**密集人群人数统计系统技术说明书**

目录

[1项目开发目的和意义 2](#_Toc27150145)

[2 总体设计 3](#_Toc27150146)

[3 算法设计 3](#_Toc27150147)

[3.1 人群数据集 3](#_Toc27150148)

[3.2 模型训练 4](#_Toc27150149)

[3.3 模型转换 4](#_Toc27150150)

[3.4 密集人数统计统计 4](#_Toc27150151)

[3.4 流程设计 4](#_Toc27150152)

[4 结果 5](#_Toc27150153)

[5 关键代码说明 5](#_Toc27150154)

[5.1 视频解码模块核心代码 5](#_Toc27150155)

[5.2 人数统计推理的核心代码 9](#_Toc27150156)

[5.3 人数统计的后处理核心代码 18](#_Toc27150157)

[6 重要问题及解决 23](#_Toc27150158)

[7 后续可扩展性 23](#_Toc27150159)

## 1项目开发目的和意义

从图像或视频中准确估计人群已成为计算机视觉技术在人群控制和公共安全中有着日益重要的应用。在某些情况下，如公众集会和体育赛事，参与人数或密度是未来赛事规划和空间设计的重要信息。好的人群计数方法也可以扩展到其他领域，例如，从显微图像中计数细胞或细菌，野生动物保护区中的动物人群估计，或估计交通枢纽或交通堵塞的车辆数量等。本文将对密集人群的人数统计的相关内容做一个较为详细的综述。

本项目在华为Atlas 200 DK开发者板上实现对本地mp4文件或者RTSP视频流进行解码，对视频帧中的人群图像进行检测并对其进行预测，生成结构化信息发送至Presenter Server端进行保存、展示。

## 2 总体设计

本系统可以分为三个部分：数据处理部分、模型构建部分和密集人群计数部分。各部分相互独立，但是存在数据关联。为了说明各系统之间的结构关系，细化结构如下图：

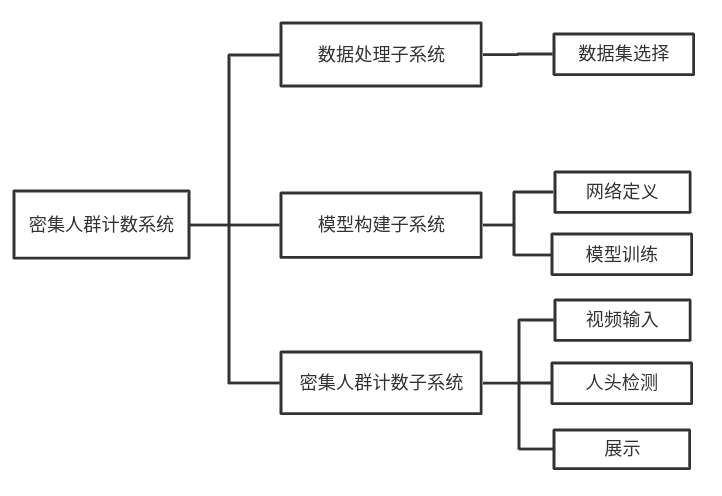


图1 系统整体功能结构图

## 3 算法设计

### 3.1 人群数据集

这里使用了大规模人群数据集，名为“上海科技”，包含近1200幅图像，其中约33万个准确标记的头部。据我们所知，它是最大的人群计数数据集方面的数字注释头。此数据集中没有两个图像是从同一个视点拍摄的。该数据集由两部分组成：A部分和B部分。A部分中的图像是从互联网上随机抓取的，其中大部分都有大量的人。B部分取自上海市区繁华的街道。

### 3.2 模型训练

模型的训练过程来源于<https://github.com/uestcchicken/crowd-counting-MCNN>

### 3.3 模型转换

训练得到的模型是h5和json文件，这里使用Keras to TensorFlow来转化成pb模型，再由pb模型在MindStudio中转换成om模型。

### 3.4 密集人数统计统计

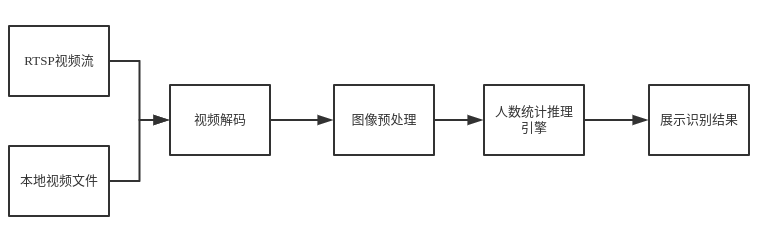


图2 人数统计流程图

人数统计流程：通过对输入的RTSP视频流或MP4视频文件，解码成YUV420SP格式的视频帧数据。当输入图片的分辨率与网络模型要求的分辨率不匹配时，使用DVPP的resize功能对图片进行预处理，经过预处理后的图片被送入密集人群计数统计网络模型进行推理，并输出原始图片及计数的结果送到presenter server进行展示。

### 3.4 流程设计

密集人群计数系统总共设计了三个Engine，分别为视频解码模块、推理模块、后处理模块。

1. **视频解码模块**

视频解码模块获取的是输入的RTSP视频流或MP4视频文件，解码成YUV420SP格式的视频帧数据。Atlas 200 DK提供了一套帮助开发者实现视频的解码的API接口，详细的接口使用方法可参考

https://ascend.huawei.com/doc/Atlas200DK/1.3.0.0/zh/zh-cn\_topic\_0197080057.html

1. **人数统计推理模块**

人数统计推理模块是当输入图片的分辨率与网络模型要求的分辨率不匹配时，使用DVPP的resize功能对图片进行预处理，经过预处理的图片会被送入推理模块中，并输出原始图片和推理的结果。

1. **后处理模块**

后处理模块接收上一个模块的原始图片和推理结果，推理的结果是得到每一个像素上的人头数量，后处理将这些结果全部相加后可以得到人群的计数统计结果，把这个结果和原始图片信息通过调用Presenter Agent的API发送到UI Host上部署的Presenter Server服务进程。Presenter Server根据接收到的推理结果，并将图像信息和检测人数的结果发送给Web UI。

## 4 结果

输入MP4视频时，推理结果展示如图3下：

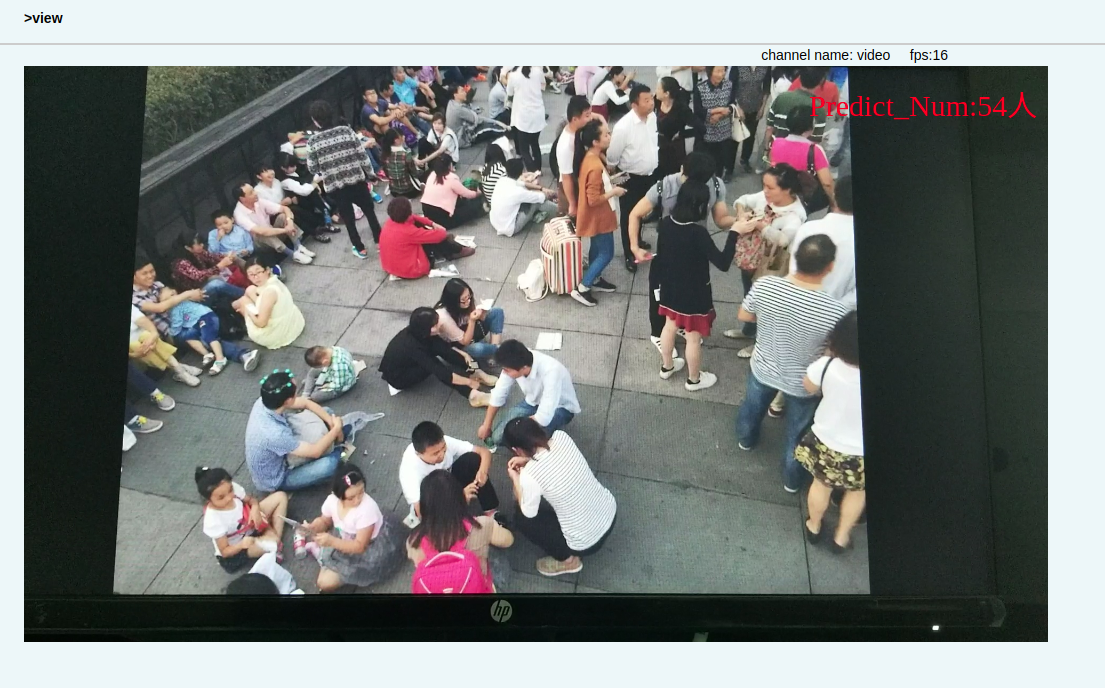


图3 推理结果展示图

可以看出，预测的人数54人和实际人数47人相差10个人以内，比较准确。

## 5 关键代码说明

### 5.1 视频解码模块核心代码

HIAI\_StatusT VideoDecode::Init(

const hiai::AIConfig &config,

const vector<hiai::AIModelDescription> &model\_desc) {

HIAI\_ENGINE\_LOG("Start process!");

// get channel values from configs item

for (int index = 0; index < config.items\_size(); ++index) {

const ::hiai::AIConfigItem &item = config.items(index);

// get channel1 value

if (item.name() == kStrChannelId1) {

channel1\_ = item.value();

continue;

}

// get channel2 value

if (item.name() == kStrChannelId2) {

channel2\_ = item.value();

continue;

}

}

// verify channel values are valid

if (!VerifyChannelValues()) {

return HIAI\_ERROR;

}

HIAI\_ENGINE\_LOG("End process!");

return HIAI\_OK;

}

bool VideoDecode::IsEmpty(const string &input\_str, const string &channel\_id) {

regex regex\_space(kRegexSpace.c\_str());

// check input string is empty or spaces

if (regex\_match(input\_str, regex\_space)) {

HIAI\_ENGINE\_LOG("The channel string is empty or all spaces, channel id:%s",

channel\_id.c\_str());

return true;

}

return false;

}

bool VideoDecode::VerifyVideoSourceName(const string &input\_str) {

// verify input string is valid mp4 file name or rtsp address

if (!IsValidRtsp(input\_str) && !IsValidMp4File(input\_str)) {

HIAI\_ENGINE\_LOG(HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,

"Invalid mp4 file name or RTSP name:%s", input\_str.c\_str());

return false;

}

return true;

}

bool VideoDecode::IsValidRtsp(const string &input\_str) {

regex regex\_rtsp\_address(kRegexRtsp.c\_str());

// verify input string is valid rtsp address

if (regex\_match(input\_str, regex\_rtsp\_address)) {

return true;

}

return false;

}

bool VideoDecode::IsValidMp4File(const string &input\_str) {

regex regex\_mp4\_file\_name(kRegexMp4File.c\_str());

// verify input string is valid mp4 file name

if (regex\_match(input\_str, regex\_mp4\_file\_name)) {

return true;

}

return false;

}

bool VideoDecode::VerifyChannelValues() {

// check channel1 and channel2 are empty

if (IsEmpty(channel1\_, kStrChannelId1)

&& IsEmpty(channel2\_, kStrChannelId2)) {

HIAI\_ENGINE\_LOG(HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,

"Both channel1 and channel2 are empty!");

return false;

}

// verify channel1 value when channel1 is not empty

if (!IsEmpty(channel1\_, kStrChannelId1)) {

// deletes the space at the head of the string

channel1\_.erase(0, channel1\_.find\_first\_not\_of(kNeedRemoveStr.c\_str()));

// deletes spaces at the end of the string

channel1\_.erase(channel1\_.find\_last\_not\_of(kNeedRemoveStr.c\_str()) + 1);

HIAI\_ENGINE\_LOG("Display channel1:%s", channel1\_.c\_str());

if (!VerifyVideoSourceName(channel1\_)) { // verify channel1

return false;

}

}

// verify channel2 value when channel1 is not empty

if (!IsEmpty(channel2\_, kStrChannelId2)) {

// deletes the space at the head of the string

channel2\_.erase(0, channel2\_.find\_first\_not\_of(kNeedRemoveStr.c\_str()));

// deletes spaces at the end of the string

channel2\_.erase(channel2\_.find\_last\_not\_of(kNeedRemoveStr.c\_str()) + 1);

HIAI\_ENGINE\_LOG("Display channel2:%s", channel2\_.c\_str());

if (!VerifyVideoSourceName(channel2\_)) { // verify channel2

return false;

}

}

return true;

}

void VideoDecode::SendImageData(shared\_ptr<VideoImageParaT> &video\_image\_data) {

HIAI\_StatusT hiai\_ret = HIAI\_OK;

if (video\_image\_data == nullptr) { // the queue is empty and return

return;

}

// send image data

do {

hiai\_ret = SendData(0, kVideoImageParaType,

static\_pointer\_cast<void>(video\_image\_data));

if (hiai\_ret == HIAI\_QUEUE\_FULL) { // check queue is full

HIAI\_ENGINE\_LOG("The queue is full when send image data, sleep 10ms");

usleep(kWait10Milliseconds); // sleep 10 ms

}

} while (hiai\_ret == HIAI\_QUEUE\_FULL); // loop while queue is full

if (hiai\_ret != HIAI\_OK) { // check send data is failed

HIAI\_ENGINE\_LOG(HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,

"Send data failed! error code: %d", hiai\_ret);

}

}

HIAI\_IMPL\_ENGINE\_PROCESS("video\_decode", VideoDecode, INPUT\_SIZE) {

av\_log\_set\_level(AV\_LOG\_INFO); // set ffmpeg log level

// verify video type

if (!VerifyVideoType()) {

SendFinishedData(); // send the flag data when finished

return HIAI\_ERROR;

}

MultithreadHandleVideo(); // handle video from file or RTSP with multi-thread

SendFinishedData();// send the flag data when finished

return HIAI\_OK;

}

### 5.2 人数统计推理的核心代码

HIAI\_StatusT ObjectDetectionInferenceEngine::Init(

const hiai::AIConfig &config,

const vector<hiai::AIModelDescription> &model\_desc) {

HIAI\_ENGINE\_LOG(HIAI\_DEBUG\_INFO, "[ODInferenceEngine] start to initialize!");

if (ai\_model\_manager\_ == nullptr) { // check ai model manager is nullptr

ai\_model\_manager\_ = make\_shared<hiai::AIModelManager>();

}

vector<hiai::AIModelDescription> od\_model\_descs;

hiai::AIModelDescription model\_description;

// load model path.

for (int index = 0; index < config.items\_size(); ++index) {

const ::hiai::AIConfigItem &item = config.items(index);

if (item.name() == kModelPath) { // current item is model path

const char\* model\_path = item.value().data();

model\_description.set\_path(model\_path);

}

}

od\_model\_descs.push\_back(model\_description);

// init ssd model

HIAI\_StatusT ret = ai\_model\_manager\_->Init(config, od\_model\_descs);

if (ret != hiai::SUCCESS) {

HIAI\_ENGINE\_LOG(HIAI\_GRAPH\_INVALID\_VALUE,

"[ODInferenceEngine] failed to initialize AI model!");

return HIAI\_ERROR;

}

// create vdec api for channel1, and check the result

if (CreateVdecApi(dvpp\_api\_channel1\_, kVdecSingleton) != kHandleSuccessful) {

HIAI\_ENGINE\_LOG(

HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,

"[ODInferenceEngine] fail to create dvpp vdec api for channel1!");

}

// create vdec api for channel2, and check the result

if (CreateVdecApi(dvpp\_api\_channel2\_, kVdecSingleton) != kHandleSuccessful) {

HIAI\_ENGINE\_LOG(

HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,

"[ODInferenceEngine] fail to create dvpp vdec api for channel2!");

}

HIAI\_ENGINE\_LOG(HIAI\_DEBUG\_INFO, "[ODInferenceEngine] engine initialized!");

return HIAI\_OK;

}

void ascend::videoanalysis::CallVpcGetYuvImage(FRAME\* frame, void\* hiai\_data) {

if (frame == nullptr || hiai\_data == nullptr) { // check input parameters

HIAI\_ENGINE\_LOG(

HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,

"The input data for function:CallVpcGetYuvImage is nullptr!");

return;

}

string channel\_id = ((HiaiDataSpSon\*) hiai\_data)->channel\_id\_;

uint32\_t frame\_id = GetFrameId(channel\_id);

// only send key frame to next engine, key frame id: 1,6,11,16...

// if (!IsKeyFrame(frame\_id)) {

// return;

// }

string channel\_name = ((HiaiDataSpSon\*) hiai\_data)->channel\_name\_;

HIAI\_ENGINE\_LOG("Get key frame, frame id:%d, channel\_id:%s, channel\_name:%s,"

" frame->realWidth:%d, frame->realHeight:%d",

frame\_id, channel\_id.c\_str(), channel\_name.c\_str(),

frame->realWidth, frame->realHeight);

IDVPPAPI\* dvpp\_api = nullptr;

CreateDvppApi(dvpp\_api);

if (dvpp\_api == nullptr) { // check create dvpp api result

HIAI\_ENGINE\_LOG(

HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,

"Fail to call CreateDvppApi for vpc, the result is nullptr!");

return;

}

int aligned\_output\_width = ALIGN\_UP(frame->width, kVpcWidthAlign);

int aligned\_output\_height = ALIGN\_UP(frame->height, kVpcHeightAlign);

// constructing input image configuration

shared\_ptr<VpcUserImageConfigure> image\_configure(

new (nothrow) VpcUserImageConfigure);

if (image\_configure.get() == nullptr) { // check new memory result

HIAI\_ENGINE\_LOG(

HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,

"Fail to new memory when construct input image configuration!");

return;

}

image\_configure->widthStride = aligned\_output\_width;

image\_configure->heightStride = aligned\_output\_height;

// check image format is nv12

if (strcmp(frame->image\_format, kImageFormatNv12.c\_str()) == kCompareEqual) {

image\_configure->inputFormat = INPUT\_YUV420\_SEMI\_PLANNER\_UV;

} else { // check image format is nv21

image\_configure->inputFormat = INPUT\_YUV420\_SEMI\_PLANNER\_VU;

}

image\_configure->outputFormat = OUTPUT\_YUV420SP\_UV;

image\_configure->isCompressData = true;

image\_configure->compressDataConfigure.lumaHeadAddr = (long) (frame->buffer

+ frame->offset\_head\_y);

image\_configure->compressDataConfigure.chromaHeadAddr = (long) (frame->buffer

+ frame->offset\_head\_c);

image\_configure->compressDataConfigure.lumaHeadStride = frame->stride\_head;

image\_configure->compressDataConfigure.chromaHeadStride = frame->stride\_head;

image\_configure->compressDataConfigure.lumaPayloadAddr = (long) (frame->buffer

+ frame->offset\_payload\_y);

image\_configure->compressDataConfigure.chromaPayloadAddr = (long) (frame

->buffer + frame->offset\_payload\_c);

image\_configure->compressDataConfigure.lumaPayloadStride = frame

->stride\_payload;

image\_configure->compressDataConfigure.chromaPayloadStride = frame

->stride\_payload;

shared\_ptr<VpcUserRoiConfigure> roi\_configure(

new (nothrow) VpcUserRoiConfigure);

if (roi\_configure.get() == nullptr) { // check new memory result

HIAI\_ENGINE\_LOG(

HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,

"Fail to new memory when initialize vpc user roi configure!");

return;

}

roi\_configure->next = nullptr;

// constructing input roi configuration

VpcUserRoiInputConfigure \*input\_configure = &roi\_configure->inputConfigure;

input\_configure->cropArea.leftOffset = 0; // 0 means without crop

// dvpp limits rightOffset is odd

input\_configure->cropArea.rightOffset =

frame->width % 2 == 0 ? frame->width - 1 : frame->width;

input\_configure->cropArea.upOffset = 0; // 0 means without crop

// dvpp limits downOffset is odd

input\_configure->cropArea.downOffset =

frame->height % 2 == 0 ? frame->height - 1 : frame->height;

int vpc\_output\_size = aligned\_output\_width \* aligned\_output\_height

\* DVPP\_YUV420SP\_SIZE\_MOLECULE / DVPP\_YUV420SP\_SIZE\_DENOMINATOR;

// check vpc output size is valid

if (vpc\_output\_size <= 0 || vpc\_output\_size > kAllowedMaxImageMemory) {

HIAI\_ENGINE\_LOG(

HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,

"The vpc\_output\_size:%d is invalid! value range: 1~67108864",

vpc\_output\_size);

return;

}

// construct vpc out data buffer

uint8\_t\* vpc\_out\_buffer = (uint8\_t \*) mmap(

0, ALIGN\_UP(vpc\_output\_size, MAP\_2M), PROT\_READ | PROT\_WRITE,

MAP\_PRIVATE | MAP\_ANONYMOUS | API\_MAP\_VA32BIT, -1, 0);

if (vpc\_out\_buffer == MAP\_FAILED) { // check mmap buffer result

HIAI\_ENGINE\_LOG(HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,

"Failed to malloc 4k memory for vpc!");

return;

}

// constructing output roi configuration

VpcUserRoiOutputConfigure \*output\_configure = &roi\_configure->outputConfigure;

output\_configure->addr = vpc\_out\_buffer;

output\_configure->bufferSize = vpc\_output\_size;

output\_configure->widthStride = aligned\_output\_width;

output\_configure->heightStride = aligned\_output\_height;

output\_configure->outputArea.leftOffset = 0; // 0 means without crop

// dvpp limits rightOffset is odd

output\_configure->outputArea.rightOffset =

frame->width % 2 == 0 ? frame->width - 1 : frame->width;

output\_configure->outputArea.upOffset = 0; // 0 means without crop

// dvpp limits downOffset is odd

output\_configure->outputArea.downOffset =

frame->height % 2 == 0 ? frame->height - 1 : frame->height;

image\_configure->roiConfigure = roi\_configure.get();

dvppapi\_ctl\_msg dvpp\_api\_ctl\_msg;

dvpp\_api\_ctl\_msg.in = static\_cast<void \*>(image\_configure.get());

dvpp\_api\_ctl\_msg.in\_size = sizeof(VpcUserImageConfigure);

// call vpc and check the result

if (DvppCtl(dvpp\_api, DVPP\_CTL\_VPC\_PROC, &dvpp\_api\_ctl\_msg)

!= kHandleSuccessful) {

HIAI\_ENGINE\_LOG(HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,

"Fail to call dvppctl VPC!");

// free vpc\_out\_buffer memory

munmap(vpc\_out\_buffer, (unsigned) (ALIGN\_UP(vpc\_output\_size, MAP\_2M)));

DestroyDvppApi(dvpp\_api);

return;

}

DestroyDvppApi(dvpp\_api);

uint8\_t\* output\_image\_buffer = new (nothrow) uint8\_t[vpc\_output\_size];

if (output\_image\_buffer == nullptr) { // check new result

HIAI\_ENGINE\_LOG(HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,

"Fail to new data when handle vpc output!");

// free vpc\_out\_buffer memory

munmap(vpc\_out\_buffer, (unsigned) (ALIGN\_UP(vpc\_output\_size, MAP\_2M)));

return;

}

int memcpy\_result = memcpy\_s(output\_image\_buffer, vpc\_output\_size,

vpc\_out\_buffer, vpc\_output\_size);

// free vpc\_out\_buffer memory

munmap(vpc\_out\_buffer, (unsigned) (ALIGN\_UP(vpc\_output\_size, MAP\_2M)));

if (memcpy\_result != EOK) { // check memcpy\_s result

HIAI\_ENGINE\_LOG(HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,

"Fail to copy vpc output image buffer, memcpy\_s result:%d",

memcpy\_result);

delete[] output\_image\_buffer;

return;

}

shared\_ptr<VideoImageParaT> video\_image\_para = make\_shared<VideoImageParaT>();

video\_image\_para->video\_image\_info.channel\_id = channel\_id;

video\_image\_para->video\_image\_info.channel\_name = channel\_name;

video\_image\_para->video\_image\_info.frame\_id = frame\_id;

video\_image\_para->video\_image\_info.is\_finished = false;

video\_image\_para->img.width = frame->realWidth;

video\_image\_para->img.height = frame->realHeight;

video\_image\_para->img.format = IMAGEFORMAT::YUV420SP;

video\_image\_para->img.size = vpc\_output\_size;

video\_image\_para->img.data.reset(output\_image\_buffer,

default\_delete<uint8\_t[]>());

AddImage2Queue(video\_image\_para);

return;

}

HIAI\_StatusT ObjectDetectionInferenceEngine::ImagePreProcess(

const ImageData<u\_int8\_t> &src\_img, ImageData<u\_int8\_t> &resized\_img) {

if (src\_img.format != IMAGEFORMAT::YUV420SP) {

// input image must be yuv420sp nv12.

HIAI\_ENGINE\_LOG(HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,

"[ODInferenceEngine] input image type does not match");

return HIAI\_ERROR;

}

ascend::utils::DvppBasicVpcPara dvpp\_basic\_vpc\_para;

/\*\*

\* when use dvpp\_process only for resize function:

\*

\* 1.DVPP limits crop\_right and crop\_down should be Odd number,

\* if it is even number, subtract 1, otherwise Equal to origin width

\* or height.

\*

\* 2.crop\_left and crop\_up should be set to zero.

\*/

dvpp\_basic\_vpc\_para.input\_image\_type = INPUT\_YUV420\_SEMI\_PLANNER\_UV; // nv12

dvpp\_basic\_vpc\_para.output\_image\_type = OUTPUT\_YUV420SP\_UV; // nv12

dvpp\_basic\_vpc\_para.src\_resolution.width = (int) src\_img.width;

dvpp\_basic\_vpc\_para.src\_resolution.height = (int) src\_img.height;

dvpp\_basic\_vpc\_para.dest\_resolution.width = kInputWidth;

dvpp\_basic\_vpc\_para.dest\_resolution.height = kInputHeight;

// DVPP limits crop\_left should be even number, 0 means without crop

dvpp\_basic\_vpc\_para.crop\_left = 0;

// DVPP limits crop\_right should be Odd number

dvpp\_basic\_vpc\_para.crop\_right =

src\_img.width % 2 == 0 ? src\_img.width - 1 : src\_img.width;

// DVPP limits crop\_up should be even number, 0 means without crop

dvpp\_basic\_vpc\_para.crop\_up = 0;

// DVPP limits crop\_down should be Odd number

dvpp\_basic\_vpc\_para.crop\_down =

src\_img.height % 2 == 0 ? src\_img.height - 1 : src\_img.height;

dvpp\_basic\_vpc\_para.is\_input\_align = true;

ascend::utils::DvppProcess dvpp\_process(dvpp\_basic\_vpc\_para);

ascend::utils::DvppVpcOutput dvpp\_out;

int ret = dvpp\_process.DvppBasicVpcProc(src\_img.data.get(),

(int32\_t) src\_img.size, &dvpp\_out);

if (ret != kDvppProcSuccess) {

HIAI\_ENGINE\_LOG(HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,

"[ODInferenceEngine] call dvpp resize failed with code %d!",

ret);

return HIAI\_ERROR;

}

// dvpp\_out->pbuf

resized\_img.data.reset(dvpp\_out.buffer, default\_delete<uint8\_t[]>());

resized\_img.size = dvpp\_out.size;

return HIAI\_OK;

}

HIAI\_StatusT ObjectDetectionInferenceEngine::SendDetectionResult(

shared\_ptr<DetectionEngineTransT> &detection\_trans, bool inference\_success,

string err\_msg) {

if (!inference\_success) {

// inference error.

HIAI\_ENGINE\_LOG(HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT, err\_msg.c\_str());

detection\_trans->status = false;

detection\_trans->msg = err\_msg;

}

HIAI\_StatusT ret;

do {

// convert the orginal image to JPEG

if(detection\_trans->status){

HIAI\_StatusT convert\_ret = ConvertImage(detection\_trans->video\_image.img);

if (convert\_ret != HIAI\_OK) {

HIAI\_ENGINE\_LOG(HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,

"Convert YUV Image to Jpeg failed!");

return HIAI\_ERROR;

}

}

// send trans\_data

std::shared\_ptr<EngineTransT> trans\_data = std::make\_shared<EngineTransT>();

uint32\_t frame\_id = detection\_trans->video\_image.video\_image\_info.frame\_id;

trans\_data->b\_info.is\_first = (frame\_id == 1);

trans\_data->b\_info.is\_last = detection\_trans->video\_image.video\_image\_info.is\_finished;

// handle one batch every time

trans\_data->b\_info.batch\_size = 1;

trans\_data->b\_info.max\_batch\_size = 1;

trans\_data->b\_info.batch\_ID = 0;

trans\_data->b\_info.channel\_ID = 0;//detection\_trans->video\_image.video\_image\_info.channel\_id;

trans\_data->b\_info.processor\_stream\_ID = 0;

trans\_data->b\_info.frame\_ID.push\_back(frame\_id);

trans\_data->b\_info.timestamp.push\_back(time(nullptr));

trans\_data->status = detection\_trans->status;

trans\_data->msg = detection\_trans->msg;

NewImageParaT img\_data;

img\_data.img = detection\_trans->video\_image.img;

trans\_data->imgs.push\_back(img\_data);

OutputT out;

out.size = detection\_trans->output\_datas[0].size;

out.data = detection\_trans->output\_datas[0].data;

trans\_data->output\_datas.push\_back(out);

// send data to next engine.

ret = SendData(kOutputPort, "EngineTransT",

static\_pointer\_cast<void>(trans\_data));

if (ret == HIAI\_QUEUE\_FULL) {

HIAI\_ENGINE\_LOG(HIAI\_DEBUG\_INFO, "[ODInferenceEngine] output queue full");

usleep(kWaitTimeShort);

}

} while (ret == HIAI\_QUEUE\_FULL);

if (ret != HIAI\_OK) {

HIAI\_ENGINE\_LOG(HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,

"[ODInferenceEngine] send inference data failed!");

return HIAI\_ERROR;

}

return HIAI\_OK;

}

void ObjectDetectionInferenceEngine::ObjectDetectInference() {

HIAI\_ENGINE\_LOG(

"[ODInferenceEngine] start object detection inference, queue size:%d",

yuv\_image\_queue.size());

while (!yuv\_image\_queue.empty()) {

// init inference results tensor shared\_ptr.

shared\_ptr<DetectionEngineTransT> detection\_trans = make\_shared<

DetectionEngineTransT>();

shared\_ptr<VideoImageParaT> video\_image = yuv\_image\_queue.Pop();

if (video\_image == nullptr) {

HIAI\_ENGINE\_LOG(

"[ODInferenceEngine] fail pop yuv image data from yuv\_image\_queue!");

continue;

}

detection\_trans->video\_image = \*video\_image;

// resize input image.

ImageData<u\_int8\_t> resized\_img;

HIAI\_StatusT dvpp\_ret = ImagePreProcess(detection\_trans->video\_image.img,

resized\_img);

if (dvpp\_ret != HIAI\_OK) {

// if preprocess error,send input image to the next engine.

SendDetectionResult(detection\_trans, false,

"[ODInferenceEngine] image preprocessed failure!");

return;

}

// inference object detection

if (PerformInference(detection\_trans, resized\_img) != HIAI\_OK) {

HIAI\_ENGINE\_LOG(

HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,

"[ODInferenceEngine] fail to inference object detection!");

}

}

}

HIAI\_IMPL\_ENGINE\_PROCESS("crowd\_counting", ObjectDetectionInferenceEngine,

INPUT\_SIZE) {

HIAI\_ENGINE\_LOG(HIAI\_DEBUG\_INFO, "[ODInferenceEngine] start process!");

if (arg0 == nullptr) {

// inputer data is nullptr.

HIAI\_ENGINE\_LOG(HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,

"[ODInferenceEngine] input data is null!");

return HIAI\_ERROR;

}

shared\_ptr<VideoImageParaT> video\_image =

static\_pointer\_cast<VideoImageParaT>(arg0);

if (video\_image->video\_image\_info.is\_finished) {

// input is finished.

return HandleFinishedData(video\_image);

}

// convert video frame to hfbc and put results to queue

if (!ConvertVideoFrameToHfbc(video\_image)) {

HIAI\_ENGINE\_LOG(

HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,

"[ODInferenceEngine] fail to call dvpp vdec convert h264 to hfbc!");

return HIAI\_ERROR;

}

// convert hfbc data to yuv image from queue, and detect object

ObjectDetectInference();

return HIAI\_OK;

}

### 5.3 人数统计的后处理核心代码

HIAI\_StatusT crowdcountingPostProcess::Init(

const hiai::AIConfig& config,

const std::vector<hiai::AIModelDescription>& model\_desc) {

HIAI\_ENGINE\_LOG("Begin initialize!");

// get configurations

if (fd\_post\_process\_config\_ == nullptr) {

fd\_post\_process\_config\_ = std::make\_shared<crowdcountingPostConfig>();

}

// get parameters from graph.config

for (int index = 0; index < config.items\_size(); index++) {

const ::hiai::AIConfigItem& item = config.items(index);

const std::string& name = item.name();

const std::string& value = item.value();

std::stringstream ss;

ss << value;

if (name == "Confidence") {

ss >> (\*fd\_post\_process\_config\_).confidence;

// validate confidence

if (IsInvalidConfidence(fd\_post\_process\_config\_->confidence)) {

HIAI\_ENGINE\_LOG(HIAI\_GRAPH\_INVALID\_VALUE,

"Confidence=%s which configured is invalid.",

value.c\_str());

return HIAI\_ERROR;

}

} else if (name == "PresenterIp") {

// validate presenter server IP

if (IsInValidIp(value)) {

HIAI\_ENGINE\_LOG(HIAI\_GRAPH\_INVALID\_VALUE,

"PresenterIp=%s which configured is invalid.",

value.c\_str());

return HIAI\_ERROR;

}

ss >> (\*fd\_post\_process\_config\_).presenter\_ip;

} else if (name == "PresenterPort") {

ss >> (\*fd\_post\_process\_config\_).presenter\_port;

// validate presenter server port

if (IsInValidPort(fd\_post\_process\_config\_->presenter\_port)) {

HIAI\_ENGINE\_LOG(HIAI\_GRAPH\_INVALID\_VALUE,

"PresenterPort=%s which configured is invalid.",

value.c\_str());

return HIAI\_ERROR;

}

} else if (name == "ChannelName") {

// validate channel name

if (IsInValidChannelName(value)) {

HIAI\_ENGINE\_LOG(HIAI\_GRAPH\_INVALID\_VALUE,

"ChannelName=%s which configured is invalid.",

value.c\_str());

return HIAI\_ERROR;

}

ss >> (\*fd\_post\_process\_config\_).channel\_name;

}

// else : nothing need to do

}

// call presenter agent, create connection to presenter server

uint16\_t u\_port = static\_cast<uint16\_t>(fd\_post\_process\_config\_

->presenter\_port);

OpenChannelParam channel\_param = { fd\_post\_process\_config\_->presenter\_ip,

u\_port, fd\_post\_process\_config\_->channel\_name, ContentType::kVideo };

Channel \*chan = nullptr;

PresenterErrorCode err\_code = OpenChannel(chan, channel\_param);

// open channel failed

if (err\_code != PresenterErrorCode::kNone) {

HIAI\_ENGINE\_LOG(HIAI\_GRAPH\_INIT\_FAILED,

"Open presenter channel failed, error code=%d", err\_code);

return HIAI\_ERROR;

}

presenter\_channel\_.reset(chan);

HIAI\_ENGINE\_LOG(HIAI\_DEBUG\_INFO, "End initialize!");

return HIAI\_OK;

}

int32\_t crowdcountingPostProcess::SendImage(uint32\_t height, uint32\_t width,

uint32\_t size, u\_int8\_t \*data, std::vector<DetectionResult>& detection\_results) {

int32\_t status = kFdFunSuccess;

// parameter

ImageFrame image\_frame\_para;

image\_frame\_para.format = ImageFormat::kJpeg;

image\_frame\_para.width = width;

image\_frame\_para.height = height;

image\_frame\_para.size = size;

image\_frame\_para.data = data;

image\_frame\_para.detection\_results = detection\_results;

PresenterErrorCode p\_ret = PresentImage(presenter\_channel\_.get(),

image\_frame\_para);

// send to presenter failed

if (p\_ret != PresenterErrorCode::kNone) {

HIAI\_ENGINE\_LOG(HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,

"Send JPEG image to presenter failed, error code=%d",

p\_ret);

status = kFdFunFailed;

}

return status;

}

HIAI\_StatusT crowdcountingPostProcess::HandleOriginalImage(

const std::shared\_ptr<EngineTransT> &inference\_res) {

HIAI\_StatusT status = HIAI\_OK;

std::vector<NewImageParaT> img\_vec = inference\_res->imgs;

// dealing every original image

for (uint32\_t ind = 0; ind < inference\_res->b\_info.batch\_size; ind++) {

uint32\_t width = img\_vec[ind].img.width;

uint32\_t height = img\_vec[ind].img.height;

uint32\_t size = img\_vec[ind].img.size;

// call SendImage

// 1. call DVPP to change YUV420SP image to JPEG

// 2. send image to presenter

vector<DetectionResult> detection\_results;

int32\_t ret = SendImage(height, width, size, img\_vec[ind].img.data.get(), detection\_results);

if (ret == kFdFunFailed) {

status = HIAI\_ERROR;

continue;

}

}

return status;

}

HIAI\_StatusT crowdcountingPostProcess::HandleResults(

const std::shared\_ptr<EngineTransT> &inference\_res) {

HIAI\_StatusT status = HIAI\_OK;

std::vector<NewImageParaT> img\_vec = inference\_res->imgs;

std::vector<OutputT> output\_data\_vec = inference\_res->output\_datas;

// dealing every image

for (uint32\_t ind = 0; ind < inference\_res->b\_info.batch\_size; ind++) {

// result

int32\_t out\_index = ind \* kDealResultIndex;

OutputT out = output\_data\_vec[out\_index];

std::shared\_ptr<hiai::AISimpleTensor> result\_tensor = std::make\_shared<

hiai::AISimpleTensor>();

result\_tensor->SetBuffer(out.data.get(), out.size);

int32\_t size = result\_tensor->GetSize() / sizeof(float);

float result[size];

errno\_t mem\_ret = memcpy\_s(result, sizeof(result),

result\_tensor->GetBuffer(),

result\_tensor->GetSize());

// memory copy failed, skip this image

if (mem\_ret != EOK) {

HIAI\_ENGINE\_LOG(HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,

"handle results: memcpy\_s() error=%d", mem\_ret);

continue;

}

uint32\_t width = img\_vec[ind].img.width;

uint32\_t height = img\_vec[ind].img.height;

uint32\_t img\_size = img\_vec[ind].img.size;

// every inference result needs 8 float

// loop the result for every inference result

std::vector<DetectionResult> detection\_results;

float \*ptr = result;

float sum = 0;

for (int i = 0; i < size; i++) {

sum += result[i];

}

ptr = result;

float attr = ptr[0];

float score = ptr[0];

DetectionResult one\_result;

int intsum = (int)sum;

one\_result.result\_text.append(kFaceLabelTextPrefix);

one\_result.result\_text.append(to\_string(intsum));

one\_result.result\_text.append(kFaceLabelTextSuffix);

detection\_results.emplace\_back(one\_result);

int32\_t ret;

ret = SendImage(height, width, img\_size, img\_vec[ind].img.data.get(), detection\_results);

// check send result

if (ret == kFdFunFailed) {

status = HIAI\_ERROR;

}

}

return status;

}

HIAI\_IMPL\_ENGINE\_PROCESS("crowd\_counting\_post\_process",

crowdcountingPostProcess, INPUT\_SIZE) {

// check arg0 is null or not

if (arg0 == nullptr) {

HIAI\_ENGINE\_LOG(HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,

"Failed to process invalid message.");

return HIAI\_ERROR;

}

// check original image is empty or not

std::shared\_ptr<EngineTransT> inference\_res = std::static\_pointer\_cast<

EngineTransT>(arg0);

if (inference\_res->imgs.empty()) {

HIAI\_ENGINE\_LOG(

HIAI\_ENGINE\_RUN\_ARGS\_NOT\_RIGHT,

"Failed to process invalid message, original image is null.");

return HIAI\_ERROR;

}

// inference failed, dealing original images

if (!inference\_res->status) {

HIAI\_ENGINE\_LOG(HIAI\_OK, inference\_res->msg.c\_str());

HIAI\_ENGINE\_LOG(HIAI\_OK, "will handle original image.");

return HandleOriginalImage(inference\_res);

}

// inference success, dealing inference results

return HandleResults(inference\_res);

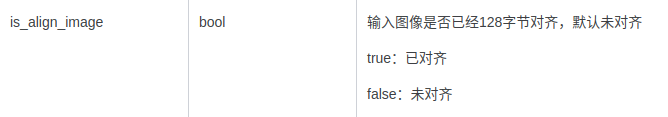
}

## 6 重要问题及解决

问题1：presenter server上显示为乱码

解答：在使用DVPP进行图片处理时，没有把对齐开关打开

即 dvpp\_to\_jpeg\_para.is\_align\_image = true;



## 7 后续可扩展性

密集人群计数可以针对目前世界各地发生的大规模踩踏事件进行预防，从图像或者视频中准确估计人群已成为计算机视觉技术在人群控制和公共安全中日益重要的应用。在某些情况下，如公众集会和体育赛事，参与人数或密度是未来赛事规划和空间设计的重要信息。好的人群计数方法也可以扩展到其他领域，例如，从显微图像中计数细胞或细菌，野生动物保护区中的动物人群估计，或估计交通枢纽或交通堵塞的车辆数量等。