Sophgo sophon-stream UserGuide

English | [简体中文](Sophon_Stream_User_Guide.md)

Catalogs

1. Quick Start

1.1 Installation and Configuration

1.1.1 x86/arm PCIe Platform

If you have installed a Deep Learning Accelerator Card (such as the SC series Deep Learning Accelerator Card) on the x86/ARM platform, you can use it directly

as your development and runtime environment. You will need to install libsophon, sophon-opencv, and sophon-ffmpeg. For specific steps, please refer to [the installation and configuration guide for x86-pcie platform](EnvironmentInstallGuide.md#3-x86-pcie平台的开发和运行环境搭建) or [installation and configuration guide for arm-pcie platform](EnvironmentInstallGuide.md#5-arm-pcie平台的开发和运行环境搭建).

1.1.2 Soc Platform

If you are using an SoC platform (such as SE or SM series edge devices), after flashing the firmware, the corresponding libsophon, sophon-opencv, and sophon-ffmpeg runtime libraries are pre-installed in `/opt/sophon/`. You can use them directly as your runtime environment. Typically, you will also need an x86 host machine as your development environment for cross-compiling C++ programs.

1.2 Compilation Commands

Please note that if you do not have the boost library installed on your host machine, you need to use the following command for installation.

```bash
sudo apt-get update
sudo apt-get install libboost-all-dev
```

Once the environment setup is complete, users can refer to the [sophon-stream compilation guide](./HowToMake.md) and use the following command for compilation.

* It's important to note that the compilation command should be executed in the sophon-stream directory.

1.2.1 x86/arm PCIe Platform

```bash
mkdir build
cd build
cmake ..
make -j
```

1.2.2 Soc Platform

Usually, when cross-compiling programs on an x86 host, you need to set up a cross-compilation environment using SOPHON SDK on the x86 host. You should package the program's required header files and library files into the 'sophon_sdk_soc' directory. For specific instructions, please refer to the [sophon-stream compilation guide](./HowToMake.md). This example relies on the 'libsophon,' 'sophon-opencv,' and 'sophon-ffmpeg' runtime library packages.

```bash
mkdir build
cd build
cmake .. -DTARGET_ARCH=soc -DSOPHON_SDK_SOC=${path_to_sophon_soc_sdk}
make -j
```

Note: When cross-compiling, the variable ${path_to_sophon_soc_sdk} should point to the arm sophon-sdk directory on the x86 host where you are running the cross-compilation command.

1.3 Compiation Results

In sophon-stream, every module, except for 'sample,' participates in the runtime as a plugin. After completing the [1.2 Compilation Commands](#12-compilation-commands), users can find dynamic library files corresponding to each compiled plugin in the ./build/lib/ directory.

The source files in the 'samples' directory result in executable programs. For example, the executable program for the 'yolox' sample can be found at [yolox](../samples/yolox/build/yolox_demo).

2. Overview

sophon-stream is a data stream processing tool designed for the Sophon development platform. This software is built on a modular concept and has been developed using C++11 to create a pipeline framework that supports concurrent processing of multiple data streams. Leveraging existing interfaces, sophon-stream offers users the advantages of ease of use and ease of secondary development, greatly simplifying the complexity of configuring projects or adding plugins. Sophon-stream is based on SophonSDK, allowing it to fully harness the hardware's encoding and decoding capabilities as well as the inference capabilities of artificial intelligence algorithms, thereby achieving higher performance.

2.1 Advantages of sophon-stream

sophon-stream has the following advantages:

- Robust and flexible foundational framework. While ensuring the robustness of the sophon-stream foundational framework, it also offers considerable flexibility. Users can easily configure JSON files to accurately and conveniently build complex business pipelines.
- Comprehensive software-hardware ecosystem. Based on the underlying characteristics of Sophon chips, sophon-stream includes hardware acceleration for encoding, conventional image processing acceleration, and inference acceleration, effectively leveraging the performance advantages of Sophon chips and significantly improving overall throughput efficiency.
- Rich algorithm library. sophon-stream supports various object detection and tracking algorithms, such as yolox, yolov5, bytetrack, and more.
- Easy deployment. sophon-stream is suitable for Sophon BM1684, BM1684X and BM1688 chips, and can be flexibly deployed in PCIe and SOC modes.

2.2 sophon-stream Software Stack

sophon-stream is designed based on SophonSDK. SophonSDK is a deep learning SDK customized by Sophgo Technologies for its self-developed Deep Learning chips. It encompasses capabilities required for neural network inference, such as model optimization and efficient runtime support, providing an easy-to-use and efficient end-to-end solution for deep learning application development and deployment.

sophon-stream consists of two main components: framework and elements. The framework serves as the overall structure, determining the fundamental operation of sophon-stream, such as graph construction and data transfer, at the foundational level. Elements, on the other hand, collectively refer to all the nodes within the graph. They are derived from the same abstract base class and are responsible for providing specific functionalities based on SophonSDK, such as video encoding and decoding, image processing, and more.

3. Framework

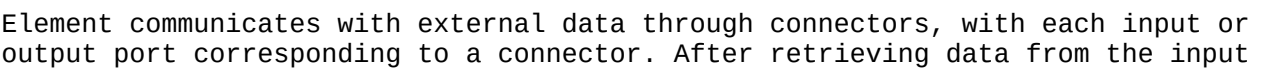
The sophon-stream framework consists of a three-tiered structure, namely Engine, Graph, and Element. The hierarchical relationship between them is illustrated in the following diagram.

Engine is the outermost structure in sophon-stream, providing interfaces to external projects. Engine oversees multiple Graphs, and each Graph is an independent directed acyclic graph that manages multiple Elements.

3.1 Element

The element class serves as the universal base class in sophon-stream, and all user-developed plugins are also based on element. As an abstract class, the element class defines the main interfaces and members for all derived classes, including how data is passed, how threads are managed, and how elements connect with each other.

The structure of an element is depicted in the following diagram:



Element communicates with external data through connectors, with each input or output port corresponding to a connector. After retrieving data from the input connector, the element processes the data in the `run()` method by calling `doWork()`. The processed data is then distributed to the connector associated with the output port, which connects to the input connector of the next element.

Key member variables of the element base class:

```cpp
int mId;             // Element ID, used to identify the element in the engine and graph, ensuring uniqueness within the graph
int mDeviceId;       // Device ID, used when involving TPU operations. Can be set as needed in PCIe mode but should be set to 0 in SOC mode.
int mThreadNumber;   // Number of threads for internal operations within the element, also equal to the number of DataPipes in the InputConnector.

// Mapping of input and output Connectors, where the key is the input or output port_id, and the value is a pointer to the Connector.
std::map<int, std::shared_ptr<framework::Connector>> mInputConnectorMap;
std::map<int, std::weak_ptr<framework::Connector>> mOutputConnectorMap;

/* Mapping of output SinkHandlers, where the key is the output port_id, and the value is a function with the signature void(std::shared_ptr<void>).
SinkHandlers provide data processing functionality for elements at the end of the graph, typically including tasks like rendering. */
std::map<int, SinkHandler> mSinkHandlerMap;
```

Key Member Functions:

```cpp
/* Static method to connect two elements. It sets the output port of srcElement and the input port of dstElement, registering dstElement's inputConnector with srcElement's mOutputConnectorMap. */
static void connect(Element& srcElement, int srcElementPort, Element& dstElement, int dstElementPort);

// Initialize element's common properties, such as element id, thread number, from a configuration file.
common::ErrorCode init(const std::string& json)

// Start and stop the element.
```

```
common::ErrorCode start();
common::ErrorCode stop();

// Push data, used to initiate the decoding task for the DecoderElement.
common::ErrorCode pushInputData(int inputPort, int dataPipeId,
std::shared_ptr<void> data);

// Thread function responsible for cyclically calling doWork() and allocating
processor time slices.
void run(int dataPipeId)

// Pure virtual function, used in derived classes to initialize custom
properties, such as algorithm-specific content.
virtual common::ErrorCode initInternal(const std::string& json) = 0;

// Pure virtual function, used in derived classes to define specific algorithm
logic, typically involving tasks like [popping data, batching, running
algorithms, pushing data, etc.]
virtual common::ErrorCode doWork(int dataPipeId) = 0;

// Cyclically call doWork(), managing thread resource scheduling.
void run(int dataPipeId);

// Push processed data to the output Connector. If the current element is a sink
element, execute the SinkHandler.
common::ErrorCode pushOutputData(int outputPort, int dataPipeId,
std::shared_ptr<void> data);
```

3.2 Graph

Instances of the graph class are managed by the engine, which provides interfaces for the engine to call, primarily when initializing or deconstructing a graph. Each graph can be configured to run on different devices, but it is not recommended to configure elements within the graph to run on different devices. The graph class's external interfaces mainly include:

```cpp
// Initialize and deinitialize the current graph.
common::ErrorCode init(const std::string& json);
void uninit();
// Start and stop the current graph.
common::ErrorCode start();
common::ErrorCode stop();
// Push data to the source element, used to initiate the decoding task for the
DecoderElement.
common::ErrorCode pushSourceData(int elementId, int inputPort,
std::shared_ptr<void> data);
// Set a data processing function for the sinkPort of the sink element, such as
rendering or sending.
void setSinkHandler(int elementId, int outputPort, SinkHandler sinkHandler);
```

3.3 Engine

The engine class is a singleton, with only one engine existing in a single process. The engine class's external interfaces mainly include:

```cpp
// Start and stop a specific graph.
common::ErrorCode start(int graphId);
common::ErrorCode stop(int graphId);
// Add a new graph.
common::ErrorCode addGraph(const std.string& json);
```

```
// Push data to the source element of a specific graph, used to initiate the
decoding function.
common::ErrorCode pushSourceData(int graphId, int elementId, int inputPort,
std::shared_ptr<void> data);
// Set a data processing function for the sinkPort of the sink element of a
specific graph, such as rendering or sending.
void setSinkHandler(int graphId, int elementId, int outputPort, SinkHandler
sinkHandler);
```

3.4 Connector

The Connector acts as a bridge for transferring data between two elements. An instance of a connector can manage multiple data pipes.

The main members of the Connector class are as follows:

```cpp
class Connector : public ::sophon_stream::common::NoCopyable {
 public:

  // Retrieve and dequeue data at the head of the queue with the specified ID.
  std::shared_ptr<void> popDataWithId(int id);

  // Push data to the queue with the specified ID.
  common::ErrorCode pushDataWithId(int id, std::shared_ptr<void> data);

  // Get the number of queues in the connector.
  int getCapacity() const;
 private:

  // Multiple DataPipes.
  std::vector<std::shared_ptr<DataPipe>> mDataPipes;
};
```

The member methods of the Connector class are used to obtain a specific data pipe using an ID and then call the corresponding methods of that data pipe.

3.5 ObjectMetadata

ObjectMetadata is a universal data structure in sophon-stream, and all functionality within elements is designed based on this structure.

The primary members of the ObjectMetadata include:

```cpp
std::shared_ptr<common::Packet> mPacket; // Stores pre-decoding information.
std::shared_ptr<common::Frame> mFrame;   // Stores post-decoding information:
bm_image, frame_id, EndOfStream indicator, and more.
std::shared_ptr<bmTensors> mInputBMtensors; // InputTensor obtained after
preprocessing the current frame.
std::shared_ptr<bmTensors> mOutputBMtensors; // OutputTensor obtained after
inference on the current frame.

// Nested ObjectMetadata, storing substructures on the current graph.
std::vector<std::shared_ptr<ObjectMetadata> > mSubObjectMetadatas;
// Detection-related information, such as box coordinates.
std::shared_ptr<common::DetectedObjectMetadata> mDetectedObjectMetadata;
// Tracking-related information, such as track_id.
std::shared_ptr<common::TrackedObjectMetadata> mTrackedObjectMetadata;
```

3.6 Frame

Frame is a structure within ObjectMetadata that stores image information, with its primary members including:

```cpp
int mChannelId;                 // Specifies the URL of the corresponding stream in the streaming service. If not specified in the configuration file, it is assigned starting from 0 by default.
int mChannelIdInternal;         // Internal channel_id, assigned starting from 0, used for calculating data flow within connectors.
std::int64_t mFrameId;          // Frame ID obtained after decoding, incrementing within a single data stream.
bool mEndOfStream;              // Indicator of the end of the data stream.
std::shared_ptr<bm_image> mSpData;      // Stores the original bm_image.
std::shared_ptr<bm_image> mSpDataOsd;   // Stores the bm_image after drawing by the OSD plugin.
```

3.7 Group

Group is a special template class designed to unify the management of three stages of an algorithm: preprocessing, inference, and post-processing.

Group inherits from the Element base class, and its template parameter is a specific algorithm class. When Group is instantiated, it automatically constructs the preprocessing, inference, and post-processing Elements corresponding to the template parameter, configures their properties in the initialization process, establishes their connections, and links the three internal Elements to the external components.

The `doWork()` method of the Group Element itself does not execute any algorithm logic; its work is carried out by the three internal Elements. The role of the Group Element is to consolidate the configuration of algorithm plugins into a single unit, greatly enhancing usability.

4. Plugins

In sophon-stream, all algorithms and multimedia functionalities are organized in the form of plugins within the `sophon_stream/element/` directory.

The `sophon-stream/element/algorithm` directory serves as a collection of algorithm plugins, currently encompassing yolox, yolov5, bytetrack, and resnet algorithms.

The `sophon-stream/element/multimedia` directory functions as a repository of multimedia plugins, which currently include encoding/decoding and OSD (On-Screen Display) functionalities.

The `sophon-stream/element/tools` directory constitutes an assembly of functional plugins, encompassing elements for data distribution, data aggregation, as well as a blank element intended for user debugging.

4.1 Algorithm

4.1.1 Overview

Algorithm plugins are modules that provide image processing and inference capabilities, leveraging the BMCV and BMRuntime libraries within the SophonSDK. These modules encompass preprocessing, inference, and post-processing components. Users can load the corresponding models according to their business requirements to activate the hardware functionalities.

Algorithm plugins exhibit the following characteristics:

- Each thread of an element is associated with a data pipe of the input connector. Batching occurs at the outset of the `doWork()` function, with data retrieval originating from the data pipe linked to the current thread.
- When transmitting data, load balancing among the threads of downstream elements is ensured.
- In cases where two modules only differ in internal model parameters, the preprocessing and post-processing elements can be reused when the preprocessing, inference, and post-processing procedures are identical.
- The ability to configure preprocessing, inference, and post-processing on different elements is supported. This configuration is designed to maximize the utilization of processor and TPU resources, thereby enhancing algorithm efficiency.

4.1.2 yolox

yolox is an object detection algorithm developed by Megvii, known for its high performance.

The configuration file for yolox is structured as follows:

```json
{
  "configure": {
    "model_path": "../data/models/BM1684X/yolox_s_int8_4b.bmodel",
    "threshold_conf": 0.5,
    "threshold_nms": 0.5,
    "bgr2rgb": true,
    "mean": [
      0,
      0,
      0
    ],
    "std": [
      1,
      1,
      1
    ],
    "stage": [
      "pre"
    ]
  },
  "shared_object": "../../../build/lib/libyolox.so",
  "id": 0,
  "device_id": 0,
  "name": "yolox",
  "side": "sophgo",
  "thread_number": 2
}
```

For a detailed explanation of the configuration parameters, please refer to the [yolox plugin documentation](../element/algorithm/yolox/README.md).

To explore yolox demos, please visit the [yolox demo](../samples/yolox/README.md).

4.1.3 yolov5

YOLOv5 is the most popular visual model in the world, widely used.

The configuration file for YOLOv5 looks like this:
```json
{
    "configure": {
```

```
        "model_path":
"../data/models/BM1684X_tpukernel/yolov5s_tpukernel_int8_4b.bmodel",
        "threshold_conf": 0.5,
        "threshold_nms": 0.5,
        "bgr2rgb": true,
        "mean": [
            0,
            0,
            0
        ],
        "std": [
            1,
            1,
            1
        ],
        "stage": [
            "pre"
        ],
        "use_tpu_kernel": true
    },
    "shared_object": "../../../build/lib/libyolov5.so",
    "id": 0,
    "device_id": 0,
    "name": "YOLOv5",
    "side": "sophgo",
    "thread_number": 1
}
```

For a detailed explanation of configuration parameters, please refer to the [YOLOv5 Plugin Introduction](../element/algorithm/yolov5/README.md).

For YOLOv5 demos, please refer to the [YOLOv5 Demo](../samples/yolov5/README.md).

4.1.4 bytetrack

bytetrack is a simple, fast, and powerful multi-object tracker jointly proposed by Huazhong University of Science and Technology, the University of Hong Kong, and ByteDance.

Its configuration file appears as follows:

```json
{
    "configure": {
        "track_thresh": 0.5,
        "high_thresh": 0.6,
        "match_thresh": 0.7,
        "frame_rate": 30,
        "track_buffer": 30
    },
    "shared_object": "../../../build/lib/libbytetrack.so",
    "id": 0,
    "device_id": 0,
    "name": "bytetrack",
    "side": "sophgo",
    "thread_number": 2
}
```

For a detailed explanation of the configuration parameters, please refer to the [bytetrack plugin documentation](../element/algorithm/bytetrack/README.md).

To explore bytetrack demos, please visit the [bytetrack demo](../samples/bytetrack/README.md).

4.1.5 openpose

openpose is a powerful pose estimation network.

Its configuration file is structured as follows:

```json
{
    "configure": {
        "model_path": "../data/models/BM1684X/pose_coco_int8_1b.bmodel",
        "threshold_nms": 0.05,
        "stage": [
            "pre"
        ]
    },
    "shared_object": "../../../build/lib/libopenpose.so",
    "name": "openpose",
    "side": "sophgo",
    "thread_number": 4
}
```

For a detailed explanation of the configuration parameters, please refer to the [openpose plugin documentation](../element/algorithm/openpose/README.md).

To explore openpose demos, please visit the [openpose demo](../samples/openpose/README.md).

4.1.6 lprnet

lprnet is a network designed for license plate recognition.

Its configuration file appears as follows:

```json
{
  "configure": {
    "model_path": "../models/BM1684/lprnet_fp32_1b.bmodel",
    "stage": ["infer"]
  },
  "shared_object": "../../../build/lib/liblprnet.so",
  "name": "lprnet",
  "side": "sophgo",
  "thread_number": 4
}
```

For a detailed explanation of the configuration parameters, please refer to the [lprnet plugin documentation](../element/algorithm/lprnet/README.md).

Lprnet does not provide a standalone demo but offers a cascading demo that combines yolov5 vehicle detection with lprnet license plate recognition. Please refer to the [license_plate_recognition demo](../samples/license_plate_recognition/README.md) for more information.

4.1.7 retinaface

retinaface is a network used for face detection.

Its configuration file is structured as follows:

```json
{
    "configure": {
        "model_path":
"../data/models/BM1684X/retinaface_mobilenet0.25_fp32_1b.bmodel",
        "max_face_count": 50,
        "score_threshold": 0.5,
        "bgr2rgb": false,
        "mean": [
            104,
            117,
            123
        ],
        "std": [
            1,
            1,
            1
        ],
        "stage": [
            "pre"
        ]
    },
    "shared_object": "../../../build/lib/libretinaface.so",
    "name": "retinaface",
    "side": "sophgo",
    "thread_number": 1
}
```

For a detailed explanation of the configuration parameters, please refer to the [retinaface plugin documentation](../element/algorithm/retinaface/README.md).

Retinaface does not provide a standalone demo but offers a demo that includes face detection and face recognition based on arcface and faiss. Please refer to the [retinaface_distributor_resnet_faiss_converger demo](../samples/retinaface_distributor_resnet_faiss_converger/README.md) for more information.

4.2 multimedia

4.2.1 decode

decode is the starting module in sophon-stream, responsible for obtaining ObjectMetadata from various types of inputs and sending them downstream to other elements.

Currently, DecoderElement supports the following types of data sources:
 - Local video files
 - Local image folders
 - RTSP streams
 - RTMP streams

As the starting module of sophon-stream, decode has some uniqueness when triggering tasks. After building the graph, you need to send a signal to the decode element to start a task. Only then will decode begin its work, and subsequent elements will receive data.

```cpp
nlohmann::json channel_config = yolox_json.channel_config;

channel_config["channel_id"] = channel_id;

auto channelTask =
    std::make_shared<sophon_stream::element::decode::ChannelTask>();
```

```
channelTask->request.operation =

sophon_stream::element::decode::ChannelOperateRequest::ChannelOperate::START;

channelTask->request.json = channel_config.dump();

sophon_stream::common::ErrorCode errorCode =
    engine.pushInputData(graph_id,
                         src_id_port.first,
                         src_id_port.second,
                         std::static_pointer_cast<void>(channelTask));
```

As you can see, when constructing channelTask, you also need to set the channel_id of the current input stream. The channel_id serves as an identifier for the stream and determines the flow of ObjectMetadata within the connector.

The configuration file for decode includes the following content:

```json
{
  "configure": {},
  "shared_object": "../../../build/lib/libdecode.so",
  "id": 0,
  "device_id": 0,
  "name": "decode",
  "side": "sophgo",
  "thread_number": 1
}
```

For a detailed explanation of the configuration parameters, please refer to the [decode documentation](../element/multimedia/decode/README.md).

4.2.2 encode

encode is typically used as the tail module in sophon-stream and is responsible for encoding processed image information into various video formats.

Currently, encode supports the following target types:
 - RTSP, RTMP, local video files
 - H.264/H.265 encoding formats
 - Pixel formats like I420, NV12, and more

The configuration file for encode includes the following content:

```json
{
  "configure": {
    "encode_type": "RTSP",
    "rtsp_port": "8554",
    "rtmp_port": "1935",
    "enc_fmt": "h264_bm",
    "pix_fmt": "I420"
  },
  "shared_object": "../../../build/lib/libencode.so",
  "id": 0,
  "device_id": 0,
  "name": "encode",
  "side": "sophgo",
  "thread_number": 4
}
```

For a detailed explanation of the configuration parameters, please refer to the [encode documentation](../element/multimedia/encode/README.md).

4.2.3 osd

The osd plugin is a visualization plugin that currently supports visualizing the results of object detection and object tracking algorithms.

The configuration file for the osd plugin includes:

```json
{
  "configure": {
    "osd_type": "TRACK",
    "class_names_file": "../data/coco.names"
  },
  "shared_object": "../../../build/lib/libosd.so",
  "id": 0,
  "device_id": 0,
  "name": "osd",
  "side": "sophgo",
  "thread_number": 1
}
```

The "osd_type" field in the configuration file identifies the algorithm type and can be set to "DET" or "TRACK."

For a detailed explanation of the configuration parameters, please refer to the [osd documentation](../element/multimedia/osd/README.md).

4.3 tools

4.3.1 distributor

The distributor plugin is a data distribution plugin that can distribute detected objects of different categories to different downstream branches. Currently, the distributor supports two distribution rules: distributing objects based on time intervals and distributing objects based on frame intervals. When configuring specific distribution rules, you can use a combination of these two methods.

The configuration file for the distributor includes:

```json
{
    "configure": {
        "default_port": 0,
        "rules": [
            {
                "time_interval": 1,
                "routes": [
                    {
                        "classes":["car"],
                        "port": 1
                    },
                    {
                        "classes":["person"],
                        "port" : 2
                    }
                ]
            }
        ],
```

```
        "class_names_file": "../data/coco.names"
    },
    "shared_object": "../../../build/lib/libdistributor.so",
    "name": "distributor",
    "side": "sophgo",
    "thread_number": 1
}
```

The distributor plugin must be used in conjunction with the converger plugin. For detailed information, please refer to the [distributor documentation](../element/tools/distributor/README.md).

4.3.2 converger

The converger is a data aggregation plugin that collects data from various downstream branches extended by the distributor and outputs it to downstream elements.

The configuration file for the converger includes:

```json
{
    "configure": {
        "default_port": 0
    },
    "shared_object": "../../../build/lib/libconverger.so",
    "name": "converger",
    "side": "sophgo",
    "thread_number": 1
}
```

The converger plugin must be used in conjunction with the distributor plugin. For detailed information, please refer to the [converger documentation](../element/tools/converger/README.md).

4.3.3 blank

The blank plugin is an empty plugin that can be connected between any two elements without affecting the pipeline in any way.

This plugin provides a complete plugin template but does not implement any configuration parameters or working logic. Users can write their own code to serve debugging purposes, among other things.

4.3.4 faiss

Faiss is a database retrieval plugin that implements Faiss::IndexFlatIP.search() on BM1684X. Considering the continuous memory of TPU on BM1684X, for a database of 1 million, it can query a maximum of about 512 inputs with 256 dimensions in a single processor.

5. Applications

Creating applications based on sophon-stream is essentially about building business pipelines based on the sophon-stream framework and elements.

After implementing the basic algorithm functionalities, you only need to write configuration files and corresponding entry programs to build the pipeline.

Next, this article introduces the key points of writing configuration files and entry programs using a typical application as an example.

5.1 Example Overview

A typical pipeline generally includes the following operations:
  - Data source decoding
  - Object detection
  - Object tracking
  - Drawing tracking results
  - Encoding output

In this section, the [reference demo](../samples/yolox_bytetrack_osd_encode/src/yolox_bytetrack_osd_encode_demo.cc) is used as an example. The graph built by this demo is shown in the following figure:

5.2 Configuration Files

First, you need to configure information for each element in the graph above. To ensure the efficiency of the detection algorithm, you can configure the three stages of the detection algorithm: preprocessing, inference, and post-processing, to be executed in three separate elements. This configuration is adopted in this demo, so the pipeline includes seven elements: decode, pre_process, inference, post_process, track, osd, and encode.

The configuration files for this demo include:

```bash
./config/
├── bytetrack.json                          # Tracking
├── decode.json                             # Decoding
├── encode.json                             # Encoding
├── engine.json                             # Overall configuration of the graph
├── engine_group.json                       # Simplified configuration of the graph
├── osd.json                                # OSD module
├── yolox_bytetrack_osd_encode_demo.json    # Overall configuration for the demo
├── yolox_group.json                        # Unified management of yolox configuration
├── yolox_infer.json                        # Yolox inference
├── yolox_post.json                         # Yolox post-processing
└── yolox_pre.json                          # Yolox preprocessing
```

The configuration files for each element have been explained in the respective sections earlier. Here, we will focus on the contents of the demo and graph configuration files.

The `yolox_bytetrack_osd_encode_demo.json` is the overall configuration for this demo, and it looks like this:

```json
{
  {
  "channels": [
    {
      "channel_id": 2,
      "url": "../data/videos/mot17_01_frcnn.mp4",
      "source_type": "VIDEO"
    },
    {
      "channel_id": 3,
      "url": "../data/videos/mot17_03_frcnn.mp4",
      "source_type": "VIDEO"
```

```
    }
  ],
  "engine_config_path": "../config/engine.json"
}
}
```

The demo configuration file includes two properties. The first one is `channels`, where all input information, such as `url`, `channel_id`, and `source_type`, is recorded in a list. It's important to note that `source_type` should be accurately set based on the [decode configuration](../element/multimedia/decode/README.md).

In this configuration file, `channel_id` is used to identify the URL of the output video stream when the encode plugin starts the streaming server. For details on configuring the streaming URL, please refer to [#4.2.2 encode](#422-encode). If you don't need this feature, you can refer to the [yolov5 demo](../samples/yolov5/src/yolov5_demo.cc) and [yolov5 configuration file](../samples/yolov5/config/yolov5_demo.json). In that case, you don't need to set `channel_id`, and it will be automatically assigned starting from 0 in the demo.

The `engine.json` file contains the graph information constructed in the current demo program. It stores information about each graph, including the elements within the graph and how these elements are connected. It has the following structure:

```json
[
    {
        "graph_id": 0,
        "graph_name": "yolox",
        "elements": [
            {
                "element_id": 5000,
                "element_config": "../config/decode.json",
                "ports": {
                    "input": [
                        {
                            "port_id": 0,
                            "is_sink": false,
                            "is_src": true
                        }
                    ],
                    "output": [
                        {
                            "port_id": 0,
                            "is_sink": false,
                            "is_src": false
                        }
                    ]
                }
            },
            {
                "element_id": 5001,
                "element_config": "../config/yolox_pre.json",
                "ports": {
                    "input": [
                        {
                            "port_id": 0,
                            "is_sink": false,
                            "is_src": false
                        }
                    ],
```

```
                    "output": [
                        {
                            "port_id": 0,
                            "is_sink": true,
                            "is_src": false
                        }
                    ]
                }
            }
        ],
        "connections": [
            {
                "src_element_id": 5000,
                "src_port": 0,
                "dst_element_id": 5001,
                "dst_port": 0
            }
        ]
    }
]
```

It's essential to pay attention to the "elements" and "connections" sections. "Elements" lists all the elements within the graph. For each element, you need to configure the element_id, the corresponding configuration file path, and port information. Different elements within the same graph should have distinct element_ids. Ports of an element include input and output ports, and ports of the same type should be distinguished by different port_ids. Each port has attributes "is_src" and "is_sink," indicating whether it is an input or output port for the entire graph.

In general, only the decode element has input ports, and there is only one decode element in a graph. For this element, you need to send a channelTask in the application to start the pipeline's operation. On the other hand, output ports are not specific to any element type. Any element can have output ports, and the configuration should be based on project requirements. For elements with output ports, you should set a SinkHandler for them, which is a callback function to handle the output data correctly.

5.3 Entry Program

For different demos, the main differences lie in the configuration files, while the entry program remains mostly consistent.

The entry program typically includes the following steps:

- Parsing the demo's configuration file.
- Parsing the engine's configuration file.
- Calling `engine.addGraph()` to initialize all the elements and their connections.
- Setting the SinkHandler for sink elements.
- Sending a channelTask to trigger the decode element's work.
- Waiting for all stream processing to finish and ending the task.
- Collecting information such as frames per second (FPS).

5.4 User-Side Information

When running a demo, the following information is sequentially printed in the command line:

 - Configuration Information of Graph

 ```bash
 ```

```
 [info] [/sophon-stream/framework/src/engine.cc:95] Add graph start, json:
{"connections":[{"dst_id":5001,"dst_port":0,"src_id":5000,"src_port":0},
{"dst_id":5002,"dst_port":0,"src_id":5001,"src_port":0},
{"dst_id":5003,"dst_port":0,"src_id":5002,"src_port":0}],"elements":
[{"configure":
{},"device_id":0,"id":5000,"name":"decode","shared_object":"../../../build/lib/
libdecode.so","side":"sophgo","thread_number":1},{"configure":
{"model_path":"../data/models/BM1684X/yolox_s_int8_4b.bmodel","stage":
["pre"],"threshold_conf":0.5,"threshold_nms":0.5},"device_id":0,"id":5001,"name"
:"yolox","shared_object":"../../../build/lib/
libyolox.so","side":"sophgo","thread_number":2},{"configure":{"model_path":"../
data/models/BM1684X/yolox_s_int8_4b.bmodel","stage":
["infer"],"threshold_conf":0.5,"threshold_nms":0.5},"device_id":0,"id":5002,"nam
e":"yolox","shared_object":"../../../build/lib/
libyolox.so","side":"sophgo","thread_number":2},{"configure":{"model_path":"../
data/models/BM1684X/yolox_s_int8_4b.bmodel","stage":
["post"],"threshold_conf":0.5,"threshold_nms":0.5},"device_id":0,"id":5003,"is_s
ink":true,"name":"yolox","shared_object":"../../../build/lib/
libyolox.so","side":"sophgo","thread_number":2}],"graph_id":0}
 ```
```

 - Configuration Information of Elements

  ```bash
 [info] [/sophon-stream/framework/src/element.cc:45] Init start, json:
{"configure":{"model_path":"../data/models/BM1684X/yolox_s_int8_4b.bmodel","stag
e":
["pre"],"threshold_conf":0.5,"threshold_nms":0.5},"device_id":0,"id":5001,"name"
:"yolox","shared_object":"../../../build/lib/
libyolox.so","side":"sophgo","thread_number":2}
 [BMRT][bmcpu_setup:349] INFO:cpu_lib 'libcpuop.so' is loaded.
 bmcpu init: skip cpu_user_defined
 open usercpu.so, init user_cpu_init
 [BMRT][load_bmodel:1079] INFO:Loading bmodel from
[../data/models/BM1684X/yolox_s_int8_4b.bmodel].  Thanks for your patience...
 [BMRT][load_bmodel:1023] INFO:pre net num: 0, load net num: 1
 *** Run in PCIE mode ***

 ########################
 NetName: yolox_s_bmnetp
 ---- stage 0 ----
   Input 0) 'x.1' shape=[ 4 3 640 640 ] dtype=INT8 scale=0.498161
   Output 0) '15' shape=[ 4 8400 85 ] dtype=FLOAT32 scale=1
 ########################
 ```

 - Information of Connections

  ```bash
 [debug] [/sophon-stream/framework/src/element.cc:26] InputConnector
initialized, mId = 5001, inputPort = 0, dataPipeNum = 2
 ```

 - Start Graph and Elements

 ```bash
[info] [/sophon-stream/framework/src/graph.cc:107] Start graph thread start,
graph id: 0
[info] [/sophon-stream/framework/src/element.cc:125] Start element thread start,
element id: 5000
[info] [/sophon-stream/framework/src/element.cc:140] Start element thread
finish, element id: 5000
[info] [/sophon-stream/framework/src/element.cc:125] Start element thread start,
element id: 5001
```
```

```
[info] [/sophon-stream/framework/src/element.cc:140] Start element thread
finish, element id: 5001
[info] [/sophon-stream/framework/src/element.cc:125] Start element thread start,
element id: 5002
[info] [/sophon-stream/framework/src/element.cc:140] Start element thread
finish, element id: 5002
[info] [/sophon-stream/framework/src/element.cc:125] Start element thread start,
element id: 5003
[info] [/sophon-stream/framework/src/element.cc:140] Start element thread
finish, element id: 5003
[info] [/sophon-stream/framework/src/graph.cc:127] Start graph thread finish,
graph id: 0
```

- Set SinkHandler

```bash
[info] [/sophon-stream/framework/src/engine.cc:143] Set sink handler, graph id:
0, element id: 5003, output port: 0
```

- Set channel_task and start decode

```bash
[info] [/sophon-stream/element/multimedia/decode/src/decode.cc:127] add one
channel task
[info] [/sophon-stream/element/multimedia/decode/src/decode.cc:163] channel
info decoder address: 0x7f7814000b70
```

- Working Status of Elements

```bash
[engine] [debug] [/sophon-stream/framework/src/element.cc:232] send data,
element id: 5000, output port: 0, data:0x7f781cab01d0
[engine] [debug] [/sophon-stream/framework/src/element.cc:232] send data,
element id: 5000, output port: 0, data:0x7f781cad9110
[engine] [debug] [/sophon-stream/framework/src/element.cc:232] send data,
element id: 5000, output port: 0, data:0x7f781c0681b0
[engine] [debug] [/sophon-stream/framework/src/element.cc:232] send data,
element id: 5002, output port: 0, data:0x7f781c0127d0
[engine] [debug] [/sophon-stream/framework/src/element.cc:232] send data,
element id: 5002, output port: 0, data:0x7f781c067480
[engine] [debug] [/sophon-stream/framework/src/element.cc:232] send data,
element id: 5002, output port: 0, data:0x7f781c0daf90
```

- Deocde to EOF

```bash
[h264_bm @ 0x7f7814007f80] av_read_frame ret(-541478725) maybe eof...
```

- engine, graph and element thread stop

```bash
[engine] [info] [/sophon-stream/framework/src/engine.cc:35] Engine stop graph
thread start, graph id: 0
[engine] [info] [/sophon-stream/framework/src/engine.cc:50] Engine stop graph
thread finish, graph id: 0
[engine] [info] [/sophon-stream/framework/src/graph.cc:132] Stop graph thread
start, graph id: 0
[engine] [info] [/sophon-stream/framework/src/element.cc:145] Stop element
thread start, element id: 5000
```

```
 [engine] [info] [/sophon-stream/element/multimedia/decode/src/decode.cc:52]
Decode stop...
 [engine] [info] [/sophon-stream/framework/src/element.cc:159] Stop element
thread finish, element id: 5000
 [engine] [info] [/sophon-stream/framework/src/element.cc:145] Stop element
thread start, element id: 5001
 [engine] [info] [/sophon-stream/framework/src/element.cc:159] Stop element
thread finish, element id: 5001
 [engine] [info] [/sophon-stream/framework/src/element.cc:145] Stop element
thread start, element id: 5002
 [engine] [info] [/sophon-stream/framework/src/element.cc:159] Stop element
thread finish, element id: 5002
 [engine] [info] [/sophon-stream/framework/src/element.cc:145] Stop element
thread start, element id: 5003
 [engine] [info] [/sophon-stream/framework/src/element.cc:159] Stop element
thread finish, element id: 5003
 [engine] [info] [/sophon-stream/framework/src/graph.cc:150] Stop graph thread
finish, graph id: 0
```

- Statistics on elapsed time, frames, fps

```bash
total time cost 5286871 us.
frame count is 1422 | fps is 268.968 fps.
```

6. Tools

6.1 Introduction to web_visualize

web_visualize is a visualization tool designed for running the sophon-stream project. This tool visualizes the project's results on the frontend, allowing users to have real-time insights into the project's inference results. With this tool, users can visually observe the processing status and outputs of each stage in real-time on the frontend. This interactive visualization provides great convenience for debugging and optimization of the project, making it accessible even to non-technical users.

Currently, web_visualize consists of two subdirectories: web_server, which is the backend program of the project, and web_ui, which is the frontend program of the project. Details on how to start and an overview of the project can be found in the [Quick Start Guide](../tools/web_visualize/README.md) below, which will provide step-by-step instructions for launching both the frontend and backend.

6.2 Features

The web_visualize visualization tool provides the following features to users:

- Graphical user interface for operation and configuration:

  We offer built-in pipeline example configurations for each pipeline, allowing you to run examples and quickly experience sophon-stream.

- Support for modifying different pipeline parameters:

  We support online configuration of various pipeline parameters. You can select different configuration files and submit them online.

- Support for previewing pipeline results:

  You can preview real-time video results of running pipelines.

6.3 Usage

To use this project tool, start the web_server backend program first and then the web_ui frontend program. Follow the introductory steps below:

First, refer to the Quick Start Guide for the backend program to launch it. For details, please see the [web_server user manual](../tools/web_visualize/web_server/README.md).

Then, refer to the Quick Start Guide for the frontend program to launch it. For details, please see the [web_ui user manual](../tools/web_visualize/web_ui/README.md).