nndeploy

一款开源的模型端到端部署框架

Always

大纲

01 需求分析

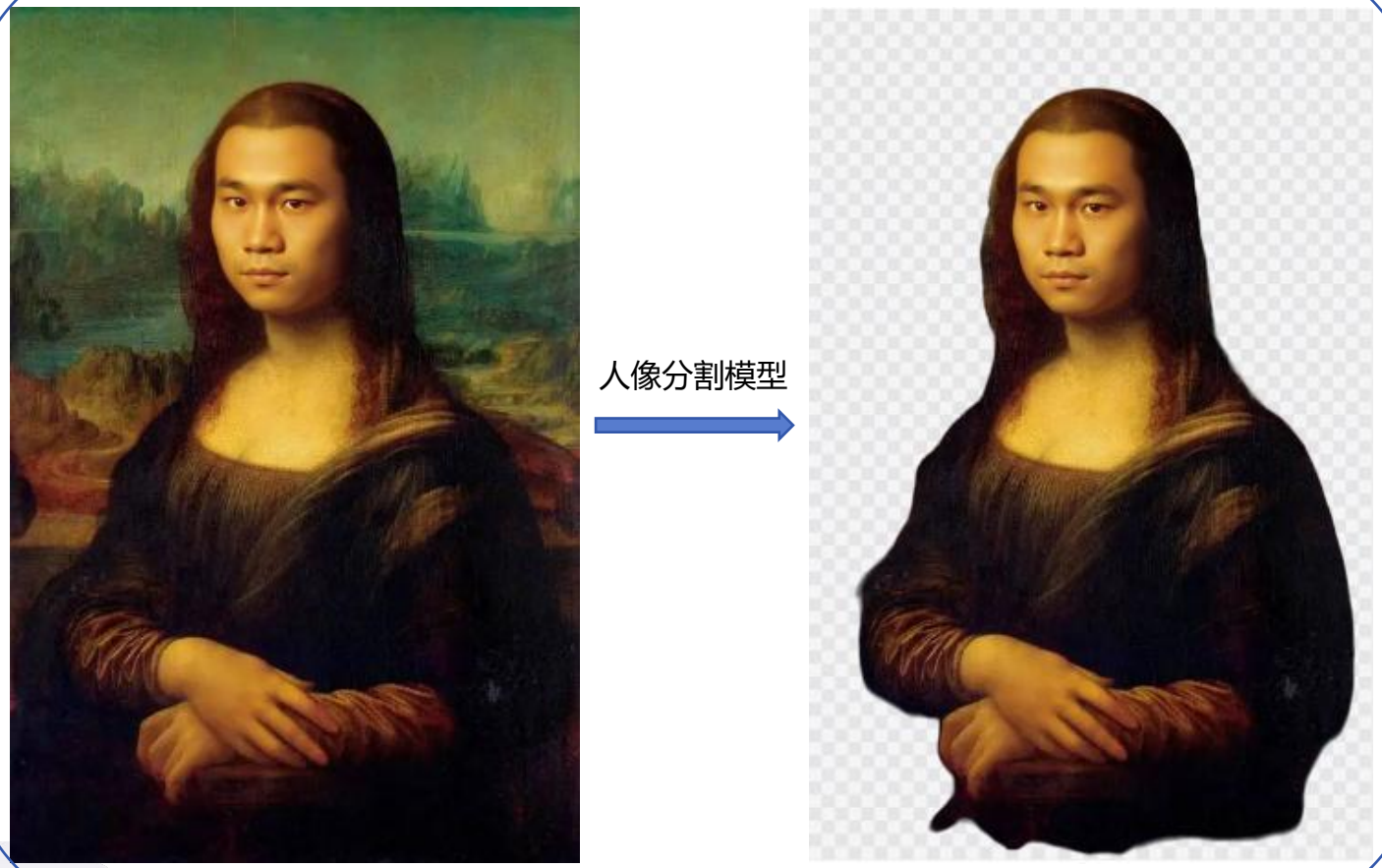
02 概述

03 架构简介

04 下一步规划

需求分析 - 为什么做nndeploy?

多端部署实际案例

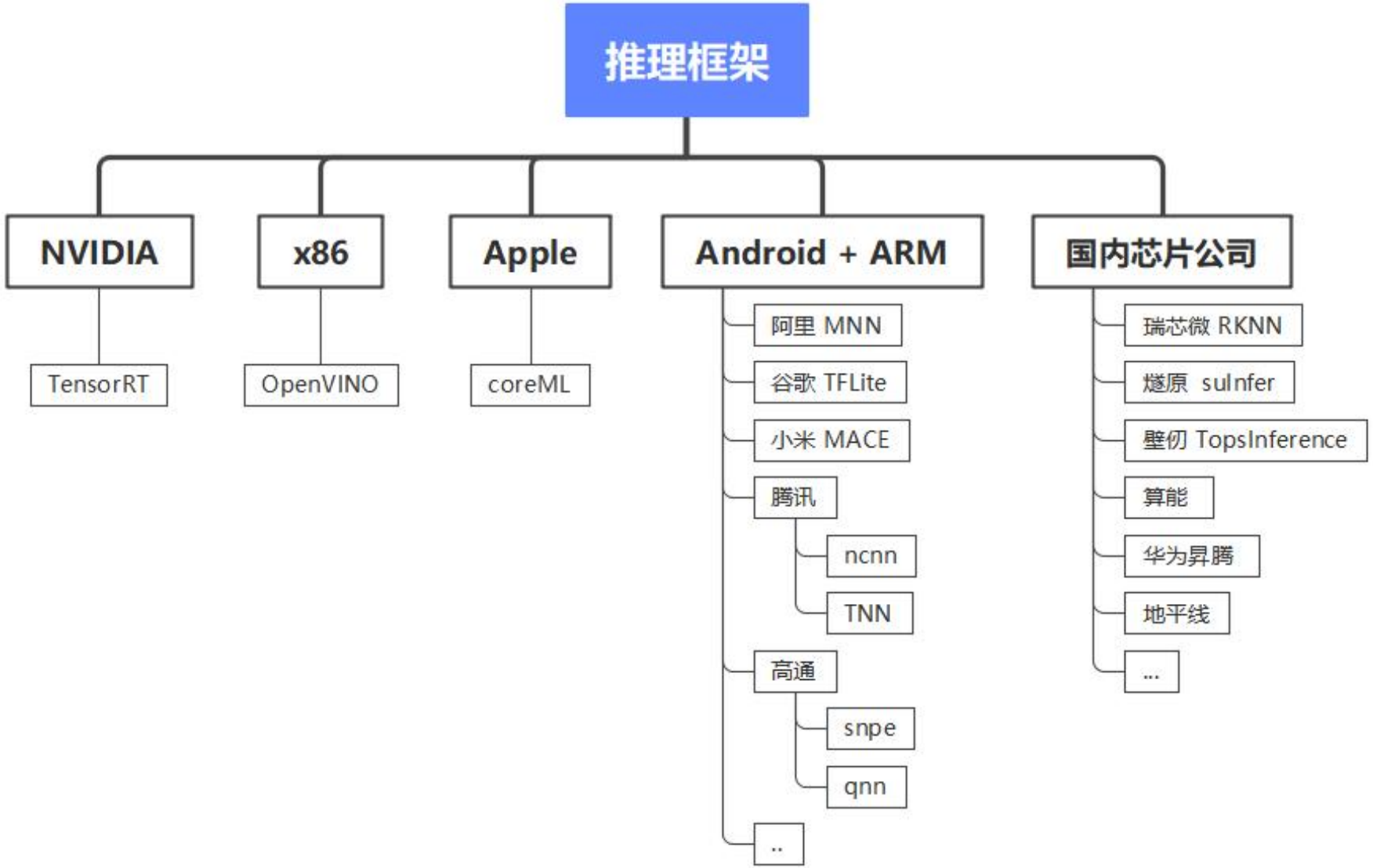


AI智能抠图

麒麟版X86
麒麟版ARM
麒麟版Loongarch
统信UOS版X86
统信UOS版ARM
统信UOS版Loongarch
中科方德版X86
中科方德版ARM
中科方德版Loongarch

跨平台P图软件

痛点一 - 推理框架的碎片化



痛点二 - 多个推理框架 的 学习成本、开发成本、维护成本

	不同的推理接口	不同的超参数配置	不同的Tensor数据结构

TensorRT	`base::UniquePtr<nvinfer1::IBuilder> builder_;`
	`base::UniquePtr<nvinfer1::INetworkDefinition> network_;`
	`base::UniquePtr<nvonnxparser::IParser> parser_;`
	`base::UniquePtr<nvinfer1::IRuntime> runtime_;`
	`std::shared_ptr<nvinfer1::ICudaEngine> engine_;`
	`std::shared_ptr<nvinfer1::IExecutionContext> context_;`
		`int max_batch_size_ = 1;`
		`size_t workspace_size_ = 1 << 30;`
		`bool is_quant_ = false;`
		`std::string int8_calibration_table_path_ = "";`
		`std::string model_save_path_ = "";`
			`std::vector<void *> bindings_;`

OpenVINO	`ov::CompiledModel compiled_model_;`
	`ov::InferRequest infer_request_;`
		`std::vector<base::DeviceType> device_types_;`
		`/// Number of streams while use OpenVINO`
		`int num_streams_ = 1;`
		`/// Affinity mode`
		`std::string affinity_ = "YES";`
		`/// Performance hint mode`
		`std::string hint_ = "UNDEFINED";`
			`ov::Tensor`

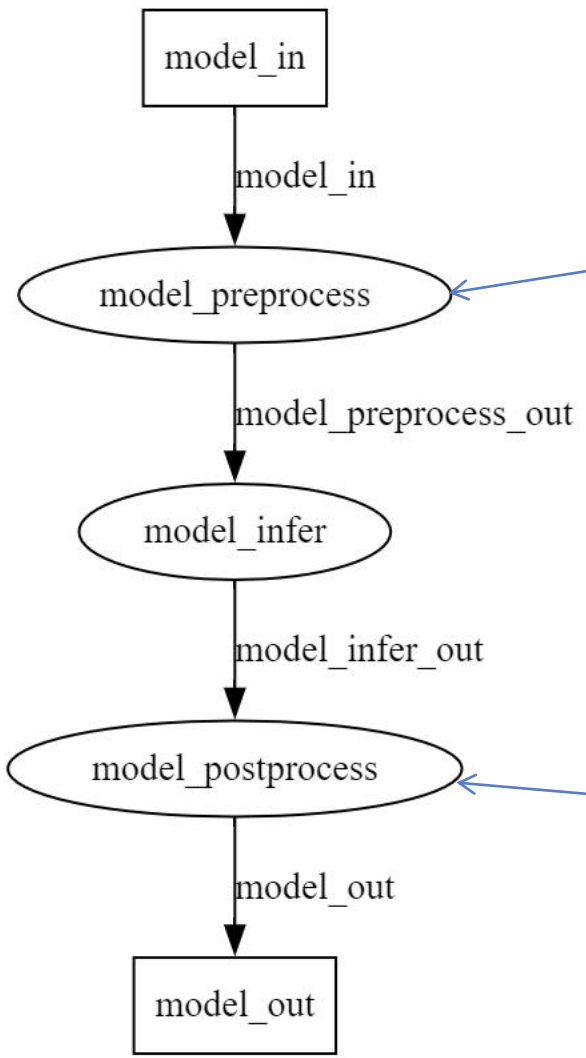
MNN	`MNN::Interpreter *interpreter_ = nullptr;`
	`MNN::Session *session_ = nullptr;`
		`std::vector<std::string> save_tensors_;`
		`MNN::ScheduleConfig::Path path_;`
		`base::DeviceType backup_device_type_;`
		`MNN::BackendConfig::MemoryMode memory_mode_ =`
		`MNN::BackendConfig::MemoryMode::Memory_Normal;`
			`MNN::Tensor`

痛点三 - 模型的多样性

模型特性	描述	TensorRT手动构图	实际算法例子
单输入	模型只有一个输入张量。	确保`NetworkDefinition`中只有一个输入节点。	图像分类模型ResNet，它接收单张图像作为输入，并输出图像的分类结果。
多输入	模型有多个输入张量。	在`NetworkDefinition`中定义多个输入节点，并在推理后处理时获取所有输入。	划痕修复模型，它接收原始图像以及划痕检测mask作为输入
单输出	模型只有一个输出张量。	确保`NetworkDefinition`中只有一个输出节点。	图像检测模型YOLOv5，将后处理融合到模型内部
多输出	模型有多个输出张量。	在`NetworkDefinition`中定义多个输出节点，并在推理后处理时获取所有输出。	图像检测模型YOLOv5，将后处理不融合到模型内部
静态形状输入	输入张量的形状在推理前已知且不变。	在`BuilderConfig`中设置固定的输入形状。	上述模型基本都为静态输入模型
动态形状输入	输入张量的形状在推理时可能变化。	使用`IOptimizationProfile`定义输入张量的动态形状，并在`ExecutionContext`中动态设置输入形状。	自适应的图像超分辨率模型，它能够接收不同尺寸的低分辨率图像作为输入，并输出高分辨率的图像。
静态形状输出	输出张量的形状在推理前已知且不变。	不需要在推理时动态调整。	除动态形状输入模型外，上述模型基本都为静态输出模型
动态形状输出	输出张量的形状在推理时可能变化。	需要在推理后处理时动态获取输出形状，并据此处理输出数据。	机器翻译模型，如Transformer，它接收任意长度的文本作为输入，并输出相应长度的目标语言翻译文本。

结合内存零拷贝优化：直接操作推理框架内部分配输入输出

痛点四 - 模型的前后处理

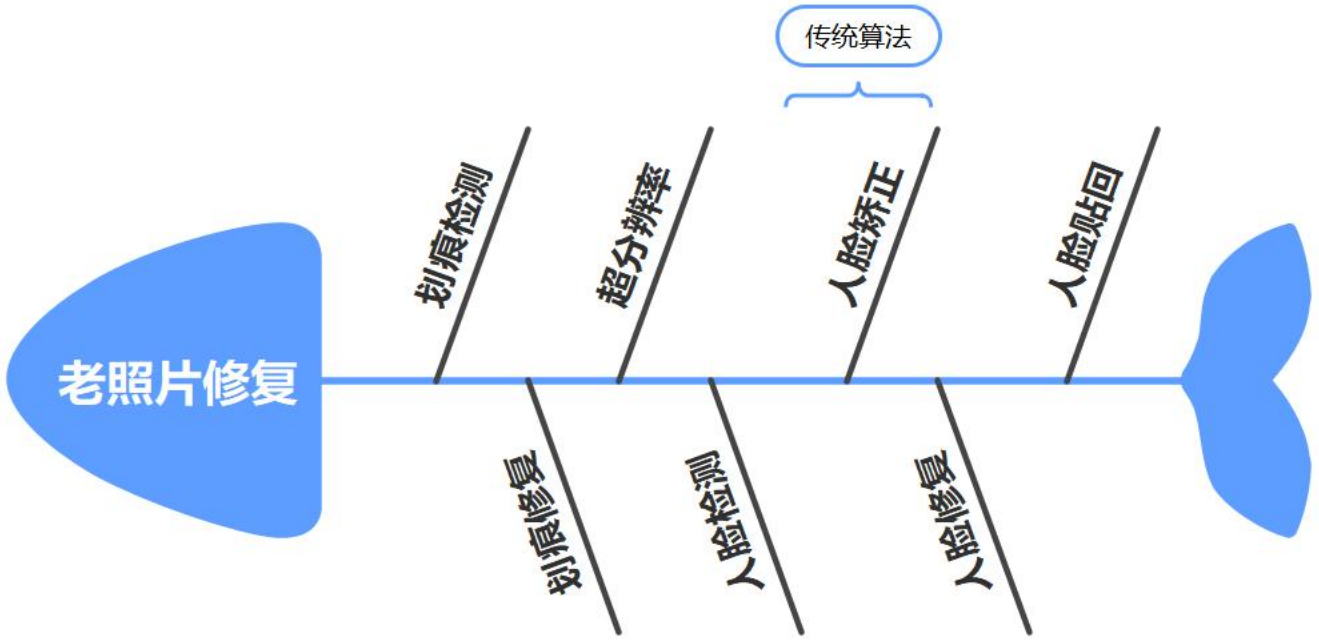


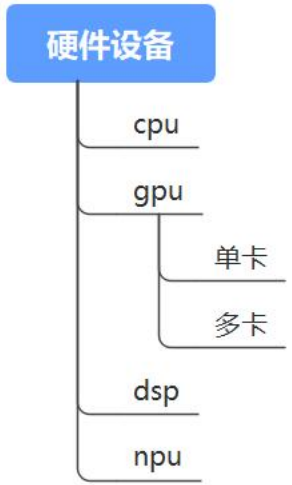

```
/**
* @brief 前处理通常由如下算子组合排列
* cvtcolor
* resize
* padding
* warp_affine
* crop
* nomalize
* transpose      Always, 7个月前 • update: 增加
* dynamic_shape
*/
```

Computer Vision

- Depth Estimation
- Image Classification
- Object Detection
- Image Segmentation
- Text-to-Image
- Image-to-Text
- Image-to-Image
- Image-to-Video
- Unconditional Image Generation
- Video Classification
- Text-to-Video
- Zero-Shot Image Classification
- Mask Generation
- Zero-Shot Object Detection
- Text-to-3D
- Image-to-3D
- Image Feature Extraction

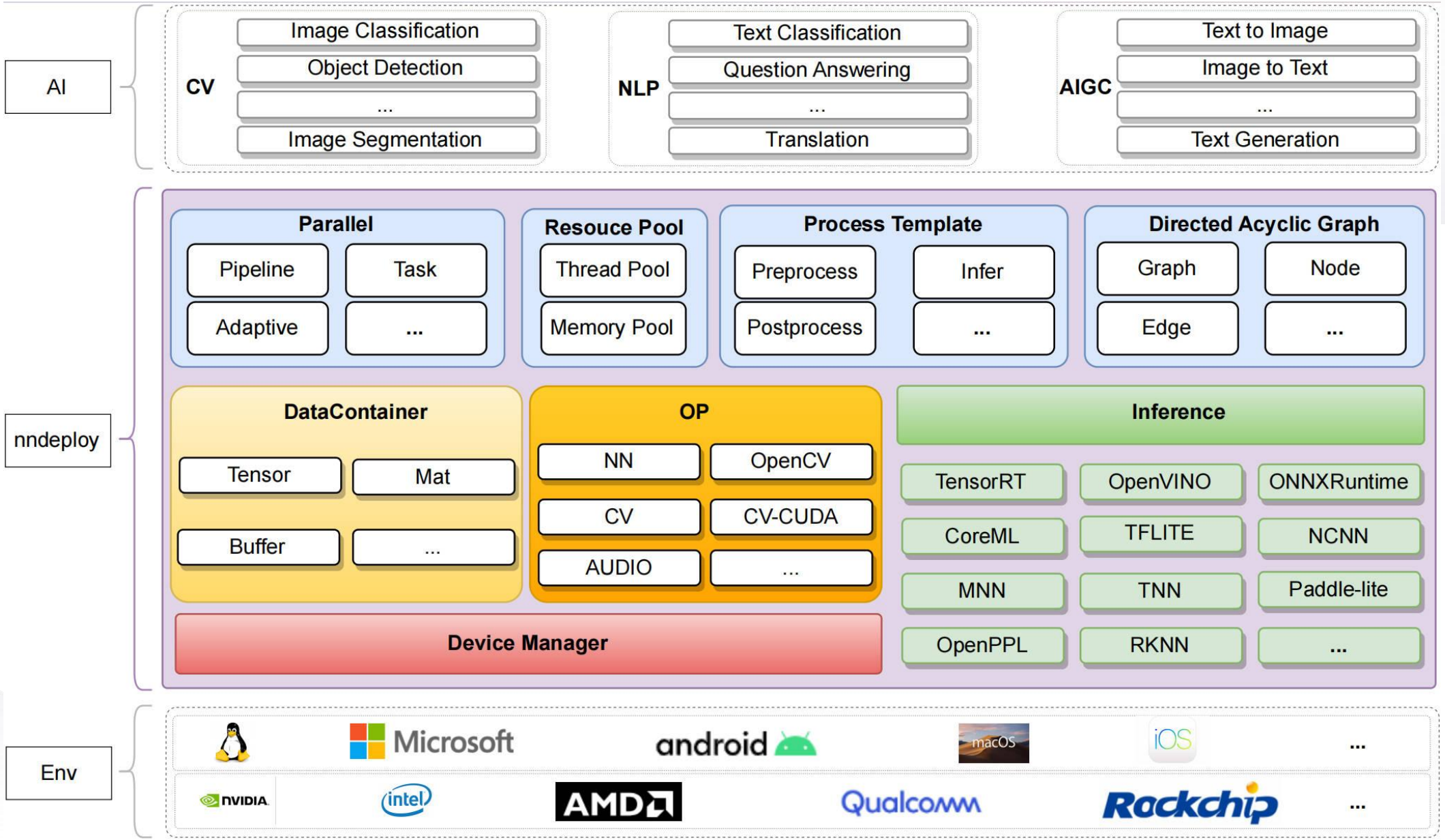
痛点五 - 多模型组合的复杂场景





概述 - nndeploy是什么?

架构


AI
CV
Image Classification
Object Detection
...
Image Segmentation
NLP
Text Classification
Question Answering
...
Translation
AIGC
Text to Image
Image to Text
...
Text Generation
nndeploy
Parallel
Pipeline
Task
Adaptive
...
Resouce Pool
Thread Pool
Memory Pool
Process Template
Preprocess
Infer
Postprocess
...
Directed Acyclic Graph
Graph
Node
Edge
...
DataContainer
Tensor
Mat
Buffer
...
OP
NN
OpenCV
CV
CV-CUDA
AUDIO
...
Inference
TensorRT
OpenVINO
ONNXRuntime
CoreML
TFLITE
NCNN
MNN
TNN
Paddle-lite
OpenPPL
RKNN
...
Device Manager
Env
Microsoft
android
macOS
iOS
...
NVIDIA
intel
AMD
Qualcomm
Rockchip
...

特点一 - 开箱即用的算法

目前已完成 YOLOV5、YOLOV6、YOLOV8 、SAM模型的部署，可供您直接使用，后续我们持续不断去部署其它开源模型，让您开箱即用

model	Inference	developer	remarks
YOLOV5	TensorRt/OpenVINO/ONNXRuntime/MNN	02200059Z、Always	
YOLOV6	TensorRt/OpenVINO/ONNXRuntime	02200059Z、Always	
YOLOV8	TensorRt/OpenVINO/ONNXRuntime/MNN	02200059Z、Always	
SAM	ONNXRuntime	youxiudeshouyeren、Always	

特点二 - 支持跨平台和多推理框架

一套代码多端部署：通过切换推理配置，一套代码即可完成模型跨多个平台以及多个推理框架部署。主要是针对痛点一（推理框架的碎片化）和痛点二（多个推理框架的学习成本、开发成本、维护成本）

当前支持的推理框架如下：

Inference/OS	Linux	Windows	Android	MacOS	IOS	developer	remarks
TensorRT	√	-	-	-	-	Always	
OpenVINO	√	√	-	-	-	Always	
ONNXRuntime	√	√	-	-	-	Always	
MNN	√	√	√	-	-	Always	
TNN	√	√	√	-	-	02200059Z	
ncnn	-	-	√	-	-	Always	
coreML	-	-	-	√	-	JoDio-zd、jaywlinux	
paddle-lite	-	-	-	-	-	qixuxiang	
AscendCL	√	-	-	-	-	CYYAI	
RKNN	√	-	-	-	-	100312dog	

特点三 - 简单易用

- **基于有向无环图部署模型**： 将 AI 算法端到端（前处理->推理->后处理）的部署抽象为有向无环图 `Graph`，前处理为一个 `Node`，推理也为一个 `Node`，后处理也为一个 `Node`。主要是针对痛点四（复用模型的前后处理）
- **推理模板Infer**： 基于`多端推理模块Inference` + `有向无环图节点Node`再设计功能强大的`推理模板Infer`，Infer推理模板可以帮您在内部处理不同的模型带来差异，例如**单输入、多输入、单输出、多输出、静态形状输入、动态形状输入、静态形状输出、动态形状输出**一系列不同。主要是针对痛点三（模型的多样性）
- **高效解决多模型的复杂场景**：在多模型组合共同完成一个任务的复杂场景下（例如老照片修复），每个模型都可以是独立的Graph，nndeploy的有向无环图支持`图中嵌入图`灵活且强大的功能，将大问题拆分为小问题，通过组合的方式快速解决多模型的复杂场景问题
- **快速构建demo**：对于已部署好的模型，需要编写demo展示效果，而demo需要处理多种格式的输入，例如图片输入输出、文件夹中多张图片的输入输出、视频的输入输出等，通过将上述编解码节点化，可以更通用以及更高效的完成demo的编写，达到快速展示效果的目的（目前主要实现了基于OpneCV的编解码节点化）

特点四 - 高性能

- **推理框架的高性能抽象**：每个推理框架也都有其各自的特性，需要足够尊重以及理解这些推理框架，才能在抽象中不丢失推理框架的特性，并做到统一的使用的体验。`nndeploy` 可配置第三方推理框架绝大部分参数，保证了推理性能。可直接操作推理框架内部分配的输入输出，实现前后处理的零拷贝，提升模型部署端到端的性能。
- **线程池**：提高模型部署的并发性能和资源利用率（thread pool）。此外，还支持CPU端算子自动并行，可提升CPU算子执行性能（parallel_for）。
- **内存池**：完成后可实现高效的内存分配与释放(TODO)
- **一组高性能的算子**：完成后将加速您模型前后处理速度(TODO)

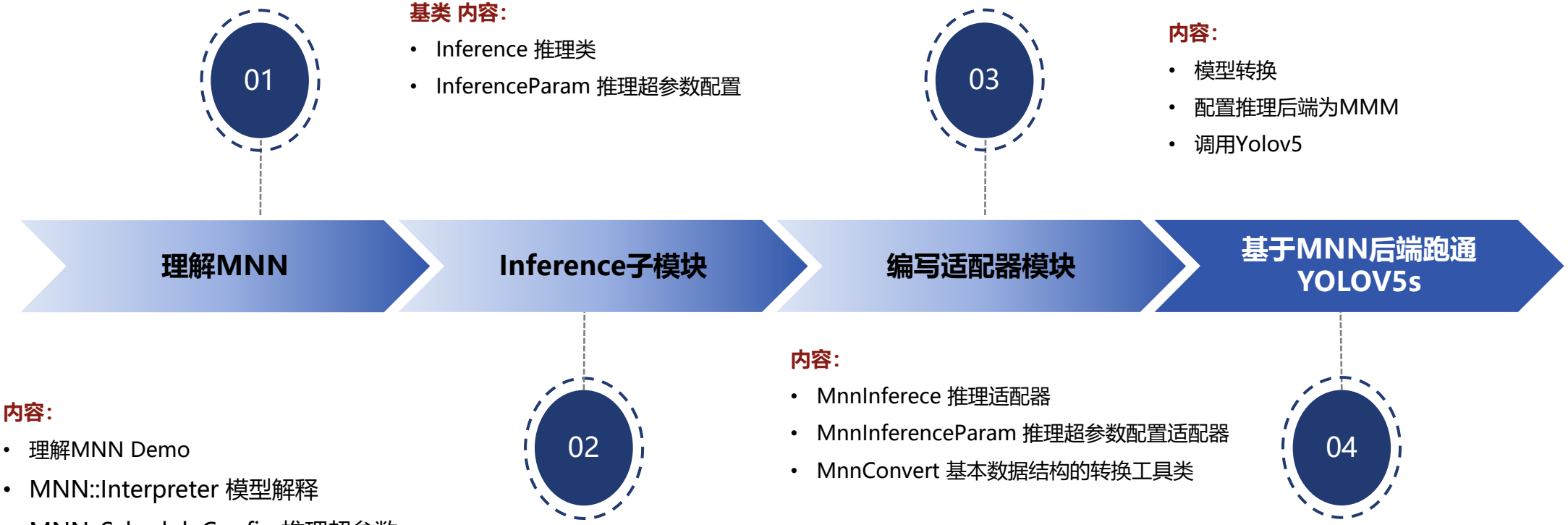
特点五 - 并行

- **串行**：按照模型部署的有向无环图的拓扑排序，依次执行每个节点。
- **流水线并行**：在处理多帧的场景下，基于有向无环图的模型部署方式，可将前处理 `Node`、推理 `Node`、后处理 `Node` 绑定三个不同的线程，每个线程又可绑定不同的硬件设备下，从而三个 `Node` 可流水线并行处理。在多模型以及多硬件设备的的复杂场景下，更加可以发挥流水线并行的优势，从而可显著提高整体吞吐量。
- **任务并行**：在多模型以及多硬件设备的的复杂场景下，基于有向无环图的模型部署方式，可充分挖掘模型部署中的并行性，缩短单次算法全流程运行耗时
- **上述模式的组合并行**：在多模型、多硬件设备以及处理多帧的复杂场景下，nndeploy的有向无环图支持图中嵌入图的功能，每个图都可以有独立的并行模式，故用户可以任意组合模型部署任务的并行模式，具备强大的表达能力且可充分发挥硬件性能。

架构简介 - nndeploy怎么做的?

多端推理子模块

- **Inference：**提供统一的模型推理的方法去操作不同的推理后端
- **接入一个新推理框架整体流程**



多端推理子模块 类定义

h inference.h 9+, M

```
/**
 * @brief 推理的基类
 * @details
 * # 根据InferencParam *param初始化
 * # 写入输入tensor数据
 * # 推理
 * # 得到输出tensor数据
 * # 其他
 * ## 获取输入输出tensor的信息
 * ### 动态输入
 * ### 动态输出
 * ### 是否可以操作推理框架分配的输入
 * ### 是否可以操作推理框架分配的输出
 * ## 获取推理初始化后的各种信息
 * ### 例如内存大小
 * ### gflops等
 *
 */
Always, 4个月前 | 1 author (Always)
class NNDEPLOY_CC_API Inference {···
```

h inference_param.h 9+

```
base::ModelType model_type_;            // 模型的类型
bool is_path_ = true;                   // model_value_是否为路径
std::vector<std::string> model_value_;  // 模型的路径或者内容
base::EncryptType encrypt_type_ =
    base::kEncryptTypeNone;     // 模型文件的加解密类型
std::string license_;           // 模型文件的加解密密钥
base::DeviceType device_type_;  // 模型推理的设备类型
int num_thread_ = 1;            // CPU推理的线程数
int gpu_tune_kernel_ = 1;       // GPU微调的模式
base::ShareMemoryType share_memory_mode_ =
    base::kShareMemoryTypeNoShare;  //  推理时的共享内存模式
base::PrecisionType precision_type_ =
    base::kPrecisionTypeFp32;  // 推理时的精度类型
base::PowerType power_type_ = base::kPowerTypeNormal;  // 推理时的功耗类型
bool is_dynamic_shape_ = false;                        // 是否是动态shape
base::ShapeMap min_shape_ = base::ShapeMap();  // 当为动态输入时最小shape
base::ShapeMap opt_shape_ = base::ShapeMap();  // 当为动态输入时最优shape
base::ShapeMap max_shape_ = base::ShapeMap();  // 当为动态输入时最大shape
std::vector<std::string> cache_path_ = {};  // 缓存路径
std::vector<std::string> library_path_ = {};  // 第三方推理框架的动态库路径

std::vector<std::string> save_tensors_;
MNN::ScheduleConfig::Path path_;
base::DeviceType backup_device_type_;
MNN::BackendConfig::MemoryMode memory_mode_ =
    MNN::BackendConfig::MemoryMode::Memory_Normal;
```

h mnn_convert.h 9+

```
class MnnConvert {
 public:
  static base::DataType convertToDataType(const halide_type_t &src);
  static halide_type_t convertFromDataType(const base::DataType &src);

  static base::DataFormat convertToDataFormat(
      const MNN::Tensor::DimensionType &src);
  static MNN::Tensor::DimensionType convertFromDataFormat(
      const base::DataFormat &src);

  static MNNForwardType convertFromDeviceType(const base::DeviceType &src);

  static MNN::BackendConfig::PowerMode convertFromPowerType(
      const base::PowerType &src);

  static MNN::BackendConfig::PrecisionMode convertFromPrecisionType(
      const base::PrecisionType &src);

  static base::Status convertFromInferenceParam(MnnInferenceParam *src,
                                                MNN::ScheduleConfig *dst);

  static device::Tensor *convertToTensor(MNN::Tensor *src, std::string name,
                                         device::Device *device);
  static MNN::Tensor *convertFromTensor(device::Tensor *src);
};
```

数据容器 Tensor && Buffer

h tensor.h 9+ ×

Always, 6个月前 | 1 author (Always)

```cpp
class NNDEPLOY_CC_API Tensor : public base::NonCopyable {

    std::string name_ = "";              // tensor name
    TensorDesc desc_;                    // tensor desc
    bool is_external_buffer_ = false;    // 是否是外部buffer
    Buffer *buffer_ = nullptr;           // buffer
};
```

```cpp
struct NNDEPLOY_CC_API TensorDesc {        Always, 8个月前 • update: 完成base、d
  TensorDesc(){};
  explicit TensorDesc(base::DataType data_type, base::DataFormat format,
                      const base::IntVector &shape,
                      const base::SizeVector &stride)···

  TensorDesc(const TensorDesc &desc) {···
  TensorDesc &operator=(const TensorDesc &desc) = default;

  virtual ~TensorDesc(){};

  bool operator==(const TensorDesc &other) {···
  bool operator!=(const TensorDesc &other) { return !(*this == other); }

  base::DataType data_type_ = base::dataTypeOf<float>();        // 数据类型
  base::DataFormat data_format_ = base::kDataFormatNotSupport;  // 数据格式
  base::IntVector shape_;                                       // 数据形状
  base::SizeVector stride_;                                     // 数据步长
};
```

h buffer.h 9+, M ×

Always, 4个月前 | 1 author (Always)

```cpp
class NNDEPLOY_CC_API Buffer : public base::NonCopyable {
    Device *device_ = nullptr;           // 内存对应的具体设备
    BufferPool *buffer_pool_ = nullptr;  // 内存来自内存池使用
    BufferDesc desc_;                    // BufferDesc
    void *data_ptr_ = nullptr;           // 设备数据可以用指针表示
    int data_id_ = -1;  // 设备数据需要用id表示，例如OpenGL设备
    // 内存类型，例如外部传入、内部分配、内存映射
    BufferSourceType buffer_source_type_ = kBufferSourceTypeNone;
    int ref_count_ = 0;  // buffer引用计数
};
```

You, 49秒钟前 | 3 authors (Always and others)

```cpp
struct NNDEPLOY_CC_API BufferDesc {
    /**
     * @brief
     * 1d size
     * 2d h w c - 例如OpenCL cl::Image2d        You
     * 3d unknown
     */
    base::SizeVector size_;
    /**
     * @brief
     * 根据不同的设备以及内存形态有不同的config_
     */
    base::IntVector config_;
};
```

设备管理

介绍

设备是nndeploy对硬件设备的抽象，通过对硬件设备的抽象，从而屏蔽不同硬件设备编程模型带来的差异性，nndeploy当前已经支持CPU、X86、ARM、CUDA、AscendCL等设备。主要功能如下

- **统一的内存分配**：为不同设备提供统一的内存分配接口，从而可简化数据容器`Buffer`、`Mat`、`Tensor`的内存分配
- **统一的内存拷贝**：为不同设备提供统一的内存拷贝接口（设备间拷贝、主从设备间上传/下载），从而可简化数据容器`Buffer`、`Mat`、`Tensor`的内存拷贝
- **统一的同步操作**：为不同设备提供统一的同步操作接口，可简化设备端模型推理、算子等同步操作
- **统一的硬件设备信息查询**：为不同设备提供统一的硬件设备信息查询接口，帮助用户更好的选择模型全流程部署的运行设备

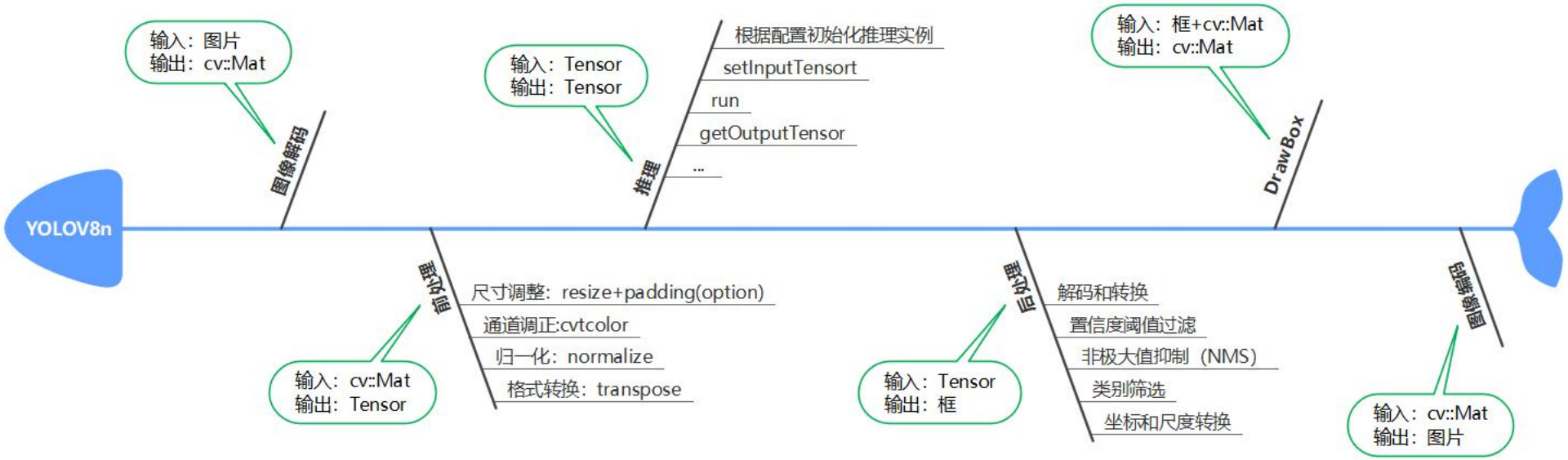
h device.h 9+, M ×

```cpp
class NNDEPLOY_CC_API Device : public base::NonCopyable {
  friend class Architecture;

  virtual Buffer *allocate(size_t size) = 0;
  virtual Buffer *allocate(const BufferDesc &desc) = 0;
  virtual void deallocate(Buffer *buffer) = 0;
  virtual base::Status copy(Buffer *src, Buffer *dst) = 0;
  virtual base::Status download(Buffer *src, Buffer *dst) = 0;
  virtual base::Status upload(Buffer *src, Buffer *dst) = 0;
  // TODO: map/unmap
  // virtual Buffer* map(Buffer* src);
  // virtual base::Status unmap(Buffer* src, Buffer* dst);
  // TODO: share? opencl / vpu / hvx?
  // virtual Buffer* share(Buffer* src);
  // virtual base::Status unshare(Buffer* src, Buffer* dst);

  virtual base::Status synchronize();

  virtual void *getContext();
  virtual void *getCommandQueue();
};
```

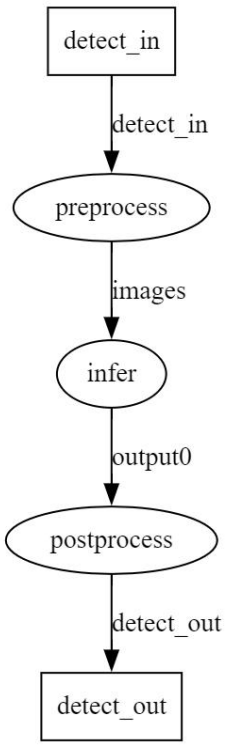
h device.h 9+, M ×

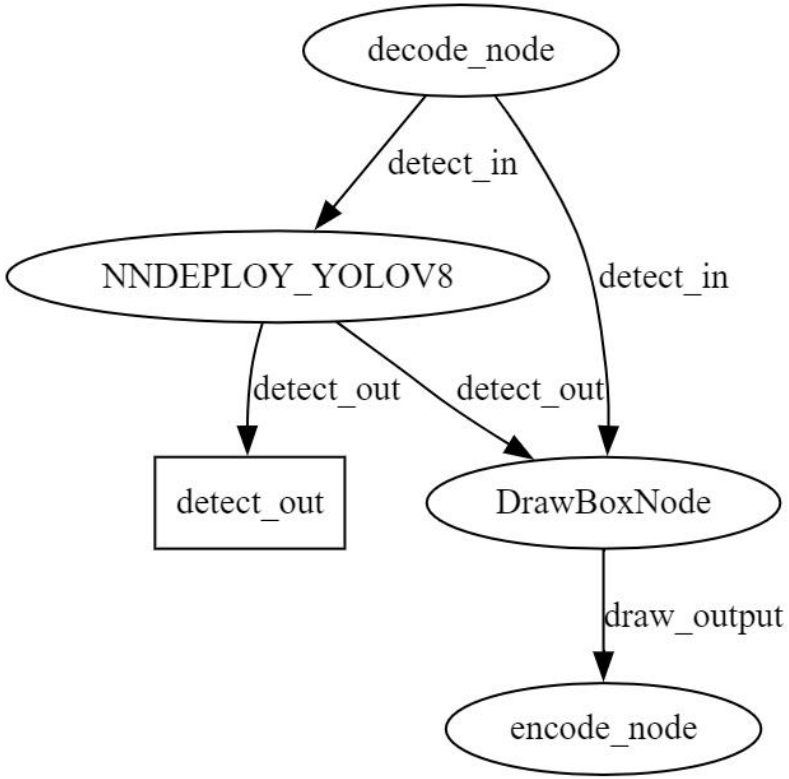
```cpp
/**
 * @brief The Architecture class
 * @note 不可以new，只能通过getArchitecture获取
 *
 */
Always, 4个月前 | 1 author (Always)
class NNDEPLOY_CC_API Architecture : public base::NonCopyable {
 public:
  explicit Architecture(base::DeviceTypeCode device_type_code);

  virtual ~Architecture();

  virtual base::Status checkDevice(int device_id = 0,
                                   void *command_queue = nullptr,
                                   std::string library_path = "") = 0;

  virtual base::Status enableDevice(int device_id = 0,
                                    void *command_queue = nullptr,
                                    std::string library_path = "") = 0;

  virtual Device *getDevice(int device_id) = 0;

  virtual std::vector<DeviceInfo> getDeviceInfo(
      std::string library_path = "") = 0;

  base::DeviceTypeCode getDeviceTypeCode();

 protected:
  std::mutex mutex_;
  /**
   * @brief device_id -> device
   *
   */
  std::map<int, Device *> devices_;      Always, 4个月前 • chore: fix cl

 private:
  base::DeviceTypeCode device_type_code_;
};
```

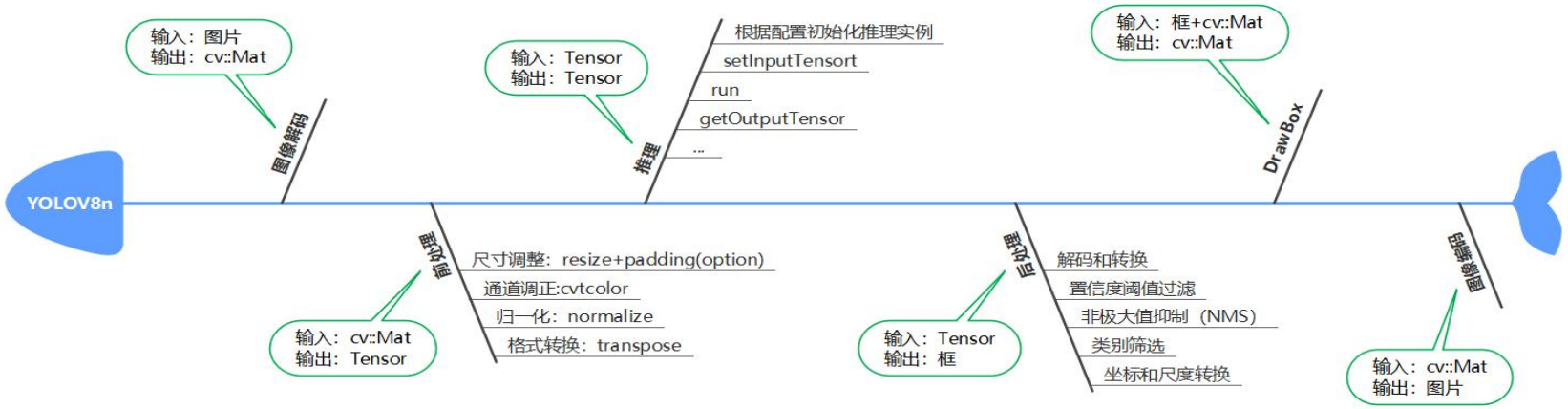
模型部署 - YOLOv8n



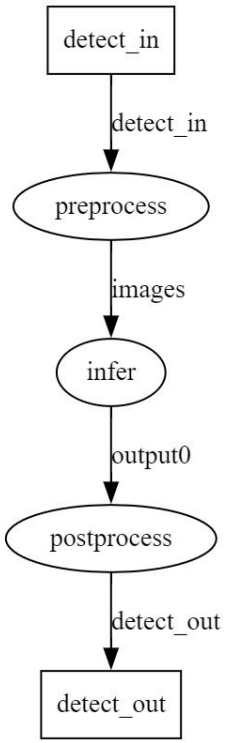
有向无环图 改造 YOLOv8n

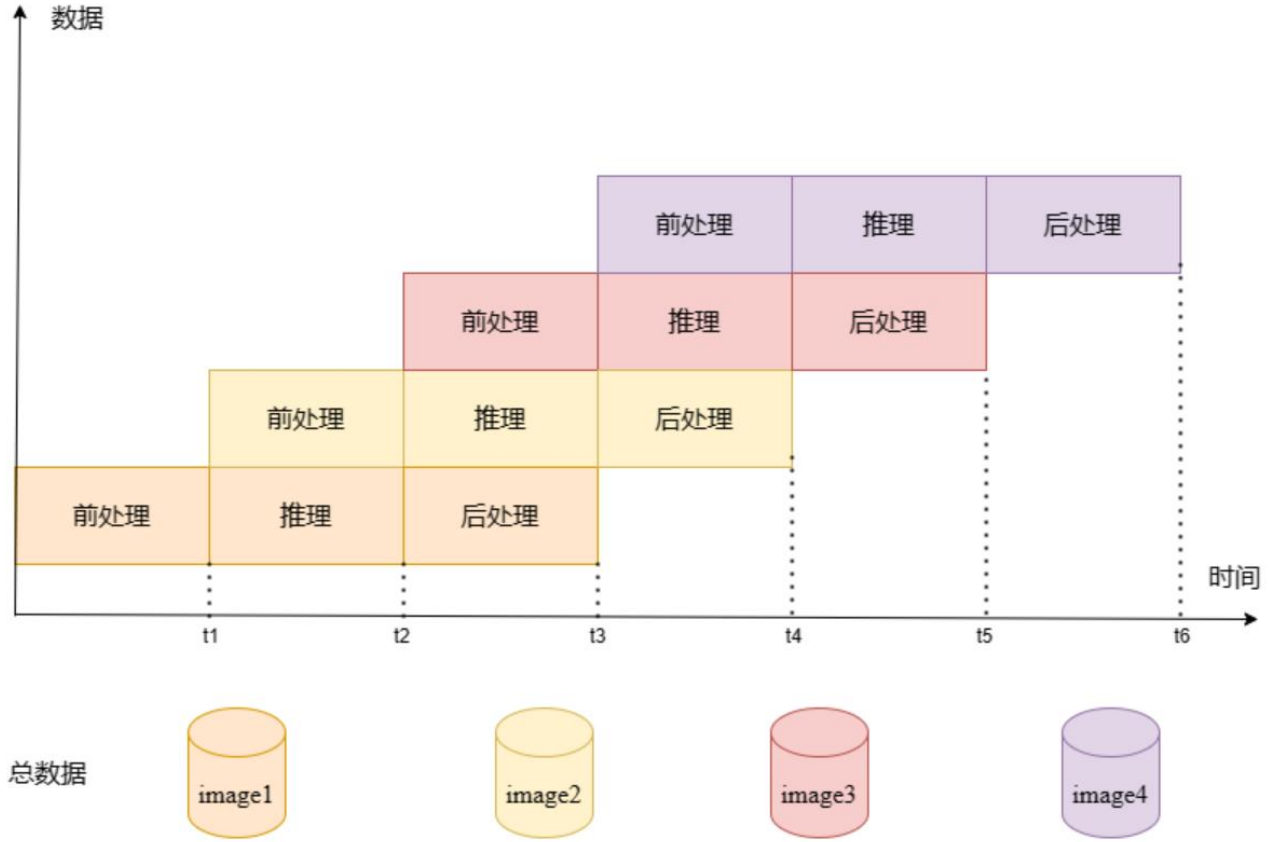






有向无环图 + 流水线并行 优化 YOLOv8n

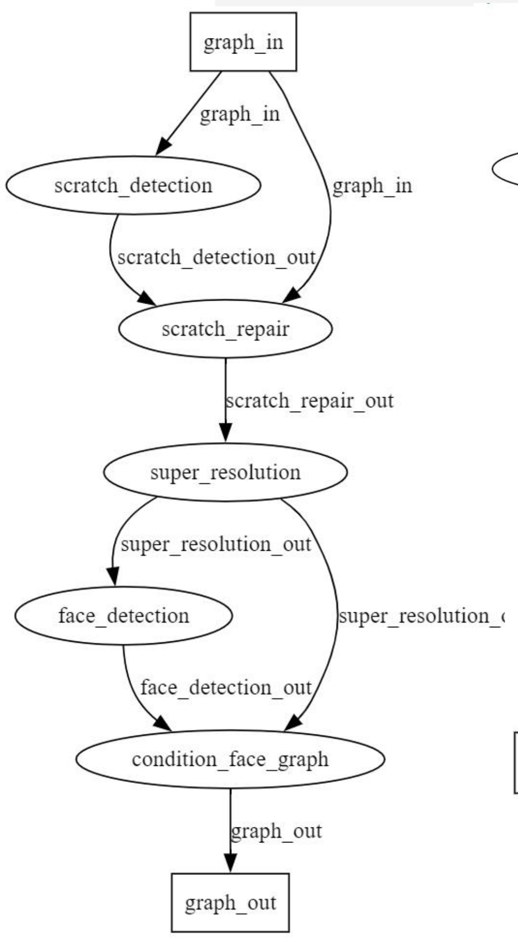


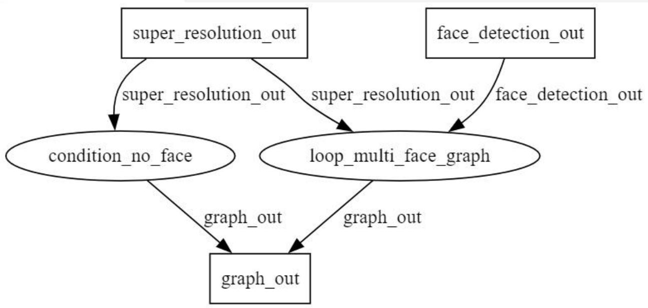


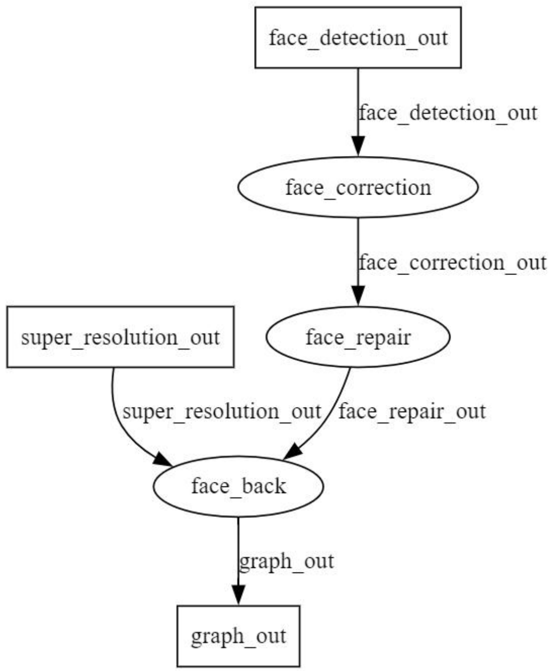
YOLOv8n处理24图片的串行vs流水线并行性能对比结果如下表所展示

模型	推理引擎	运行方式	耗时（ms）
YOLOv8n	onnxruntime	串行	936.359
YOLOv8n	onnxruntime	流水线并行	796.763
YOLOv8n	TensorRT	串行	327.416
YOLOv8n	TensorRT	流水线并行	120.463

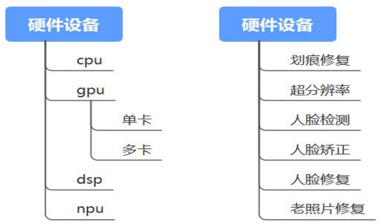
模型部署 - 老照片修复

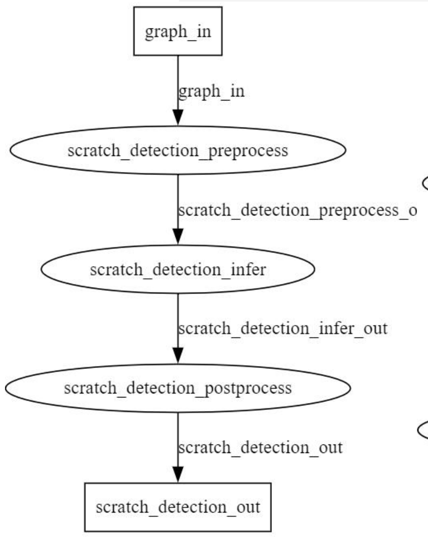

graph_in
graph_in
graph_in
scratch_detection
scratch_detection_out
scratch_repair
scratch_repair_out
super_resolution
super_resolution_out
super_resolution_
face_detection
face_detection_out
condition_face_graph
graph_out
graph_out

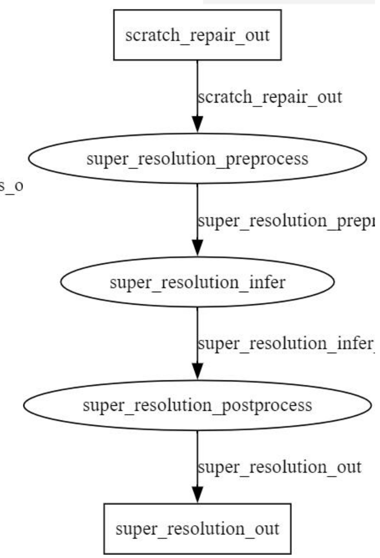

super_resolution_out
face_detection_out
super_resolution_out
super_resolution_out
face_detection_out
condition_no_face
loop_multi_face_graph
graph_out
graph_out
graph_out

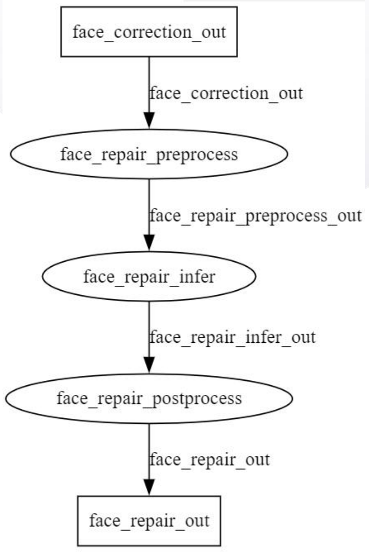

face_detection_out
face_detection_out
face_correction
face_correction_out
super_resolution_out
face_repair
super_resolution_out
face_repair_out
face_back
graph_out
graph_out

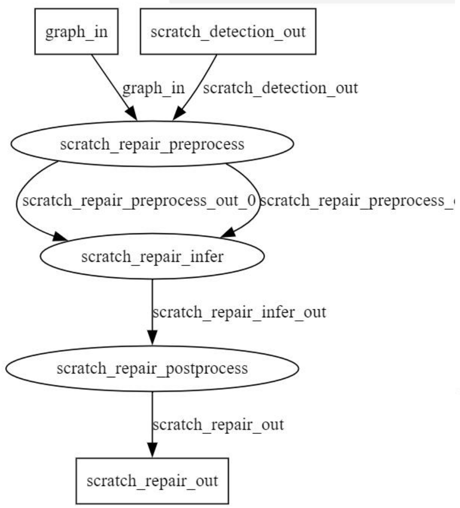

传统算法
老照片修复
划痕检测
超分辨率
人脸矫正
人脸贴回
划痕修复
人脸检测
人脸修复

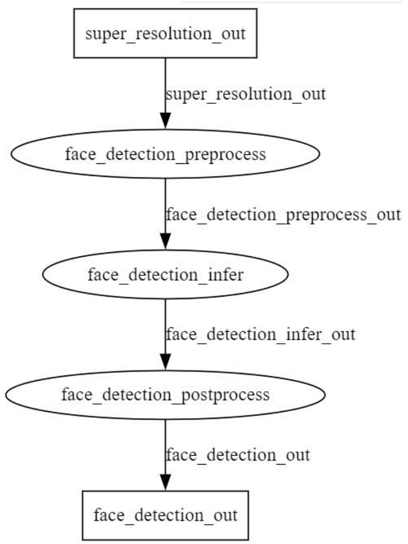

硬件设备
cpu
gpu
单卡
多卡
dsp
npu
硬件设备
划痕修复
超分辨率
人脸检测
人脸矫正
人脸修复
老照片修复

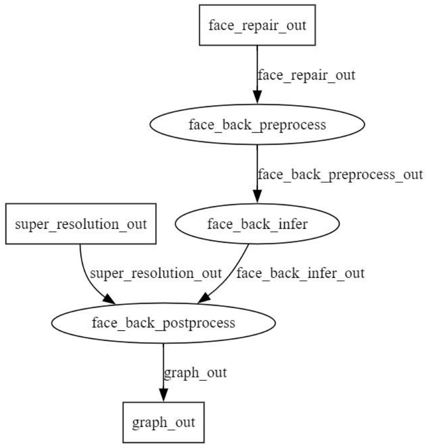

graph_in
graph_in
scratch_detection_preprocess
scratch_detection_preprocess_o
scratch_detection_infer
scratch_detection_infer_out
scratch_detection_postprocess
scratch_detection_out
scratch_detection_out


scratch_repair_out
scratch_repair_out
super_resolution_preprocess
super_resolution_prepr
super_resolution_infer
super_resolution_infer
super_resolution_postprocess
super_resolution_out
super_resolution_out


face_correction_out
face_correction_out
face_repair_preprocess
face_repair_preprocess_out
face_repair_infer
face_repair_infer_out
face_repair_postprocess
face_repair_out
face_repair_out


graph_in
scratch_detection_out
graph_in
scratch_detection_out
scratch_repair_preprocess
scratch_repair_preprocess_out_0
scratch_repair_preprocess_
scratch_repair_infer
scratch_repair_infer_out
scratch_repair_postprocess
scratch_repair_out
scratch_repair_out


super_resolution_out
super_resolution_out
face_detection_preprocess
face_detection_preprocess_out
face_detection_infer
face_detection_infer_out
face_detection_postprocess
face_detection_out
face_detection_out


face_repair_out
face_repair_out
face_back_preprocess
face_back_preprocess_out
super_resolution_out
face_back_infer
super_resolution_out
face_back_infer_out
face_back_postprocess
graph_out
graph_out

下一步规划

下一步规划

- 推理后端
  - 完善已接入的推理框架coreml
  - 完善已接入的推理框架paddle-lite
  - 接入新的推理框架TFLite
- 设备管理模块
  - 新增OpenCL的设备管理模块
  - 新增ROCM的设备管理模块
  - 新增OpenGL的设备管理模块
- 内存优化
  - 主从内存拷贝优化：针对统一内存的架构，通过主从内存映射、主从内存地址共享等方式替代主从内存拷贝
  - 内存池：针对nndeploy的内部的数据容器Buffer、Mat、Tensor，建立异构设备的内存池，实现高性能的内存分配与释放
  - 多节点共享内存机制：针对多模型串联场景下，基于模型部署的有向无环图，在串行执行的模式下，支持多推理节点共享内存机制
  - 边的环形队列内存复用机制：基于模型部署的有向无环图，在流水线并行执行的模式下，支持边的环形队列共享内存机制
- stable diffusion model
  - 部署stable diffusion model
  - 针对stable diffusion model搭建stable_diffusion.cpp（推理子模块，手动构建计算图的方式）
  - 高性能op
  - 分布式
    - 在多模型共同完成一个任务的场景里，将多个模型调度到多个机器上分布式执行
    - 在大模型的场景下，通过切割大模型为多个子模型的方式，将多个子模型调度到多个机器上分布式执行

谢谢！

仓库：https://github.com/DeployAI/nndeploy

文档：https://nndeploy-zh.readthedocs.io/zh/latest/

视频：https://www.bilibili.com/video/BV1VA4m1A7Bk/?spm_id_from=333.337.search-card.all.click&vd_source=c5d7760172919cd367c00bf4e88d6f57