Atlas 500

Application Software Development Guide

Issue	01
Date	2020-05-30

HUAWEI TECHNOLOGIES CO., LTD.

Trademarks and Permissions


and other Huawei trademarks are trademarks of Huawei Technologies Co., Ltd.
All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Contents

1 Before You Start

This topic describes the basic knowledge, requirements, and precautions for using the Atlas 500 to develop services.

You are advised to read this section carefully before starting the development.

Application Scenario

This document applies to inference tasks using the Atlas 500.

Key Concepts

Table 1-1 Key concepts

Concept	Description
Ascend 310	The Ascend 310 is a high-performance and low–power consumption AI chip designed for scenarios such as image recognition, video processing, inference computing, and machine learning. The chip has two built-in AI cores, supports 128-bit LPDDR4X, and provides up to 16 TOPS (Float16/INT8) computing capability.
DDK	The Mind Studio solution provides the Digital Development Kit (DDK) for developers. You can install the DDK to obtain the APIs, libraries, and tool chains required for development on Mind Studio.
Graph	Graph is a concept in the HiAI framework instead of the computational graph in the deep learning framework. In the HiAI framework, a graph describes the entire service processing flow. It is a program processing flow consisting of multiple engines.
HiAI Engine	HiAI Engine is a universal service flow execution engine. It consists of Agent that runs on the host and Manager that runs on the device. Each engine provides a function implemented by user code, that is, the engine processing program is implemented by users.
Host	The host is the OS of the Hi3559A CPU.

Concept	Description
Device	The device is the OS of the Ascend 310.
DVPP	Digital vision pre-processing (DVPP) supports pre-processing operations such as image/video decoding and scaling. It is also capable of encoding and outputting precessed videos and images.
AIPP	AI pre-processing (AIPP) provides functions such as format conversion and padding/cropping, CSC (YUV2RGB or RGB2YUV), scaling, and channel data exchange.
OMG	Offline model generator (OMG) converts models trained by using frameworks such as Caffe and TensorFlow into offline models supported by Huawei chips. The OMG also supports model optimization functions that are performed independent from the device, such as operator scheduling optimization, weight data rearrangement and compression, and memory usage optimization.
OME	Offline model executor (OME) loads converted offline models for inference.
Ctrl CPU	One Ascend 310 chip has four Ctrl CPUs, which are used for service logic processing.
AI CPU	One Ascend 310 chip has four AI CPUs, which are used for operator task scheduling and implementation of some operators.
AI Core	One Ascend 310 chip has two AI cores, which are used for matrix computing.
IPC	An IP camera (IPC) provides RTSP data streams.

2 Introduction to the Atlas 500 and Software

The Atlas 500 is a lightweight edge device designed for a wide range of edge applications. It features powerful computing performance, large-capacity storage, flexible configuration, compact size, operating under a wide temperature range, strong environment adaptability, and easy maintenance and management. It is ideal for intelligent video surveillance, analysis, and data storage application scenarios, and can be deployed across edge and central equipment rooms, meeting application requirements in diverse environments, such as public security departments, communities, campuses, shopping malls, and supermarkets.

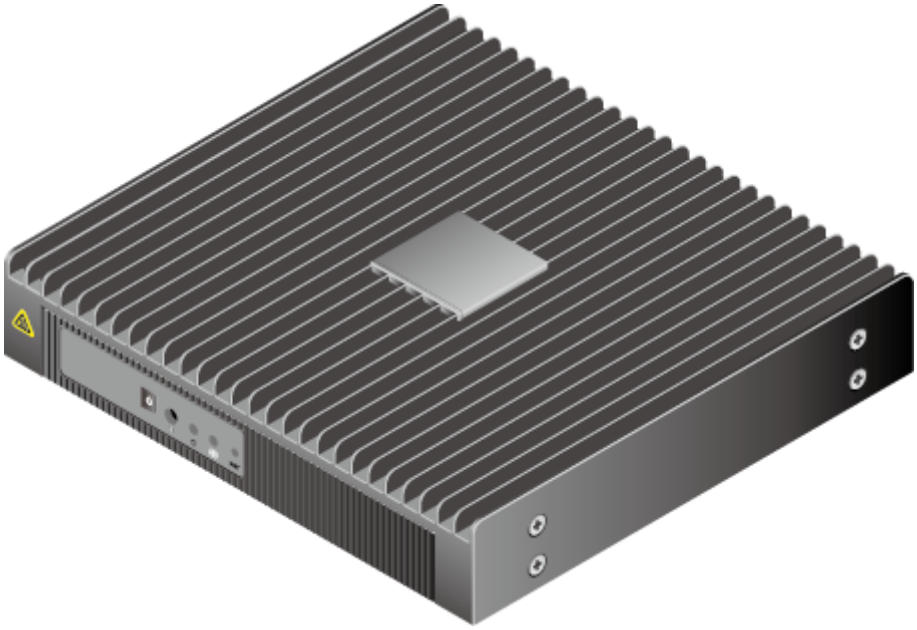
2.1 Atlas 500 Hardware

2.1.1 Atlas 500 Product Form

The Atlas 500 has two models with different drive configurations:

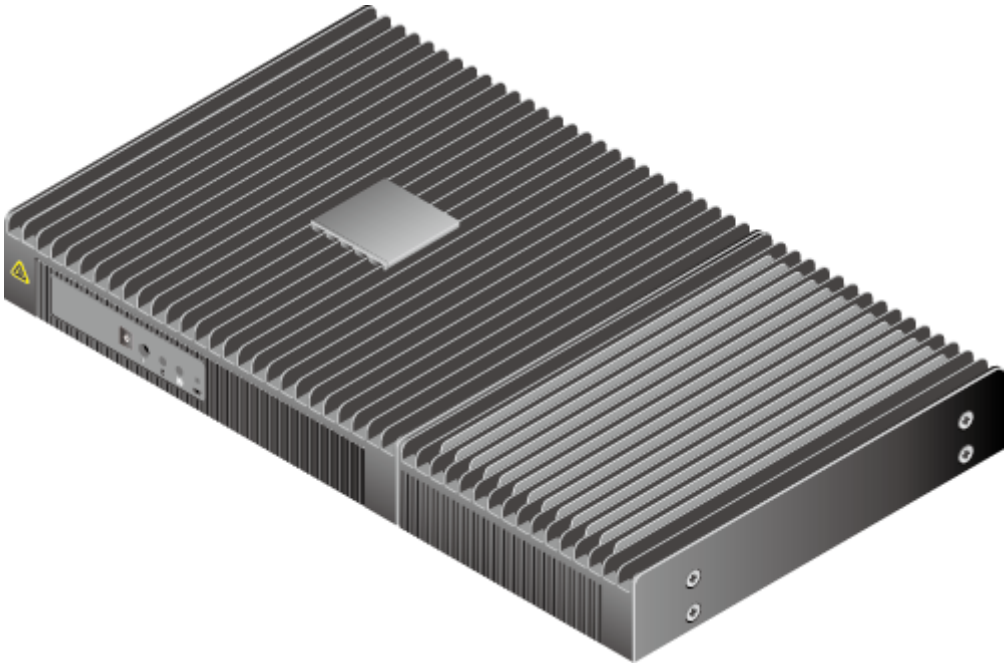
- **Figure 2-1** shows the Atlas 500 without a drive.

Figure 2-1 Atlas 500 (without a drive)



- **Figure 2-2** shows the Atlas 500 with a 3.5-inch drive configured in the drive tray on the right.

Figure 2-2 Atlas 500 (with a drive)



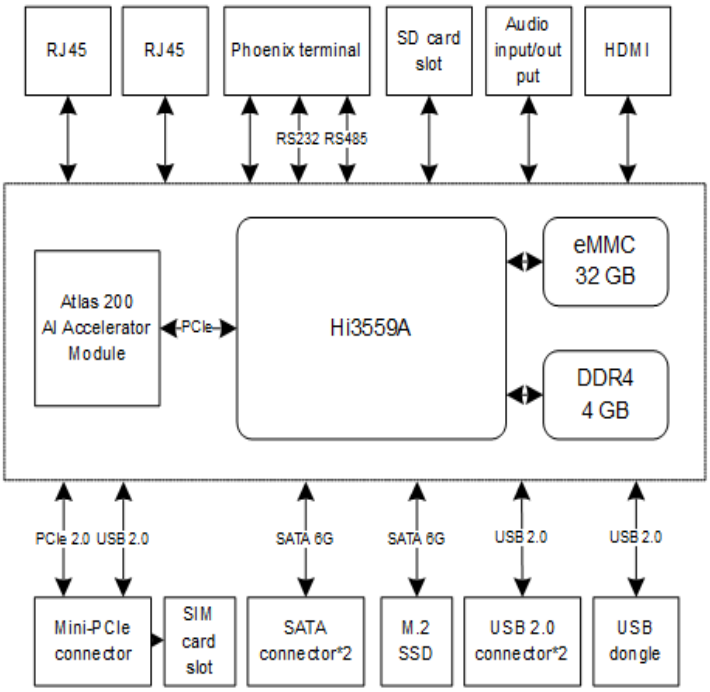
Atlas 500 PSU specifications:

The Atlas 500 uses 12 V DC/5 A power supply units (PSUs). You are advised to use 60 W industrial AC PSUs. For details about cable connections and recommended PSU models, see the *Atlas 500 AI Edge Station User Guide (Models 3000, 3010)*.

2.1.2 Atlas 500 System Architecture

Figure 2-3 shows the system architecture of the Atlas 500. It uses the Huawei-developed HiSilicon Hi3559A chip as the processor and works with the Atlas 200 AI Accelerator Card (optional) to provide 16 TOPS compute power on INT8 data. For details, see the *Huawei Atlas 500 White Paper*.

Figure 2-3 Atlas 500 system architecture



2.1.3 Atlas 200 AI Accelerator Card

Integrated with the HiSilicon Ascend 310 AI processor, the Atlas 200 allows data analysis, inference, and computing for various data such as images and videos. For details about the Atlas 200 AI Accelerator Card, see the *Huawei Atlas 500 White Paper*.

Project	Description
(Optional) Atlas 200	• Two Da Vinci AI cores • 8-core Arm Cortex-A55 1.6 GHz CPU • Computing performance (multiplication and addition): 8 TFLOPS for FP16 and 16 TOPS for INT8 • Memory specifications: LPDDR4X, 128-bit, 8 GB/4 GB, interface rate at 3200 Mbit/s

Project	Description
Atlas 200 codec capability	• H.264 hardware decoding, 16-channel 1080p 30 FPS (2-channel 3840 x 2160 60 FPS), YUV420 • H.265 hardware decoding, 16-channel 1080p 30 FPS (2-channel 3840 x 2160 60 FPS), YUV420 • H.264 hardware encoding, 1-channel 1080p 30 FPS, YUV420 • H.265 hardware encoding, 1-channel 1080p 30 FPS, YUV420 • JPEG decoding capability at 1080p 256 FPS and encoding capability at 1080p 64 FPS, up to 8192 x 8192 resolution • PNG decoding at 1080p 48 FPS, up to 4096 x 4096 resolution

2.2 Atlas 500 Software

2.2.1 OS Introduction

The master chip (Hi3559A) of the Atlas 500 runs a custom Linux kernel-based operating system (OS) Euler, which is used to manage hardware resources. To manage hardware resources and service software, EulerOS integrates software such as the intelligent management system. For details, see the *Atlas 500 Intelligent Edge System (V2.2.200.010 or Later) User Guide (Models 3000, 3010)*.

NOTE

You can log in to the EulerOS CLI over SSH. For details about the IP address, user name, and password, see "Default Parameters" in the *Atlas 500 Intelligent Edge System (V2.2.200.010 or Later) User Guide (Models 3000, 3010)*. The file systems and commands of EulerOS are similar to those of CentOS. You can configure the network and query hardware resources on the CLI.

2.2.2 Atlas 200 Driver

The Atlas 200 is mounted to EulerOS as a PCIe slave device for service software to call. You do not need to install or upgrade the Atlas 200 driver because it has been integrated into EulerOS.

2.2.3 Introduction to the Intelligent Management System

The Atlas Intelligent Edge System (IES) is used to enable and manage the intelligent edge computing hardware. It provides a secure, easy-to-use, and reliable edge AI hardware platform, making it easier for Huawei Atlas solution service software or third-party service software to use edge computing capabilities. With the IES, you can deploy media analysis and processing services in a low-cost, secure, reliable, and flexible manner.

The Atlas IES supports the following features:

- Network configuration
- Time synchronization
- Drive partitioning
- Software installation
- Certificate management
- Edge-cloud synergy
- System maintenance, including firmware upgrade, system restart, and log collection

For details about the Atlas IES, see the *Atlas 500 Intelligent Edge System (V2.2.200.010 or Later) User Guide (Models 3000, 3010)*.

3 Preparations

3.1 Obtaining Software Packages

Step 1 Log in to the **Huawei enterprise product support website**.

Step 2 Choose **TECHNICAL SUPPORT > AI Computing Platform > Atlas** and select the product.

Select A500-3000 or A500-3010 to go to the details page of the A500-3000 or A500-3010. The software packages on the A500-3000 page are the same as those on the A500-3010 page. You can select either of them.

Step 3 Click the **Software Download** tab and select the required software version.

Step 4 Obtain the Atlas 500 software packages, as listed in **Table 3-1**.

Table 3-1 Software packages

Name	Package	Description
Host cross compiler file	Euler_compile_env_cross.tar.gz	Used to compile the programs on the Atlas 500 host (Hi3559A).
OS source code package	A500-3000_A500-3010-EulerOSx.x.x.xxx_64bit_aarch64_basic.tar.gz	Contains the OS kernel source code of the Atlas 500 host.
DDK	A500-3000_A500-3010_A200-3000HiLens-DDK-Vx.x.x.x.tar.gz	Contains the development and debugging tools, header files, dependent libraries, and device compilation toolchain of the Atlas 500.

Name	Package	Description
System software upgrade package	A500-3000_A500-3010-ESP-FIRMWARE-Vx.x.x.xxx.zip	Used to upgrade the Atlas 500 OS, driver, and intelligent management system.

NOTE

In the package names, *xxx* indicates the software version.

----End

3.2 Obtaining the Sample Program

Open source code samples are available for the software components and inference services of the Ascend AI processor to improve the code development efficiency. You can refer to the code samples for code development. The following describes how to obtain the code samples.

- Obtain the sample program.
  Download URL: **https://gitee.com/HuaweiAtlas/samples**
  Usage: For details, see the **README.md** file of each sample provided on gitee.
- Obtain the demo program.
  Download URL: **https://gitee.com/HuaweiAtlas**
  Usage: For details, see the **README.md** file of the demo provided on gitee.

4 Configuring the Atlas 500

4.1 Powering On an Atlas 500

Step 1 Power on the system by referring to "Power-On Procedure" in the *Atlas 500 AI Edge Station User Guide (Models 3000, 3010)*.

Step 2 Wait 1-2 minutes after the device is powered on for the OS and device management software to start.

Step 3 After the system is started, check the status of the indicators and buttons on the front panel of the Atlas 500 by referring to "Indicators and Buttons" in the *Atlas 500 AI Edge Station User Guide (Models 3000, 3010)*.

- For the Atlas 500 (without a disk), both the graceful power-off indicator and health indicator are green. If the green indicators are steady on, the system is normal.
- For the Atlas 500 (with a disk), the graceful power-off indicator, health indicator, and disk indicator are green. If the green indicators are steady on, the system is normal. Otherwise, the system is abnormal.

----End

4.2 Obtaining a PC

Obtain a PC for accessing the IES and CLI of the Atlas 500 and configure network data to enable communication between the PC and the Atlas 500.

Step 1 Add an IP address in **192.168.2.xxx** for the PC, set the subnet mask to **255.255.255.0** and the gateway address to **192.168.2.1**.

Step 2 Connect GE port 1 of the Atlas 500 and the PC network port using an RJ45 network cable.

Step 3 On the PC, run the **ping** *Atlas 500 IP address* command to check whether the communication between the Atlas 500 and the PC is successful.

----End

4.3 Logging In to the Atlas IES

Step 1 Open your browser, enter https://*Atlas IES IP address* in the address box, and press **Enter**. The default IP address of port 1 is **192.168.2.111**, and the default IP address of the port 2 is **192.168.3.111**.

NOTE

Google Chrome 69 or later and Internet Explorer 11 are supported.

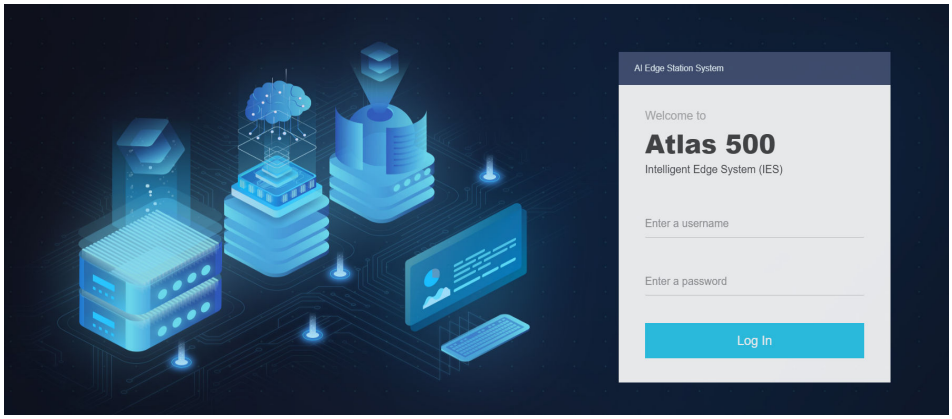
Step 2 Enter the user name and password.

- Default user name: **admin**
- Default password: **Huawei12#$**

NOTE

- The system locks a user account if the user enters incorrect passwords for consecutive five times. The locked account will be automatically unlocked in 5 minutes.
- For security purposes, change the initial password upon the first login and change the password periodically.

Figure 4-1 Atlas IES login page



Step 3 Click **Log In**.

The Atlas IES home page is displayed.

----End

4.4 Configuring a Network for the Atlas 500

To connect the Atlas 500 to a LAN not in the **192.168.2.xxx** network segment, you need to change the network configurations of the Atlas 500. For details about how to configure a network for the Atlas 500, see "Changing the Initial IP Address for the IES" in the *Atlas 500 AI Edge Station User Guide (Models 3000, 3010)*.

4.5 Checking the Atlas 500 Software Version

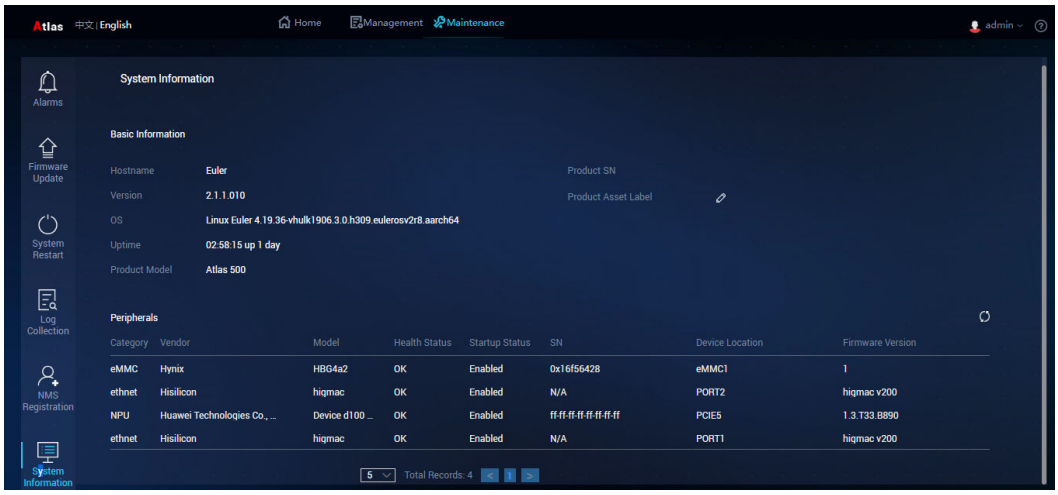
Step 1 Log in to the Atlas 500 IES.

Step 2 In the **System Information** area at the lower left corner on the homepage of the IES, click **More** to view system details.

Alternatively, on the **Maintenance** tab, click **System Information** on the left to view more.

Step 3 On the **System Information** page, check whether the system version, OS version, and NPU driver version are consistent with those in the obtained development software version. For details about how to obtain the development software and how to check its version, see **3.1 Obtaining Software Packages**.

Figure 4-2 Checking system information



----End

4.6 Upgrading the Atlas 500 Software

Decompress the **A500-3000_A500-3010-ESP-FIRMWARE-Vx.x.x.xxx.zip** package and obtain the **A500-3000_A500-3010-FW-Vx.x.xxx.xxx.hpm** file. To ensure that the software can run on the Atlas 500, use the **A500-3000_A500-3010-FW-Vx.x.xxx.xxx.hpm** file to upgrade the system to the same version as the development package.

Step 1 Log in to the Atlas 500 IES.

Step 2 Choose **Maintenance** > **Firmware Update**.

Step 3 On the **Firmware Update** page, click ··· on the right of **Firmware Package** and select a firmware package.

NOTE

The firmware format must be *.hpm.

Step 4 Upgrade firmware.

NOTICE

Do not power off the device during the update. Otherwise, the device may be damaged.

1. Click **Update**. A confirmation dialog box is displayed.
2. (Optional) Select **After the update is complete, the system automatically restarts for the update to take effect**.

   NOTE

   - If you select this option, the system automatically restarts for the update to take effect after the update is complete.
   - If you do not select this option, you need to manually restart the system for the update to take effect. For details, see **Step 5**.
3. Click **OK**.

You can view the update version and update progress on the page.

Step 5 After the update is complete, click **Restart to Take Effect**.

Step 6 Click **OK**. Wait for about 10 minutes until the update is complete.

----End

4.7 Logging In to and Using the Atlas 500 CLI

You can run commands on the Atlas 500 CLI to manage the Atlas 500. You can log in to the Atlas 500 CLI using either of the following:

- SSH
- serial port

Logging In to the OS Using SSH

Step 1 Prepare a PC installed with SSH client software (such as PuTTY and MobaXterm).

Step 2 Connect the PC and GE port 1 of the Atlas 500 to the same LAN, or use a network cable to directly connect the PC to the Atlas 500.

Step 3 On the PC, ping the Atlas 500 IP address to check whether the network between the Atlas 500 and the PC is connected.

Step 4 Start the SSH client software on the PC, set parameters, and log in to the Atlas 500.

Example:

- **Host Name** (or **IP address**): IP address of the Atlas 500. The default IP address of the Atlas 500 is **192.168.2.111**.
- **Port**: The default value is **22**.
- **Connection type** (or protocol): Select **SSH**.
- **Username**: Set it to **admin**.
- **Password**: The default password of **admin** is **Huawei12#$**.

  After setting the parameters, start the connection and log in to the Atlas 500 OS.

Step 5 Access the Atlas 500 CLI.

The host name of the login device is displayed on the left of the prompt.

Step 6 Run an Atlas 500 CLI command.

For details about the Atlas 500 CLI commands and their instructions, see "CLI" in the *Atlas 500 Intelligent Edge System (V2.2.200.010 or Later) User Guide (Models 3000, 3010)*.

----End

Logging In to the OS over a Serial Port

Step 1 Prepare a PC with a serial port, install the serial port client software (PuTTY or MobaXterm) on the PC, and prepare a Phoenix connector-to-serial cable.

Step 2 Connect the male Phoenix connector to the female Phoenix connector on the rear panel of the Atlas 500, and connect the serial cable to the PC. Locate the female Phoenix connector on the rear panel of the Atlas 500 by referring to "Rear Panel" in the *Atlas 500 AI Edge Station User Guide (Models 3000, 3010)*.

Step 3 Start the serial port client software and set login parameters.

Example:

- **Serial Line to connect to**: **COMn**
- **Speed (baud)**: **115200**
- **Data bits**: **8**
- **Stop bits**: **1**
- **Parity**: **None**
- **Flow control**: **None**

NOTE

In COM*N*, *N* indicates the serial port number, and the value is an integer.

After the parameters are set, start the connection.

Step 4 Enter the user name and password.

The default user name and password of the Atlas 500 are **admin** and **Huawei12#$**, respectively.

Step 5 Access the Atlas 500 CLI.

The host name of the login device is displayed on the left of the prompt.

Step 6 Run an Atlas 500 CLI command.

For details about the Atlas 500 CLI commands and their instructions, see "CLI" in the *Atlas 500 Intelligent Edge System (V2.2.200.010 or Later) User Guide (Models 3000, 3010)*.

----End

4.8 Accessing the Atlas 500 Development Mode

On the Atlas 500 CLI, run the **develop** command and enter the password **Huawei@SYS3** to switch to the Atlas 500 development mode.

In the displayed EulerOS CLI, you can run common Linux commands.

NOTE

It is recommended that the Atlas 500 development mode be enabled only in the software development and debugging phases.

5 Development Environment

5.1 Setting Up the Software Development Environment

After obtaining the software by referring to **3.1 Obtaining Software Packages** and verifying the software version consistency by referring to **4.5 Checking the Atlas 500 Software Version**, set up the software development environment.

Step 1 Prepare an x86 CPU server (or PC) as the development host. Install the Ubuntu 16.04 LTS OS on the development host. (If the development host has been installed with another OS version, you can install a VM on the development host and then the Ubuntu 16.04 LTS OS for development.)

URL for downloading the Ubuntu 16.04 LTS OS image: **http://old-releases.ubuntu.com/releases/**

Step 2 Configure the development host network and ensure that the development host and Atlas 500 are in the same LAN.

Step 3 Copy the software packages obtained in **3.1 Obtaining Software Packages** to a directory, for example **/home**, on the development host.

Step 4 Run the following command to create a directory for installing the Atlas 500 DDK, for example, **/home/Atlas500_DDK**:

mkdir -p /home/Atlas500_DDK

Step 5 Run the following command in the directory where the **A500-3000_A500-3010_A200-3000HiLens-DDK-Vx.x.x.x.tar.gz** file is located to install the Atlas 500 DDK file in the created directory:

tar -zxvf A500-3000_A500-3010_A200-3000HiLens-DDK-Vx.x.x.x.tar.gz -C /home/Atlas500_DDK

In **A500-3000_A500-3010_A200-3000HiLens-DDK-Vx.x.x.x.tar.gz**, *Vx.x.x.x* indicates the version number.

Step 6 In the directory where **Euler_compile_env_cross.tar.gz** is located, run the following command to install the cross compilation toolchain of the Atlas 500 host to the Atlas 500 DDK installation directory:

tar -zxvf Euler_compile_env_cross.tar.gz -C /home/Atlas500_DDK/toolchains

Step 7 Run the **cd /home/Atlas500_DDK** command to go to the Atlas 500 DDK installation directory.

Run the **cat ddk_info** command to check the version of the installed Atlas 500 DDK.

- **VERSION**: consistent with the NPU driver version obtained in **4.5 Checking the Atlas 500 Software Version**.
- **NAME**: DDK
- **TARGET**: ASIC

Step 8 Run the **vi ~/.bashrc** command to open the environment configuration file of the current user and add the following content to the end of the file:

```
export DDK_HOME=/home/Atlas500_DDK
export PATH=$PATH:$DDK_HOME/host/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$DDK_HOME/host/lib
```

Run the **source ~/.bashrc** command for the settings to take effect.

Step 9 Run the following commands to add a soft link to the **lib64** folder in **$DDK_HOME**:

```
cd /home/Atlas500_DDK/lib64
ln -s libssl.so.1.1 libssl.so
ln -s libcrypto.so.1.1 libcrypto.so
ln -s libprotobuf.so.15 libprotobuf.so
```

Step 10 If no error is reported in the preceding steps and the following information is displayed after you run the **omg -h** command in a directory of the OS, the Atlas 500 development environment has been successfully set up. If the Atlas 500 development environment fails to be set up, see Getting Help.

```
omg: usage: ./omg <args>
example:
./omg --model=./alexnet.prototxt --weight=./alexnet.caffemodel
--framework=0 --output=./domi
aguments explain:
  --model             Model file
  --weight            Weight file. Required when framework is Caffe
  --framework         Framework type(0:Caffe; 3:Tensorflow)
  --output            Output file path&name(needn't suffix, will add .om automatically)
  --encrypt_mode      Encrypt flag. 0: encrypt; -1(default): not encrypt
  --encrypt_key       Encrypt_key file
  --certificate       Certificate file
  --hardware_key      ISV file
  --private_key       Private key file
  --input_shape       Shape of input data. E.g.: "input_name1:n1,c1,h1,w1;input_name2:n2,c2,h2,w2"
  --h/help            Show this help message
  --cal_conf          Calibration config file
  --insert_op_conf    Config file to insert new op
  --op_name_map       Custom op name mapping file
  --plugin_path       Custom op plugin path. Default value is: "./plugin". E.g.: "path1;path2;path3".
                    Note: A semicolon(;) cannot be included in each path, otherwise the resolved path will not
match the expected one.
  --om                The model file to be converted to json
  --json              The output json file path&name which is converted from a model
```



```
  --mode              Run mode. 0(default): model => davinci; 1: framework/davinci model => json; 3: only pre-check
  --target            Target platform. (mini)
  --out_nodes         Output nodes designated by users. E.g.: "node_name1:0;node_name1:1;node_name2:0"
  --input_format      Format of input data. E.g.: "NCHW"
  --perf_level        Performance level. -1(default): generate a task-sink-model with ub-fuison and l2-fusion;
                      3: generate task-sink-model without l2-fusion; 4: task-sink-model without ub-fusion and l2-fusion.
  --check_report      The pre-checking report file. Default value is: "check_result.json"
  --input_fp16_nodes  Input node datatype is fp16 and format is NCHW. E.g.: "node_name1;node_name2"
  --is_output_fp16    Net output node datatype is fp16 and format is NCHW, or not. E.g.: "false,true,false,true"
  --ddk_version       The ddk version. E.g.: "x.y.z.Patch.B350"
  --net_format        Set net prior format. ND: select op's ND format preferentially; 5D: select op's 5D format preferentially
  --output_type       Set net output type. Support FP32 and UINT8
  --fp16_high_prec    FP16 high precision. 0(default): not use fp16 high precision; 1: use fp16 high precision
```

----End

5.2 DDK Directory Distribution

After the Atlas500 DDK is installed, its directory structure is as follows:

```
.
├── bin
├── conf
├── device
├── host
├── include
├── lib
├── lib64
├── packages
├── sample
├── scripts
├── toolchains
│   ├── Euler_compile_env_cross
│   ├── aarch64-linux-gcc6.3
└── uihost
```

Table 5-1 Description of the Atlas 500 DDK directory

Directory Name	Description
bin	Contains tools used on the development host, such as the offline model conversion tool OMG.
conf	Contains header files related to TE operator development.
device	Contains the link library files that device (Ascend 310) programs of the Atlas 500 depend on. This directory is linked to **../lib/aarch64-linux-gcc6.3**.
host	Contains the tools and libraries used on the development host.
include	Contains the header files required for the development of software running on the Atlas 500.

Directory Name	Description
lib	Contains the link library files that device programs of the Atlas 500 and programs on the development host depend on.
lib64	Contains the link library files that host programs of the Atlas 500 depend on.
packages	-
sample	-
scripts	-
toolchains	Contains the cross compilation toolchain of the Atlas 500.
toolchains/Euler_compile_env_cross	Contains the cross compilation toolchain for the host side of the Atlas 500.
toolchains/aarch64-linux-gcc6.3	Contains the cross compilation toolchain for the device side of the Atlas 500.
uihost	-

5.3 Header Files and Link Libraries

5.3.1 Header Files

The Atlas 500 host and device programs share a set of header files during development. All header files are stored in the **include** directory in the Atlas 500 DDK installation directory. The **include** directory structure is as follows:

```
include
├── inc
├── libc_sec
└── third_party
```

Table 5-2 Description of header files

Header File	Description
include/inc	Contains header files of service functions of Atlas 500 programs.
include/libc_sec	Contains header files of security functions.
include/third_party	Contains header files of third-party libraries.

5.3.2 Link libraries

The link libraries for Atlas 500 program development include the link libraries on which host and device programs of the Atlas 500 depend.

- The link library files on the host are stored in the **lib64** directory in the Atlas 500 DDK installation directory. The **lib64** directory does not contain subdirectories and contains all link library files and soft links required by the Atlas 500 host.
- The link library files of the device are stored in the **device/lib** directory in the Atlas500 DDK installation directory. The **device/lib** directory does not contain subdirectories and contains all link library files and soft links required by the Atlas 500 device.

5.4 Compilation Toolchains

5.4.1 Host Compilation Toolchain

The Atlas 500 host uses Hi3559A as the CPU, which is a 64-bit CPU based on the Armv8-A architecture. The Atlas 500 does not have any compilation tool running on Hi3559A. Therefore, you need to perform cross compilation on the x86 platform and then copy the compiled software to Hi3559A for running.

The Atlas 500 host software development and compilation toolchain is stored in the **toolchains/Euler_compile_env_cross/arm/cross_compile/install/bin** directory of the Atlas 500 DDK installation directory.

5.4.2 Device Compilation Toolchain

The Atlas 500 device uses the Arm Cortex-A55 CPU embedded in the Ascend 310 chip. Arm Cortex-A55 is a 64-bit CPU based on the Armv8-A architecture. The Atlas 500 does not have any compilation tool running on the Arm Cortex-A55 CPU. Therefore, you need to perform cross compilation on the x86 platform and then copy the compiled software to the Atlas 500 host for running. During program running, copy the generated executable file or dynamic library to the Atlas 500 device for running.

The software development and compilation toolchain for the Atlas 500 device is stored in **toolchains/aarch64-linux-gcc6.3/bin** under the Atlas 500 DDK installation directory.

5.5 DDK Tools

- DDK tool directory: **$DDK_HOME/bin/x86_64-linux-gcc5.4**

  $DDK_HOME is the installation directory of the Atlas 500 DDK, for example, **/home/Atlas500_DDK**.
- For details about tool functions, see **Table 5-3**.

Table 5-3 Description of tool functions

Name	Function	Description
IDE-daemon-client	IDE daemon command toolkit	For details, see the *IDE-daemon-client Command Reference*.
IDE-daemon-hiai	Data backhaul tool	• Data is sent back during image pre-processing. • Operator data is transmitted from the device to the host.
omg	Model conversion tool	The Caffe or TensorFlow model can be converted into a .om model file supported by the DDK. For details, see the *Model Conversion Guide*.
protoc	Tool for the third-party library protobuf	This tool converts .proto files into protocols supported by each language.

6 General Inference Service Flow

6.1 Service Flow

Figure 6-1 Service flow

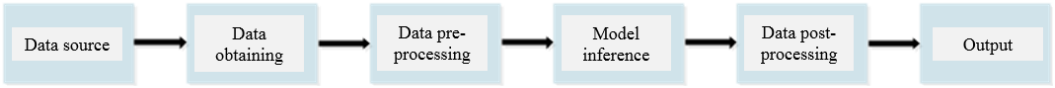


Figure 6-1 shows the software service flow. For details, see **Table 6-1**.

Table 6-1 Service flow description

Process	Description	Remarks
Data source	Indicates the video or image source.	The source can be RTSP streams from an IP camera (IPC) or offline videos or pictures in a disk.
Data obtaining	Implements data obtaining.	You can pull streams from the open-source FFmpeg function library or using custom implementation code. HostCPU implements data obtaining.

Process	Description	Remarks
Data pre-processing	Implements image pre-processing functions such as decoding, scaling, and color gamut conversion.	• If software decoding is used, the open-source OpenCV function library is called. The data pre-processing software runs on the host CPU or Ctrl CPU. • If hardware decoding is used, the API for DVPP decoding and AIPP CSC is called to implement pre-processing. Then the corresponding hardware modules are started on the device side.
Inference	Implements the model inference function.	The function runs on the device-side AI Core or AI CPU.
Post-processing	Implements post-processing of model inference results.	The function runs on the device-side Ctrl CPU or host CPU.
Output	Displays the implementation result.	-

6.2 Mapping Between Software and Hardware Modules

- Pre-processing and post-processing can be performed on the host or device side based on actual requirements.
- Pre-processing can be implemented by either hardware or software.

Figure 6-2 Software modules

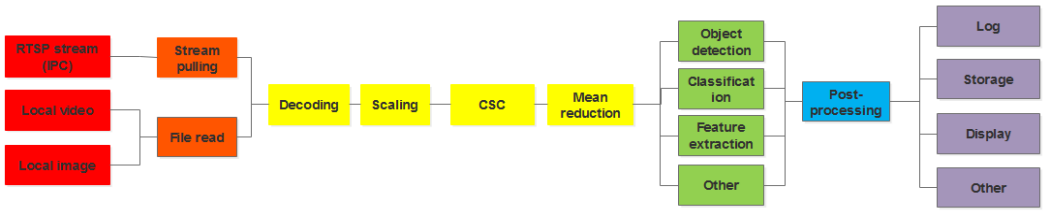
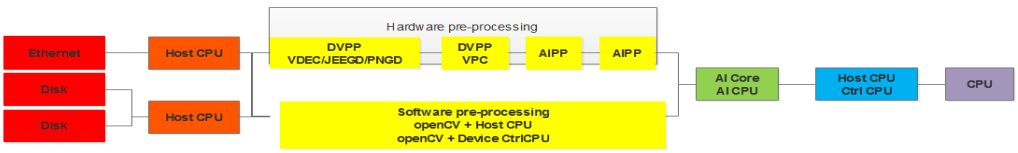


Figure 6-3 Hardware modules



7 Running a Code Sample

7.1 Obtaining the HelloDavinci Code

Download URL: **https://gitee.com/HuaweiAtlas/samples**

7.2 Description of the HelloDavinci File

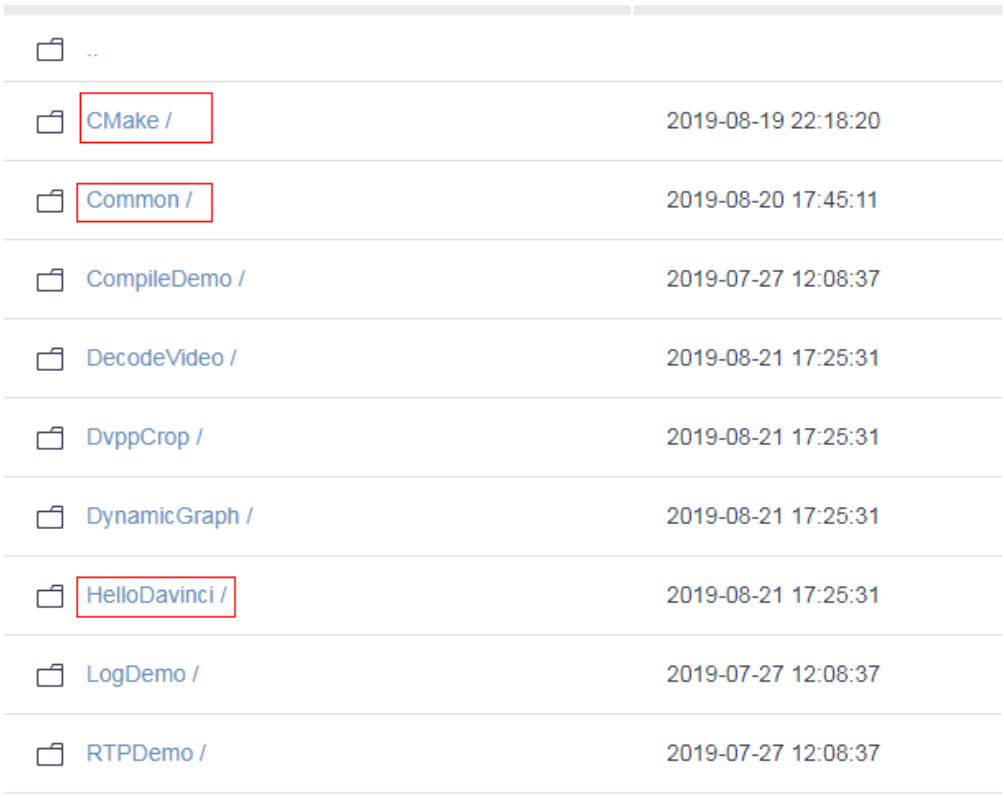
The **samples** directory contains compilation configurations and program samples (HelloDavinci included) of the Atlas. To run the HelloDavinci code file independently, obtain the **Samples/Cmake**, **Samples/Common**, and **Samples/HelloDavinci** folders, as shown in **Figure 7-1**.

- **CMake**: Stores the CMake configuration file.
- **Common**: Stores common code.
- **HelloDavinci**: Indicates the HelloDavinci project directory, including the .build file, source code file, graph configuration file, and **README.md**.

  NOTE

  Retain the relative paths of the three folders.

Figure 7-1 File directory



The directory structure of the CMake files is as follows:

```
├──Ascend.cmake    // device-side compilation chain
├──Euler.cmake     // host-side compilation chain
└──FindDDK.cmake   // file for searching for the DDK module
```

The directory structure of HelloDavinci is as follows:

```
├── build              // compilation folder, including the compilation on the host and device sides
│   ├── CMakeLists.txt
│   ├── device
│   └── host
├── build.sh           // compilation script
├── README.md          //README.md
├── main.cpp           // main function entry
├── include            // HelloDavinci common module
├── DstEngine          // DstEngine (host side)
│   ├── DstEngine.cpp
│   └── DstEngine.h
├── graph.config       // graph configuration file
├── HelloDavinci       // HelloDavinci engine (device side)
│   ├── HelloDavinci.cpp
│   └── HelloDavinci.h
└── SrcEngine          // SrcEngine engine (host side)
    ├── SrcEngine.cpp
    └── SrcEngine.h
```

7.3 HelloDavinci Process Framework

This section describes the process of implementing the HelloDavinci sample code.

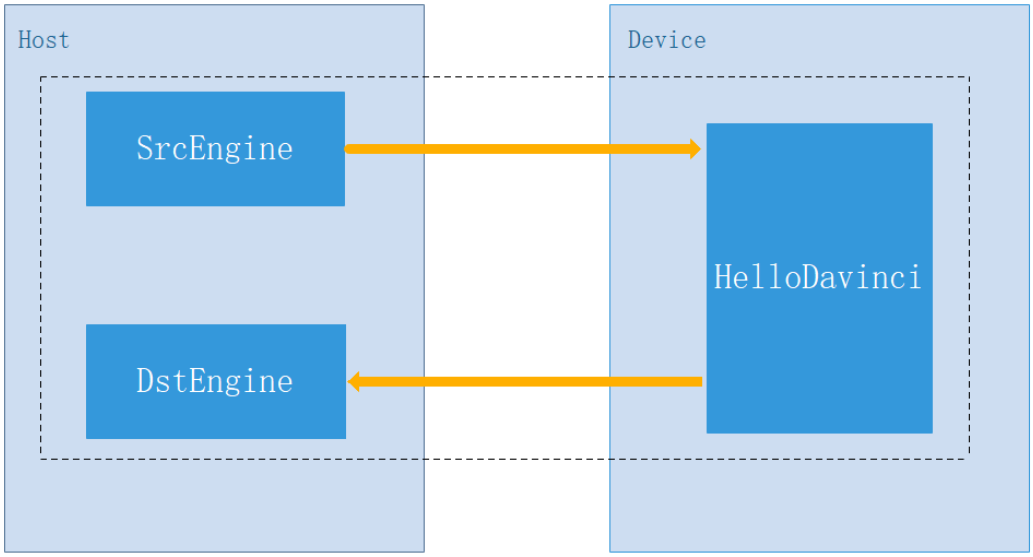
NOTE

If you are familiar with this process, directly go to **7.4 HelloDavinci Compilation and Running** and check the running result.

This sample demonstrates how to send data from the host to the device, return the generated character string from the device to the host, save the result, and print it to the terminal. As shown in **Figure 7-2**, the program is divided into two parts running on the host side (including SrcEngine and DstEngine) and device side (including HelloDavinci), respectively. The running process is as follows:

1. The **main** function is called to send data to SrcEngine.
2. SrcEngine forwards the received data to HelloDavinci, which generates the character string "This message is from HelloDavinci" and sends it to DstEngine.
3. After receiving the character string, DstEngine saves it to **${workPath}/out/dacvinci_log_info.txt** and sends a signal indicating the operation completion to the **main** function. **${workPath}** is the root directory of the project.
4. After the **main** function receives the completion signal, the graph is destroyed, information indicating the operation completion is printed on the terminal, and the program exits.

Figure 7-2 HelloDavinci process framework



7.4 HelloDavinci Compilation and Running

Prerequisites

- The Atlas 500 DDK has been installed.
- The third-party CMake compilation tool (2.8.4 or later) has been installed.

  If the CMake compilation tool is not available, install it by referring to **8.5.1 Compiling Service Software Using CMake**.

Compilation Procedure

Step 1 Copy the samples file obtained in **7.2 Description of the HelloDavinci File** to a directory on the development host.

Step 2 Run the **export DDK_HOME=***Atlas 500 DDK installation directory* command.

Step 3 Go to the **HelloDavinci** directory.

Step 4 Run the **bash build.sh A500** command to perform compilation. After the compilation is successful, a .main executable file and the target dynamic library file are generated in the **${workPath}/out/** directory.

${workPath} indicates the **HelloDavinci** directory.

Step 5 Copy the **out** folder to Atlas 500 and run the **./out/main** command to view the output result. If the following information is displayed, the execution is successful:

```
Euler:~ # ./out/main
Hello Davinci!
The sample end!!
Euler:~ # cat out/davinci_log_info.txt
This message is from HelloDavinci
```

----End

8 Software Code Development

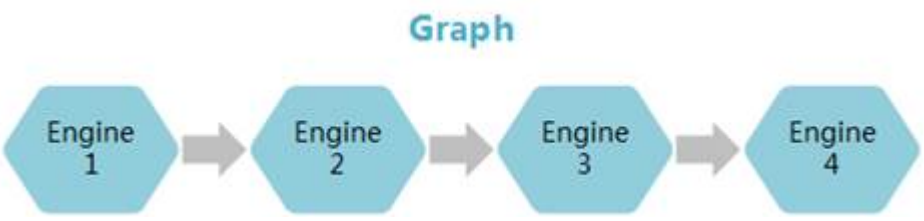
8.1 Configuring the Matrix Framework

Service Framework

To maximize the computing power of Ascend 310 chips, Huawei provides the Matrix framework to migrate inference services. The Matrix framework provides the following functions:

- Process orchestration
  a. An engine is defined as a basic functional unit in the process, and its implementation can be customized (for example, inputting image data, classifying images, and outputting the predicted class of images). By default, each engine corresponds to a thread on Ascend 310.
  b. A graph is defined to manage multiple engines. Each graph corresponds to a process on Ascend 310. **Figure 8-1** shows the relationship between the graph and engines.

  Figure 8-1 Relationship between the graph and engines

  In the graph configuration file, configure the serial connections of engine nodes and node properties (parameters required for node running). The

actual data flow direction is implemented on the nodes based on the specific service. The entire engine computation process is started by inputting data to the start node of the service.

- Media pre-processing

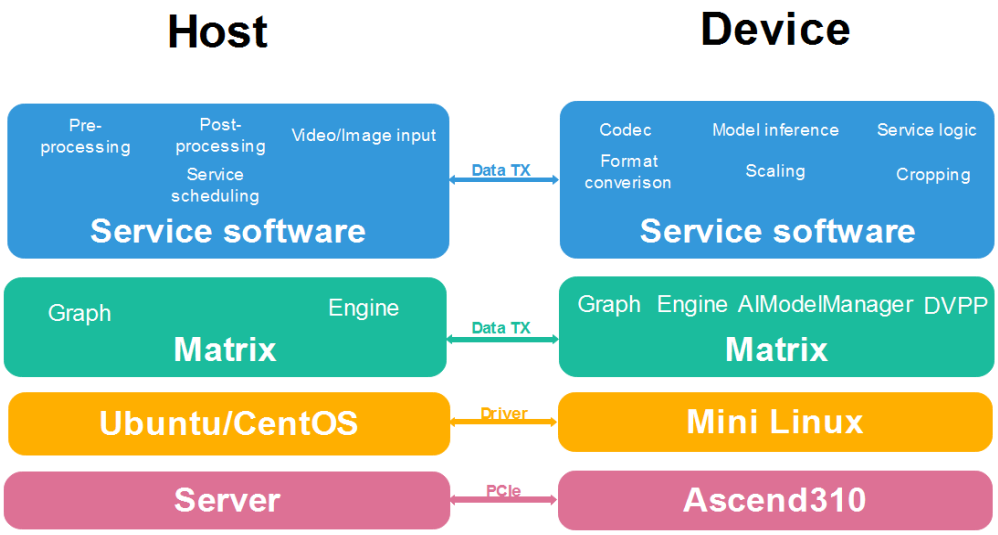
  Engines running on Ascend 310 can directly call the APIs provided by the DVPP to implement the media pre-processing capability.

- Offline model loading and running

  Engines running on Ascend 310 can directly call the APIs provided by the model manager (AIModelManger) to load offline models and perform inference.

  The following figure shows a user's service software structure based on the Matrix framework. The user creates a service flow (graph) that consists of custom engines. Engines running on the device side can call the APIs of the DVPP and AIModelManager to utilize Ascend 310-enabled media pre-processing and the hardware acceleration function of model inference. Engines on the host side are used to implement the service software logic and exchange data with the engines on the device side.

Figure 8-2 Service software framework



8.1.1 Configuring, Creating, and Destroying a Graph

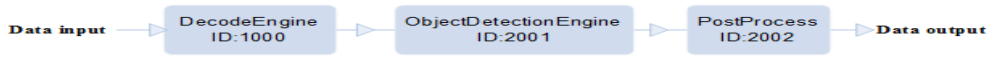
The graph describes the data transmission relationship between engines in a service. The Matrix framework defines the graph data structure in a protobuf file. You can define the graph configurations in the configuration file.

NOTE

For details about the graph keywords, see **10.1 Graph Keywords**. For more details, see the *Matrix API Reference*.

The following provides a simple graph configuration file, which will create a graph service flow with the ID of **1000**. The service flow contains three engines. **Figure 8-3** shows the data transmission relationship between the engines.

Figure 8-3 Data transmission relationship between engines




```
graphs {
    graph_id: 1000          # A single chip supports multiple graphs, whose IDs must be greater
than 0 and must be unique within the chip.
    device_id: "0"          # ID of the chip on which the graph is running
    priority: 1             # priority

    # user-defined engine, which can be instantiated into multiple engines. Multiple engines are
differentiated by ID.
    engines {
        id: 2001                            # engine ID
        engine_name: "ObjectDetectionEngine"     # class name of the user-defined engine
        side: DEVICE                        # whether the engine runs on the host or device
        so_name: "./libObjectDetectionEngine.so" # name and host-side path of the dynamic
library, which is copied by the Matrix framework to the device side
        # user-definable parameters
        ai_config{
            items{
                name: "model"               # model name
                value: "./FaceDetection.om"     # host-side path of the model, which is copied by the
Matrix framework to the device side
            }
            items{
                name: "mode"
                value: "test"
            }
        }
    }
    engines {
        id: 1000
        engine_name: "DecodeEngine"
        side: DEVICE
        so_name: "./libDecodeEngine.so"
    }
    engines {
        id: 2002
        engine_name: "PostProcess"
        side: HOST
    }

    # Describes interface connections between engines.
    connects {
        src_engine_id: 1000                 # source engine ID for data transmission
        src_port_id: 0                      # source port ID for data transmission
        target_engine_id: 2001              # target engine ID for data transmission
        target_port_id: 0                   # target port ID for data transmission
    }
    connects {
        src_engine_id: 2001
        src_port_id: 0
        target_engine_id: 2002
        target_port_id: 0
    }
}
```


Table 8-1 lists three frequently used graph APIs provided by the Matrix. For details, see the *Matrix API Reference*.

Table 8-1 API description

API	Description
HIAI_StatusT HIAI_Init(uint32_t deviceID)	Initializes an Ascend 310 chip. Note that the chip ID used must be the absolute number of the Ascend 310 chip. The chip ID queried on the Atlas 500 is fixed to **0**, while the chip ID queried using the **npu-smi** command is a relative number.
static HIAI_StatusT Graph::CreateGraph(const std::string& configFile)	Reads the graph configuration file, initializes engines, and creates threads and channels for data transmission to initialize the service flow.
static HIAI_StatusT Graph::DestroyGraph(uint32_t graphID)	Destroys the graph and runs the destructor functions of engines.

8.1.2 Configuring Engine

An engine is a basic functional unit of service software defined by the Matrix framework. Users inherit the engine template class defined by the Matrix framework and create engines for each functional module in services (such as reading input files, image pre-processing, neural network inference, inference result post-processing, and host/device data transmission). For details about the functional modules, see the *Matrix API Reference*. Each engine defines the **Init()** and **Process()** functions. When the graph is initialized, the **Init()** function is automatically executed to initialize engine parameters (including memory allocation and model loading). The **Process()** function implements data transmission and service logic.

- Table 8-2 describes the common APIs.

  Table 8-2 API description

API	Description	Remarks
HIAI_DEFINE_PROCESS (inputPortNum, outputPortNum)	Specifies the number of interfaces for inputting and outputting engine data.	The Matrix framework creates a queue for each interface to cache data. The transmission relationship between engines is specified in the graph configuration file. For details, see **10.1 Graph Keywords**.

API	Description	Remarks
HIAI_StatusT Engine::Init(const AIConfig &config, const vector<AIModelDescription>&modelDesc)	Initializes the engine.	During graph creation, this API is called to initialize engines. The **ai_config** value defined in the graph configuration file is transferred to the argument **config** of the function. You can add custom items to the configuration file and use them in the initialization function.
HIAI_IMPL_ENGINE_PROCESS(name, engineClass, inPortNum)	Indicates the **Process()** function of an engine, corresponding to a device-side thread.	The **Process()** function of the engine is driven by data on the device side. That is, after the input interface receives data, the framework starts the process. If multiple input interfaces are configured, each input interface triggers a process after receiving data. Therefore, if service processing depends on multiple inputs, you need to implement the synchronization logic of multiple inputs. The framework has encapsulated the **Process()** function into a macro definition for implementation.

- The engine reads data from the input interface. The framework supports a maximum of 16 input interfaces (arg0–arg15) that can be directly used. From the perspective of service applications, shared pointers are transmitted. The following shows the transmission code sample:

```
// Transmission engine: transmits the user-defined data USER_DEFINE_TYPE to the target engine.
std::shared_ptr<USER_DEFINE_TYPE > streamData= std::make_shared< USER_DEFINE_TYPE >();
// After a value is assigned to streamData, call Senddata to send the value. The shared pointer needs to be converted to the void type for data transmission.
hiai::Engine::SendData(0, " USER_DEFINE_TYPE ", std::static_pointer_cast<void>(deviceStreamData));
// Receiving engine: receives data and converts it to the user-defined type USER_DEFINE_TYPE.
std::shared_ptr< USER_DEFINE_TYPE > inputArg = std::static_pointer_cast< USER_DEFINE_TYPE >(arg0);
```

8.1.3 Configuring Data Transmission

Based on service applications, data transmission defined by the Matrix framework is classified into the following types (The used APIs are similar and the transmitted objects are shared pointers, but the API usage varies according to scenarios):

- Input API for transmitting data outside a graph to engines. For details, see "Graph::SendData" in the *Matrix API Reference*.

- Output API for transmitting data of engines in a graph to the outside of the graph. For details, see "Graph::SetDataRecvFunctor" and "Engine::SetDataRecvFunctor" in the *Matrix API Reference*.
- Data is transmitted between engines in a graph. For details, see "Engine::SendData" in the *Matrix API Reference*.

Data Transmission Between Engines

The Matrix framework divides service software into the software on the host side (x86/Arm server) and the software on the device side (Ascend 310 chip). Therefore, data transmission between engines is classified into cross-side transmission and intra-side transmission.

- Cross-side transmission: The data to be transmitted needs to be serialized into binary data. After the data is transmitted by using hardware such as PCIe or DMA, the data is deserialized into valid data. Therefore, you need to customize serialization and deserialization functions for custom data structures. The Matrix framework allows you to define serialization and deserialization functions using either a common or a high-speed interface. The common interface is applicable to data transmission at a rate below 256 kbit/s, whereas the high-speed interface is applicable to data transmission at a rate of 256 kbit/s or higher. The high-speed interface operates at a speed similar to that of a common interface when transmitting small memory blocks. For details, see "Data Type Serialization and Deserialization (C++ Language)" in the *Matrix API Reference*.
- Intra-side transmission: The transferred data is the address of the shared pointer and is not copied. As the **Process()** function of engines is instantiated into threads, intra-side transmission implements data transmission in memory-sharing mode. Ensure that there is no unauthorized access. The Matrix framework recommends that the local engine does not modify the shared pointer after transmitting it to the next engine.
- After the serialization and deserialization functions are implemented for cross-side transmission, the API used in the service application is the same as that used in intra-side transmission.

```
HIAI_StatusT Engine::SendData(uint32_t portId, const std::string& messageName,
const shared_ptr<void>& dataPtr, uint32_t timeOut = TIME_OUT_VALUE);
```

Input API for Transmitting Data Outside a Graph to Engines

- The service flow composed of engines is driven by data. This API allows data outside a graph to be sent to engines in the graph.
- The following APIs can be called to implement cross-side and intra-side transmission, with similar requirements to those for data transmission between engines. For details, see "Graph::SendData" in the *Matrix API Reference*.

```
HIAI_StatusT Graph::SendData(const EnginePortID& targetPortConfig, const
std::string& messageName, const std::shared_ptr<void>& dataPtr,const uint32_t
timeOut = 500)
```

Output API for Transmitting Data of Engines in a Graph to the Outside of the Graph

- The Matrix framework transmits the data of engines in a graph to the outside of the graph using either of the following callback functions:

  - Callback on the output API for an engine (node of a service flow). For details, see "Graph::SetDataRecvFunctor" in the *Matrix API Reference*.
  - For details about the callback on the output API in other scenarios, see "Engine::SetDataRecvFunctor" in the *Matrix API Reference*.
- The framework provides the template class **DataRecvInterface** of callback functions.
- The framework requires that the callback function and the output engine run on the same side. That is, only intra-side data transmission is supported.

8.2 Using DVPP APIs

DVPP is an image pre-processing hardware acceleration module provided by Ascend 310. This module integrates the following six functions. For details about the APIs and their usage, see the *DVPP API Reference*.

- Format conversion, image cropping, and scaling (by the VPC)
- H.264/H.265 video decoding (by the VDEC)
- H.264/H.265 video encoding (by the VENC)
- JPEG image decoding (by the JPEGD)
- JPEG image encoding (by the JPEGE)
- PNG image decoding (by the PNGD)

8.2.1 Using DVPP APIs

The DVPP provides the following three types of APIs by using handles, namely, APIs for creating, using, and destroying handles.

The VPC, JPEGE, JPEGD, and PNGD components share the same set of APIs with varying input parameters. VDEC and VENC each have a separate set of APIs.

- VDEC uses the following three APIs for decoding, which can be called in asynchronous mode. The **VdecCtl** API is used to transfer configurations including the callback function) and H.264/H.265 data. After hardware decoding, the Matrix framework calls the callback function to return the result.

  NOTE

  The data decoded by the VDEC is in HFBC format (internal format), which has to be converted to the YUV420SP format using the VPC. For details, see samples in "Implementing the VDEC Function" of the *DVPP API Reference*.

  ```
  int CreateVdecApi(IDVPPAPI *&pIDVPPAPI, int singleton)
  int VdecCtl(IDVPPAPI *&pIDVPPAPI, int CMD, dvppapi_ctl_msg *MSG, int singleton)
  int DestroyVdecApi(IDVPPAPI *&pIDVPPAPI, int singleton)
  ```
- The VPC, JPEGE, JPEGD, and PNGD use the same interfaces. The configuration parameters vary according to the function. For details, see "VPC/JPEGE/JPEGD/PNGD Interfaces" in the *DVPP API Reference*.

  ```
  int CreateDvppApi(IDVPPAPI *&pIDVPPAPI)
  int DvppCtl(IDVPPAPI *&pIDVPPAPI, int CMD, dvppapi_ctl_msg *MSG)
  int DestroyDvppApi(IDVPPAPI *&pIDVPPAPI)
  ```
- The DVPP is restricted by hardware during its usage. To speed up data read and write, an image's length and width must be aligned to the specified size

without affecting the valid region. The image's length and width are aligned to the specified size by padding **0**s to leftward and downward.

For example, for a 300 x 300 YUV420SP_UV image, the size must be aligned to 304 x 300 (The width is 16-pixel aligned, and the height is 2-pixel aligned). The valid region ranges from [0, 0] to [300, 300]. In this case, you need to pad **0**s rightward to column 304.

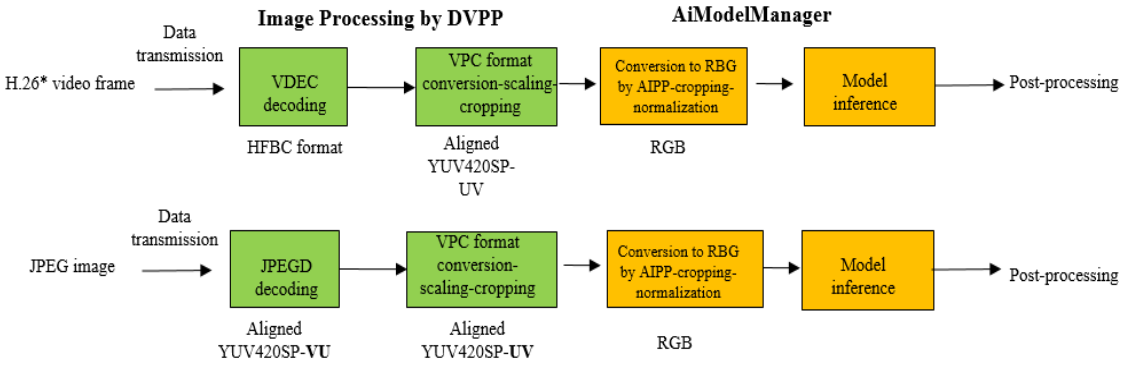
- When the JPEGD, VDEC, and PNGD components of the DVPP are used to read input images, the decoded images must meet the length and width alignment requirements. In this case, you need to apply for memory for output images based on the size of the aligned images.

  For example, for a 300 x 300 YUV420SP_UV image, you need to apply for memory with the size of (304*300*3/2) bytes. Each pixel of a YUV420SP image requires a 1.5-byte storage space.

  - VPC: input and output memory address aligned by 16 bytes
  - VPC: output image width aligned by 16 bytes
  - VPC: output image height aligned by two bytes
  - VPC: input image width aligned by 16 bytes
  - VPC: input image height aligned by two bytes
  - JPEGD: output image width aligned by 128 bytes
  - JPEGD: output image height aligned by 16 bytes
- DVPP components pose many restrictions on output images based on the processing speed and memory usage. For example, the length and width of output images must be aligned, and the output format must be **YUV420SP**. However, the model input is usually in RGB or BGR format, and the sizes of the input images are different. Therefore, the Ascend 310 chip provides AI pre-processing (AIPP) for image format conversion and image cropping. For details, see the *Model Conversion Guide*.

  Figure 8-4 shows the handling process of the JPEG image input and H.26* video input.

Figure 8-4 Handling process of video and image inputs



8.2.2 Applying for DVPP Memory

The Matrix framework provides the memory allocation API **HIAI_DVPP_DMalloc** and memory freeing API **HIAI_DVPP_DFree**. **HIAI_DVPP_DMalloc** is used to allocate memory that meets DVPP alignment requirements. The two APIs must be used in pairs. To prevent memory leakage, you are advised to use the shared pointer to manage the allocated memory. The implementation code is as follows:

```
uint8_t* buffer = nullptr;
HIAI_StatusT ret = hiai::HIAIMemory::HIAI_DVPP_DMalloc(dataSize,  (void*&) buffer);
std::shared_ptr<uint8_t> dataBuffer =  std::shared_ptr<uint8_t>( buffer, \
[](std::uint8_t* data) hiai::HIAIMemory::HIAI_DVPP_DFree(data);});
```

In addition, these APIs are available only on the device side. The memory allocated by **HIAI_DVPP_DMalloc** can be used for high-speed data transmission from the device to host. In this mode, the memory is freed by the Matrix framework. For details, see the sample code **DvppDecodeResize**.

8.3 Offline Model Inference

The Ascend 310 chip is capable of accelerating inference under the Caffe and TensorFlow frameworks. After model training is complete, you need to convert the trained model to the model file (.om file) supported by Ascend 310, compile service code, and call APIs provided by the Matrix framework to implement service functions.

The Matrix framework encapsulates the AI pre-processing (AIPP) and model inference functions into a module. After the inference API is called, the Matrix framework calls the AIPP to pre-process input images, inputs the pre-processed images into the model inference module, and returns the inference result.

8.3.1 Configuring AIPPs

AIPP is a hardware image pre-processing function provided by Ascend 310. The pre-processing includes CSC, image normalization (by subtracting the mean value or multiplying a coefficient), image cropping (by specifying the start point of cropping and cropping the image to the size required by the neural network), and much more.

AIPP supports static and dynamic modes.

- Static AIPP: In this mode, parameters are set during model conversion. The model inference process uses fixed AIPP pre-processing (cannot be modified). For details about the keywords and configuration file template of static AIPP, see "Configuration File Template" in the *Model Conversion Guide*.
- Dynamic AIPP: During model conversion, AIPP is set to dynamic mode. Before model inference, set AIPP pre-processing parameters as required. Dynamic AIPP is used when pre-processing parameters have to be changed based on service requirements. For example, cameras use different normalization parameters, and the input image format must be compatible with YUV420 and RGB. For details about how to use dynamic AIPP, see "AIPP Configuration APIs" in the *Matrix API Reference*.

NOTE

- The input format of AIPP is **YUV420SP_U8** (the default format is **YUV420SP_UV**). If the format is **YUV420SP_VU**, change the value of **rbuv_swap_switch**. Otherwise, the output result will be affected.
- The model input is in either **RGB_U8** or **BGR_U8** format, corresponding to different color gamut conversion matrices.

Images output by the DVPP module are the aligned YUV420SP images. The output does not support the RGB format. Therefore, the service flow needs to use converted and aligned YUV420SP images of the AIPP module, and crop the image to the size in line with the model input.

For example, the model requires the input of a 300 x 300 RGB image. After DVPP APIs are called for processing (such as JPEG decoding and scaling), the DVPP module outputs a 384 x 304 image YUV420SP_UV image (the valid region size is 300 x 300, and **0**s are padded to rightward and downward.

The following shows the static AIPP configuration. The file configures the start coordinates of the image to be cropped. The length and width of the image to be cropped are set according to the model input by default. The image normalization parameters are the mean value and the reciprocal of the variance (The final value is obtained by subtracting the mean value and multiplying this coefficient).

```
aipp_op{
# Sets AIPP to static mode.
aipp_mode: static
# Enables image cropping.
crop: true
# Sets the format and size for an input image.
input_format : YUV420SP_U8
src_image_size_w : 384
src_image_size_h : 304
# Sets the start coordinates for cropping. The width and height of the cropped region are in line with the model input by default.
load_start_pos_h : 0
load_start_pos_w : 0

# Enables format conversion. The conversion matrix converts YUV420SP_UV to RGB888.
csc_switch : true
matrix_r0c0 : 298
matrix_r0c1 : 516
matrix_r0c2 : 0
matrix_r1c0 : 298
matrix_r1c1 : -100
matrix_r1c2 : -208
matrix_r2c0 : 298
matrix_r2c1 : 0
matrix_r2c2 : 409
input_bias_0 : 16
input_bias_1 : 128
input_bias_2 : 128

# Enables data normalization and configure the mean value and the reciprocal of variance.
mean_chn_0 : 125
mean_chn_1 : 125
mean_chn_2 : 125
var_reci_chn_0 : 0.0039
var_reci_chn_1 : 0.0039
var_reci_chn_2 : 0.0039
}
```


8.3.2 Converting an Offline Model

The Ascend 310 chip is capable of accelerating inference under the Caffe and TensorFlow framework models. During model conversion, operator scheduling tuning, weight data rearrangement, quantization compression, and memory usage tuning can be implemented, and model preprocessing can be completed without using devices. After model training is complete, you need to convert the trained model to the model file (.om file) supported by Ascend 310, compile service code, and call APIs provided by the Matrix framework to implement service functions.

The offline model conversion tool is stored in the DDK in the **<$DDK_HOME>/uihost/bin/omg** directory. The offline model generator (OMG) tool is a CLI tool (parameters can be obtained by using the **-h** command). It is used to convert models under Caffe and TensorFlow frameworks into .om files supported by Ascend 310. For details about how to use the OMG tool, see "Model Conversion Using OMG" in the *Model Conversion Guide*.

1. Caffe model conversion:
```
#omg --framework 0 --model <model.prototxt> --weight <model.caffemodel> --output <output name> --insert_op_conf <aipp.cfg>
```
2. TensorFlow model conversion:
```
#omg --framework 3 --model <model.pb> --input_shape "input_name:1,112,112,3" --output <output_name> --insert_op_conf <aipp.cfg>
```

Table 8-3 Parameter description

Parameter	Description
framework=0	Indicates a Caffe model.
framework=3	Indicates a TensorFlow model.
model	Specifies a model file.
weight	Specifies a weight file for the Caffe model.
output	Specifies a name of the .om file.
input_shape	Specifies the name and size of the input layer. The default value for TensorFlow models is **input_layer_name: n, h, w, c**.
insert_op_conf	Specifies an AIPP configuration file.

8.3.3 Performing Model Inference

The Matrix framework provides the **AIModelManager** class to implement model loading and inference. For details, see the *Matrix API Reference*.

Model Inference Initialization

Step 1 Set the model path in the graph configuration file of the custom inference model engine (add items to **ai_config** and set the model path on the host).

Step 2 Use the Matrix framework to transfer the model file to the device.

Step 3 Parse the custom items in the custom engine to obtain the path of the model on the device side.

Step 4 Call **AIModelManager::Init()** to complete the initialization. The implementation is as follows:

```
/* Define member variables of a custom engine. */
std::shared_ptr<hiai::AIModelManager> modelManager;
/* Implement AIModelManager initialization with the Init function of the custom engine. */
std::vector<hiai::AIModelDescription> model_desc_vec;
hiai::AIModelDescription model_desc_;
......
/* Parse the model path from the ai_config structure in the graph configuration file. */
model_desc_.set_path(model_path);// Set the model path.
model_desc_vec.push_back(model_desc_);
ret = modelManager->Init(config, model_desc_vec);// If configurations are meaningless, input the arguments of Engine::Init.
```

----End

Setting the Input and Output of Model Inference

- The Matrix framework defines the **IAITensor** class for managing the input and output matrices of model inference. For ease of use, the Matrix framework derives **AISimpleTensor** and **AINeuralNetworkBuffer** based on the **IAITensor** class.
- Memory for the model inference input and output is allocated by calling **HIAI_DMalloc**, which reduces memory copy.
- Even though the Matrix framework can automatically free the memory managed by **AISimpleTensor**, you are advised to apply for and free the memory by yourself to prevent memory leakage or repeated freeing.
- During model conversion, if the functions such as image cropping, format conversion, and image normalization of the AIPP module are enabled, the input data must be processed by the AIPP module before it is used for model inference.

Input and Output Implementation

The code for input and output implementation is as follows:

```
/* Obtain the descriptions of input and output tensors of the inference model. */
std::vector<hiai::TensorDimension> inputTensorDims;
std::vector<hiai::TensorDimension> outputTensorDims;
ret = modelManager->GetModelIOTensorDim(modelName, inputTensorDims,
outputTensorDims);

/* Set the input. If there are multiple inputs, create and set them in sequence. */
std::shared_ptr<hiai::AISimpleTensor> inputTensor =
std::shared_ptr<hiai::AISimpleTensor>(new hiai::AISimpleTensor());
inputTensor->SetBuffer (< memory address of the input data >, < length of the input data >);
inputTensorVec.push_back(inputTensor);

/* Set the output. */
for (uint32_t index = 0; index < outputTensorDims.size(); index++) {
hiai::AITensorDescription outputTensorDesc = hiai::AINeuralNetworkBuffer::GetDescription();
uint8_t* buf = (uint8_t*)HIAI_DMalloc(outputTensorDims[index].size);
......
```



```
std::shared_ptr<hiai::IAITensor> outputTensor = hiai::AITensorFactory::GetInstance()-
>CreateTensor(
outputTensorDesc, buf, outputTensorDims[index].size);
outputTensorVec.push_back(outputTensor);
}
```

Model Inference

- The Matrix framework supports either synchronous inference and asynchronous inference. Synchronous inference is used by default. You can set the **AIContext** configuration item and call the callback function to implement asynchronous inference.

  ```
  /* Model inference */
  hiai::AIContext aiContext;
  HIAI_StatusT ret = modelManager->Process(aiContext, inputTensorVec, outputTensorVec,
  0);
  ```

- If the **AIModelManager** object loads multiple models, you can set the **AIContext** configuration item to set model parameters (initialization phase and model name). For details, see "Offline Model Manager" in the *Matrix API Reference*.

Model Inference Post-Processing

The result matrix of model inference is stored in the **IAITensor** object as the memory+description information. You need to parse the memory information into valid output based on the actual output format (data type and data sequence) of the model.

```
/* Parse the inference result. */
for (uint32_t index = 0; index < outputTensorVec.size(); index++) {
shared_ptr<hiai::AINeuralNetworkBuffer> resultTensor =
std::static_pointer_cast<hiai::AINeuralNetworkBuffer>(outputTensorVec[i]);
// resultTensor->GetNumber()  -- N
// resultTensor->GetChannel() -- C
// resultTensor->GetHeight()  -- H
// resultTensor->GetWidth()   -- W
// resultTensor->GetSize() -- memory size
// resultTensor->GetBuffer() -- memory address
}
```

For details about the post-processing of common classification models, see the sample code **InferClassification**. For details about the post-processing of the SSD object detection model, see the sample code **InferObjectDetection**.

8.4 Commissioning Software Logs

8.4.1 Configuring Log System

Configuring Log Levels

The framework provides five log levels: error > warning > info > debug > event.

Event logs record the most critical logs of the system and must be set separately.

For the other four log levels, logs of the specified level and higher can all be printed.

Step 1 Enter the developer mode by following the instructions provided in **4.8 Accessing the Atlas 500 Development Mode**.

Step 2 Run the **vi /etc/slog.conf** command to open the log configuration file. Its default content is as follows:

```
# Global log level
global_level=3

# User
user=HwHiAiUser

# Share memory size of node 512k * 32 biggest support
maxNodeSize=524272

# Share memory count of queue
maxQueueCount=40

# log-agent-host #
logAgentMaxFileNum=8
# set host one log file max size, range is (0, 104857600]
logAgentMaxFileSize=10485760
# set host log dir
logAgentFileDir=/var/dlog
...
```

Step 3 Change the **global_level** value in the log configuration file to change the log level. The **global_level** value corresponding to each log level is as follows. After the modification, save the **/etc/slog.conf** file.

0: debug

1: info

2: warning

3: error

Step 4 Restart the OS for the modifications to take effect.

----End

8.4.2 Viewing Logs

The log files of the Atlas 500 are in the **/var/dlog** directory.

- The log files prefixed by **host-** are log files on the host (Hi3559A).
- The files prefixed by **device-** are the log files on the device (Ascend 310).

Procedure

Step 1 Access the development mode of the Atlas 500 by referring to **4.8 Accessing the Atlas 500 Development Mode**.

Step 2 Go to the **/var/dlog** directory.

Step 3 Check whether the value of **zip_switch** is **0** before viewing logs.

In the **/etc/slog.conf** file, check the value of **zip_switch**. If the value is not **0**, open the **/etc/slog.conf** file as the **root** user, change the value of **zip_switch** to **0**, and save the file.

Step 4 Restart the Atlas 500 OS for the modification to take effect.

You can use the editor to view logs generated after **zip_switch** is set.

----End

8.4.3 Log API Usage

Defining the Log Module

Step 1 Define an ID for a log module. The value must be unique.

```
#define USER_DEFINE_ERROR 0x6001
```

NOTE

The **include/inc/hiaiengine/status.h** file in the Atlas 500 DDK installation directory contains some module IDs. Ensure that new module IDs do not conflict with those defined in the file.

Step 2 Define the error code enumeration of the log module. If there are multiple errors, define multiple error codes in the enumeration.

```
typedef enum
{
USER_DEFINE_OK_CODE,
USER_DEFINE_INVALID_CODE
}
USER_DEFINE_CODE;
```

Step 3 Register the error codes defined in **Step 2** with **HIAI_DEF_ERROR_CODE** (macro definition). **moduleId** corresponds to the module ID defined in **Step 1**. **loglevel** is the log level registered for the error code. The value can be **HIAI_ERROR**, **HIAI_INFO**, **HIAI_DEBUG**, or **HIAI_WARNING**. **codeName** corresponds to the content of the enumeration ID (excluding **_CODE**) of the error code in **Step 2**. **codeDesc** indicates the error code description. The following describes the implementation of the macro definition and use sample:

```
HIAI_DEF_ERROR_CODE(moduleId, logLevel, codeName, codeDesc)
HIAI_DEF_ERROR_CODE(USER_DEFINE_ERROR,HIAI_ERROR,USER_DEFINE_OK,"OK")
```

----End

Exporting Logs to the .dlog Log File

The Atlas 500 service software can call **HIAI_ENGINE_LOG** to output logs to the .dlog log file. For details, see "Logs (C++)" in the *Matrix API Reference*. There are eight log formats. This section describes one of them based on the log registration content. The definition and use sample are as follows:

- The function format is as follows. For details about the parameters, see **Table 8-4**.

  ```
  #define HIAI_ENGINE_LOG(...) \
  HIAI_ENGINE_LOG_IMPL(__FUNCTION__, __FILE__, __LINE__, ##__VA_ARGS__)
  void HIAI_ENGINE_LOG_IMPL(const char* funcPointer, const char* filePath, int lineNumber,
  const uint32_t  errorCode, const char* format, ...);
  ```


Table 8-4 Parameter description

Parameter	Description
errorCode	Error code
format	Log description
...	This is a variable parameter in the format and is added based on the log content.

- **HIAI_ENGINE_LOG** is called as follows:
```
HIAI_ENGINE_LOG(HIAI_INVALID_INPUT_MSG, "RUNNING OK");
```

8.5 Service Software Compilation

A complete program running on the Atlas 500 can be divided into two parts, which are running on the Atlas 500 host CPU (Hi3559A) and the Atlas 500 device CPU (Arm Cortex-A55), respectively.

The Atlas 500 does not provide compilation toolchains running on the host or device CPU. Therefore, cross compilation is required. The methods of cross-compiling software for the Atlas 500 host and device are similar. They differ only in the used tools and link libraries. This document describes the method of cross-compiling the host software and device software. To improve software compilation efficiency, a build automation tool is used. The Atlas 500 supports software building using CMake, which is a cross-platform build tool with simple syntax and strong portability. The Atlas 500 also supports software compilation using CMake.

8.5.1 Compiling Service Software Using CMake

Step 1 Run the **sudo apt install cmake** command on the development host to install the CMake tool.

Step 2 Run the **cmake -version** command to check the CMake version. The version must be later than **2.8.4**.

Step 3 Create a CMake compilation project by referring to the settings of the Atlas 500 sample project.

Step 4 Copy the **CMake** directory in the root directory of the Atlas 500 sample project to the upper-level directory in the home directory of the target project.

Step 5 Copy the **build.sh** file in a subproject (for example, HelloDavinci) of the Atlas 500 sample project to the home directory of the target project and modify the file.

Step 6 Delete **source $path_cur/../Common/scripts/build_tools.sh** from the file.

Step 7 Go to the root directory of the project and run the following command to compile the program. After the compilation, check the generated target program in the **out** folder under the root directory.

export DDK_HOME=*Atlas 500 DDK installation directory*&&bash build.sh A500

Step 8 Copy the **out** folder to the Atlas 500 host for running.

----End

9 Software Packaging and Deployment

9.1 Importing the Base Image

Prerequisites

Ensure that the Docker program has been installed in the packaging environment.

NOTE

The **dockerfile** command is a built-in command of Docker. Therefore, Docker 18.09 that matches the Atlas 500 is recommended.

Procedure

Step 1 In the development environment, go to the directory where **A500-3000_A500-3010-EulerOSx.x.x.xxx_64bit_aarch64_basic.tar.gz** is located.

In **A500-3000_A500-3010-EulerOS*x.x.x.xxx*_64bit_aarch64_basic.tar.gz**, *x.x.x.xxx* indicates the version number.

Step 2 Run the following commands to decompress the **A500-3000_A500-3010-EulerOS*x.x.x.xxx*_64bit_aarch64_basic.tar.gz** package and obtain the **Atlas500_EulerOS*x.x.x.xxx*_64bit_aarch64_basic.tar.gz** package:

tar -mxvf A500-3000_A500-3010-EulerOSx.x.x.xxx_64bit_aarch64_basic.tar.gz

Step 3 Run the following command to import the EulerOS base image:

docker load -i Atlas500_EulerOSx.x.x.xxx_64bit_aarch64_basic.tar.gz

Step 4 Run the following command to view the imported images:

docker images

Step 5 Run the following command to rename the base image as **euler**:

docker tag atlas500_eulerosx.x.x.xxx_64bit_aarch64_basic euler

atlas500_euleros*x.x.x.xxx***_64bit_aarch64_basic** indicates the image name obtained in **Step 4**.

----End

9.2 Creating an Image

Step 1 Run the **vi Dockerfile** command in the project directory to generate and compile the **Dockerfile** file. The content is as follows:

```
FROM euler                          #Specify the renamed base image euler.
WORKDIR /app                        #Specify the working directory app.
COPY --chown=1001:1001 out /app/    #Copy the generated program to the /app directory. By default, images are deployed on the management platform as the HwHiAiUser user. Add --chown=1001:1001.
ENTRYPOINT [./main"]                #Specify the container boot program.
```

Step 2 Run the following command to create an image:

docker build -t myapp .

Step 3 Run the following command to export the image:

docker save myapp -o myapp.tar

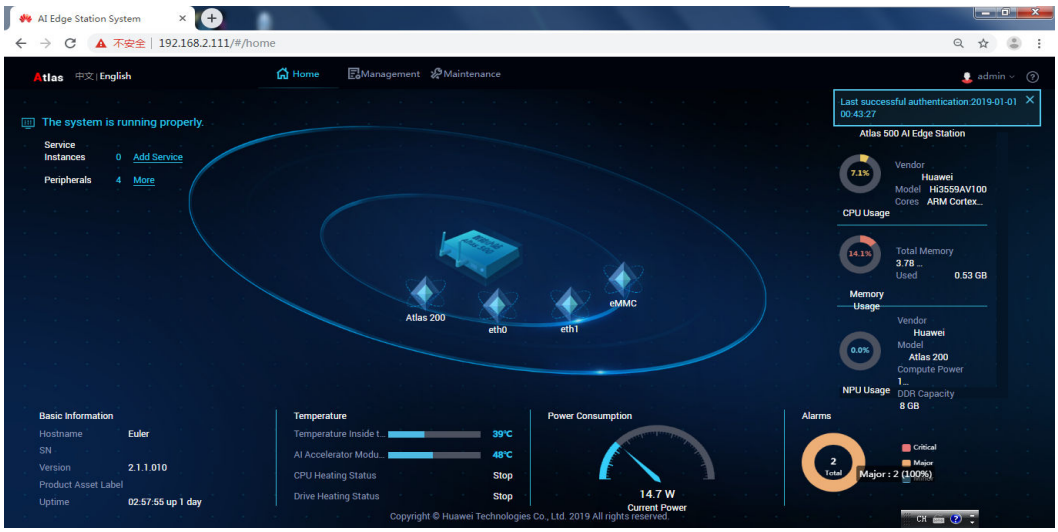
----End

9.3 Deploying the Image

9.3.1 Deployment Using the IES

Step 1 Open the browser, enter https://*Atlas 500 IES IP address* in the address box and press **Enter** and enter the user name and password to log in to the Atlas 500 IES. The default user name is **admin**, and the default password is **Huawei12#$**. **Figure 9-1** shows the home page.

Figure 9-1 Atlas 500 IES



Step 2 Choose **Management** > **Service Software** > **Service Instances** > **Add Service**. The page for service instance creation is displayed in the right pane.

Step 3 Enter basic information, such as the service instance name and its description.

Step 4 Set container information as follows:

- **Container Image File**: Click ... to upload a file (for example, **myapp.tar** exported in **9.2 Creating an Image**).

  NOTE

  The file size (including the size after decompression) cannot exceed 512 MB. The file can be in the *.tar or *.tar.gz format.

- Click ... on the right of **Additional Configuration File** to upload a file and mount the file to the container directory.

  NOTE

  An additional configuration file indicates the directory where the configuration and data files of a container service are located. Generally, an additional configuration file is used to store inference model files, images, and video data.

  The file size cannot exceed 512 MB. The file can be in the *.tar or *.tar.gz format.

- **Resource Restriction**
  - **CPU**: Select this option and enter the maximum CPU usage for a container.
  - **Memory**: Select this option and enter the maximum memory capacity for a container.
  - **AI Compute Power**: Select the option to allow the container to use AI compute power.

Step 5 Set environment variables, including the variable name and value.

- To delete an environment variable, click **Delete** in the **Operations** column.
- To add an environment variable, click ⊕.

  NOTE

  1. System environment variables can be configured in the container running environment and can be modified even after the service instance is deployed.
  2. The values of environment variables are displayed in plaintext. Do not enter sensitive information. If sensitive information is involved, encrypt it to prevent information leakage.

Step 6 Configure drive partitions.

1. Click ⌄ under **Drive Partition Name** and select a drive partition name to view the total capacity (GB) and available capacity (GB). Set the host mount point of the drive partition and container mount directory, and select the permission.
2. To delete the drive partition, click **Delete** in the **Operations** column.
3. Click ⊕ to mount the drive partition.

   NOTE

   Set the local drive partition that is mounted to the container to implement persistent data file storage.

Step 7 Configure the container restart policy and container service network.

1. The container restart policies include:
   a. **Restart Upon Failure**: The system restarts the container only if the container is exited abnormally.
   b. **Not Restart**: The system does not restart the container regardless of whether the container is exited normally or abnormally.
   c. **Always Restart**: The system restarts the container regardless of whether the container is exited normally or abnormally.
2. The available container service network configurations include:
   a. Host network: Configure the container to directly use the host network.
   b. Port mapping: Map host ports to container ports.
3. To delete a port mapping, click **Delete** in the **Operations** column.
4. To add a port mapping, click  .

Step 8 Check the configuration and click **OK** to start the deployment. After the deployment is complete, the new service is displayed in the list.

NOTICE

- The deployment requires a long time. Do not close the page before the deployment result is displayed. Otherwise, the deployment will fail.

----End

9.3.2 Deployment Using the CLI

Step 1 Upload the image file (for example, **myapp.tar** exported in **9.2 Creating an Image**) to the Atlas 500.

Step 2 Run the following command on the Atlas 500 to import images:

docker load –i myapp.tar

Step 3 Start deployment.

The following device parameters need to be added for the Atlas 500 to load the Atlas 200 AI accelerator module:

```
docker run \
-it \
--device=/dev/davinci_manager \
--device=/dev/hisi_hdc \
--device=/dev/davinci0 \
myapp
```

----End

10 Appendix

10.1 Graph Keywords

Parameter		Description	Mandatory (M) or Optional (O)
graph_id	-	Graph ID, which is a positive integer	O
priority	-	Priority, which does not need to be adjusted	O
device_id	-	Device ID. Different graphs can run on one or more chips or on the chips of different PCIe cards. If they run on different chips, the **device_id** field has to be added to the graph configuration file to specify the device ID on which the graphs are running. If **device_id** is not specified, the graphs are running on the chips of device 0 by default. The value of **device id** ranges from **0** to **N-1** (**N** indicates the number of devices).	O
engines	id	Engine ID	M
	engine_name	Engine name	M
	side	Target for the engine to run. The value can be **HOST** or **DEVICE**, which is determined based on service requirements.	M

Parameter		Description	Mandatory (M) or Optional (O)
NOTE You are advised to configure multiple engines for multi-channel decoding. One engine corresponds to one thread. If one engine corresponds to multiple threads, the sequence cannot be ensured during decoding.	so_name	During engine running, you need to copy the dynamic library file in .so format from the host to the device. If FrameworkerEngine running depends on a third-party library file or user-defined library file, the dependent .so file must be configured in the graph file. (Replace **xxx** in the following example with the actual name of the dependent library.) so_name: "./libFrameworkerEngine.so" so_name: "./libxxx.so" so_name: "./libxxx.so"	O
	thread_num	Number of threads. You are advised to set this parameter to **1** for multi-channel decoding. If the value of **thread_num** is greater than **1**, the decoding sequence of threads cannot be ensured.	M
	thread_priority	Thread priority. The value ranges from **1** to **99**. This parameter is used to set the priority of the data processing thread corresponding to the engine. A higher priority is configured for the corresponding engine according to the SCHED_RR scheduling policy.	O
	queue_size	Queue size. The default value is **200**. The parameter value needs to be adjusted based on the service load fluctuation, size of data received by the engine, and system memory.	O

Parameter		Description	Mandatory (M) or Optional (O)

	ai_config	Configuration example:
		ai_config{
		items{
		name: "model_path"
		value: "./test_data/model/resnet18.om"
		}
		}
		• **name** does not need to be set.
		• Set the value to the path of the model file, including the file name. The file name can contain only digits, letters, underscores (_), and dots (.). Alternatively, you can set the value to the path of a single model file, for example, **./test_data/model/resnet18.om**. You can also compress the model file into a .tar package and set the parameter to the path **./test_data/model/resnet18.tar** where the .tar package is stored. If there are multiple AI configuration items, ensure that files with the same name but in different formats (for example, **./test_data/model/test.zip** and **./test_data/model/test.tar**) do not exist in the same directory.
			O

	ai_model	Configuration example:
		ai_model{
		name: "" // model name
		type:"" // model type
		version:"" // model version
		size:"" // model size
		path:"" // model path
		sub_path:"" // auxiliary model path
		key:"" // model key
		sub_key: "" // auxiliary model key
		frequency:"UNSET" // device frequency. Value **UNSET** or **0** indicates that the setting is canceled. Value **LOW** or **1** indicates low frequency. Value **MEDIUM** or **2** indicates medium frequency. Value **HIGH** or **3** indicates high frequency.
		device_frameworktype:"NPU" // type of the device on which a model is running. The value is **NPU**, **IPU**, **MLU**, or **CPU**.
		framework: "OFFLINE" // execution framework. The value is **OFFLINE**, **CAFFE**, or **TENSORFLOW**.
		}
			O

Parameter		Description	Mandatory (M) or Optional (O)

	oam_config	Mode of dumping algorithm data. If the inference is inaccurate, you can view the algorithm execution results of some or all layers.
		Configuration example:
		oam_config{
		items{
		model_name: "" // relative path+model name +suffix
		is_dump_all: "" // whether to dump all data. The value is **true** or **false**.
		layer: "" // layer
		}
		dump_path:"" // dump path
		}
			O

	internal_so_name	Dynamic library file embedded on the device. You can directly use the file without copying it from the host to the device.	O
	wait_inputdata_max_time	Maximum timeout of waiting for the next piece of data after the current data is received ● Unit: millisecond ● The default value is **0**, indicating that the wait does not time out.	O
	is_repeat_timeout_flag	Whether to perform cyclic timeout processing (wakeup) when the engine does not receive data. This parameter is used together with **wait_inputdata_max_time**. The options are as follows: ● 0: Cyclic timeout processing is not performed. The default value is **0**. ● 1: Cyclic timeout processing is performed.	O
	holdModelFileFlag	Whether to retain the model file of the engine. The options are as follows: ● 0: no ● Other values: yes	O
connects	src_engine_id	ID of the source engine	M

Parameter		Description	Mandatory (M) or Optional (O)

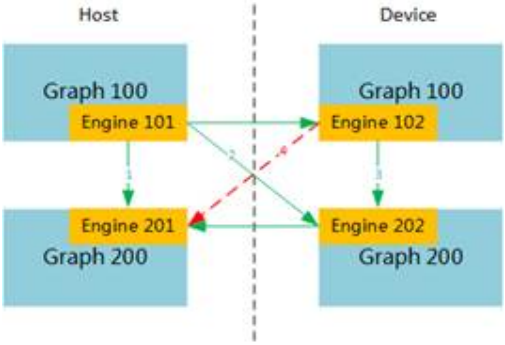
	src_port_id	Sending port number of the source engine.
		The port number starts from 0. The number of ports is defined by the **HIAI_DEFINE_PROCESS** macro in the corresponding header file. The implementation code is as follows:
		#define SOURCE_ENGINE_INPUT_SIZE 1
		#define SOURCE_ENGINE_OUTPUT_SIZE 1
		class SrcEngine : public Engine {
		
		HIAI_DEFINE_PROCESS(SOURCE_ENGINE_INPUT_SIZE, SOURCE_ENGINE_OUTPUT_SIZE)
		}
			M

	target_engine_id	ID of the target engine	M
	target_port_id	Receive port number of the target engine	M

Parameter		Description	Mandatory (M) or Optional (O)
	target_graph_id	ID of the target graph. Engines can be connected across graphs. In this scenario, the **target_graph_id** field needs to be added to the connection configuration file to indicate the graph ID at the receive end. If **target_graph_id** is not specified, engines are connected within the same graph by default. When multiple Ascend 310 chips are available, you can enable models to run on different Ascend 310 chips by connecting engines across graphs. Therefore, you are advised to place the same type of models on the same chip for inference, which avoids unnecessary memory consumption. Scenario description: • As shown in **Figure 10-1**, engines support inter-graph serial connection in the following scenarios: 1. 1: The host engine of graph A directly sends data to the host engine of graph B. 2. 2: The host engine of graph A directly sends data to the device engine of graph B. 3. 3: The device engine of graph A directly sends data to the device engine of graph B. • Engines do not support inter-graph serial connection in the following scenarios: 4: The device engine of graph A directly sends data to the host engine of graph B, as shown in **Figure 10-1**.	O

Parameter		Description	Mandatory (M) or Optional (O)
	receive_memory_without_dvpp	• The default value is **0**, indicating that the RX memory of the target engines running on the device side must meet the address space limit of 4 GB. • If this parameter is set to **1**, the target engines running on the device side does not need to meet the address space limit of 4 GB. • In the graph configuration file, you can set the **receive_memory_without_dvpp** parameter (under the **connects** property) to **1** for all target engines running on the device. Alternatively, you can set the **receive_memory_without_dvpp** parameter (under the **connects** property) to **1** for all target engines, because this parameter setting does not affect the target engines running on the host. In this way, the Matrix RX memory pool is not limited by the 4 GB address space, which improves the memory utilization. **NOTE** In the current DVPP, the input memory for the VPC, JPEGE, JPEGD, and PNGD must meet the address space limit of 4 GB, which is not mandatory for the input memory of the VDEC and VENC.	O

Figure 10-1 Engine connection across graphs



10.2 Change History

Release Date	Description
2020-05-30	This issue is the first official release.

