Check ====
[OK] dxrt_driver module is loaded.
[OK] dx_dma module is loaded.
[OK] PCIe 02:00.0 driver probe is success.
[OK] PCIe 07:00.0 driver probe is success.

================================================================================
```

```
** Sanity check PASSED!
================================================================================
```

3. Installation on Windows

This chapter describes the instructions for installing and using **DX-RT** on a Windows system.

3.1 System Requirements

This section describes the hardware and software requirements for running **DX-RT** on Windows.

- **RAM:** 8GB RAM (16GB RAM or higher is recommended)
- **Storage:** 4GB or higher available disk space
- **OS:** Windows 10 / 11
- **Python:** Version 3.8 or higher (for Python module support)
- **Compiler:** Visual Studio 2022 (required for building C++ examples)
- **Hardware:** The system **must** support connection to an **M1 M.2** module with the **M.2 interface** on the host PC.

The current version **only** supports **Single-process** and does **not** support Multi-process.

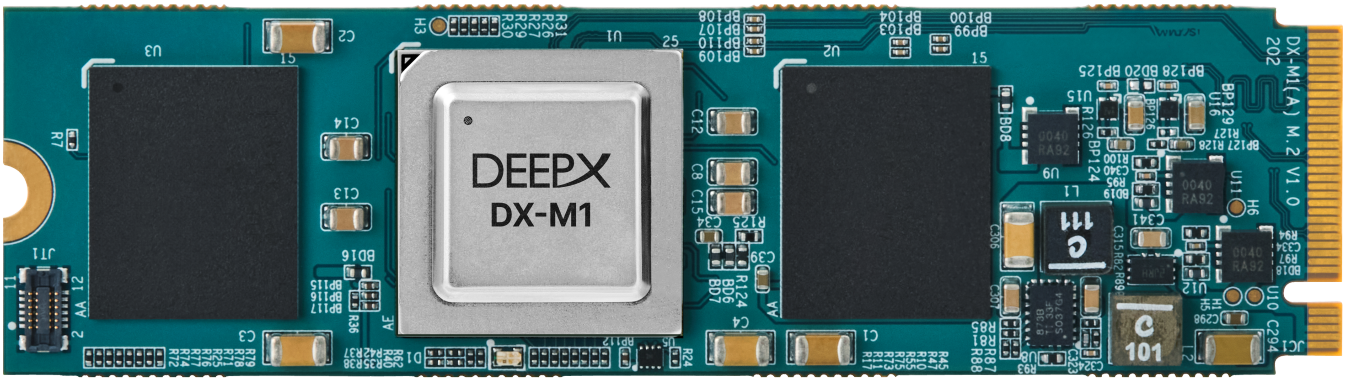

Figure. DX-M1 M.2 Module

3.2 Execute Installer

DEEPX provides the Windows installer executable file for **DX-RT**.

- `DXNN_Runtime_v[version]_windows_[architecture].exe`

Here is an example of the execution file.

- `DXNN_Runtime_v2.X.X_windows_amd64.exe`

Default Directory Path

- 'C:/DevTools/DXNN/dxrt_v[version]'

Once you install the exe file, the driver will be installed automatically. So you can verify the installation via Device Manager under DEEPX_DEVICE.

Note.
If **Visual Studio 2022** is **not** installed, you may be prompted to install the **Microsoft Visual C++ Redistributable** (`VC_redist.x64.exe`) using administrator permissions.

3.3 File Structure

```
├── bin
├── docs
├── drivers
├── examples
├── firmware
├── include
├── lib
└── python_package
```

- `bin` : Compiled binary executables
- `docs` : Markdown documents
- `examples` : Inference example files
- `include` : Header files for DX-RT libraries
- `lib` : Pre-built DX-RT libraries
- `python_package` : Python modules for DX-RT
- `sample` : Sample input files for demo apps
- `service` : Service unit files for runtime management
- `tool` : Profiler result visualization tools

3.4 Running Examples

DX-RT includes sample programs in both C++ and Python.

3.4.1 Building C++ Examples

Visual Studio 2022 should be installed on your PC.

1. Open the solution file in the following location.

`examples\<example-name>\msvc\<example-name>.sln`

2. In Visual Studio, Click Rebuild Solution.

Once the build is complete, an `x64` directory is generated in the same location as the solution file. The executable file of the sample includes the Debug or Release sub-folder.

3.4.2 Running C++ Examples

1. Run the executable file of the sample at the following location.

`examples\<example-name>\msvc\x64\[Debug|Release]\<example-name>.exe`

Example

```
C:\...> cd .\examples\run_async_model\msvc\x64\Release
C:\...\examples\run_async_model\msvc\x64\Release> .\run_async_model.exe model.dxnn 100
```

3.5 Python Package Installation

DEEPX provides a Python module named `dx_engine` for Python 3.8 or later.

1. Build and Install the Package
Navigate to the Python package directory and install the module.

```
C:\...\dxrt_v2.8.2> cd python_package/
C:\...\dxrt_v2.8.2\python_package> pip install .
```

2. Verify the Installation

Open a Python shell and check the installed version.

```
C:\...> python
...
>>> from dx_engine import version
>>> print(version.__version__)
1.0.1
```

Examples

```
cd examples/python
C:\...\examples\python> python run_async_model.py ...model.dxnn 10
```

4. Model Inference

4.1 Model File Format

The original ONNX model is converted by **DX-COM** into the following structure.

```
Model dir.
  └── graph.dxnn
```

- `graph.dxnn`

  A unified DEEPX artifact that contains NPU command data, model metadata, model parameters.

This file is used directly for inference on DEEPX hardware or simulator.

4.2 Inference Workflow

Here the inference workflow using the DXNN Runtime as follows.

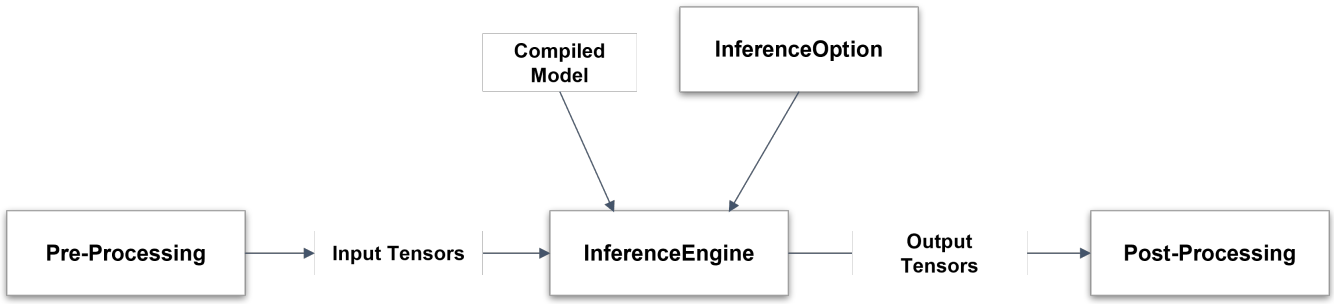

Figure. Inference Workflow

- **1**. Compiled Model and optional InferenceOption are provided to initialize the InferenceEngine.
- **2**. Pre-processed Input Tensors are passed to the InferenceEngine for inference.
- **3**. The InferenceEngine produces Output Tensors as a result of the inference.
- **4**. These outputs are then passed to the Post-Processing stage for interpretation or further action.

4.2.1 Prepare the Model

Choose one of the following options.

- Use a pre-built model from **DX ModelZoo**
- Compile an ONNX model into the **DX-RT** format using **DX-COM** (Refer to the **DX-COM User Guide** for details.)

4.2.2 Configure Inference Options

Create a `dxrt::InferenceOption` object to configure runtime settings for the inference engine.

Note. This option is temporarily unsupported in the current version, and will be available in the next release.

4.2.3 Load the Model into the Inference Engine

Create a `dxrt::InferenceEngine` instance using the path to the compiled model directory. Hardware resources are automatically initialized during this step.

If `dxrt::InferenceEngine` is **not** provided, a default option is applied.

```
auto ie = dxrt::InferenceEngine("yolov5s.dxnn");
auto ie = dxrt::InferenceEngine("yolov5s.dxnn", &option);
```

4.2.4 Connect Input Tensors

Prepare input buffers for inference.

The following example shows how to initialize the buffer with the appropriate size.

```
std::vector<uint8_t> inputBuf(ie.GetInputSize(), 0);
```

Refer to **DX-APP User Guide** for practical examples on connecting inference engines to image sources such as cameras or video, along with the preprocessing routines.

4.2.5 Inference

DX-RT provides both synchronous and asynchronous execution modes for flexible inference handling.

1. Run - Synchronous Execution

Use the `dxrt::InferenceEngine::Run()` method for blocking, single-core inference.

```
auto outputs = ie.Run(inputBuf.data());
```

- This method processes input and output on the same thread.
- This method is suitable for simple and sequential workloads.

2. Run - Asynchronous Execution

a. With `Wait()`

Use `RunAsync()` to perform the inference in non-blocking mode, and retrieve results later with `Wait()`.

```
auto jobId = ie.RunAsync(inputBuf.data());
auto outputs = ie.Wait(jobId);
```

- This method is ideal for parallel workloads where inference can run in the background.
- This method is continuously executed while waiting for the result.

b. With Callback

Use a callback function to handle output as soon as inference completes.

```
std::function<int(vector<shared_ptr<dxrt::Tensor>>, void*)> postProcCallBack = \
    [&](vector<shared_ptr<dxrt::Tensor>> outputs, void *arg)
    {
        /* Process output tensors here */
        ... ...
        return 0;
    };
ie.RegisterCallback(postProcCallBack)
```

- The callback is triggered by a background thread after inference.
- You can pass a custom argument to track input/output pairs.

Note. Output data is **only** valid within the callback scope.

4.2.6 Process Output Tensors

Once inference is complete, the output tensors are processed using Tensor APIs and custom post-processing logic. You can find the templates and example code in **DX-APP** to help you implement post-

process smoothly.
As noted earlier, using callbacks allows for more efficient and real-time post-processing.

4.3 Multiple Device Inference

This feature is **not** applicable to single-NPU devices. Basically, the inference engine schedules and manages multiple devices in real time.
If the inference option is explicitly set, the inference engine may **only** use specific devices during real-time inference for the model.

4.4 Data Format of Device Tensor

Compiled models use the **NHWC** format by default.

However, the input tensor formats on the device side may vary depending on the hardware's processing type.

Input Tensor Formats

Type	Compiled Model Format	Device Format	Data Size
`Formatter`	`[N, H, W, C]`	`[N, H, W, C]`	8-bit
`IM2COL`	`[N, H, W, C]`	`[N, H, align64(W*C)]`	8-bit

- Formatter Type Example: `[1, 3, 224, 224] (NCHW) -> [1, 224, 224, 3] (NHWC)`
- IM2COL Type Example: `[1, 3, 224, 224] (NCHW) -> [1, 224, 224*3+32] (NH, aligned width x channel)`

Output Tensor Formats

The output tensor format is also aligned with the NHWC format, but with padding applied for alignment.

Type	Compiled Model Format	Device Format
`Aligned NHWC`	`[N, H, W, C]`	`[N, H, W, align64(C)]`

- Output Example: `[1, 40, 52, 36] (NCHW) -> [1, 52, 36, 40+24]` (Channel size is aligned for optimal memory access.)

Post-processing can be performed directly without converting formats.
API to convert from device format to **NCHW/NHWC** format will be supported in the next release.

4.5 Profile Application

4.5.1 Gather Timing Data per Event

You can profile events within your application using the Profiler APIs. Please refer to **Section 8. API reference**.

Here is a basic usage example.

```
auto& profiler = dxrt::Profiler::GetInstance();
profiler.Start("1sec");
sleep(1);
profiler.End("1sec");
```

After the application is finished, `profiler.json` is created in the working directory.

4.5.2 Visualize Profiler Data

You can visualize the profiling results using the following Python script.

```
python3 tool/profiler/plot.py --input profiler.json
```

This generates an image file named `profiler.png`, providing a detailed view of runtime event timing for performance analysis.

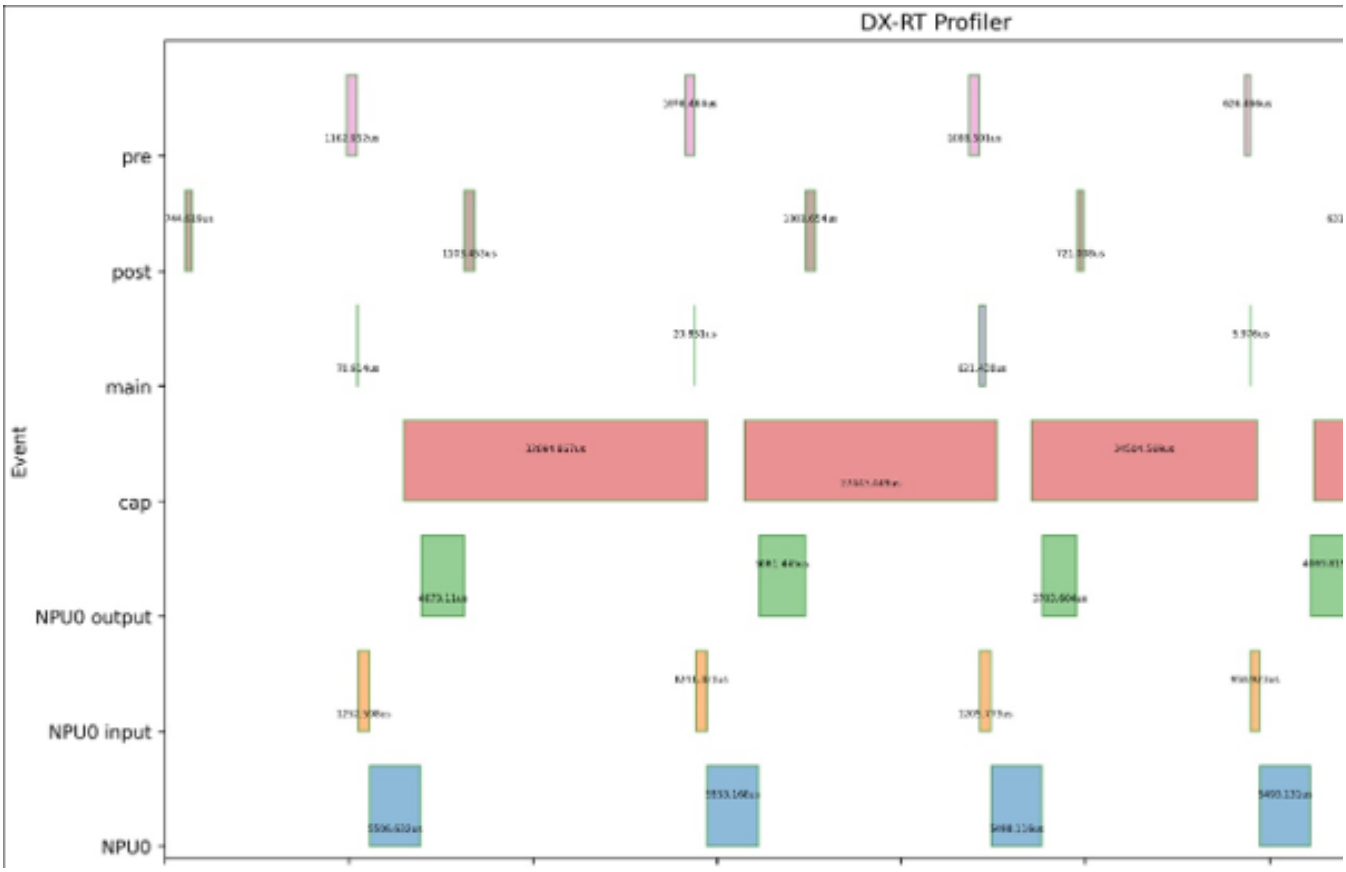

Figure. DX-RT Profiling Report

Script Usage: `tool/profiler/plot.py`

Use this script to draw a timing chart from profiling data generated by **DX-RT**.

```
usage: plot.py [-h] [-i INPUT] [-o OUTPUT] [-s START] [-e END] [-g]
```

Optional Arguments

- `-h, --help` : Show help message and exit
- `-i INPUT, --input INPUT` : Input `.json` file to visualize (e.g., `profiler.json`)
- `-o OUTPUT, --output OUTPUT` : Output image file name to save (e.g., profiler.png)
- `-s START, --start START` : Starting position (normalized, > 0.0) within the time interval [0.0-1.0]
- `-e END, --end END` : End position (normalized, < 1.0) within the time interval [0.0-1.0]
- `-g, --show_gap` : Show time gaps between the start point of each event

Please refer to usage of `tool/profiler/plot.py`.

```
usage: plot.py [-h] [-i INPUT] [-o OUTPUT] [-s START] [-e END] [-g]
```

4.6 How To Create an Application Using DX-RT

This guide provides step-by-step instructions for creating a new CMake project using the **DX-RT** library.

1. Build the DX-RT Library
Before starting, make sure the **DX-RT** library is already built.

Refer to **Chapter 2. Installation on Linus** and **Chapter 3. Installation on Windows** for detailed build instructions.

2. Create a New CMake Project
Create a project directory and an initial `CMakeLists.txt` file.

```
mkdir MyProject
cd MyProject
touch CMakeLists.txt
```

3. “Hello World” with DX-RT API
Create a simple source file (`main.cpp`) that uses a **DX-RT** API.

```
#include "dxrt/dxrt_api.h"
using namespace std;

int main(int argc, char *argv[])
{
 auto& devices = dxrt::CheckDevices();
 cout << "hello, world" << endl;
 return 0;
}
```

4. Modify CMakeLists.txt
Edit the `CMakeLists.txt` file as follows.

```
cmake_minimum_required(VERSION 3.14)
project(app_template)
```

```
set(CMAKE_CXX_STANDARD_REQUIRED "ON")
set(CMAKE_CXX_STANDARD "14")

# Set the DX-RT library installation path (adjust as needed)
set(DXRT_LIB_PATH "/usr/local/lib")

# Locate the DX-RT library
find_library(DXRT_LIBRARY REQUIRED NAMES dxrt_${CMAKE_SYSTEM_PROCESSOR} PATHS $
{DXRT_LIB_PATH})

# Add executable and link libraries
add_executable(HelloWorld main.cpp)
target_link_libraries(HelloWorld PRIVATE ${DXRT_LIBRARY} protobuf)
```

Replace `/usr/local/lib` with the actual path where the **DX-RT** library is installed.

5. Build the Project
Compile your project using the following commands.

```
mkdir build
cd build
cmake ..
make
```

6. Run the Executable
After a successful build, run the generated executable.

```
./HelloWorld
```

You now successfully create and build a CMake project using the **DX-RT** library.

4.7 (Optional) Improving CPU Task Throughput with DXRT_DYNAMIC_CPU_THREAD

The USE_ORT option allows for enabling ONNX Runtime to handle operations that are not supported by the NPU. When this option is active, the model's CPU tasks are executed via ONNX Runtime.

To mitigate potential bottlenecks in these CPU tasks, especially under varying Host CPU conditions, an optional dynamic multi-threading feature is provided. This feature monitors the input queue load to identify CPU task congestion. If a high load is detected, it dynamically increases the number of threads allocated to CPU tasks, thereby improving their throughput. This dynamic CPU threading can be

enabled by setting the DXRT_DYNAMIC_CPU_THREAD=ON environment variable (e.g., export DXRT_DYNAMIC_CPU_THREAD=ON).

Additionally, if the system observes that CPU tasks are experiencing significant load, it will display a message: "To improve FPS, set: 'export DXRT_DYNAMIC_CPU_THREAD=ON'", recommending the activation of this feature for better performance.

Warning: Enabling the DXRT_DYNAMIC_CPU_THREAD=ON option does not always guarantee an FPS increase; its effectiveness can vary depending on the specific workload and system conditions.

5. Command Line Interface

This chapter introduces **DX-RT** command-line tools for model inspection, execution, and device management.

5.1 Parse Model

This tool is used to parse and inspect a compiled model file (`.dxnn`), printing model structure and metadata.

Source: `bin/parse_model.cpp`

Usage

```
parse_model -m <model_dir>
```

Option

- `-m, --model` : Path to the compiled model file (`.dxnn`)
- `-h, --help` : Show help message

Example

```
$./parse_model -m model.dxnn
```

5.2 Run Model

This tool runs a compiled model and performs a basic inference test. It measures inference time, validates output data against a reference, and optionally runs in a loop for stress testing.

Source: `bin/run_model.cpp`

Usage

```
run_model -m <model_dir> -i <input_bin> -o <output_bin> -r <reference output_bin> -l
<number of loops>
```

Option

- `-c, --config` : Path to a JSON configuration file
- `-m, --model` : Path to the compiled model file (`.dxnn`)
- `-i, --input` : Input binary file
- `-o, --output` : Output file to save inference results
- `-r, --ref` : Reference output file to compare results
- `-l, --loop` : Number of inference iteration to run (loop test)
- `--use-ort` : use ONNX Runtime
- `-h, --help` : Show help message

Example

```
$ run_model -m /.../model.dxnn -i /.../input.bin -l 100
```

5.3 DX-RT CLI Tool (Firmware Interface)

This tool provides a command-line interface to interact with DX-RT accelerator devices. It supports querying device status, resetting hardware, updating firmware, and more.

Note. This tool is applicable **only** for accelerator devices.

Usage

```
dxrt-cli <option> <argument>
```

Option

- `-s, --status` : Get current device status
- `-i, --info` : Display basic device information
- `-m, --monitor` : Monitoring device status every [arg] seconds (arg > 0)
- `-r, --reset` : Reset device (0: NPU only, 1: full device) (default: 0)
- `-d, --device` : Specify device ID (default: -1 for all device)
- `-u, --fwupdate` : Update firmware with a Deepx firmware file (options: force:, unreset)
- `-w, --fwupload` : Update firmware file (2nd_boot or rtos)
- `-g, --fwversion` : Check firmware version from a firmware file
- `-p, --dump` : Dump initial device state to a file
- `-l, --fwlog` : Extract firmware logs to a file
- `-h, --help` : Show help message

Example

```
$ dxrt-cli --status

$ dxrt-cli --reset 0

$ dxrt-cli --fwupdate fw.bin

$ dxrt-cli -m 1
```

6. Tutorials

6.1 C++ Tutorials

6.1.1 C++ Tutorials

Run (Synchronous)

The synchronous Run method uses a single NPU core to perform inference in a blocking manner. It can be configured to utilize multiple NPU cores simultaneously by employing threads to run each core independently.

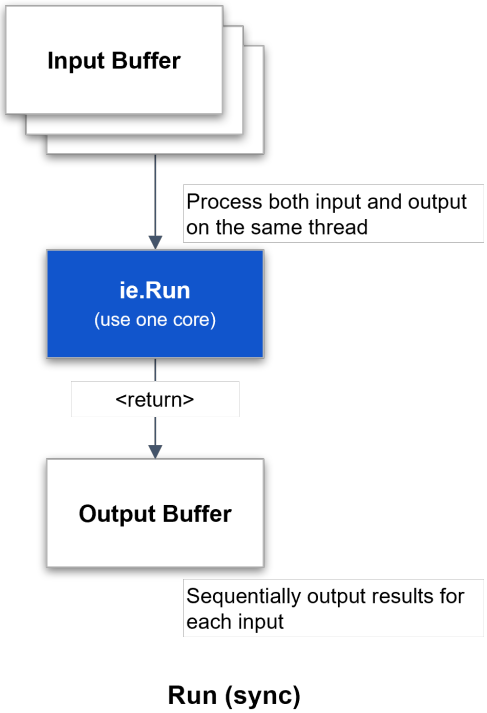

Figure. Synchronous Inference Operation

Inference Engine Run synchronous

- Inference synchronously
- Use **only** one npu core

The following is the simplest example of synchronous inference.

`run_sync_model.cpp`

```
// DX-RT includes
#include "dxrt/dxrt_api.h"
...

int main()
{
    std::string modelPath = "model-path";

    try
    {
        // create inference engine instance with model
        dxrt::InferenceEngine ie(modelPath);

        // create temporary input buffer for example
        std::vector<uint8_t> inputPtr(ie.GetInputSize(), 0);

        // inference loop
        for(int i = 0; i < 100; ++i)
        {
            // inference synchronously
            // use only one npu core
            auto outputs = ie.Run(inputPtr.data());

            // post processing
            postProcessing(outputs);

        } // for i
    }
    catch(const dxrt::Exception& e)  // exception for inference engine
    {
        std::cerr << e.what() << " error-code=" << e.code() << std::endl;
        return -1;
    }
    catch(const std::exception& e)
    {
        std::cerr << e.what() << std::endl;
        return -1;
    }

    return 0;
}
```

RunAsync (Asynchronous)

The asynchronous Run mode is a method that performs inference asynchronously while utilizing multiple NPU cores simultaneously. It can be implemented to maximize NPU resources through a callback function or a thread wait mechanism.

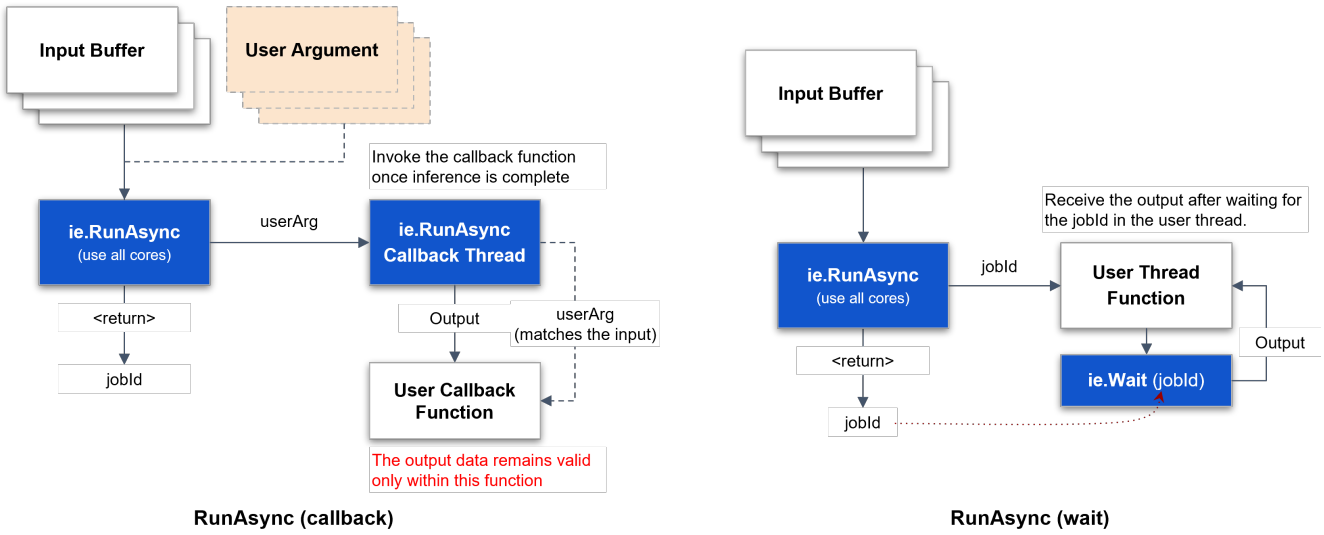

Figure. Asynchronous Inference Operation

Inference Engine RunAsync, Callback, User Argument

- the outputs are guaranteed to be valid **only** within this callback function
- processing this callback functions as quickly as possible is beneficial for improving inference performance
- inference asynchronously, use all npu cores
- if `device-load >= max-load-value`, this function will block

The following is an example of asynchronous inference using a callback function. A user argument can be used to synchronize the input with the output of the callback.

`run_async_model.cpp`

```
// DX-RT includes
#include "dxrt/dxrt_api.h"
...


static std::atomic<int> gCallbackCnt = {0};
static ConcurrentQueue<int> gResultQueue(1);
static std::mutex gCBMutex;

// invoke this function asynchronously after the inference is completed
static int onInferenceCallbackFunc(dxrt::TensorPtrs &outputs, void *userArg)
{

    // user data type casting
    std::pair<int, int>* user_data = reinterpret_cast<std::pair<int, int>*>(userArg);
```

```
    // post processing with outputs
    // ...
    (void)outputs;


    {
        // Mutex locks should be properly adjusted
        // to ensure that callback functions are thread-safe.
        std::lock_guard<std::mutex> lock(gCBMutex);

        gCallbackCnt ++;

        // end of the loop
        if ( user_data->second == gCallbackCnt.load() ) // check loop count
        {
            gResultQueue.push(gCallbackCnt);
        }
    }

    // delete argument object
    delete user_data;

    return 0;
}


int main(int argc, char* argv[])
{
    ...

    int total_callback_count = 0;

    try
    {

        // create inference engine instance with model
        dxrt::InferenceEngine ie(modelPath);

        // register call back function
        ie.RegisterCallback(onInferenceCallbackFunc);

        // create temporary input buffer for example
        std::vector<uint8_t> inputPtr(ie.GetInputSize(), 0);

        // inference loop
        for(int i = 0; i < loop_count; ++i)
        {
            // user argument
            std::pair<int, int> *userData = new std::pair<int, int>(i, loop_count);

            // inference asynchronously, use all npu cores
            ie.RunAsync(inputPtr.data(), userData);

        }
```

```
        // wait until all callbacks have been processed
        total_callback_count = gResultQueue.pop();

    }
    catch (const dxrt::Exception& e)
    {
        std::cerr << e.what() << " error-code=" << e.code() << std::endl;
        return -1;
    }
    catch (const std::exception& e)
    {
        std::cerr << e.what() << std::endl;
        return -1;
    }
    catch(...)
    {
        std::cerr << "Exception" << std::endl;
        return -1;
    }

    return (total_callback_count == loop_count ? 0 : -1);
}
```

The following is an example where multiple threads start input and inference, and a single callback processes the output.

Inference Engine RunAsync, Callback, User Argument, Thread

- the outputs are guaranteed to be valid **only** within this callback function
- processing this callback functions as quickly as possible is beneficial for improving inference performance
- inference asynchronously, use all npu cores
- if `device-load >= max-load-value`, this function will block

`run_async_model_thread.cpp`

```
// DX-RT includes
#include "dxrt/dxrt_api.h"
...

static const int THREAD_COUNT = 3;
static std::atomic<int> gResultCount = {0};
static std::atomic<int> gTotalCount = {0};
static ConcurrentQueue<int> gResultQueue(1);
static std::mutex gCBMutex;

static int inferenceThreadFunc(dxrt::InferenceEngine& ie, std::vector<uint8_t>&
inputPtr, int threadIndex, int loopCount)
{
```

```
    // inference loop
    for(int i = 0; i < loopCount; ++i)
    {
        // user argument
        UserData *userData = new UserData();

        // thread index
        userData->setThreadIndex(threadIndex);

        // total loop count
        userData->setLoopCount(loopCount);

        // loop index
        userData->setLoopIndex(i);

        try
        {
            // inference asynchronously, use all npu cores
            // if device-load >= max-load-value, this function will block
            ie.RunAsync(inputPtr.data(), userData);
        }
        catch(const dxrt::Exception& e)
        {
            std::cerr << e.what() << " error-code=" << e.code() << std::endl;
            std::exit(-1);
        }
        catch(const std::exception& e)
        {
            std::cerr << e.what() << std::endl;
            std::exit(-1);
        }

    } // for i

    return 0;

}

// invoke this function asynchronously after the inference is completed
static int onInferenceCallbackFunc(dxrt::TensorPtrs &outputs, void *userArg)
{

    // the outputs are guaranteed to be valid only within this callback function
    // processing this callback functions as quickly as possible is beneficial
    // for improving inference performance

    // user data type casting
    UserData *user_data = reinterpret_cast<UserData*>(userArg);

    // thread index
    int thread_index = user_data->getThreadIndex();

    // loop index
    int loop_index = user_data->getLoopIndex();
```

```
    // post processing
    // transfer outputs to the target thread by thread_index
    // postProcessing(outputs, thread_index);
    (void)outputs;


    // result count
    {
        // Mutex locks should be properly adjusted
        // to ensure that callback functions are thread-safe.
        std::lock_guard<std::mutex> lock(gCBMutex);

        gResultCount++;
        if ( gResultCount.load() == gTotalCount.load() ) gResultQueue.push(0);
    }

    // delete argument object
    delete user_data;

    return 0;
}


int main(int argc, char* argv[])
{
    ...

    bool result = false;

    try
    {
        // create inference engine instance with model
        dxrt::InferenceEngine ie(modelPath);

        // register call back function
        ie.RegisterCallback(onInferenceCallbackFunc);

        // create temporary input buffer for example
        std::vector<uint8_t> inputPtr(ie.GetInputSize(), 0);

        gTotalCount.store(loop_count * THREAD_COUNT);

        // thread vector
        std::vector<std::thread> thread_array;

        for(int i = 0; i < THREAD_COUNT; ++i)
        {
            // create thread
            thread_array.push_back(std::thread(inferenceThreadFunc, std::ref(ie),
std::ref(inputPtr), i, loop_count));
        }

        for(auto &t : thread_array)
        {
            t.join();
```

```
        } // for t


        // wait until all callbacks have been processed
        gResultQueue.pop();

    }
    catch (const dxrt::Exception& e)
    {
        std::cerr << e.what() << " error-code=" << e.code() << std::endl;
        return -1;
    }
    catch (const std::exception& e)
    {
        std::cerr << e.what() << std::endl;
        return -1;
    }
    catch(...)
    {
        std::cerr << "Exception" << std::endl;
        return -1;
    }

    return result ? 0 : -1;
}
```

The following is an example of performing asynchronous inference by creating an inference wait thread. The main thread starts input and inference, and the inference wait thread retrieves the output data corresponding to the input.

Inference Engine RunAsync, Wait

- inference asynchronously, use all npu cores
- if `device-load >= max-load-value`, this function will block

`run_async_model_wait.cpp`

```
// DX-RT includes
#include "dxrt/dxrt_api.h"
...

// concurrent queue is a thread-safe queue data structure
// designed to be used in a multi-threaded environment
static ConcurrentQueue<int> gJobIdQueue;

// user thread to wait for the completion of inference
static int inferenceThreadFunc(dxrt::InferenceEngine& ie, int loopCount)
{
    int count = 0;

    while(...)
```

```
    {
        // pop item from queue
        int jobId = gJobIdQueue.pop();

        try
        {
            // waiting for the inference to complete by jobId
            auto outputs = ie.Wait(jobId);

            // post processing
            postProcessing(outputs);

        }
        catch(const dxrt::Exception& e)  // exception for inference engine
        {
            std::cerr << e.what() << " error-code=" << e.code() << std::endl;
            std::exit(-1);
        }
        catch(const std::exception& e)
        {
            std::cerr << e.what() << std::endl;
            std::exit(-1);
        }

        // something to do

        count++;
        if ( count >= loopCount ) break;

    } // while

    return 0;
}

int main()
{
    const int LOOP_COUNT = 100;
    std::string modelPath = "model-path";

    try
    {
        // create inference engine instance with model
        dxrt::InferenceEngine ie(modelPath);

        // do not register call back function
        // inferenceEngine.RegisterCallback(onInferenceCallbackFunc);

        // create temporary input buffer for example
        std::vector<uint8_t> inputPtr(ie.GetInputSize(), 0);

        // create thread
        auto t1 = std::thread(inferenceThreadFunc, std::ref(ie), LOOP_COUNT);

        // inference loop
        for(int i = 0; i < LOOP_COUNT; ++i)
```

```
        {

            // no need user argument
            // UserData *userData = getUserDataInstanceFromDataPool();

            // inference asynchronously, use all npu cores
            // if device-load >= max-load-value, this function will block
            auto jobId = ie.RunAsync(inputPtr.data());

            // push jobId in global queue variable
            gJobIdQueue.push(jobId);

        } // for i

        t1.join();
    }
    catch(const dxrt::Exception& e)  // exception for inference engine
    {
        std::cerr << e.what() << " error-code=" << e.code() << std::endl;
        return -1;
    }
    catch(std::exception& e)
    {
        std::cerr << e.what() << std::endl;
        return -1;
    }

    return 0;
}
```

Run (Batch)

The following is an example of batch inference with multiple inputs and multiple outputs.

`run_batch_model.cpp`

```
int main(int argc, char* argv[])
{
    ...

    try
    {

        // create inference engine instance with model
        dxrt::InferenceEngine ie(modelPath);

        // create temporary input buffer for example
        std::vector<uint8_t> inputBuffer(ie.GetInputSize(), 0);

        // input buffer vector
        std::vector<void*> inputBuffers;
        for(int i = 0; i < batch_count; ++i)
```

```
        {
            // assigns the same buffer pointer in this example
            inputBuffers.emplace_back(inputBuffer.data());
        }

        // output buffer vector
        std::vector<void*> output_buffers(batch_count, 0);

        // create user output buffers
        for(auto& ptr : output_buffers)
        {
            ptr = new uint8_t[ie.GetOutputSize()];
        } // for i

        // batch inference loop
        for(int i = 0; i < loop_count; ++i)
        {
            // inference asynchronously, use all npu core
            auto outputPtrs = ie.Run(inputBuffers, output_buffers);

            // postProcessing(outputs);
            (void)outputPtrs;
        }

        // Deallocated the user's output buffers
        for(auto& ptr : output_buffers)
        {
            delete[] static_cast<uint8_t*>(ptr);
        } // for i

    }
    catch (const dxrt::Exception& e)
    {
        std::cerr << e.what() << " error-code=" << e.code() << std::endl;
        return -1;
    }
    catch (const std::exception& e)
    {
        std::cerr << e.what() << std::endl;
        return -1;
    }
    catch(...)
    {
        std::cerr << "Exception" << std::endl;
        return -1;
    }

    return 0;
}
```

Run & RunAsync

The method for converting a synchronous inference approach using one NPU core into an asynchronous inference approach using multiple NPU cores is as follows. It requires the use of callbacks or threads, as well as the implementation of multiple input buffers to support concurrent operations effectively.

Converting Run(Sync) to RunAsync

- Shift from Single NPU Core to Multiple Cores
  : Modify the existing Run(Sync) structure, which utilizes a single NPU core, to RunAsync structure capable of leveraging multiple NPU cores simultaneously.
- Create Multiple Input/Output Buffers
  : Implement multiple input/output buffers to prevent overwriting. Ensure an appropriate number of buffers are created to support concurrent operations effectively.
- Introduce Multi-Buffer Concept
  : To handle simultaneous inference processes, integrate a multi-buffer mechanism. This is essential for managing concurrent inputs and outputs without data conflicts.
- Asynchronous Inference with Threads or Callbacks
  : Adjust the code to ensure that inference inputs and outputs operate asynchronously using threads or callbacks for efficient processing.
- Thread-Safe Data Exchange
  : For data exchange between threads or callbacks, use a thread-safe queue or structured data mechanisms to avoid race conditions and ensure integrity.

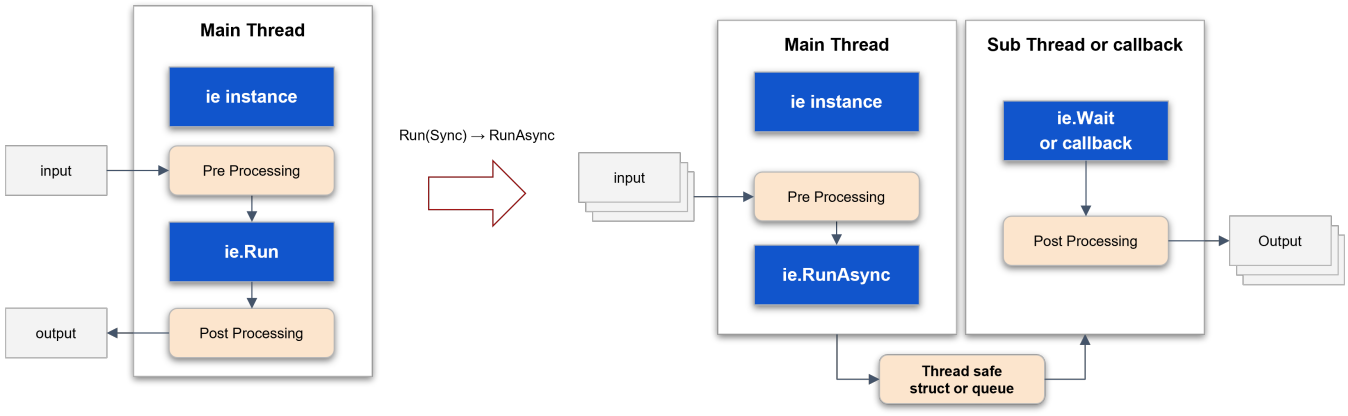

Figure. Converting Run(Sync) to RunAsync

Inference Option

The following inference options allow you to specify an NPU core for performing inference.

Inference Engine Run, Inference Option

- Select devices
  : default devices is `{}`
  : Choose devices to utilize

- Select bound option per device
  : `dxrt::InferenceOption::BOUND_OPTION::NPU_ALL`
  : `dxrt::InferenceOption::BOUND_OPTION::NPU_0`
  : `dxrt::InferenceOption::BOUND_OPTION::NPU_1`
  : `dxrt::InferenceOption::BOUND_OPTION::NPU_2`

- Use onnx runtime library (`ORT`)
  : `useORT` on or off

`run_sync_model_bound.cpp`

```
// DX-RT includes
#include "dxrt/dxrt_api.h"
...

int main()
{
    std::string modelPath = "model-path";

    try
    {

        // select bound option NPU_0 to NPU_2 per device
        dxrt::InferenceOption op;

        // first device only, default null
        op.devices.push_back(0); // use device 0
        op.devices.push_back(3); // use device 3

        // use BOUND_OPTION::NPU_0 only
        op.boundOption = dxrt::InferenceOption::BOUND_OPTION::NPU_0;

        // use ORT
        op.useORT = false;

        // create inference engine instance with model
        dxrt::InferenceEngine ie(modelPath, op);
```

```
        // create temporary input buffer for example
        std::vector<uint8_t> inputPtr(ie.GetInputSize(), 0);

        // inference loop
        for(int i = 0; i < 100; ++i)
        {
            // input
            uint8_t* inputPtr = readInputData();

            // inference synchronously with boundOption
            // use only one npu core
            // ownership of the outputs is transferred to the user
            auto outputs = ie.Run(inputPtr.data());

            // post processing
            postProcessing(outputs);

        } // for i
    }
    catch(const dxrt::Exception& e)  // exception for inference engine
    {
        std::cerr << e.what() << " error-code=" << e.code() << std::endl;
        return -1;
    }
    catch(const std::exception& e)
    {
        std::cerr << e.what() << std::endl;
        return -1;
    }

    return 0;
}
```

Camera / Inference / Display

The following is an example of a pattern that performs inference using two models on a single camera input and combines the results from both models for display.

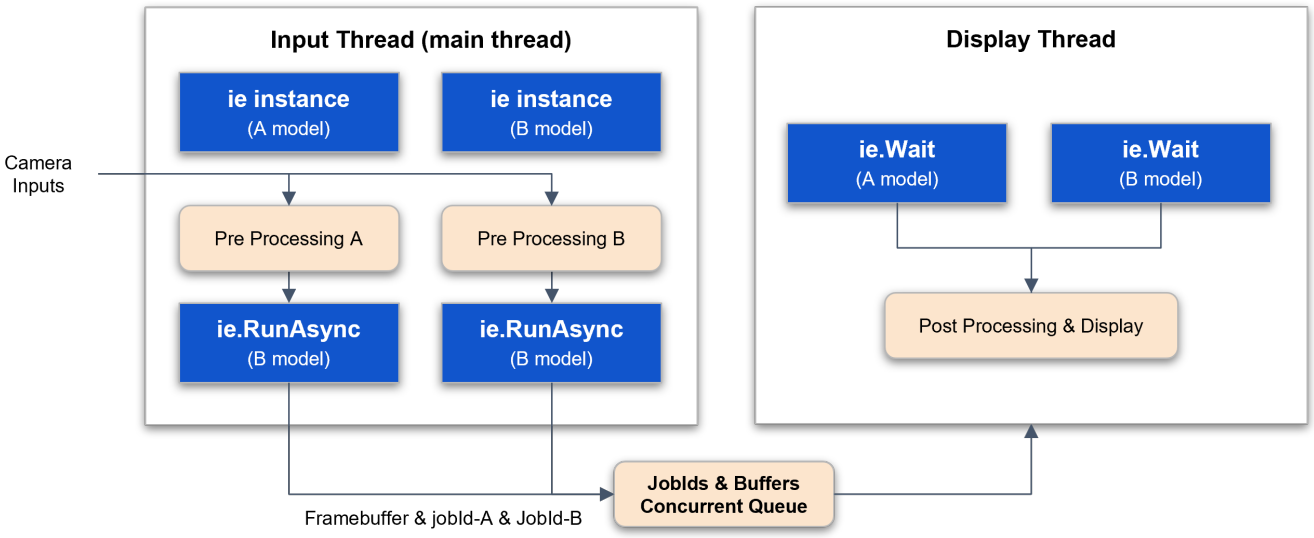

Figure. Multi-model and Multi-output

Multi-model, Async, Wait Thread `(CPU_1 → {NPU_1 + NPU_2} → CPU_2`

`display_async_wait.cpp`

```
// DX-RT includes
#include "dxrt/dxrt_api.h"
...

// input processing main thread with 2 InferenceEngine (asynchronous)
// display thread

struct FrameJobId {
    int jobId_A = -1;
    int jobId_B = -1;
    void* frameBuffer = nullptr;
    int loopIndex = -1;
};

static const int BUFFER_POOL_SIZE = 10;
static const int QUEUE_SIZE = 10;

static ConcurrentQueue<FrameJobId> gFrameJobIdQueue(QUEUE_SIZE);
static std::shared_ptr<SimpleCircularBufferPool<uint8_t>> gInputBufferPool_A;
static std::shared_ptr<SimpleCircularBufferPool<uint8_t>> gInputBufferPool_B;
static std::shared_ptr<SimpleCircularBufferPool<uint8_t>> gFrameBufferPool;

// total display count
static std::atomic<int> gTotalDisplayCount = {0};

static int displayThreadFunc(int loopCount, dxrt::InferenceEngine& ieA,
dxrt::InferenceEngine& ieB)
{
```

```
    while(gTotalDisplayCount.load() < loopCount)
    {
        // consumer framebuffer & jobIds
        auto frameJobId = gFrameJobIdQueue.pop();

        // output data of ieA
        auto outputA = ieA.Wait(frameJobId.jobId_A);

        // output data of ieB
        auto outputB = ieB.Wait(frameJobId.jobId_B);

        // post-processing w/ outputA & outputB
        postProcessing(outputA, outputB);

        gTotalDisplayCount++;

        // display (update framebuffer)
    }

    return 0;
}


int main(int argc, char* argv[])
{
    ...

    try
    {

        // create inference engine instance with model
        dxrt::InferenceEngine ieA(modelPath_A);

        gInputBufferPool_A =
std::make_shared<SimpleCircularBufferPool<uint8_t>>(BUFFER_POOL_SIZE,
ieA.GetInputSize());

        // create inference engine instance with model
        dxrt::InferenceEngine ieB(modelPath_B);

        gInputBufferPool_B =
std::make_shared<SimpleCircularBufferPool<uint8_t>>(BUFFER_POOL_SIZE,
ieB.GetInputSize());

        const int W = 512, H = 512, CH = 3;
        gFrameBufferPool =
std::make_shared<SimpleCircularBufferPool<uint8_t>>(BUFFER_POOL_SIZE, W*H*CH);

        // create thread
        std::thread displayThread(displayThreadFunc, loop_count, std::ref(ieA),
std::ref(ieB));

        // input processing
        for(int i = 0; i < loop_count; ++i)
        {
```

```
            uint8_t* frameBuffer = gFrameBufferPool->pointer();
            readFrameBuffer(frameBuffer, W, H, CH);

            uint8_t* inputA = gInputBufferPool_A->pointer();
            preProcessing(inputA, frameBuffer);

            uint8_t* inputB = gInputBufferPool_B->pointer();
            preProcessing(inputB, frameBuffer);

            // struct to pass to cpu operation thread
            FrameJobId frameJobId;

            // start inference of A model
            frameJobId.jobId_A = ieA.RunAsync(inputA);

            // start inference of B model
            frameJobId.jobId_B = ieB.RunAsync(inputB);

            // framebuffer used for input data
            frameJobId.frameBuffer = frameBuffer;
            frameJobId.loopIndex = i;

            // producer frame & jobId
            gFrameJobIdQueue.push(frameJobId);

        }

        displayThread.join();


    }
    catch (const dxrt::Exception& e)
    {
        std::cerr << e.what() << " error-code=" << e.code() << std::endl;
        return -1;
    }
    catch (const std::exception& e)
    {
        std::cerr << e.what() << std::endl;
        return -1;
    }
    catch(...)
    {
        std::cerr << "Exception" << std::endl;
        return -1;
    }

    return 0;
}
```

The following is an example of a pattern that sequentially performs operations using two models and CPU processing. The inference result from Model A is processed through CPU computation and then used as input data for Model B. Finally, the result from Model B is handled for display.

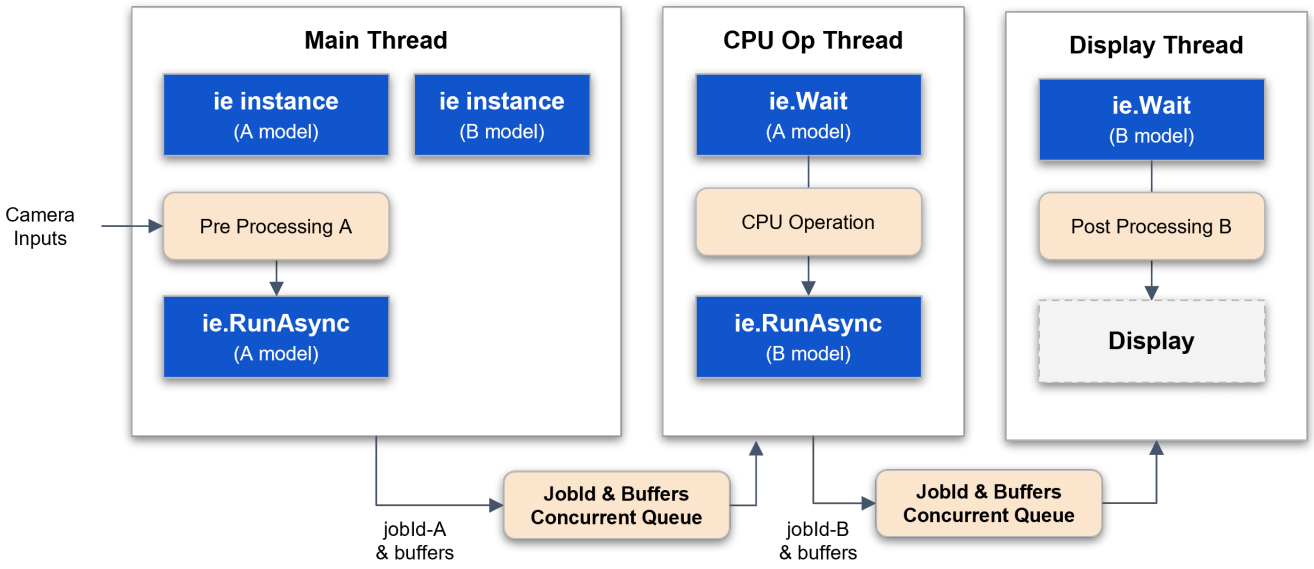

Figure. CPU and NPU Pipeline Operation

Multi-model, Async, Wait Thread `(CPU_1 → NPU_1 → CPU_2 → NPU_2 → CPU_3)`

`display_async_pipe.cpp`

```
// DX-RT includes
#include "dxrt/dxrt_api.h"
...

// input main thread
// 1 cpu processing thread
// 1 display thread

struct FrameJobId {
    int jobId_A = -1;
    int jobId_B = -1;
    uint8_t* inputBufferA;
    uint8_t* inputBufferB;
    void* frameBuffer = nullptr;

    int loopIndex;
};

static const int BUFFER_POOL_SIZE = 10;
static const int QUEUE_SIZE = 10;

static ConcurrentQueue<FrameJobId> gCPUOPQueue(QUEUE_SIZE);
static ConcurrentQueue<FrameJobId> gDisplayQueue(QUEUE_SIZE);
static std::shared_ptr<SimpleCircularBufferPool<uint8_t>> gInputBufferPool_A;
static std::shared_ptr<SimpleCircularBufferPool<uint8_t>> gInputBufferPool_B;
static std::shared_ptr<SimpleCircularBufferPool<uint8_t>> gFrameBufferPool;

// total display count
```

```
static std::atomic<int> gTotalDisplayCount = {0};


static int displayThreadFunc(int loopCount, dxrt::InferenceEngine& ieB)
{

    while(gTotalDisplayCount.load() < loopCount)
    {
        // consumer framebuffer & jobIds
        auto frameJobId = gDisplayQueue.pop();

        // output data of ieB
        auto outputB = ieB.Wait(frameJobId.jobId_B);

        // post-processing w/ outputA & outputB
        postProcessingB(outputB);

        gTotalDisplayCount++;

        // display (update framebuffer)
        if ( frameJobId.loopIndex == (loopCount - 1)) break;
    }

    return 0;
}

static int cpuOperationThreadFunc(int loopCount, dxrt::InferenceEngine& ieA,
dxrt::InferenceEngine& ieB)
{

    while(gTotalDisplayCount.load() < loopCount)
    {
        // consumer framebuffer & jobIds
        auto frameJobIdA = gCPUOPQueue.pop();

        // output data of ieA
        auto outputA = ieA.Wait(frameJobIdA.jobId_A);

        // post-processing w/ outputA
        postProcessingA(frameJobIdA.inputBufferB, outputA);

        FrameJobId frameJobIdB;
        frameJobIdB.loopIndex = frameJobIdA.loopIndex;
        frameJobIdB.jobId_B = ieB.RunAsync(frameJobIdA.inputBufferB);

        gDisplayQueue.push(frameJobIdB);

        // display (update framebuffer)
        if ( frameJobIdA.loopIndex == (loopCount - 1)) break;
    }

    return 0;
}
```

```
int main(int argc, char* argv[])
{

    ...

    try
    {

        // create inference engine instance with model
        dxrt::InferenceEngine ieA(modelPath);

        gInputBufferPool_A =
std::make_shared<SimpleCircularBufferPool<uint8_t>>(BUFFER_POOL_SIZE,
ieA.GetInputSize());

        // create inference engine instance with model
        dxrt::InferenceEngine ieB(modelPath);

        gInputBufferPool_B =
std::make_shared<SimpleCircularBufferPool<uint8_t>>(BUFFER_POOL_SIZE,
ieB.GetInputSize());

        const int W = 512, H = 512, CH = 3;
        gFrameBufferPool =
std::make_shared<SimpleCircularBufferPool<uint8_t>>(BUFFER_POOL_SIZE, W*H*CH);

        // create thread
        std::thread cpuOperationThread(cpuOperationThreadFunc, loop_count,
std::ref(ieA), std::ref(ieB));
        std::thread displayThread(displayThreadFunc, loop_count, std::ref(ieB));


        // input processing
        for(int i = 0; i < loop_count; ++i)
        {
            uint8_t* frameBuffer = gFrameBufferPool->pointer();
            readFrameBuffer(frameBuffer, W, H, CH);

            uint8_t* inputA = gInputBufferPool_A->pointer();
            preProcessing(inputA, frameBuffer);

            // struct to pass to a thread
            FrameJobId frameJobId;

            frameJobId.inputBufferA = inputA;
            frameJobId.inputBufferB = gInputBufferPool_B->pointer();

            // start inference of A model
            frameJobId.jobId_A = ieA.RunAsync(inputA);

            // framebuffer used for input data
            frameJobId.frameBuffer = frameBuffer;
            frameJobId.loopIndex = i;

            // producer frame & jobId
```

```
            gCPUOPQueue.push(frameJobId);

        }

        cpuOperationThread.join();
        displayThread.join();

    }
    catch (const dxrt::Exception& e)
    {
        std::cerr << e.what() << " error-code=" << e.code() << std::endl;
        return -1;
    }
    catch (const std::exception& e)
    {
        std::cerr << e.what() << std::endl;
        return -1;
    }
    catch(...)
    {
        std::cerr << "Exception" << std::endl;
        return -1;
    }

    return 0;
}
```

Exception

The error codes and types of exceptions for error handling are as follows.

```
enum ERROR_CODE {
        DEFAULT = 0x0100,
        FILE_NOT_FOUND,
        NULL_POINTER,
        FILE_IO,
        INVALID_ARGUMENT,
        INVALID_OPERATION,
        INVALID_MODEL,
        MODEL_PARSING,
        SERVICE_IO,
```

```
        DEVICE_IO
    };
```

- FileNotFoundException
- NullPointerException
- FileIOException
- InvalidArgumentException
- InvalidOperationException
- InvalidModelException
- ModelParsingException
- ServiceIOException
- DeviceIOException

```
    // try/catch prototype

    try
    {
        // DX-RT APIs ...
    }
    catch(const dxrt::Exception& e)  // exception for inference engine
    {
        std::cerr << e.what() << " error-code=" << e.code() << std::endl;
        return -1; // or std::exit(-1);
    }
    catch(std::exception& e)
    {
        std::cerr << e.what() << std::endl;
        return -1;  // or std::exit(-1);
    }
```

Examples

The examples provided earlier are actual code samples that can be executed. Please refer to them for practical use.

- `display_async_pipe`
  : An example using `[CPU_1 → {NPU_1 + NPU_2} → CPU_2]` pattern
- `display_async_wait`
  : An example using `[CPU_1 → NPU_1 → CPU_2 → NPU_2 → CPU_3]` pattern
- `display_async_thread`
  : An example using single model and multi threads
- `display_async_models_1`
  : An example using multi models and multi threads (Inference Engine is created within each thread)
- `display_async_models_2`
  : An example using multi models and multi threads (Inference Engine is created in the main thread)
- `run_async_model`
  : A performance-optimized example using a callback function
- `run_async_model_thread`
  : An example using a single inference engine, callback function, and thread
  : Usage method when there is a single AI model and multiple inputs
- `run_async_model_wait`
  : An example using threads and waits
- `run_sync_model`
  : An example using a single thread
- `run_sync_model_thread`
  : An example running an inference engine on multiple threads
- `run_sync_model_bound`
  : An example of specifying an NPU using the bound option

6.2 Python Tutorials

Run (Synchronous)

The synchronous Run method uses a single NPU core to perform inference in a blocking manner. It can be configured to utilize multiple NPU cores simultaneously by employing threads to run each core independently. (Refer to **Figure** in **Section 5.2. Inference Workflow**)

Inference Engine Run (Python)

`run_sync_model.py`

```
# DX-RT importes
from dx_engine import InferenceEngine
...


if __name__ == "__main__":
    ...

    # create inference engine instance with model
    ie = InferenceEngine(modelPath)

    input = [np.zeros(ie.GetInputSize(), dtype=np.uint8)]

    # inference loop
    for i in range(loop_count):

        # inference synchronously
        # use only one npu core
        outputs = ie.Run(input)

        # post processing
        postProcessing(outputs)

    exit(0)
```

RunAsync (Asynchronous)

The asynchronous Run mode is a method that performs inference asynchronously while utilizing multiple NPU cores simultaneously. It can be implemented to maximize NPU resources through a callback function or a thread wait mechanism.

Inference Engine RunAsync, Callback, User Argument

- the outputs are guaranteed to be valid **only** within this callback function
- processing this callback functions as quickly as possible is beneficial for improving inference performance
- inference asynchronously, use all npu cores
- if `device-load >= max-load-value`, this function will block

The following is an example of asynchronous inference using a callback function. A user argument can be used to synchronize the input with the output of the callback.

Inference Engine RunAsync, Callback, User Argument (Python)

`run_async_model.py`

```
from dx_engine import InferenceEngine
...

q = queue.Queue()
gLoopCount = 0

lock = threading.Lock()

def onInferenceCallbackFunc(outputs, user_arg):

    # the outputs are guaranteed to be valid only within this callback function
    # processing this callback functions as quickly as possible is beneficial
    # for improving inference performance

    global gLoopCount

    # Mutex locks should be properly adjusted
    # to ensure that callback functions are thread-safe.
    with lock:
        # user data type casting
        index, loop_count = user_arg


        # post processing
        #postProcessing(outputs);

        # something to do

        print("Inference output (callback) index=", index)

        gLoopCount += 1
        if ( gLoopCount == loop_count ) :
            print("Complete Callback")
            q.put(0)
```

```
    return 0


if __name__ == "__main__":

    ...

    # create inference engine instance with model
    ie = InferenceEngine(modelPath)

    # register call back function
    ie.register_callback(onInferenceCallbackFunc)


    input = [np.zeros(ie.GetInputSize(), dtype=np.uint8)]

    # inference loop
    for i in range(loop_count):

        # inference asynchronously, use all npu cores
        # if device-load >= max-load-value, this function will block
        ie.RunAsync(input, user_arg=[i, loop_count])

        print("Inference start (async)", i)

    exit(q.get())
```

The following is an example where multiple threads start input and inference, and a single callback processes the output.

Inference Engine RunAsync, Callback, User Argument, Thread

- the outputs are guaranteed to be valid **only** within this callback function
- processing this callback functions as quickly as possible is beneficial for improving inference performance
- inference asynchronously, use all npu cores
- if `device-load >= max-load-value`, this function will block

Inference Engine RunAsync, Callback, User Argument, Thread (Python)

`run_async_model_thread.py`

```
from dx_engine import InferenceEngine
...

THRAD_COUNT = 3
total_count = 0
q = queue.Queue()

lock = threading.Lock()
```

```
def inferenceThreadFunc(ie, threadIndex, loopCount):

    # input
    input = [np.zeros(ie.get_input_size(), dtype=np.uint8)]

    # inference loop
    for i in range(loopCount):

        # inference asynchronously, use all npu cores
        # if device-load >= max-load-value, this function will block
        ie.RunAsync(input,user_arg = [i, loopCount, threadIndex])

    return 0

def onInferenceCallbackFunc(outputs, user_arg):
    # the outputs are guaranteed to be valid only within this callback function
    # processing this callback functions as quickly as possible is beneficial
    # for improving inference performance

    global total_count

    # Mutex locks should be properly adjusted
    # to ensure that callback functions are thread-safe.
    with lock:
        # user data type casting
        index = user_arg[0]
        loop_count = user_arg[1]
        thread_index = user_arg[2]

        # post processing
        #postProcessing(outputs);

        # something to do

        total_count += 1

        if ( total_count ==  loop_count * THRAD_COUNT) :
            q.put(0)

    return 0


if __name__ == "__main__":
    ...

    # create inference engine instance with model
    ie = InferenceEngine(modelPath)

    # register call back function
    ie.register_callback(onInferenceCallbackFunc)


    t1 = threading.Thread(target=inferenceThreadFunc, args=(ie, 0, loop_count))
```

```
    t2 = threading.Thread(target=inferenceThreadFunc, args=(ie, 1, loop_count))
    t3 = threading.Thread(target=inferenceThreadFunc, args=(ie, 2, loop_count))

    # Start and join
    t1.start()
    t2.start()
    t3.start()


    # join
    t1.join()
    t2.join()
    t3.join()


    exit(q.get())
```

The following is an example of performing asynchronous inference by creating an inference wait thread. The main thread starts input and inference, and the inference wait thread retrieves the output data corresponding to the input.

Inference Engine RunAsync, Wait

- inference asynchronously, use all npu cores
- if `device-load >= max-load-value`, this function will block

Inference Engine RunAsync, Wait (Python)

`run_async_model_wait.py`

```
# DX-RT imports
from dx_engine import InferenceEngine
...

q = queue.Queue()


def inferenceThreadFunc(ie, loopCount):

    count = 0

    while(True):

        # pop item from queue
        jobId = q.get()

        # waiting for the inference to complete by jobId
        # ownership of the outputs is transferred to the user
        outputs = ie.Wait(jobId)

        # post processing
```

```
        # postProcessing(outputs);

        # something to do

        count += 1
        if ( count >= loopCount ):
            break

    return 0


if __name__ == "__main__":
    ...

    # create inference engine instance with model
    with InferenceEngine(modelPath) as ie:

        # do not register call back function
        # ie.register_callback(onInferenceCallbackFunc)

        t1 = threading.Thread(target=inferenceThreadFunc, args=(ie, loop_count))

        t1.start()

        input = [np.zeros(ie.get_input_size(), dtype=np.uint8)]

        # inference loop
        for i in range(loop_count):

            # inference asynchronously, use all npu cores
            # if device-load >= max-load-value, this function will block
            jobId = ie.run_async(input, user_arg=0)

            q.put(jobId)

        t1.join()

    exit(0)
```

Run (Batch)

The following is an example of batch inference with multiple inputs and multiple outputs.

`run_batch_model.py`

```
import numpy as np
import sys
from dx_engine import InferenceEngine
from dx_engine import InferenceOption


if __name__ == "__main__":
```

```
    ...

    # create inference engine instance with model
    with InferenceEngine(modelPath) as ie:

        input_buffers = []
        output_buffers = []
        index = 0
        for b in range(batch_count):
            input_buffers.append([np.array([np.random.randint(0, 255)],
dtype=np.uint8)])
            output_buffers.append([np.zeros(ie.get_output_size(), dtype=np.uint8)])
            index = index + 1

        # inference loop
        for i in range(loop_count):

            # batch inference
            # It operates asynchronously internally
            # for the specified number of batches and returns the results
            results = ie.run_batch(input_buffers, output_buffers)

            # post processing

    exit(0)
```

Inference Option

The following inference options allow you to specify an NPU core for performing inference.

Inference Engine Run, Inference Option

- select devices

  default device is `[]`

  Choose the device to utilize (ex. `[0, 2]`)

- select bound option per device

  `InferenceOption.BOUND_OPTION.NPU_ALL`

  `InferenceOption.BOUND_OPTION.NPU_0`

  `InferenceOption.BOUND_OPTION.NPU_1`

  `InferenceOption.BOUND_OPTION.NPU_2`

- use onnx runtime library (`ORT`)

  `set_use_ort / get_use_ort`

NPU_ALL / NPU_0 / NPU_1 / NPU_2

```
# DX-RT imports
from dx_engine import InferenceEngine, InferenceOption
...

if __name__ == "__main__":
    ...

    # inference option
    option = InferenceOption()

    print("Inference Options:")

    # select devices
    option.devices = [0]

    # NPU bound opion (NPU_ALL or NPU_0 or NPU_1 or NPU_2)
    option.bound_option = InferenceOption.BOUND_OPTION.NPU_ALL

    # use ONNX Runtime (True or False)
    option.use_ort = False

    # create inference engine instance with model
    with InferenceEngine(modelPath, option) as ie:
```

```
        input = [np.zeros(ie.get_input_size(), dtype=np.uint8)]

        # inference loop
        for i in range(loop_count):

            # inference synchronously
            # use only one npu core
            # ownership of the outputs is transferred to the user
            outputs = ie.run(input)

            # post processing
            #postProcessing(outputs)
            print("Inference outputs ", i)

    exit(0)
```

Examples

The examples provided earlier are actual code samples that can be executed. Please refer to them for practical use. (`examples/python`)

- `run_async_model.py`

  A performance-optimized example using a callback function

- `run_async_model_thread.py`

  An example using a single inference engine, callback function, and thread

  Usage method when there is a single AI model and multiple inputs

- `run_async_model_wait.py`

  An example using threads and waits

- `run_sync_model.py`

  An example using a single thread

- `run_sync_model_thread.py`

  An example running an inference engine on multiple threads

- `run_sync_model_bound.py`

  An example of specifying an NPU using the bound option

7. API Reference

7.1 C++ API Reference

7.1.1 C++ API Reference

Inference Engine

```
class InferenceEngine
This class provides an abstraction for the runtime inference executor of the user's
compiled model. Once the user loads a compiled model into the InferenceEngine, real-
time device tasks are managed and scheduled by internal runtime libraries. It supports
both synchronous and asynchronous inference modes, depending on the user's request.


InferenceEngine(const std::string& modelPath, InferenceOption& option =
DefaultInferenceOption)
   Constructs an InferenceEngine instance.
   parameters:
      [in] modelPath    : Path to the model file.
      [in] option   : Inference options, including device and NPU core settings.
                      (default: DefaultInferenceOption).


std::vector<TensorPtrs> GetAllTaskOutputs()
   Retrieves the output tensors for all tasks.
   returns:
      A vector containing the output tensors of all tasks.


std::vector<TensorPtrs> get_outputs()
   Retrieves the output tensors for all tasks.
   returns:
      A vector containing the output tensors of all tasks.


std::string GetCompileType()
   Retrieves the compile type of the model.
   returns:
      The compile type of the model as a std::string.


std::string get_compile_type()
   [[deprecated("Use GetCompileType() instead")]]
   Retrieves the compile type of the model.
   returns:
      The compile type of the model as a std::string.
```

```
Tensors GetInputs(void* ptr = nullptr, uint64_t phyAddr = 0)
   Retrieves the input tensor.
   This function provides access to the input tensor based on the provided
   virtual and physical address pointers.
   parameters:
      [in] ptr      : pointer to virtual the address (optional)
      [in] phyAddr  : pointer to physical the address (optional)
   returns:
     If `ptr` is `nullptr`, returns the input memory area in the engine.
     If both `ptr` and `phyAddr` are provided, returns input tensors containing
     output addresses.


Tensors inputs(void* ptr = nullptr, uint64_t phyAddr = 0)
   [[deprecated("Use GetInputs() instead")]]
   Retrieves the input tensor.
   This function provides access to the input tensor based on the provided
   virtual and physical address pointers.
   parameters:
      [in] ptr      : pointer to virtual the address (optional)
      [in] phyAddr  : pointer to physical the address (optional)
   returns:
     If `ptr` is `nullptr`, returns the input memory area in the engine.
     If both `ptr` and `phyAddr` are provided, returns input tensors containing
     output addresses.


std::vector<Tensors> GetInputs(int devId)
   Retrieves the input tensors.
   parameters:
      [in] devId        : Device Id.
   returns:
      A vector of the input tensors.


std::vector<Tensors> inputs(int devId)
   [[deprecated("Use GetInputs() instead")]]
   Retrieves the input tensors.
   parameters:
      [in] devId        : Device Id.
   returns:
      A vector of the input tensors.


uint64_t GetInputSize()
   Retrieves the total size of input tensors.
   This function returns the total memory size required for the input tensors
   of a single inference operation.
   returns:
      Input size in bytes for one inference.


uint64_t input_size()
   [[deprecated("Use GetInputSize() instead")]]
```

```
   Retrieves the total size of input tensors.
   This function returns the total memory size required for the input tensors
   of a single inference operation.
   returns:
      Input size in bytes for one inference.


int GetLatency()
   Retrieves the most recent inference latency.
   This function returns the latency time taken for the most recent inference operation.
   returns:
      Latency in microseconds


int latency()
   [deprecated("Use GetLatency() instead")]]
   Retrieves the most recent inference latency.
   This function returns the latency time taken for the most recent inference operation.
   returns:
      Latency in microseconds


int GetLatencyCnt()
   Retrieves the number of latency measurements.
   This function returns the total count of latency measurements.
   returns:
      The count of latency measurements.


std::vector<int> GetLatencyVector()
   Retrieves a vector of inference latency values.
   returns:
      A vector of inference latency values in microseconds.


double GetLatencyMean()
   Retrieves the average inference latency.
   This function returns the mean latency of inference execution.
   returns:
      The average inference latency in microseconds.


double GetLatencyStdDev()
   Retrieves the standard deviation of inference latency.
   This function calculates the variation in inference latency time.
   returns:
      The standard deviation of inference latency in microseconds.


std::string GetModelName()
   Retrieves the name of the model.
   This function returns the model's name.
   returns:
      The name of the model as a string.
```

```
std::string name()
   [[deprecated("Use GetModelName() instead")]]
   Retrieves the name of the model.
   This function returns the model's name.
   returns:
      The name of the model as a string.


uint32_t GetNpuInferenceTime()
   Retrieves the most recent inference time.
   This function returns the duration of the most recent inference operation.
   returns:
      Inference time in microseconds.


uint32_t inference_time()
   [[deprecated("Use GetNpuInferenceTime() instead")]]
   Retrieves the most recent inference time.
   This function returns the duration of the most recent inference operation.
   returns:
      Inference time in microseconds.


int GetNpuInferenceTimeCnt()
   Retrieves the number of NPU inference time measurements.
   This function returns the total count of NPU inference time measurements.
   returns:
      The count of NPU inference time measurements.


std::vector<uint32_t> GetNpuInferenceTimeVector()
   Retrieves a vector of NPU inference time values.
   returns:
      A vector of inference time values in microseconds.


double GetNpuInferenceTimeMean()
   Retrieves the average NPU inference time.
   This function returns the mean execution time for NPU inference.
   returns:
      The average NPU inference time in microseconds.


double GetNpuInferenceTimeStdDev()
   Retrieves the standard deviation of NPU inference time.
   This function returns the standard deviation of the inference time.
   returns:
      The standard deviation of inference time in microseconds.


int GetNumTailTasks()
   Retrieves the number of tail tasks in the model.
   Tail tasks are tasks that do not have any subsequent tasks in the model's task chain.
   This function returns the count of such tail tasks.
```

```
   returns:
      The number of tail tasks.


int get_num_tails()
   [[deprecated("Use GetNumTailTasks() instead")]]
   Retrieves the number of tail tasks in the model.
   Tail tasks are tasks that do not have any subsequent tasks in the model's task chain.
   This function returns the count of such tail tasks.
   returns:
      The number of tail tasks.


Tensors GetOutputs(void *ptr=nullptr, uint64_t phyAddr=0)
   Retrieves the output tensor.
   parameters:
      [in] ptr      : Pointer to the virtual address.
      [in] phyAddr  : Pointer to the physical address.
   returns:
      If `ptr` is `nullptr`, the output memory area in the engine is returned
      If both `ptr` and `phyAddr` are provided, returns output tensors containing
      the output addresses.


Tensors outputs(void *ptr=nullptr, uint64_t phyAddr=0)
   [[deprecated("Use GetOutputs() instead")]]
   Retrieves the output tensor.
   parameters:
      [in] ptr      : Pointer to the virtual address.
      [in] phyAddr  : Pointer to the physical address.
   returns:
      If ptr is null, the output memory area in the engine is returned
      If both ptr and phyAddr are provided, returns output tensors containing
      the output addresses.


uint64_t GetOutputSize()
   Retrieves the total size of output tensors.
   This function returns the total memory size required for the output tensors
   of a single inference operation.
   returns:
      Output size in bytes for one inference.


uint64_t output_size()
   [[deprecated("Use GetOutputSize() instead")]]
   Retrieves the total size of output tensors.
   This function returns the total memory size required for the output tensors
   of a single inference operation.
   returns:
      Output size in bytes for one inference.


std::vector<string> GetTaskOrder()
   Retrieves the execution order of tasks in the model.
```

```
   This function returns the sequence of tasks as they are executed within the model.
   returns:
      A vector of strings representing the task execution order.


std::vector<string> task_order()
   [[deprecated("Use GetTaskOrder() instead")]]
   Retrieves the execution order of tasks in the model.
   This function returns the sequence of tasks as they are executed within the model.
   returns:
      A vector of strings representing the task execution order.


bool IsPPU()
   Checks whether PPU is utilized.
   returns:
     true if PPU is being utilized, otherwise false.


bool is_PPU()
   [[deprecated("Use IsPPU() instead")]]
   Checks whether PPU is utilized.
   returns:
     true if PPU is being utilized, otherwise false.


void RegisterCallback(std::function<int(TensorPtrs &outputs, void *userArg)>
callbackFunc)
   Register user callback function to be called by inference completion.
   parameters:
      [in] callbackFunc : A function that is called when inference is complete.
                          It receives two arguments: outputs and userArg.
                          outputs : output tensors data
                          userArg : userArg given by RunAsync();



void RegisterCallBack(std::function<int(TensorPtrs &outputs, void *userArg)>
callbackFunc)
   [[deprecated("Use RegisterCallback() instead")]]
   Register user callback function to be called by inference completion.
   parameters:
      [in] callbackFunc : A function that is called when inference is complete.
                          It receives two arguments: outputs and userArg.
                          outputs : output tensors data
                          userArg : userArg given by RunAsync();


TensorPtrs Run(void* inputPtr, void * userArg = nullptr, void* outputPtr = nullptr)
   Runs the inference engine synchronously using the specified input pointer.
   This function executes inference with the given input data and returns the output
tensors.
   parameters:
      [in] inputPtr : Pointer to the input data for inference.
      [in] userArg  : User-defined arguments as a void pointer (optional).
```

```
                        (e.g. original frame data, metadata about input, etc.)
      [out] outputPtr   : Pointer to store the output data. If `nullptr`, the output
data is
                         stored in an internal buffer.
   returns:
      Output tensors as a vector of smart pointer instances.


int RunAsync(void* inputPtr, void * userArg = nullptr, void* outputPtr = nullptr)
   Initiates an asynchronous inference request and returns a job ID.
   This function performs a non-blocking inference operation using the specified input
   pointer. It returns a job ID that can be used with the `wait()` function to retrieve
   the results.
   parameters:
      [in] inputPtr : Pointer to the input data for inference.
      [in] userArg  : User-defined arguments as a void pointer (optional).
                        (e.g. original frame data, metadata about input, etc.)
      [out] outputPtr   : Pointer to store the output data. If `nullptr`, the output
data is
                         stored in an internal buffer.
   returns:
      Job ID that can be used with the `wait()` function to retrieve the inference
result.


float RunBenchmark(int num, void* inputPtr = nullptr)
   Runs a benchmark test by executing inference multiple times.
   This function performs inference in a loop for the specified number of times
   and calculates the average frames per second (FPS).
   parameters:
      [in] num      : Number of inference iterations.
      [in] inputPtr : Pointer to the input data for inference (optional).
                         If `nullptr`, default input data is used.
   returns:
      Average FPS (frames per second) over the benchmark runs.


float RunBenchMark(int num, void* inputPtr = nullptr)
   [[deprecated("Use RunBenchmark() instead")]]
   Runs a benchmark test by executing inference multiple times.
   This function performs inference in a loop for the specified number of times
   and calculates the average frames per second (FPS).
   parameters:
      [in] num      : Number of inference iterations.
      [in] inputPtr : Pointer to the input data for inference (optional).
                         If `nullptr`, default input data is used.
   returns:
      Average FPS (frames per second) over the benchmark runs.


TensorPtrs Wait(int jobId)
   Waits for the completion of an asynchronous inference request.
   This function blocks execution until the specified inference job, identified by
`jobId`,
   is complete and then returns the output tensors.
```

```
   parameters:
      [in] jobId    : Job ID returned by `RunAsync()`, used to track the inference
request.
   returns:
      Output tensors as a vector of smart pointer instances.
```

Inference Option

```
class InferenceOption
This struct specifies inference options applied to dxrt::InferenceEngine.
User can configure which npu device is used to inference.

enum BOUND_OPTION {
        NPU_ALL = 0,
        NPU_0,
        NPU_1,
        NPU_2
    };

uint32_t boundOption = BOUND_OPTION::NPU_ALL
   variables:
   Select the NPU core inside the device.
   NPU_ALL is an option that uses all NPU cores simultaneously. NPU_0, NPU_1, and NPU_2
are
   options that allow using only a single NPU core.


std::vector< int> devices = {}
   variables:
   device ID list to use
   make a list which contains a list of device ID to use. if it is empty
   (or use default value),
   then all devices are used. list of device ID to use (it is empty by default, then
all
   devices are used.)
```

Profiler

```
class Profiler
Provides a time measurement API based on timestamps.
The `Profiler` class is used to measure execution time using timestamps,
enabling performance analysis of various operations.

void Add(const std::string& event)
   Registers an event in the profiler.
   This function records an event with the specified name. If the profiler is used
   in a multi-threaded environment, this function should be called first to ensure
   proper event tracking.
   parameters:
      [in] event    : Name of the event to be registered.
```

```
void AddTimePoint(const std::string& event, TimePointPtr tp)
   Adds a timing data point for the specified event.
   This function records a time point associated with a given event name,
   allowing precise measurement of execution timing.
   parameters:
      [in] event    : Name of the event to associate with the time point.
      [in] tp       : Pointer to the timing data.


void End(const std::string& event)
   Records the end point of a specified event.
   This function marks the completion of an event, allowing for
   measurement of the event's duration when used with corresponding start points.
   parameters:
      [in] event    : Name of the event to mark as completed.


void Erase(const std::string& event)
   Clears the timing data of a specified event.
   parameters:
      [in] event    : Name of the event whose timing data should be cleared.


uint64_t Get(const std::string& event)
   Retrieves the most recent elapsed time of a specified event.
   This function returns the elapsed time (in microseconds) for the given event,
   based on the most recent recorded start and end points.
   parameters:
      [in] event    : Name of the event for which the elapsed time is requested.
   returns:
      Elapsed time in microseconds.


double GetAverage(const std::string& event)
   Retrieves the average elapsed time of a specified event.
   This function returns the average elapsed time (in microseconds) for the given
event,
   calculated from all recorded start and end points.
   parameters:
      [in] event    : Name of the event for which the average elapsed time is requested.
   returns:
      Average elapsed time in microseconds.


static Profiler& GetInstance()
   Retrieves the singleton instance of the Profiler.
   This function returns a reference to the pre-created singleton instance of
   the `Profiler`. Users should not create their own instance.
   returns:
      A reference to the singleton instance of `dxrt::Profiler`.


void Save(const std::string& file)
```

```
   Saves the timing data of all events to a specified file.
   This function exports all recorded timing data for each event and saves it to the
   given file. The data can be used for further analysis or reporting.
   parameters:
      [in] file     : Name of the file where the timing data will be saved.


void Show(bool  showDurations = false)
   Displays the elapsed times for all events.
   This function prints the elapsed times for each event. If `showDurations` is set to
true,
   the durations of each event will also be shown.
   parameters:
      [in] showDurations : If true, displays the durations of each event.
                           If false, only the elapsed times are shown.


void Start(const std::string& event)
   Records the start point of a specified event.
   This function marks the beginning of an event, allowing for the calculation of
   its duration when paired with an end point.
   parameters:
      [in] event    : Name of the event to mark as started.
```

Tensor

```
class Tensor
Represents a DX-RT tensor object, which defines a data array composed of uniform
elements.
The `Tensor` class abstracts a tensor object that holds a multi-dimensional array of
data
elements, typically used in machine learning models. This tensor is generally connected
to
inference engine objects for computations.

void* data(int  height, int width, int channel)
   Retrieves a pointer to a specific element in the tensor by its indices.
   This function returns the address of the element at the specified indices
   (height, width, channel) for a tensor in NHWC (height, width, channel) data format.
   parameters:
      [in] height   : The height index of the desired element.
      [in] width    : The width index of the desired element.
      [in] channel  : The channel index of the desired element.
   returns:
      A pointer to the specified tensor element at the given indices (height, width,
channel).
```

7.2 Python API Reference

7.2.1 Python API Reference

Inference Engine

```
class InferenceEngine
This class provides an abstraction for the runtime inference executor of the user's
compiled model. Once the user loads a compiled model into the InferenceEngine, real-
time device tasks are managed and scheduled by internal runtime libraries. It supports
both synchronous and asynchronous inference modes, depending on the user's request.

InferenceEngine(model_path: str, inference_option=None)
   Constructs an InferenceEngine instance.
   parameters:
      [in] model_path       : Path to the model file.
      [in] inference_option     : Inference options, including device and NPU core
settings.
                                (default: DefaultInferenceOption).


get_all_task_outputs()
   Retrieves the outputs from all tasks in sequence.
   returns:
      The outputs from all tasks in order.


get_outputs()
   @deprecated: Use `get_all_task_outputs()` instead.
   Retrieves the outputs from all tasks in sequence.
   returns:
       The outputs from all tasks in order.


get_compile_type()
   Retrieves the compile type of the model.
   returns:
      The compile type of the model.


get_input_data_type()
   Retrieves the required output data types as a list of strings.
   returns:
      A list of strings representing the input data types.


input_dtype()
   @deprecated: Use `get_input_data_type()` instead.
   Retrieves the required output data types as a list of strings.
   returns:
```

```
      A list of strings representing the input data types.


get_input_size()
   Retrieves the total size of input tensors.
   This function returns the total memory size required for the input tensors
   of a single inference operation.
   returns:
      Input size in bytes for one inference.


input_size()
   @deprecated: Use `get_input_size()` instead.
   Retrieves the total size of input tensors.
   This function returns the total memory size required for the input tensors
   of a single inference operation.
   returns:
      Input size in bytes for one inference.


get_latency()
   Retrieves the most recent inference latency.
   This function returns the latency taken for the most recent inference operation.
   returns:
      Latency in microseconds.


latency()
   @deprecated: Use `get_latency()` instead.
   Retrieves the most recent inference latency.
   This function returns the latency taken for the most recent inference operation.
   returns:
      Latency in microseconds.


get_latency_count()
   Retrieves the number of latency measurements.
   This function returns the total count of latency measurements.
   returns:
      The count of latency measurements.


get_latency_list()
   Retrieves a list of inference latency values.
   returns:
      A list of inference latency values in microseconds.


get_latency_mean()
   Retrieves the average inference latency.
   This function returns the mean latency of inference execution.
   returns:
      The average inference latency in microseconds.
```

```
get_latency_std()
   Retrieves the standard deviation of inference latency.
   This function calculates the variation in inference latency time.
   returns:
      The standard deviation of inference latency in microseconds.


get_npu_inference_time()
   Retrieves the most recent inference time.
   This function returns the duration of the most recent inference execution.
   returns:
      Inference time in microseconds.


inference_time()
   @deprecated: Use `get_npu_inference_time()` instead.
   Retrieves the most recent inference time.
   This function returns the duration of the most recent inference execution.
   returns:
      Inference time in microseconds.


get_npu_inference_time_count()
   Retrieves the number of NPU inference time measurements.
   This function returns the total count of NPU inference time measurements.
   returns:
      The count of NPU inference time measurements.


get_npu_inference_time_list()
   Retrieves a list of NPU inference time values.
   returns:
      A list of inference time values in microseconds.


get_npu_inference_time_mean()
   Retrieves the average NPU inference time.
   This function returns the mean execution time for NPU inference.
   returns:
      The average NPU inference time in microseconds.


get_npu_inference_time_std()
   Retrieves the standard deviation of NPU inference time.
   This function returns the standard deviation of the inference time.
   returns:
      The standard deviation of inference time in microseconds.


get_num_tail_tasks()
   Retrieves the number of tail tasks in the model.
   Tail tasks are tasks that do not have any subsequent tasks in the model's task chain.
   This function returns the count of such tail tasks.
   returns:
      The number of tail tasks.
```

```
get_num_tails()
   @deprecated: Use `get_num_tail_tasks()` instead.
   Retrieves the number of tail tasks in the model.
   Tail tasks are tasks that do not have any subsequent tasks in the model's task chain.
   This function returns the count of such tail tasks.
   returns:
      The number of tail tasks.


get_output_data_type()
   @deprecated: Use `get_output_data_type()` instead.
   Retrieves the required output data types as a list of strings.
   returns:
      A list of strings representing the output data types.


output_dtype()
   Retrieves the required output data types as a list of strings.
   returns:
      A list of strings representing the output data types.


get_output_size()
   Retrieves the total size of output tensors.
   This function returns the total memory size required for the output tensors
   of a single inference operation.
   returns:
     Output size in bytes for one inference.


output_size()
   @deprecated: Use `get_output_size()` instead.
   Retrieves the total size of output tensors.
   This function returns the total memory size required for the output tensors
   of a single inference operation.
   returns:
     Output size in bytes for one inference.


get_task_order()
   Retrieves the execution order of tasks in the model.
   This function returns the sequence of tasks as they are executed within the model.
   returns:
      A vector of strings representing the task execution order.


task_order()
   @deprecated: Use `get_task_order()` instead.
   Retrieves the execution order of tasks in the model.
   This function returns the sequence of tasks as they are executed within the model.
   returns:
      A vector of strings representing the task execution order.
```

```
is_ppu()
   Checks whether PPU is utilized.
   returns:
     True if PPU is being utilized, otherwise False.


is_PPU()
   @deprecated: Use `is_ppu()` instead.
   Checks whether PPU is utilized.
   returns:
     True if PPU is being utilized, otherwise False.


register_callBack(callback)
   Register user callback function to be called by inference completion.
   parameters:
      [in] callback : A function that is called when inference is complete.
                      It receives two arguments: outputs and user_arg.
                      - outputs: The data from the output tensors.
                      - user_arg: A user-defined argument passed via RunAsync().


RegisterCallBack(callback)
   @deprecated: Use `register_callback()` instead.
   Register user callback function to be called by inference completion.
   parameters:
      [in] callback : A function that is called when inference is complete.
                      It receives two arguments: outputs and user_arg.
                      - outputs: The data from the output tensors.
                      - user_arg: A user-defined argument passed via RunAsync().


run(input_feed_list: List[np.ndarray])
   Runs the inference engine synchronously using the specified input data.
   This function executes inference with the provided input data and returns
   the output tensors.
   parameters:
      [in] input_feed_list  : A list of Numpy arrays representing the input data.
   returns:
      A list of Numpy arrays representing the output tensors.


Run(input_feed_list: List[np.ndarray])
   @deprecated: Use run() instead.
   Runs the inference engine synchronously using the specified input data.
   This function executes inference with the provided input data and returns
   the output tensors.
   parameters:
      [in] input_feed_list  : A list of Numpy arrays representing the input data.
   returns:
      A list of Numpy arrays representing the output tensors.


run_async(input_feed_list: List[np.ndarray], user_arg)
```

```
   Initiates an asynchronous inference request and returns a job ID.
   This function performs a non-blocking inference operation using the specified input
   data. It returns a job ID that can be used with the `wait()` function to retrieve
   the results.
   parameters:
      [in] input_feed_list  : input data for inference.
      [in] user_arg     : user-defined arguments.
                          (e.g. original frame data, metadata about input, ... )
   returns:
      Job ID that can be used with the `wait()` function to retrieve the inference
result.


RunAsync(input_feed_list: List[np.ndarray], user_arg)
   @deprecated: Use `run_async()` instead.
   Initiates an asynchronous inference request and returns a job ID.
   This function performs a non-blocking inference operation using the specified input
   data. It returns a job ID that can be used with the `wait()` function to retrieve
   the results.
   parameters:
      [in] input_feed_list  : input data for inference.
      [in] user_arg     : user-defined arguments.
                          (e.g. original frame data, metadata about input, ... )
   returns:
      Job ID that can be used with the `wait()` function to retrieve the inference
result.


run_benchmark(loop_cnt,input_feed_list)
   Runs a benchmark test by executing inference multiple times.
   This function performs inference in a loop for the specified number of times
   and calculates the average frames per second (FPS).
   parameters:
      [in] loop_cnt     : Number of inference iterations.
      [in] input_feed_list  : the input data for inference
   returns:
      Average FPS (frames per second) over the benchmark runs.


RunBenchMark(loop_cnt,input_feed_list)
   @deprecated: Use `run_benchmark()` instead.
   Runs a benchmark test by executing inference multiple times.
   This function performs inference in a loop for the specified number of times
   and calculates the average frames per second (FPS).
   parameters:
      [in] loop_cnt     : Number of inference iterations.
      [in] input_feed_list  : the input data for inference
   returns:
      Average FPS (frames per second) over the benchmark runs.


wait(int jobId)
   Wait until a request is complete and returns output.
   parameters:
      [in] jobId    : job Id returned by run_async()
```

```
   returns:
      output tensors as vector


wait(int jobId)
   @deprecated: Use `wait()` instead.
   Wait until a request is complete and returns output.
   parameters:
      [in] jobId    : job Id returned by RunAsync()
   returns:
      output tensors as vector
```

8. Change Log

8.1 v2.9.5 (May 2025)

- Added full support for Python run_model.
- Updated the run_model option and its description
- Improve the Python API
- InferenceOption is now supported identically to the C++ API.
- set_devices(...) → devices = [0]
- set_bound_option(...) → bound_option = InferenceOption.BOUND_OPTION.NPU_ALL
- set_use_ort(...) → use_ort = True
- Callback functions registered via register_callback now accept user_arg of custom types. (removed .value)
- user_arg.value → user_arg
- run() now supports both single-input and batch-input modes, depending on the input format.
- Modify the build.sh script according to cmake options.
- CMake option USE_ORT=ON, running build.sh --clean installs ONNX Runtime.
- CMake option USE_PYTHON=ON, running build.sh installs the Python package.
- CMake option USE_SERVICE=ON, running build.sh starts or restarts the service.
- Add dxrt-cli -v to display minimum driver & compiler versions
- Addressed multithreading issues by implementing additional locks, improving stability under heavy load.
- Fix crash on multi-device environments with more than 2 H1 cards. (>=8 devices)
- Resolved data corruption errors that could occur in different scenarios, ensuring data integrity.
- Fix profiler bugs.
- Addressed issues identified by static analysis and other tools, enhancing code quality and reliability.
- Add --use_ort flag to the run_model.py example for ONNX Runtime.
- Add run batch function. (Python & C++)
- batch inference with multiple inputs and multiple outputs.
- Minimum model file versions
- .dxnn file format version >= v6
- compiler version >= v1.15.2

- Minimum Driver and Firmware versions
- RT Driver Version >= v1.5.0
- PCIe Driver Version >= v1.4.0
- Firmware Version >= v2.0.5

8.2 v2.8.2 (April 2025)

- Modify Inference Engine to be used with 'with' statements, and update relevant examples.
- Add Python inference option interface with the following configurations
- NPU Device Selection / NPU Bound Option / ORT Usage Flag
- Display dxnn versions in parse_model (.dxnn file format version & compiler version)
- Added instructions on how to retrieve device status information
- Driver and Firmware versions
- RT Driver >= v1.3.3
- Firmware >= v1.6.3