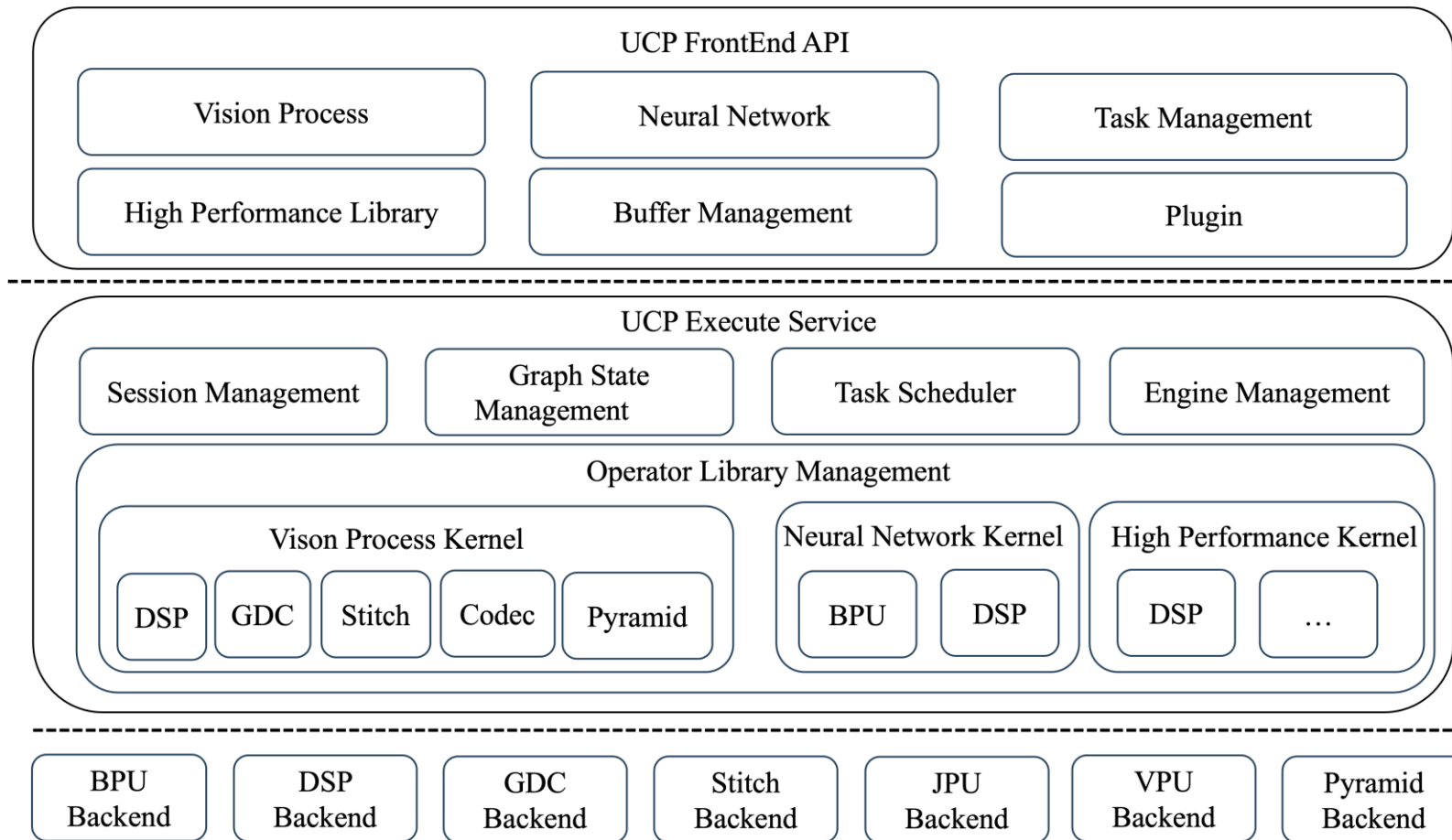


# 天工开物 J6算法工具链

模型板端部署详解-UCP

# UCP简介-功能架构



统一计算平台（Unify Compute Platform，以下简称 UCP）定义了一套高度抽象和集成的统一异构编程接口，提供API实现对SoC上所有资源的调用。UCP将SoC上的功能硬件抽象出来并进行封装，对外提供基于功能的API，用于创建相应的UCP任务（如VP算子任务、模型推理任务等），并支持设置硬件Backend提交至UCP调度器，UCP可基于硬件资源，完成SoC上任务的统一调度。具体提供了以下几个功能：视觉处理(Vision Process)、神经网络模型推理（Neural Network）、高性能计算库（High Performance Library）、自定义算子插件开发（开发中）。

# UCP简介-Backend

| Backend | 描述                                                                         |
|---------|----------------------------------------------------------------------------|
| BPU     | Brain Process Unit, 地平线神经网络计算单元                                            |
| DSP     | Digital Signal Processor, 数字信号处理器, 是一个可编程的硬件单元                             |
| GDC     | Geometric Distortion Correction, 是ARM上的一个硬件IP, 可将输入图像进行视角变换、畸变矫正、图像仿射变换等操作 |
| STITCH  | stitch是J6的一个IP单元, 可对输入的图像进行裁剪, 拼接, 拼接模式分别有: alpha融合、alpha beta融合、直接拷贝      |
| JPU     | JPEG Processing Unit, 主要用以完成JPEG的编解码功能                                     |
| VPU     | Video Processing Unit, 是一种专用的视觉处理单元                                        |
| PYRAMID | 全称Image Pyramid, 是一个硬件处理模块, 可对整幅原始图像进行缩小                                   |

Backend是指UCP任务执行时的后端计算硬件, 当前UCP支持的Backend包括BPU、DSP、GDC、STITCH、JPU、VPU、PYRAMID。

# UCP简介-交付物

## 开发环境:

| 环境/工具  | 支持版本                    |
|--------|-------------------------|
| 操作系统   | Linux                   |
| 开发板    | J6开发板                   |
| 开发语言   | C++11                   |
| 交叉编译器  | Linaro 11.4.0           |
| 工具链DSP | Cadence Vsion Q8 2021.7 |

注：x86端UCP仿真使用Docker  
镜像自带的编译器环境即可

## OE包示例:

```
samples/ucp_tutorial/ # UCP示例
├─ deps_aarch64 # aarch64依赖库
├─ deps_x86 # x86仿真依赖库
├─ dnn # 神经网络模型推理 (Neural Network) 推理示例
│   └─ ai_benchmark
│   └─ basic_samples
├─ hpl # 高性能计算库算子示例
│   └─ README.md
│   └─ code
│   └─ hpl_samples
├─ tools # 工具目录, 包含hrt_model_exec、hrt_ucp_monitor、trace工具
│   └─ hrt_model_exec
│   └─ monitor
│   └─ trace
└─ vp # 视觉处理算子示例
    └─ code
    └─ vp_samples
```

注：当前为试用版本内容，正式版  
本将会有更丰富内容。

# 模型板端部署详解-UCP

神经网络模型推理

视觉处理

高性能计算库

UCP工具

# 神经网络模型推理-注意事项

由于J6工具链编译器架构和硬件对齐规则发生了比较大的变化，模型的板端部署方式相比以前版本例如J5发生了改变，体现在模型部署侧有以下几点：

- Pyramid输入和Resizer输入需沿Batch维度拆分，例如1个Batch 8输入分支将拆分为8个Batch 1的输入分支；
- Pyramid输入的stride信息存在动态维度，Resizer输入的valid shape和stride信息存在动态维度，需指定具体数值（使用hrt\_model\_exec工具通过input\_valid\_shape和input\_stride参数传入），并且stride需32位对齐。此外原来的aligned shape信息需通过valid shape和stride解析，无法直接从properties获取，并且aligned shape会在后期废弃；
- Pyramid输入的nv12数据将拆分为y和uv两个分量输入；Resizer输入的nv12数据将拆分为y、uv和roi三个分量输入，需分别准备输入数据，例如1个Batch 8 Pyramid单输入的模型，部署时模型的输入分支将是16个。

stride的计算方式如右图：

$$stride[idx] \geq stride[idx + 1] * validShape.dimensionSize[idx + 1]$$

以y分量为例：

valid shape为(1, 224, 224, 1)，stride为(-1, -1, 1, 1)，stride[0]和stride[1]为动态，因此

$$stride[1] = \text{ALIGN\_32}(stride[2] * \text{validShape.dimensionSize}[2]) = \text{ALIGN\_32}(1 * 224) = 224$$

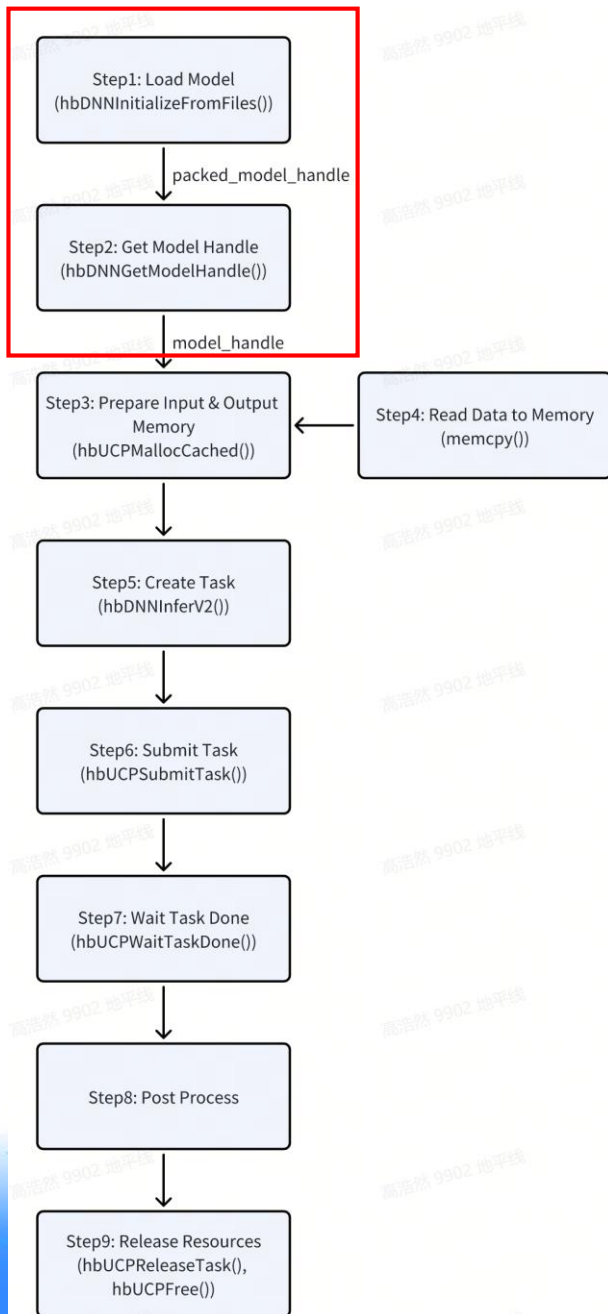
$$stride[0] = \text{ALIGN\_32}(stride[1] * \text{validShape.dimensionSize}[1]) = \text{ALIGN\_32}(224 * 224) = 50176$$

# 神经网络模型推理-基本流程



# 神经网络模型推理-示例解读

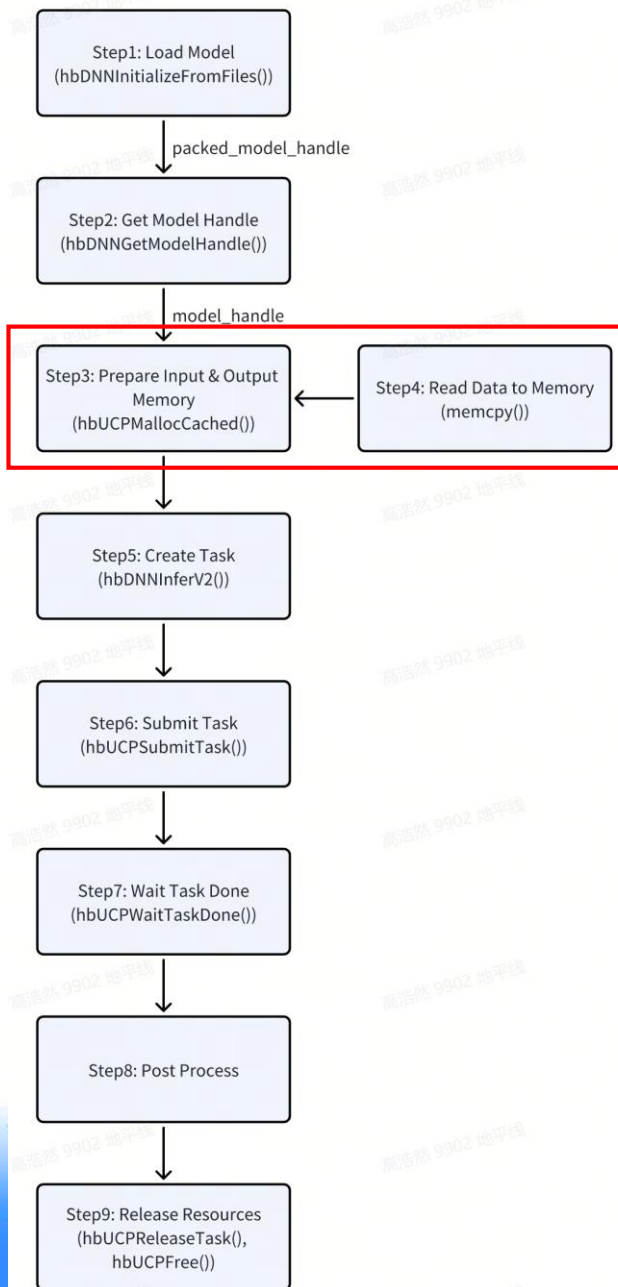
我们以samples/ucp\_tutorial/dnn/basic\_samples/code/00\_quick\_start为例，使用ResNet50进行单张图片模型推理和结果解析：



```
int main(int argc, char **argv) {  
    // Parsing command line arguments  
    gflags::SetUsageMessage(argv[0]);  
    gflags::ParseCommandLineFlags(&argc, &argv, true);  
    std::cout << gflags::GetArgv() << std::endl;  
  
    // Init logging  
    hobot::hlog::HobotLog::Instance()->SetLogLevel(  
        "DNN_BASIC_SAMPLE", hobot::hlog::LogLevel::log_info);  
  
    hbdnnPackedHandle_t packed_dnn_handle;  
    hbdnnHandle_t dnn_handle;  
    const char **model_name_list;  
    auto modelFileName = FLAGS_model_file.c_str();  
    int model_count = 0;  
  
    // Step1: get model handle  
    {  
        HB_CHECK_SUCCESS(  
            hbDNNInitializeFromFiles(&packed_dnn_handle, &modelFileName, 1),  
            "hbDNNInitializeFromFiles failed");  
        HB_CHECK_SUCCESS(hbDNNGetModelNameList(&model_name_list, &model_count,  
            packed_dnn_handle),  
            "hbDNNGetModelNameList failed");  
        HB_CHECK_SUCCESS(  
            hbDNNGetModelHandle(&dnn_handle, packed_dnn_handle, model_name_list[0]),  
            "hbDNNGetModelHandle failed");  
    }  
}
```

# 神经网络模型推理-示例解读

我们以samples/ucp\_tutorial/dnn/basic\_samples/code/00\_quick\_start为例，使用ResNet50进行单张图片模型推理和结果解析：

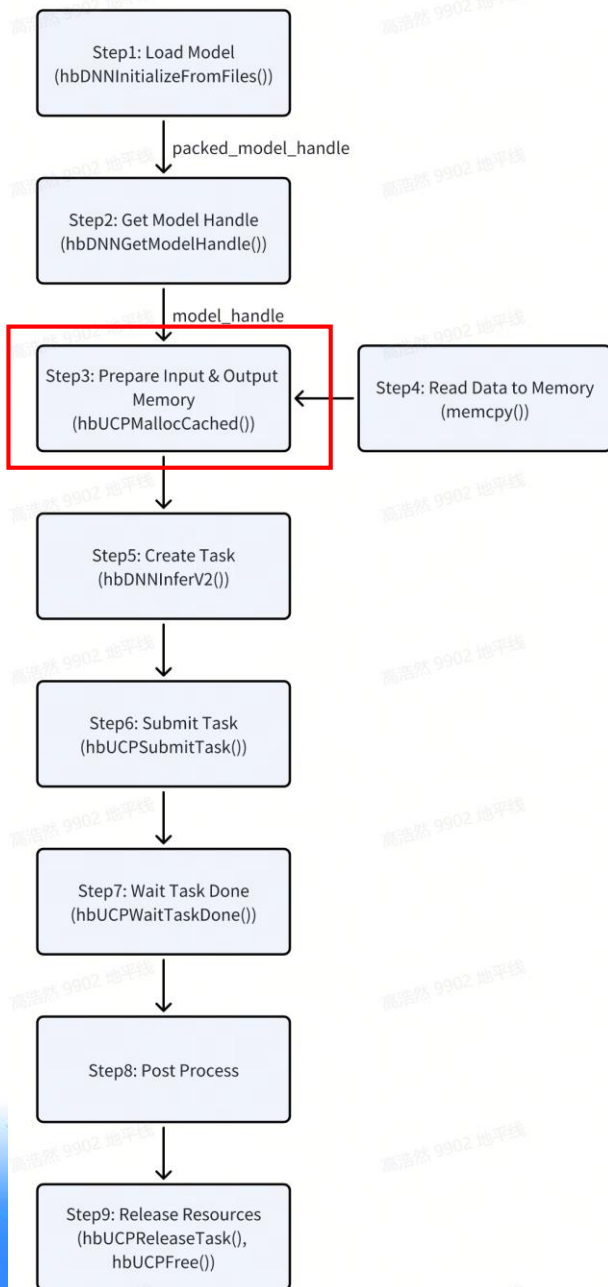


```
std::vector<hbDNNTensor> input_tensors;
std::vector<hbDNNTensor> output_tensors;
int input_count = 0;
int output_count = 0;
// Step2: prepare input and output tensor
{
    HB_CHECK_SUCCESS(hbDNNGetInputCount(&input_count, dnn_handle),
        "hbDNNGetInputCount failed");
    HB_CHECK_SUCCESS(hbDNNGetOutputCount(&output_count, dnn_handle),
        "hbDNNGetOutputCount failed");
    input_tensors.resize(input_count);
    output_tensors.resize(output_count);
    prepare_tensor(input_tensors.data(), output_tensors.data(), dnn_handle);
}

// Step3: set input data to input tensor
{
    // read a single picture for input_tensor[0], for multi_input model, you
    // should set other input data according to model input properties.
    HB_CHECK_SUCCESS(
        read_image_2_tensor_as_nv12(FLAGS_image_file, input_tensors.data()),
        "read_image_2_tensor_as_nv12 failed");
    LOGI("read image to tensor as nv12 success");
}
```

# 神经网络模型推理-示例解读

我们以samples/ucp\_tutorial/dnn/basic\_samples/code/00\_quick\_start为例，使用ResNet50进行单张图片模型推理和结果解析：



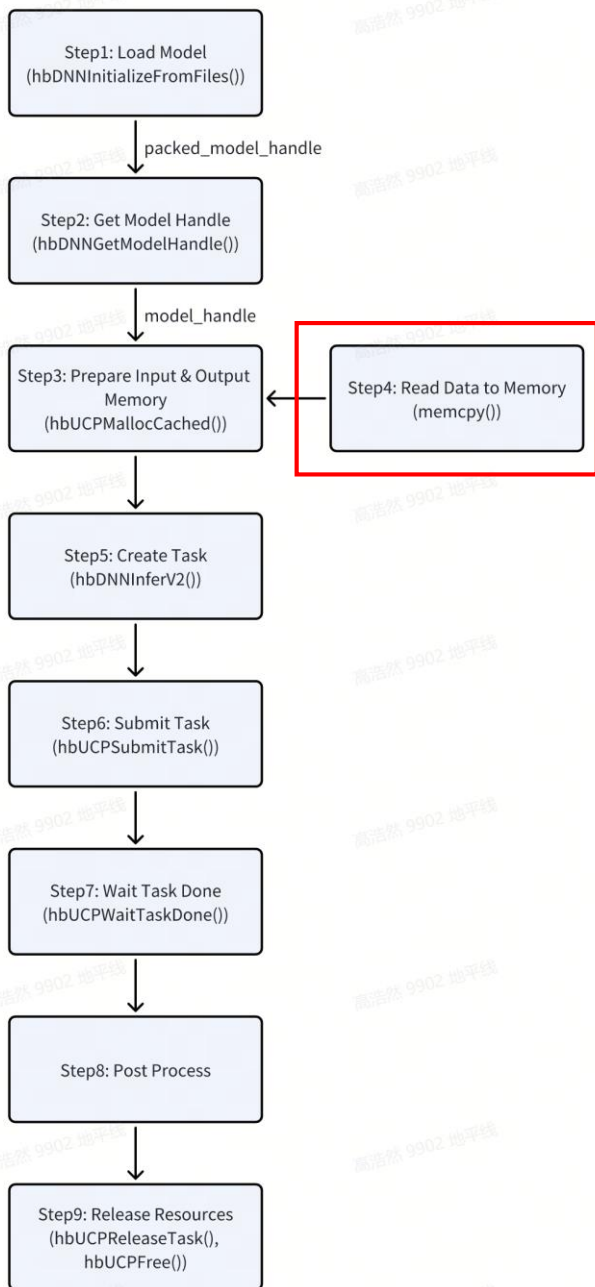
```
int prepare_tensor(hbDNNTensor *input_tensor, hbDNNTensor *output_tensor,
/** Tips:
    * For output memory size.
    * *   output_memSize = output[i].properties.alignedByteSize
    */
hbDNNTensor *input = input_tensor;
for (int i = 0; i < input_count; i++) {
    HB_CHECK_SUCCESS(
        hbDNNGetInputTensorProperties(&input[i].properties, dnn_handle, i),
        "hbDNNGetInputTensorProperties failed");

/** Tips:
    * For input tensor, usually need to pad the input data according to stride obtained
    * but here for dynamic stride of y and uv, user needs to specify a value which should
    */
    auto dim_len = input[i].properties.validShape.numDimensions;
    for (int32_t dim_i = dim_len - 1; dim_i >= 0; --dim_i) {
        if (input[i].properties.stride[dim_i] == -1) {
            auto cur_stride =
                input[i].properties.stride[dim_i + 1] *
                input[i].properties.validShape.dimensionSize[dim_i + 1];
            input[i].properties.stride[dim_i] = ALIGN_32(cur_stride);
        }
    }

    int input_memSize = input[i].properties.stride[0] *
        input[i].properties.validShape.dimensionSize[0];
    HB_CHECK_SUCCESS(hbUCPMallocCached(&input[i].sysMem[0], input_memSize, 0),
        "hbUCPMallocCached failed");
```

# 神经网络模型推理-示例解读

我们以samples/ucp\_tutorial/dnn/basic\_samples/code/00\_quick\_start为例，使用ResNet50进行单张图片模型推理和结果解析：



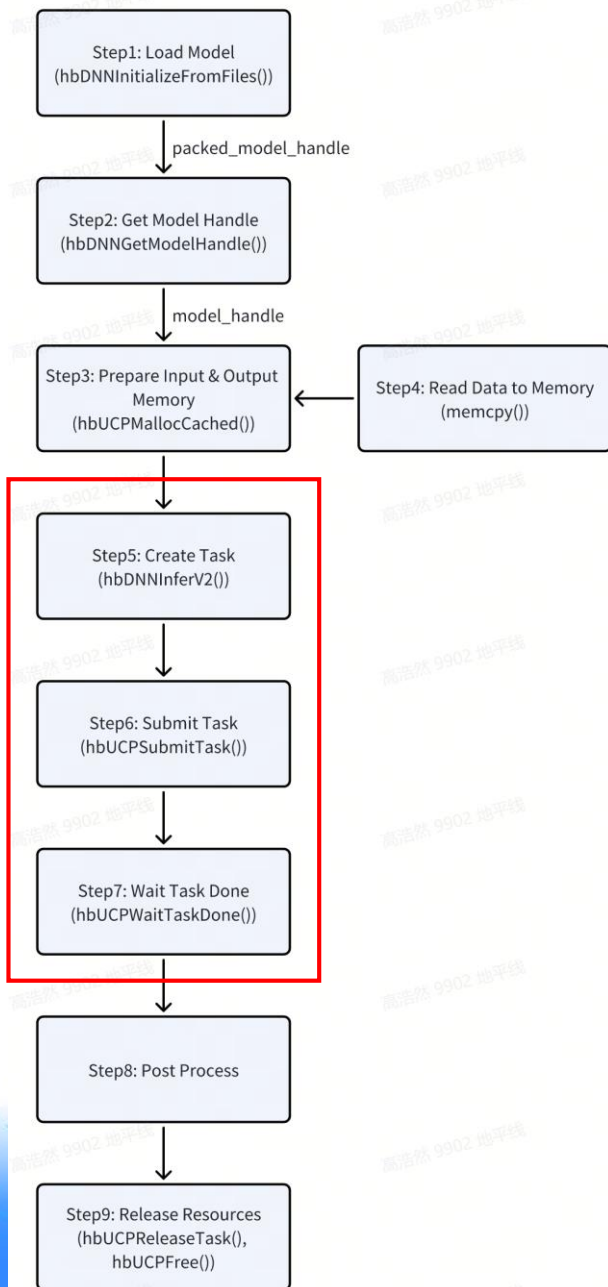
```
int32_t read_image_2_tensor_as_nv12(std::string &image_file,
                                     int input_h, int input_w, int input_c) {
    cv::Mat yuv_mat;
    cv::cvtColor(mat, yuv_mat, cv::COLOR_BGR2YUV_I420);
    uint8_t *yuv_data = yuv_mat.ptr<uint8_t>();
    uint8_t *y_data_src = yuv_data;
```

```
// copy y data
uint8_t *y_data_dst =
    reinterpret_cast<uint8_t *>(input_tensor[0].sysMem[0].virAddr);
for (int32_t h = 0; h < input_h; ++h) {
    memcpy(y_data_dst, y_data_src, input_w);
    y_data_src += input_w;
    // add padding
    y_data_dst += input_tensor[0].properties.stride[1];
}
```

```
// copy uv data
int32_t uv_height = input_tensor[1].properties.validShape.dimensionSize[1];
int32_t uv_width = input_tensor[1].properties.validShape.dimensionSize[2];
uint8_t *uv_data_dst =
    reinterpret_cast<uint8_t *>(input_tensor[1].sysMem[0].virAddr);
uint8_t *u_data_src = yuv_data + input_h * input_w;
uint8_t *v_data_src = u_data_src + uv_height * uv_width;
```

# 神经网络模型推理-示例解读

我们以samples/ucp\_tutorial/dnn/basic\_samples/code/00\_quick\_start为例，使用ResNet50进行单张图片模型推理和结果解析：



```
hbUCPTaskHandle_t task_handle{nullptr};
hbDNNTensor *output = output_tensors.data();
// Step4: run inference
{
    // make sure memory data is flushed to DDR before inference
    for (int i = 0; i < input_count; i++) {
        hbUCPMemFlush(&input_tensors[i].sysMem[0], HB_SYS_MEM_CACHE_CLEAN);
    }

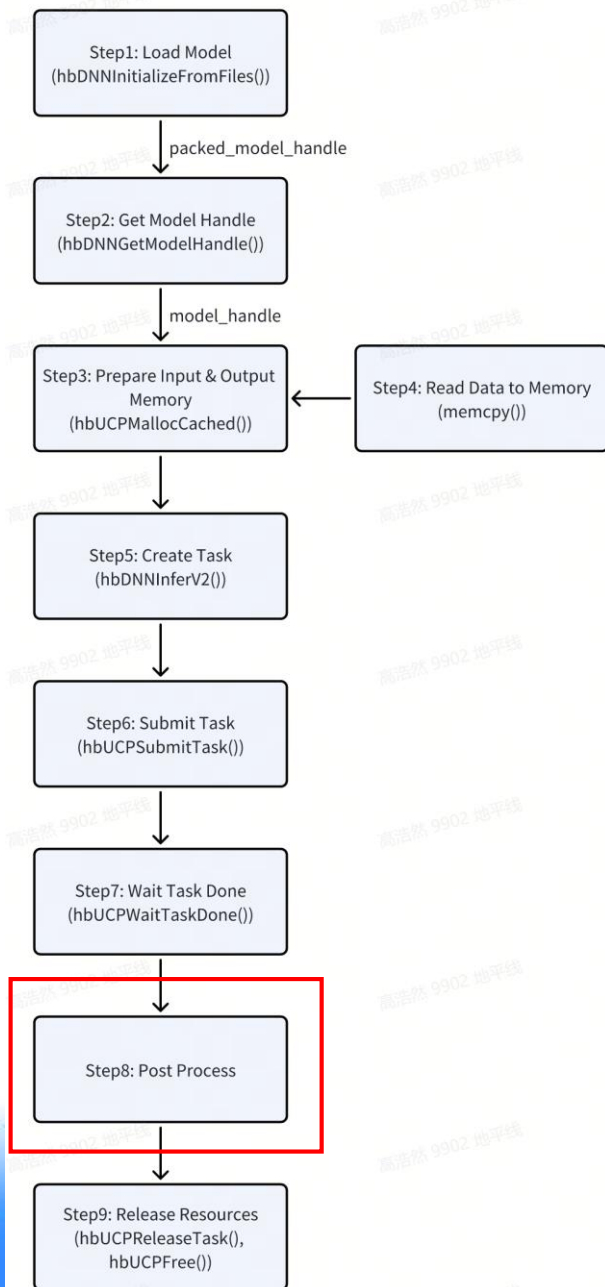
    // generate task handle
    HB_CHECK_SUCCESS(
        hbDNNInferV2(&task_handle, output, input_tensors.data(), dnn_handle),
        "hbDNNInferV2 failed");

    // submit task
    hbUCPSchedParam ctrl_param;
    HB_UCP_INITIALIZE_SCHED_PARAM(&ctrl_param);
    ctrl_param.backend = HB_UCP_BPU_CORE_ANY;
    HB_CHECK_SUCCESS(hbUCPSubmitTask(task_handle, &ctrl_param),
        "hbUCPSubmitTask failed");

    // wait task done
    HB_CHECK_SUCCESS(hbUCPWaitTaskDone(task_handle, 0),
        "hbUCPWaitTaskDone failed");
}
```

# 神经网络模型推理-示例解读

我们以samples/ucp\_tutorial/dnn/basic\_samples/code/00\_quick\_start为例，使用ResNet50进行单张图片模型推理和结果解析：

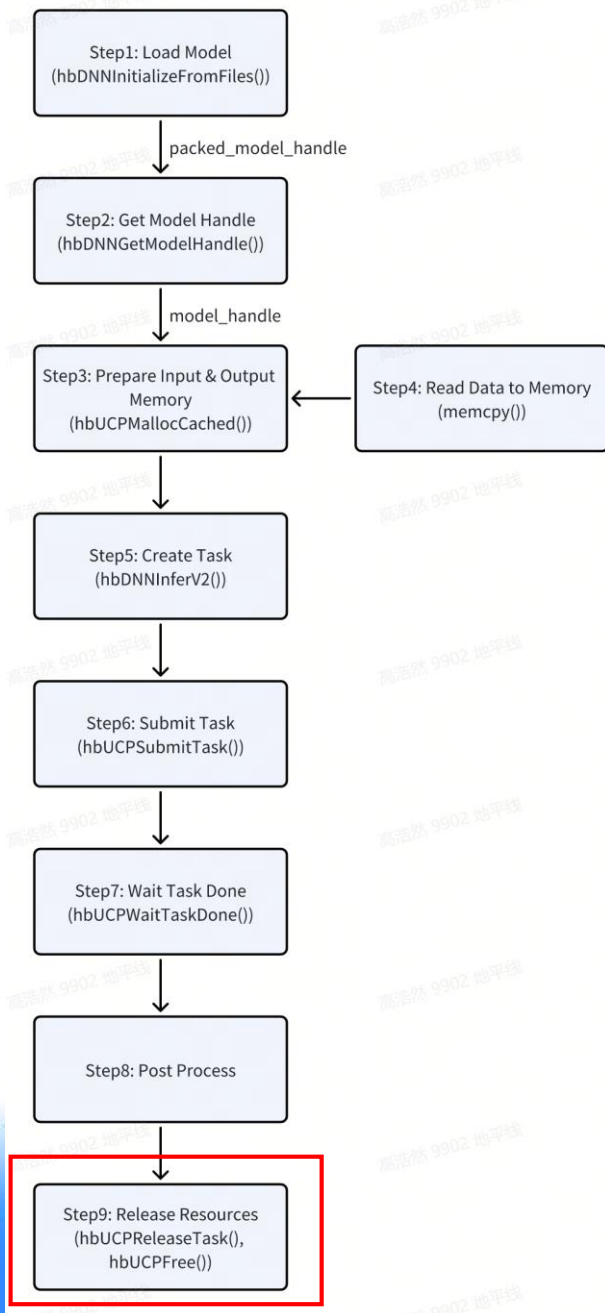


```
// Step5: do postprocess with output data
std::vector<Classification> top_k_cls;
{
    // make sure CPU read data from DDR before using output tensor data
    for (int i = 0; i < output_count; i++) {
        hbUCPMemFlush(&output_tensors[i].sysMem[0], HB_SYS_MEM_CACHE_INVALIDATE);
    }

    get_topk_result(output, top_k_cls, FLAGS_top_k);
    for (int i = 0; i < FLAGS_top_k; i++) {
        LOGI("TOP {} result id: {}", i, top_k_cls[i].id);
    }
}
```

# 神经网络模型推理-示例解读

我们以samples/ucp\_tutorial/dnn/basic\_samples/code/00\_quick\_start为例，使用ResNet50进行单张图片模型推理和结果解析：



```
// Step6: release resources
{
    // release task handle
    HB_CHECK_SUCCESS(hbUCPReleaseTask(task_handle), "hbUCPReleaseTask failed");
    // free input mem
    for (int i = 0; i < input_count; i++) {
        HB_CHECK_SUCCESS(hbUCPFree(&(input_tensors[i].sysMem[0])),
            "hbUCPFree failed");
    }
    // free output mem
    for (int i = 0; i < output_count; i++) {
        HB_CHECK_SUCCESS(hbUCPFree(&(output_tensors[i].sysMem[0])),
            "hbUCPFree failed");
    }
    // release model
    HB_CHECK_SUCCESS(hbDNNRelease(packed_dnn_handle), "hbDNNRelease failed");
}
```

# 神经网络模型推理-基础示例交付物

基础示例包位于 horizon\_j6\_open\_explorer 发布物的 samples/ucp\_tutorial/dnn/basic\_samples/ 路径下，主要包括以下内容：

```
basic_samples/
├── README.md
├── code # 示例源码
│   ├── 00_quick_start # 快速入门示例, 用resnet50读取单张图片进行推理的示例代码
│   │   ├── CMakeLists.txt
│   │   └── src
│   ├── 01_api_tutorial
│   │   ├── CMakeLists.txt
│   │   ├── mem
│   │   ├── model
│   │   └── roi_infer
│   ├── CMakeLists.txt
│   ├── build.sh # 编译脚本
│   ├── build_aarch64.sh # 编译脚本, aarch64环境运行
│   ├── build_x86.sh # 编译脚本, x86环境运行
│   └── resolve.sh # 运行依赖获取脚本
└── runtime
    ├── model
    │   ├── README.md
    │   └── runtime -> ../../../../../../model_zoo/runtime/basic_samples/
    ├── script
    │   ├── 00_quick_start
    │   ├── 01_api_tutorial
    │   └── README.md
    └── script_x86
        ├── 00_quick_start
        ├── 01_api_tutorial
        └── README.md
```

注：当前为试用版本内容，更多示例以及算子开发中。

# 神经网络模型推理-AI Benchmark交付物

AI Benchmark位于 horizon\_j6\_open\_explorer 发布物的 samples/ucp\_tutorial/dnn/ai\_benchmark/ 路径下，主要包括以下内容：

```
ai_benchmark/  
├── README.md  
├── code  
│   ├── CMakeLists.txt  
│   ├── README.md  
│   ├── build.sh  
│   ├── build_ptq_j6.sh # PTQ方案程序一键编译脚本  
│   ├── build_qat_j6.sh # QAT方案程序一键编译脚本  
│   ├── include  
│   ├── resolve.sh # 下载模型性能评测数据集的脚本  
│   └── src # 示例源码  
└── j6  
    ├── ptq # PTQ方案模型示例  
    │   ├── README.md  
    │   ├── data # 模型评测数据集  
    │   ├── model  
    │   ├── script # 示例执行脚本  
    │   └── tools # 精度评测工具  
    └── qat # QAT方案模型示例  
        ├── README.md  
        ├── data  
        ├── model  
        ├── script  
        └── tools
```

注：当前为试用版本内容，更多示例以及算子开发中。

# 模型板端部署详解-UCP

神经网络模型推理

视觉处理

高性能计算库

UCP工具

# 视觉处理

视觉处理VP模块在以NN为中心的计算方案中，主要作用于模型的前后处理环节。在地平线统一架构中，多种硬件均已搭载了图像处理的算子，而VP模块将图像处理相关的硬件调用进行了封装，通过设置backend来选择不同的硬件方案（若不指定backend，UCP会自动适配负载更低的处理单元），从而平衡开发板负载，充分对开发板算力进行挖掘，规避了不同硬件调用区别带来的不便

Remap

Resize

Codec

Affine

Pyrdown

.....

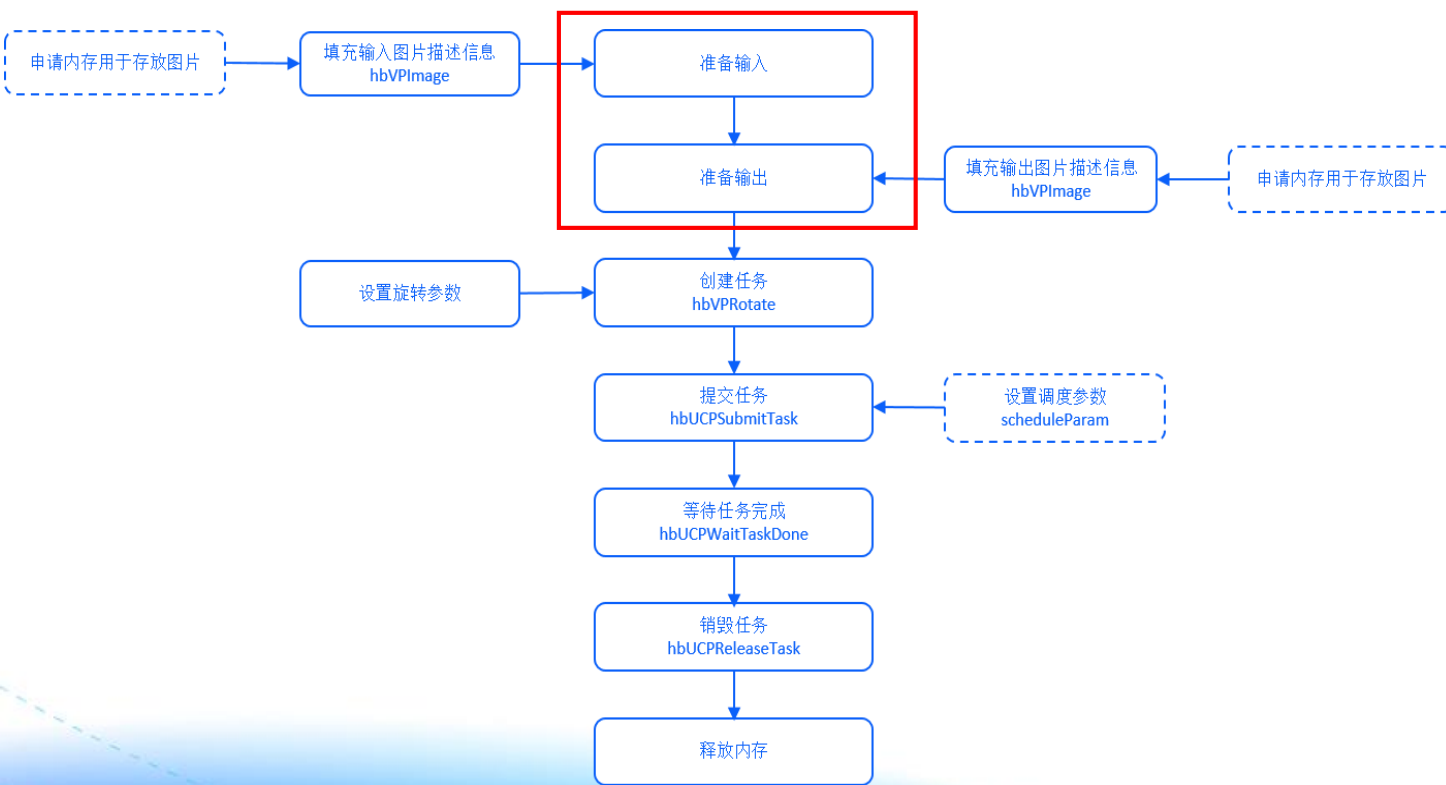
注：VP算子相关性能指标请参考用户手册

# 视觉处理-基本流程



# 视觉处理-示例解读

我们通过一个简单的算子调用展示了如何使用VP封装的算子实现图片处理的功能，使用hbVPRotate算子将图片旋转90度：



```
#include <cstring>

#include "img_util.h"
#include "log_util.h"
#include "opencv2/opencv.hpp"
#include "hobot/vp/hb_vp_rotate.h"

// 读取输入图片
std::string src_img = "../../data/images/input.jpg";
cv::Mat src_mat = cv::imread(src_img.c_str(), cv::IMREAD_GRAYSCALE);
LOGE_AND_RETURN_IF(src_mat.empty(), HB_UCP_INVALID_ARGUMENT,
                    "Read image {} failed", src_img.c_str());
LOGI("Read image {} success", src_img);

const int32_t src_width = src_mat.cols;
const int32_t src_height = src_mat.rows;
const int32_t src_stride = src_width;
const int32_t dst_width = src_height;
const int32_t dst_height = src_width;
const int32_t dst_stride = dst_width;

// 初始化输入输出内存，并将图片填充到输入内存中
hbUCPSysMem src_mem, dst_mem;
hbUCPMallocCached(&src_mem, src_stride * src_height, 0);
hbUCPMallocCached(&dst_mem, dst_stride * dst_height, 0);
memcpy(src_mem.virAddr, src_mat.data, src_stride * src_height);
hbUCPMemFlush(&src_mem, HB_SYS_MEM_CACHE_CLEAN);
```

# 视觉处理-示例解读

我们通过一个简单的算子调用展示了如何使用VP封装的算子实现图片处理的功能，使用hbVPRotate算子将图片旋转90度：



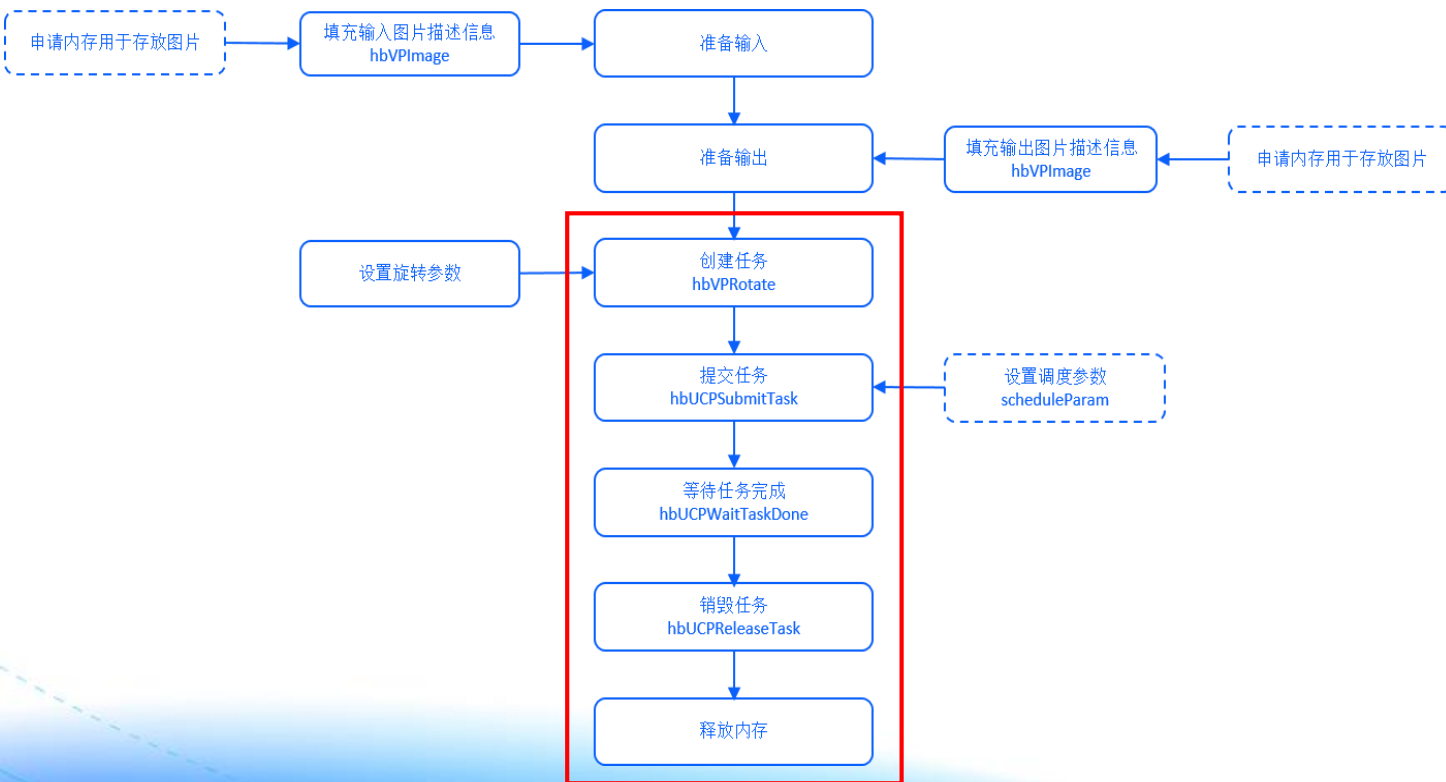
```
// 填充输入图片信息
hbVPImage src{HB_VP_IMAGE_FORMAT_Y,
               HB_VP_IMAGE_TYPE_U8C1,
               src_width,
               src_height,
               src_stride,
               src_mem.virAddr,
               src_mem.phyAddr,
               0,
               0,
               0};

// 填充输出图片信息
hbVPImage dst{HB_VP_IMAGE_FORMAT_Y,
               HB_VP_IMAGE_TYPE_U8C1,
               dst_width,
               dst_height,
               dst_stride,
               dst_mem.virAddr,
               dst_mem.phyAddr,
               0,
               0,
               0};

// 填充算子参数
int8_t rotate_code = HB_VP_ROTATE_90_CLOCKWISE;
```

# 视觉处理-示例解读

我们通过一个简单的算子调用展示了如何使用VP封装的算子实现图片处理的功能，使用hbVPRotate算子将图片旋转90度：



```
// 通过VP提供的算子接口创建任务，任务句柄支持设置为nullptr，使用同步模式执行此任务
hbUCPTaskHandle_t rotate_task{nullptr}; // UCP任务句柄
hbVPRotate(&rotate_task/*任务句柄*/,
           &dst/*输出图片*/,
           &src/*输入图片*/,
           rotate_code/*算子参数*/);

// 设置调度参数调整任务优先级、选择执行终端等
hbScheParam sched_param;
HB_UCP_INITIALIZE_SCHED_PARAM(&sched_param);
sched_param.backend = HB_UCP_DSP_CORE_0; /*指定执行设备ID*/

// 提交任务，sche_param 支持设置为nullptr，使用默认调度参数
hbUCPSubmitTask(rotate_task, &sched_param);

// 等待任务完成，设置超时参数，值为0时表示一直等待
hbUCPWaitTaskDone(rotate_task, 500);

// 释放任务句柄
hbUCPReleaseTask(rotate_task);

// 释放内存资源
hbUCPFree(&src_mem);
hbUCPFree(&dst_mem);
```

# 视觉处理-交付物

## OE包示例:

```
└─ vp                                // vp示例
  └─ code                            // 示例源码目录
    └─ 01_basic_processing           // 基础图像处理示例
    └─ 02_transformation             // 图像变换示例
    └─ 03_feature_extraction         // 图像特征提取示例
    └─ 04_optical_flow              // 光流处理示例
    └─ 05_avm                       // 环视图像拼接示例
    └─ build_aarch64.sh             // aarch64示例编译脚本
    └─ build.sh                     // 编译vp示例最小可执行环境脚本
    └─ build_x86.sh                 // x86仿真示例编译脚本
    └─ CMakeLists.txt               // cmake文件
  └─ vp_samples                      // vp示例最小可执行环境
    └─ data                         // 图像数据目录
    └─ script                       // aarch64示例脚本目录
      └─ 01_basic_processing         // 基础图像处理示例脚本目录
      └─ 02_transformation           // 图像变换示例脚本目录
      └─ 03_feature_extraction       // 图像特征提取示例脚本目录
      └─ 04_optical_flow            // 光流处理示例脚本目录
      └─ 05_avm                    // 环视图像拼接示例脚本目录
      └─ dsp_deploy.sh              // 部署板上的dsp运行环境脚本
      └─ README.md                  // readme文件
    └─ script_x86                   // x86仿真示例脚本目录
      └─ 01_basic_processing         // 基础图像处理示例脚本目录
      └─ 02_transformation           // 图像变换示例脚本目录
      └─ 03_feature_extraction       // 图像特征提取示例脚本目录
      └─ 04_optical_flow            // 光流处理示例脚本目录
      └─ 05_avm                    // 环视图像拼接示例脚本目录
      └─ README.md                  // readme文件
```

## OE包示例后端使用:

| 示例名                                | 说明       | DSP | GDC | STITCH | JPU |
|------------------------------------|----------|-----|-----|--------|-----|
| <a href="#">basic process</a>      | 基础图像处理示例 | Y   | N   | N      | N   |
| <a href="#">transformation</a>     | 图像转换示例   | Y   | N   | N      | N   |
| <a href="#">feature extraction</a> | 特征提取示例   | Y   | N   | N      | N   |
| <a href="#">optical flow</a>       | 光流示例     | Y   | N   | N      | Y   |
| <a href="#">avm</a>                | 环视图像拼接示例 | Y   | Y   | Y      | Y   |

注：当前为试用版本内容，更多示例以及算子开发中。

# 模型板端部署详解-UCP

神经网络模型推理

视觉处理

高性能计算库

UCP工具

# 高性能计算库

HPL模块在以NN为中心的计算方案中，主要作用于模型的前后处理环节。在地平线统一架构中，多种硬件均已搭载了常用的高性能的算子，而HPL模块将高性能算子相关的硬件调用进行了封装，通过设置backend来选择不同的硬件方案（若不指定backend，UCP会自动适配负载更低的处理单元），从而平衡开发板负载，充分对开发板算力进行挖掘，规避了不同硬件调用区别带来的不便

1D-FFT

1D-IFFT

2D-FFT

2D-IFFT

FIR Filter

.....

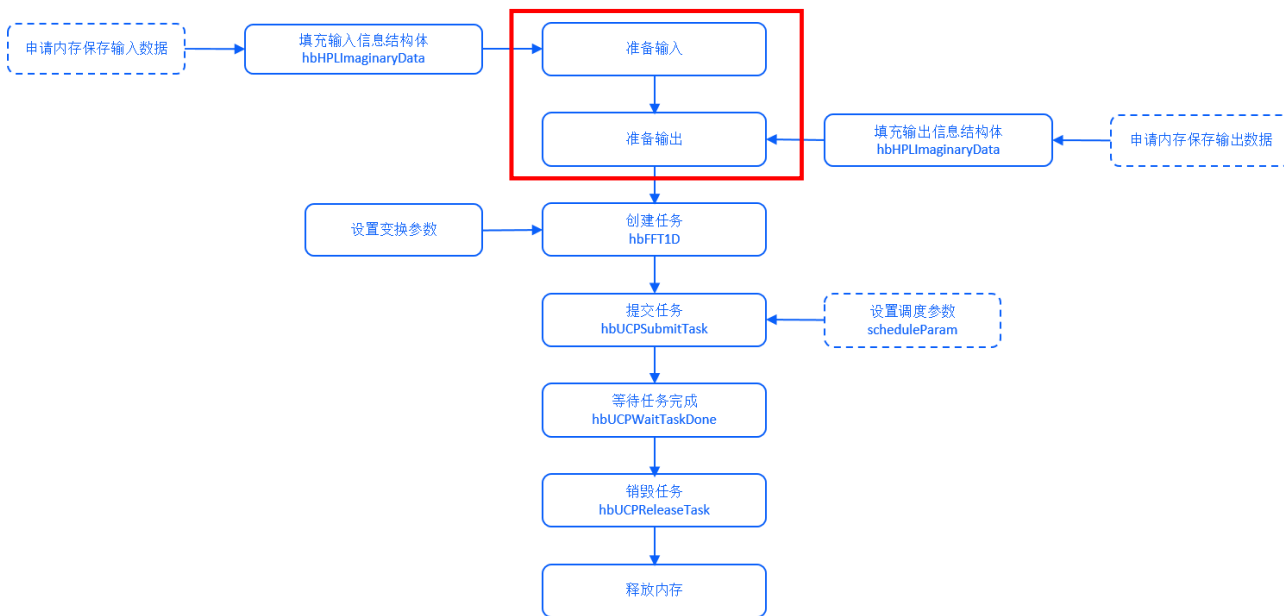
注：HPL算子相关性能指标请参考用户手册

# 高性能计算库-调用基本流程



# 高性能计算库-示例解读

我们通过一个简单的算子调用展示如何使用HPL封装的算子实现快速傅里叶变换的功能，使用hbFFT1D算子对输入数据进行FFT变换：



```
#include <cstring>

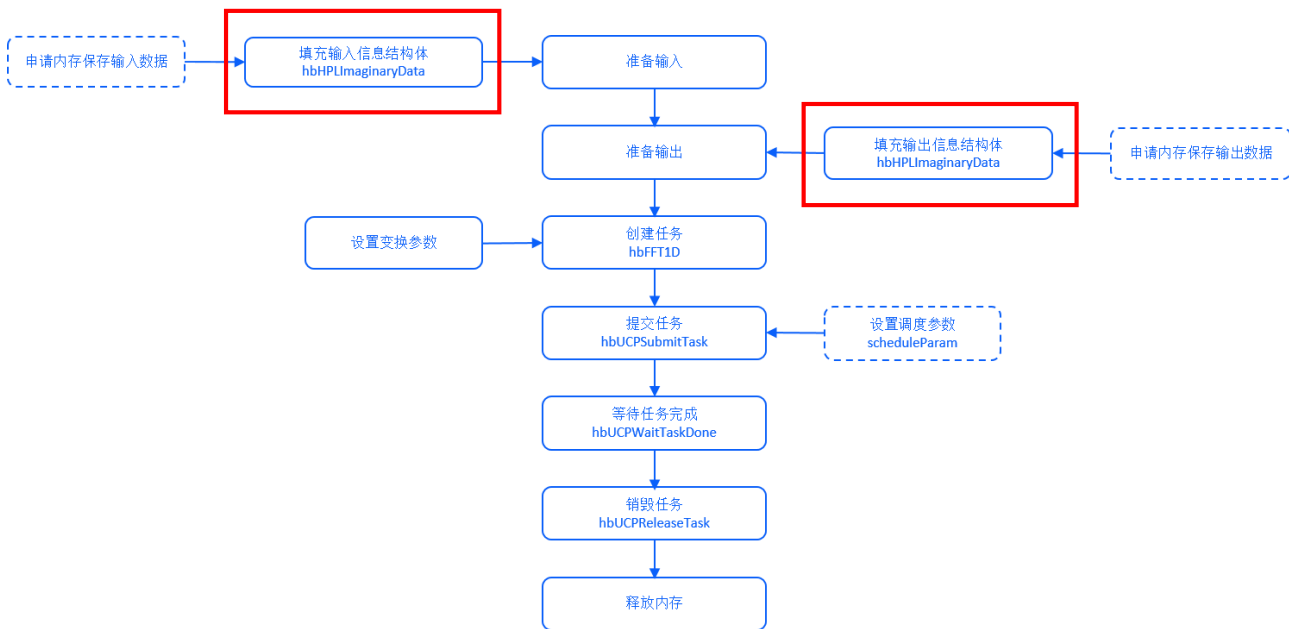
#include "util.h"
#include "log_util.h"
#include "hobot/hpl/hb_fft.h"

// 初始化输入输出内存
hbUCPSysMem src_re_mem, src_im_mem, dst_re_mem, dst_im_mem;
int32_t src_length = 1024 * 4;
hbUCPMallocCached(&src_re_mem, src_length, 0);
hbUCPMallocCached(&src_im_mem, src_length, 0);
hbUCPMallocCached(&dst_re_mem, src_length, 0);
hbUCPMallocCached(&dst_im_mem, src_length, 0);

// 将数据填充到输入内存中
std::string src_img = "./fft_input.txt";
std::ifstream ifs(src_path, std::ios::in | std::ios::binary);
ifs.read(static_cast<char*>(src_re_mem.virAddr), src_length);
ifs.read(static_cast<char*>(src_im_mem.virAddr), src_length);
hbUCPMemFlush(&src_re_mem, HB_SYS_MEM_CACHE_CLEAN);
hbUCPMemFlush(&src_im_mem, HB_SYS_MEM_CACHE_CLEAN);
```

# 高性能计算库-示例解读

我们通过一个简单的算子调用展示如何使用HPL封装的算子实现快速傅里叶变换的功能，使用hbFFT1D算子对输入数据进行FFT变换：



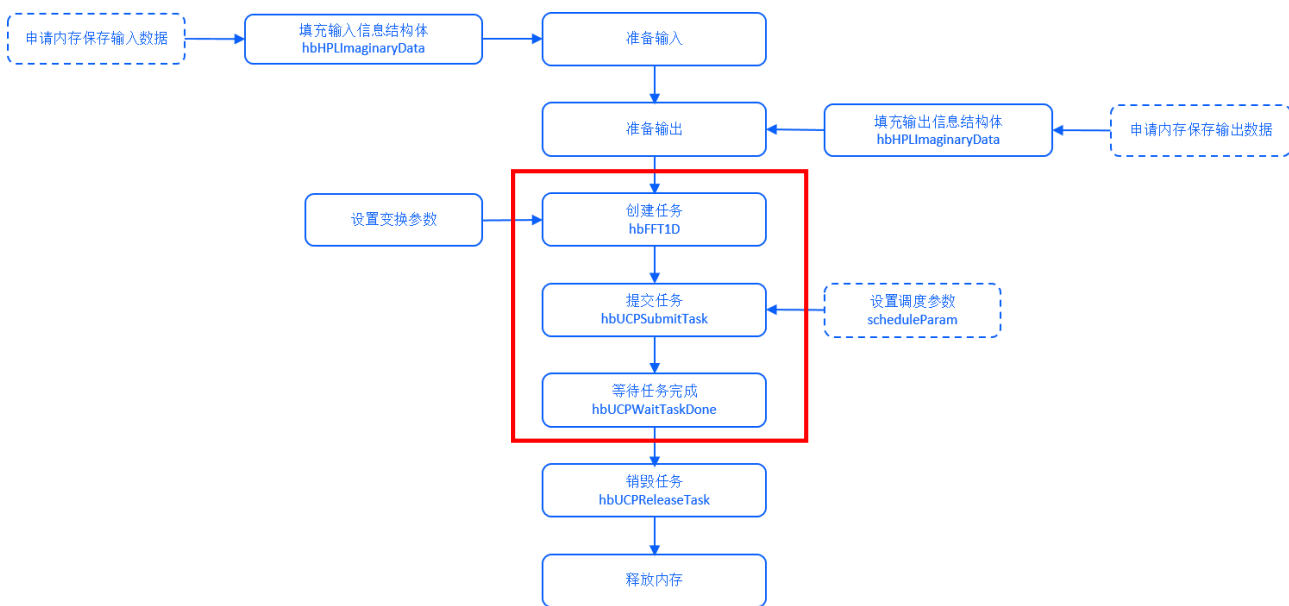
```
// 填充输入输出信息
hbHPLImaginaryData src, dst;
src.realDataVirAddr = src_re_mem.virAddr;
src.realDataPhyAddr = src_re_mem.phyAddr;
src.imDataVirAddr = src_im_mem.virAddr;
src.imDataPhyAddr = src_im_mem.phyAddr;
src.numDimensionSize = 1;
src.dataType = HB_HPL_DATA_TYPE_I16;
src.imFormat = HB_IM_FORMAT_SEPARATE;
src.dimensionSize[0] = src_length / sizeof(int16_t);

dst.realDataVirAddr = dst_re_mem.virAddr;
dst.realDataPhyAddr = dst_re_mem.phyAddr;
dst.imDataVirAddr = dst_im_mem.virAddr;
dst.imDataPhyAddr = dst_im_mem.phyAddr;
dst.numDimensionSize = 1;
dst.dataType = HB_HPL_DATA_TYPE_I16;
dst.imFormat = HB_IM_FORMAT_SEPARATE;
dst.dimensionSize[0] = src_length / sizeof(int16_t);

// 填充算子参数
hbFFTParam param;
param.pSize = HB_HPL_FFT32;
param.normalize = 0;
```

# 高性能计算库-示例解读

我们通过一个简单的算子调用展示如何使用HPL封装的算子实现快速傅里叶变换的功能，使用hbFFT1D算子对输入数据进行FFT变换：



```
// 通过HPL提供的算子接口创建任务，任务句柄支持设置为nullptr，使用异步模式执行此任务
hbUCPTaskHandle_t task_handle{nullptr};

// 设置调度参数调整任务优先级、选择执行终端等参数
hbUCPSchedParam sched_param;
HB_UCP_INITIALIZE_SCHED_PARAM(&sched_param);
sched_param.backend = HB_UCP_DSP_CORE_0; // 指定执行设备ID
sched_param.priority = 0;                // 指定任务优先级

hbFFT1D(&task_handle, // UCP任务句柄
        &dst,          // 输出数据
        &src,          // 输入数据
        &param         // 算子参数
);

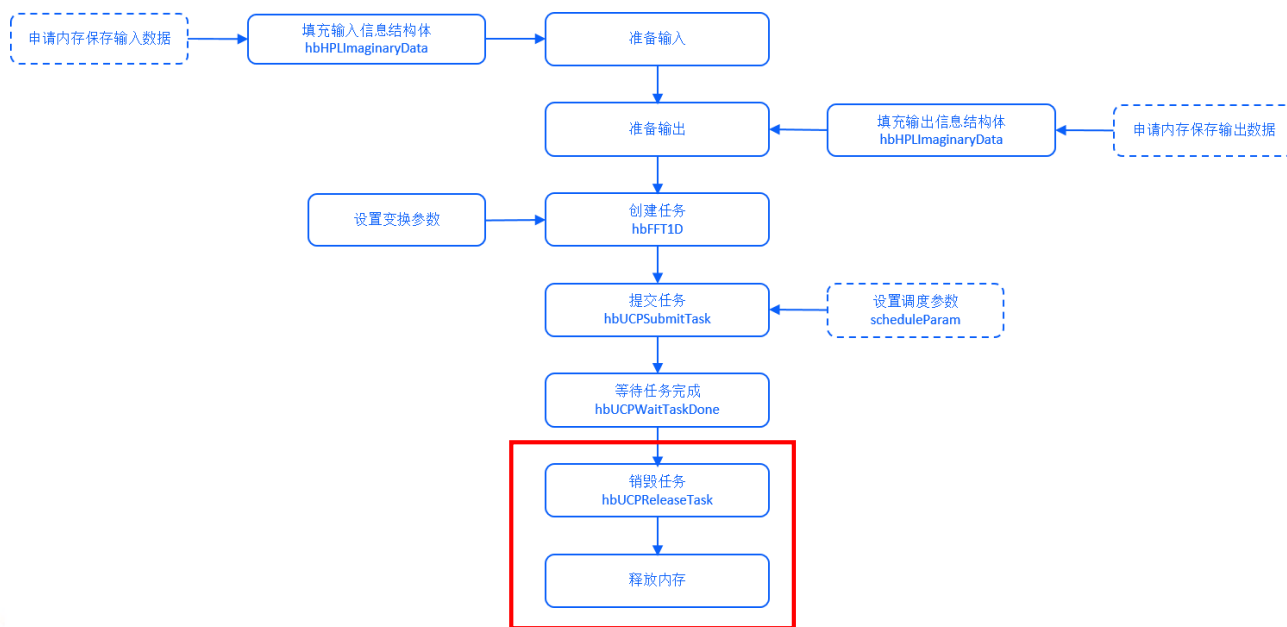
// 设置调度参数调整任务优先级、选择执行终端等参数
hbScheParam sched_param;
HB_UCP_INITIALIZE_SCHED_PARAM(&sched_param);
sched_param.backend = HB_UCP_DSP_CORE_0; /*指定执行设备ID*/

// 提交任务，sched_param 支持设置为nullptr，使用默认调度参数
hbUCPSubmitTask(task_handle, &sched_param);

// 等待任务完成，设置超时参数，值为0时表示一直等待
hbUCPWaitTaskDone(task_handle, 0);
```

# 高性能计算库-示例解读

我们通过一个简单的算子调用展示如何使用HPL封装的算子实现快速傅里叶变换的功能，使用hbFFT1D算子对输入数据进行FFT变换：



```
// 释放任务句柄
hbUCPReleaseTask(task_handle);

// 释放内存资源
hbUCPFree(&src_re_mem);
hbUCPFree(&src_im_mem);
hbUCPFree(&dst_re_mem);
hbUCPFree(&dst_im_mem);
```

# 高性能计算库

## OE包示例:

```
└─ hpl                                // hpl示例
  └─ code                             // 示例源码目录
    └─ 01_fft_ifft_transform          // fft ifft转换示例
    └─ util                           // 公共代码目录
    └─ build_aarch64.sh               // aarch64示例编译脚本
    └─ build.sh                       // 编译hpl示例最小可执行环境脚本
    └─ build_x86.sh                   // x86仿真示例编译脚本
    └─ CMakeLists.txt                 // cmake文件
  └─ hpl_samples                       // hpl示例最小可执行环境
    └─ data                           // 数据目录
    └─ script                         // aarch64示例脚本目录
      └─ 01_fft_ifft_transform        // 示例脚本目录
      └─ dsp_deploy.sh                // 部署板上的dsp运行环境脚本
    └─ script_x86                     // x86仿真示例脚本目录
      └─ 01_fft_ifft_transform        // 示例脚本目录
      └─ README.md                    // readme文件
```

## OE包示例后端使用:

| 算子名                                               | 说明         | DSP |
|---------------------------------------------------|------------|-----|
| <a href="#">Fast Fourier Transform</a>            | 快速傅里叶变换    | Y   |
| <a href="#">Inverse Fast Fourier Transform</a>    | 快速傅里叶逆变换   | Y   |
| <a href="#">Fast Fourier Transform 2D</a>         | 二维快速傅里叶变换  | Y   |
| <a href="#">Inverse Fast Fourier Transform 2D</a> | 二维快速傅里叶逆变换 | Y   |

注：当前为试用版本内容，更多示例以及算子开发中。

# 模型板端部署详解-UCP

神经网络模型推理

视觉处理

高性能计算库

UCP工具

# UCP工具-hrt\_model\_exec

hrt\_model\_exec工具可以帮助用户快速了解模型的实际上板性能，同时可以帮助用户获取模型信息以及推理模型。工具参数说明可以运行hrt\_model\_exec -h查看

| 编号 | 子命令                     | 说明                   |
|----|-------------------------|----------------------|
| 1  | <code>model_info</code> | 获取模型信息，如：模型的输入输出信息等。 |
| 2  | <code>infer</code>      | 执行模型推理，获取模型推理结果。     |
| 3  | <code>perf</code>       | 执行模型性能分析，获取性能分析结果。   |

# UCP工具-hrt\_model\_exec

运行hrt\_model\_exec工具查看Pyramid和Resizer输入模型信息model\_info时，终端打印信息如下：

```
[model name]: resnet50
[model desc]: TODO 输入源为Pyramid，输入数据格式为nv12，默认y和uv分量拆开
input[0]:
name: input_y
input source: HB_DNN_INPUT_FROM_DDR
valid shape: (1,224,224,1,)
aligned shape: (0,0,0,0,)
aligned byte size: -1
tensor type: HB_DNN_TENSOR_TYPE_U8
tensor layout: HB_DNN_LAYOUT_NONE J6支持更灵活的数据排布，Layout概念取消
quanti type: NONE
stride: (-1,-1,1,1,) -1为占位符，表示该维度为动态维度，需在推理前指定具体值

input[1]:
name: input_uv
input source: HB_DNN_INPUT_FROM_DDR
valid shape: (1,112,112,2,)
aligned shape: (0,0,0,0,)
aligned byte size: -1
tensor type: HB_DNN_TENSOR_TYPE_U8
tensor layout: HB_DNN_LAYOUT_NONE
quanti type: NONE
stride: (-1,-1,2,1,)

output[0]:
name: output
valid shape: (1,1,1,1000,) aligned shape建议根据valid shape和stride来解析:
aligned shape: (1,1,1,1000,) (1, stride[0]/stride[1], stride[1]/stride[2], stride[2]/stride[3])
aligned byte size: 1024
tensor type: HB_DNN_TENSOR_TYPE_S8
tensor layout: HB_DNN_LAYOUT_NONE
quanti type: SCALE
stride: (1000,1000,1000,1,)
scale data: 0.00787402, 反量化节点被删除，保留反量化参数
zero_point data: ,
op_type: HB_DNN_OUTPUT_OPERATOR_TYPE_UNKNOWN
```

```
input[0]: 输入源为Resizer
name: data_y_imageY
input source: HB_DNN_INPUT_FROM_DDR
valid shape: (1,-1,-1,1,) 动态valid
aligned shape: (0,0,0,0,) shape
aligned byte size: -1
tensor type: HB_DNN_TENSOR_TYPE_U8
tensor layout: HB_DNN_LAYOUT_NONE
quanti type: NONE
stride: (-1,-1,1,1,)

input[1]:
name: data_uv_imageUV
input source: HB_DNN_INPUT_FROM_DDR
valid shape: (1,-1,-1,2,)
aligned shape: (0,0,0,0,)
aligned byte size: -1
tensor type: HB_DNN_TENSOR_TYPE_U8
tensor layout: HB_DNN_LAYOUT_NONE
quanti type: NONE
stride: (-1,-1,2,1,)

input[2]:
name: data_roi_imageRoi
input source: HB_DNN_INPUT_FROM_DDR
valid shape: (1,4,)
aligned shape: (1,4,)
aligned byte size: 16
tensor type: HB_DNN_TENSOR_TYPE_S32
tensor layout: HB_DNN_LAYOUT_NONE
quanti type: NONE
stride: (16,4,)
```

# UCP工具-hrt\_model\_exec

执行hrt\_model\_exec perf可以在板端评测模型的动态性能，其中指标Latency是指单流程推理模型所耗费的平均时间，重在表示在资源充足的情况下推理一帧的平均耗时，体现在上板运行是单核单线程统计；FPS是指多流程同时进行模型推理平均一秒推理的帧数，重在表示充分使用资源情况下模型的吞吐，体现在上板运行为单核多线程。耗时统计在代码中的位置如下图：

```
// Load model and prepare input and output tensor
...

// Loop run inference and get latency
{
    int32_t const loop_num{1000};
    start = std::chrono::steady_clock::now();
    for(int32_t i = 0; i < loop_num; i++){
        hbUCPSchedParam sched_param{};
        HB_UCP_INITIALIZE_SCHED_PARAM(&sched_param);
        // create task
        hbDNNInferV2(&task_handle, output_tensor, input_tensor, dnn_handle);
        // submit task
        hbUCPSubmitTask(task_handle, &sched_param);
        // wait task done
        hbUCPWaitTaskDone(task_handle, 0);
        // release task handle
        hbUCPReleaseTask(task_handle);
        task_handle = nullptr;
    }
    end = std::chrono::steady_clock::now();
    latency = (end - start) / loop_num;
}

// release tensor and model
...
```

# UCP工具-hrt\_model\_exec

hrt\_model\_exec工具源码和编译脚本路径如左图，执行编译脚本编译后的生成物如右图

```
samples/ucp_tutorial/tools/hrt_model_exec/  
├─ CMakeLists.txt  
├─ README.md  
├─ build.sh # 工具编译脚本  
├─ build_aarch64.sh # aarch64工具编译脚本  
├─ build_x86.sh # x86工具编译脚本  
├─ include # 头文件  
│   └─ util  
├─ script # aarch64工具运行脚本  
│   └─ run_hrt_model_exec.sh  
├─ script_x86 # x86工具运行脚本  
│   └─ run_hrt_model_exec.sh  
└─ src # 工具源码  
    ├── cnpy  
    ├── main.cpp  
    └─ util
```

```
# 运行aarch64工具编译脚本后  
output_shared_J6_aarch64/  
├─ aarch64  
│   ├── bin  
│   └─ lib  
└─ script  
    └─ run_hrt_model_exec.sh  
  
# 运行x86工具编译脚本后  
output_shared_J6_x86/  
├─ script_x86  
│   └─ run_hrt_model_exec.sh  
└─ x86  
    ├── bin  
    └─ lib
```

# UCP工具-hrt\_ucp\_monitor

hrt\_ucp\_monitor 是一个监控硬件 IP 占用率的工具，支持的 IP 包括BPU, DSP, GDC, STITCH, PYM, ISP, Codec (VPU 和 JPU) , 将OE开发包samples/ucp\_tutorial/tools/monitor路径下的可执行文件拷贝至板端，执行./hrt\_ucp\_monitor -h可查看参数说明

运行hrt\_ucp\_monitor时，如果不指定参数，则使用默认参数运行。默认开启所有硬件IP监控，以交互模式运行， BPU 和 DSP 每秒采样 500 次，硬件 IP 占用率每 1000ms 刷新一次

| Monitor 1.0.0 |          |
|---------------|----------|
| IP            | %LOADING |
| BPU0          | 0.0      |
| DSP0          | 0.0      |
| GDC0          | 0.0      |
| ISP0          | 0.0      |
| ISP1          | 0.0      |
| JPU0          | 0.0      |
| PYM0          | 0.0      |
| PYM1          | 0.0      |
| PYM2          | 0.0      |
| STITCH0       | 0.0      |
| VPU0          | 0.0      |

# UCP工具-trace

UCP trace 通过在 UCP 执行的关键路径上嵌入 trace 记录（任务记录和算子记录），提供深入分析 UCP 应用程序调度逻辑的能力。在出现性能异常时，可以通过分析UCP trace，快速找到异常发生的时间点。UCP trace 提供了两种 trace 后端选项：Perfetto Trace 和 Chrome Trace。您可以通过设置环境变量，在这两者之间进行选择，以满足您特定的性能跟踪需求：

- Perfetto Trace可以获取到 UCP 记录的 trace，以及系统状态，ftrace 等信息；
- Chrome Trace只能获取 UCP 记录的 trace，主要用于分析 UCP 本身的调度逻辑。

UCP trace 工具和配置文件位于 samples/ucp\_tutorial/tools/trace 路径下，目录结构为：

```
└─ trace                                # trace工具
    ├── catch_trace.sh                  # chrome trace捕获脚本
    ├── configs                          # 参考配置文件
    │   ├── ucp_in_process.cfg           # in_process模式的perfetto配置文件
    │   ├── ucp_in_process.json          # in_process模式的ucp配置文件
    │   ├── ucp_system.cfg               # system模式的perfetto配置文件
    │   └─ ucp_system.json               # system模式的ucp配置文件
    ├── perfetto                        # 触发trace抓取的工具
    ├── traced                           # trace服务程序
    └─ traced_probes                     # trace数据抓取程序
```

# 性能评测实操

# 实操要求

获取ResNet50模型的ONNX文件，使用`hb_compile --fast-perf`工具快速完成转换编译得到可以板端运行且性能最优的hbm模型，进行静态（PC端）和动态（板端）的性能评测。

# 实操提示

1. 运行启动docker脚本进入开发环境，需指定挂载的本地数据集路径和cpu/gpu docker: `sh run_docker.sh /home/data gpu`;
2. 我们实操选取ResNet50，了解模型快速评测的流程以及对生成物进行解读，模型示例路径为 `samples/ai_toolchain/horizon_model_convert_sample/03_classification/03_resnet50`，首先运行00\_init.sh脚本获取原始onnx模型，默认下载保存在 `samples/ai_toolchain/horizon_model_convert_sample/01_common/model_zoo/mapper/classification/resnet50`路径下，拷贝至 `samples/ai_toolchain/horizon_model_convert_sample/03_classification/03_resnet50` ;
3. 执行命令`hb_compile --fast-perf --model resnet50.onnx --march nash-m`，进行PTQ快速模型转换，获取可部署的性能最优模型（不可用作精度验证），查看model\_output路径下生成物静态性能评测数据。

# 实操提示

4. 将hbm模型拷贝至开发板端，配置PATH链接推理库：

```
#!/bin/sh
```

```
export PATH=/map/env_setup/runtime_env:${PATH}
```

```
export LD_LIBRARY_PATH=/map/env_setup/runtime_env/oe3017:${LD_LIBRARY_PATH}
```

```
export _HB_UCP_PROFILER_LOG_PATH=./
```

5. 执行命令hrt\_model\_exec -h验证推理环境是否已正确配置；

6. 执行命令hrt\_model\_exec model\_info --model\_file resnet50.hbm查看模型信息；

7. 执行板端动态Perf的命令为hrt\_model\_exec perf --model\_file resnet50.hbm --thread\_num 8 --input\_stride "50176,224,1,1;25088,224,2,1" --frame\_count 200评测吞吐率，查看当前路径下生成的BPU和CPU耗时统计信息。

# Thanks