曦云®系列通用计算 GPU

AI 推理用户手册

CSOG-23025-020-F3_V09　　2024-09-18

沐曦专有和三级保密信息　　本文档受 NDA 管控

更新记录

版本	日期	更新说明
V09	2024-09-18	更新以下章节： 2.3 使用 MacaRT 容器镜像
V08	2024-08-05	新增以下章节： 3.5 自定义算子
V07	2024-06-14	新增以下章节： 7 MacaRT-vLLM
V06	2024-05-15	更新以下章节： 6.3.1.2 转换为 PMX 模型 6.3.1.3 模型切分 6.3.1.4 模型合并（可选） 6.3.1.5 PMX 模型测试 6.3.1.6 导出为 ONNX 模型
V05	2024-03-15	新增曦云®系列 GPU 产品信息 更新以下章节： 6.3.2 本地模型部署精度验证 6.3.4.1 服务端部署
V04	2024-02-29	更新以下章节： 6.3.2 本地模型部署精度验证
V03	2024-01-31	更新以下章节： 6.3.1.2 转换为 PMX 模型 6.3.1.5 PMX 模型测试 6.3.1.6 导出为 ONNX 模型
V02	2023-12-29	更新以下章节： 2.2.2 MacaRT Python 3.4.2 MacaRT 显存管理 5.1.1 安装 5.2.1 安装 新增以下章节： 6 MacaRT-LLM
V01	2023-10-16	正式版本首次发布

目录

图目录

表目录

1. 概述

本文档主要用于指导用户如何使用 MacaRT 推理引擎，并快速有效地将训练好的模型部署到曦云®系列 GPU 上。

1.1 MacaRT 介绍

MacaRT（MXMACA® Runtime）是在曦云系列 GPU 上进行人工智能算法模型部署和执行的推理引擎，它是在 ONNX Runtime 上扩展了 Maca 后端。在进行推理时，只需指定 Maca Execution Provider（MacaEP）就可以在曦云系列 GPU 上完成模型推理。MacaRT 的使用方法与 ONNX Runtime 完全兼容。

MacaRT 进行模型推理的整体架构，如图 1-1 所示，可以分为以下部分：

- 解析 ONNX 模型，获取模型图和参数信息
- 对模型图进行处理，包括图优化、图拆分、图编译等
- 通过 MacaEP 在曦云系列 GPU 上执行模型

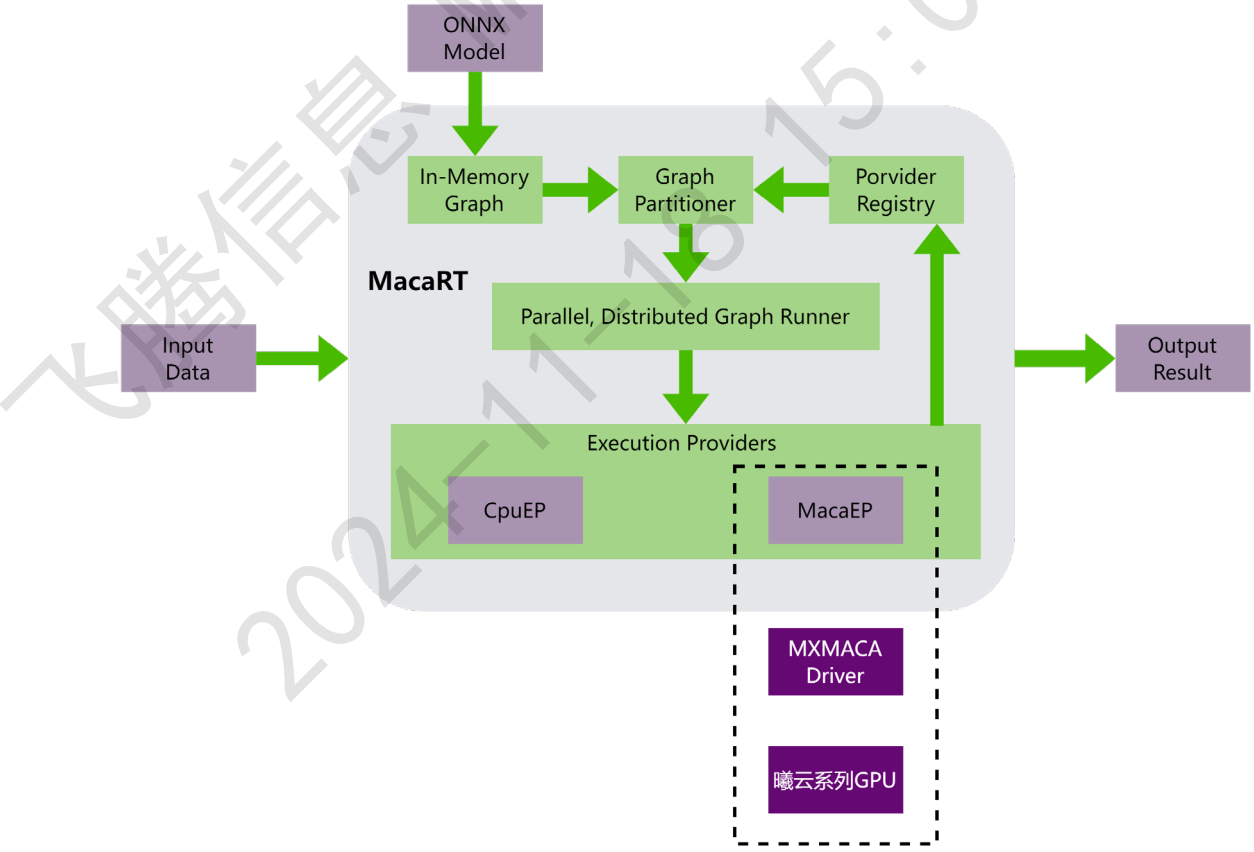

图 1-1 MacaRT 的模型推理整体架构

1.2 MacaRT 功能

MacaRT 包含了以下的功能和特性：

- MacaRT 提供了 C++和 Python 接口
- 支持多种模型数据类型，包括 float32、float16、int8、uint8 等
- 支持动态 Batch 推理
- 支持多线程调用和多进程调用
- 支持模型图优化、模型量化等特性
- MacaRT 工具链，包括 MacaConverter、MacaQuantizer、MacaPrecision

2. 环境依赖及 MacaRT 安装

2.1 环境依赖

在使用 MacaRT 之前，必须确保服务器已安装曦云 GPU 板卡及其驱动程序。

2.2 安装 MacaRT

操作步骤

1. 执行以下命令，快速安装 MacaRT 的 deb 软件安装包。deb 软件安装包可以在相应 GPU 的软件发布包中找到。

```
dpkg -i onnxruntime-maca_*.deb
```

安装结束后，MacaRT 相关文件均会安装到**/opt/maca-ai/onnxruntime-maca** 目录中。

2.2.1 MacaRT C++

操作步骤

1. 执行以下命令，快速验证是否已正确安装 MacaRT C++库。

```
/opt/maca-ai/onnxruntime-maca/bin/maca_test --model_dir /opt/maca-ai/onnxruntime-maca/sample/test_add --check_mode 1
```

对获取到的 test_add 中模型在 CpuEP 和 MacaEP 上执行的结果进行对比，若对比结果相同则代表正确安装了 MacaRT。

2.2.2 MacaRT Python

MacaRT Python 版本的 wheel 安装包可在**/opt/maca-ai/onnxruntime-maca/python** 中发现。在安装 wheel 包之前，须确保当前 Python 版本和 wheel 包上标识的 Python 版本一致。当前为 Python 3.8。

操作步骤

1. 执行以下命令，安装 MacaRT Python 版本 wheel 包：**onnxruntime_gpu-1.12.0+mc*.whl**。

```
pip install /opt/maca-ai/onnxruntime-maca/python/onnxruntime_gpu-1.12.0+mc*.whl
```

2. 执行以下命令，快速验证是否已正确安装 MacaRT。

```
python /opt/maca-ai/onnxruntime-maca/sample/python/maca_test.py -m /opt/maca-ai/onnxruntime-maca/sample/test_add
```

说明

MacaRT Python 主要依赖 numpy、protobuf 等，在使用 pip 进行安装时会自动安装 MacaRT Python 的依赖包。

2.3 使用 MacaRT 容器镜像

从发布的软件包中获取 **onnxruntime** 容器镜像并启动。**onnxruntime** 容器镜像包含 MacaRT 及工具链。容器镜像的使用，参见《曦云®系列通用计算 GPU 用户指南》中“容器相关场景支持”章节。

3. MacaRT C++ API

3.1 部署 ONNX 模型

MacaRT 支持使用 C++ API 将 ONNX 模型部署到曦云系列 GPU 上，并完成推理。

3.1.1 创建 Session

操作步骤

1. 配置日志对象 `Ort::Env`。

```
#include <onnxruntime_cxx_api.h>
std::string ep_name = "MacaEP";
Ort::Env env(OrtLoggingLevel::ORT_LOGGING_LEVEL_INFO, ep_name.c_str());
```

2. 在 `SessionOption` 中添加 MacaEP 的信息。

```
OrtMACAProviderOptions maca_options;
maca_options.device_id=0;
Ort::SessionOptions sessionOptions;
sessionOptions.AppendExecutionProvider_MACA(maca_options);
```

说明

`SessionOption` 用于在创建 Session 时，指定 MacaRT 所使用的 EP 信息。

`SessionOption` 的详细信息，可参考 ONNX Runtime 相关文档。

3. 创建 Session 对象。

```
Ort::Session session(env, your_onnx_model_path, sessionOptions);
```

3.1.2 获取模型图的输入输出信息

操作步骤

1. 执行以下命令，获取模型图的输入输出信息。

```
Ort::AllocatorWithDefaultOptions allocator;
std::vector<const char*> inputNames, outputNames;
for ( size_t i=0; i< session.GetInputCount(); i++){
  inputNames.push_back(session.GetInputName(i , allocator));
}
for ( size_t i=0; i< session.GetOutputCount(); i++){
  outputNames.push_back(session.GetOutputName(i , allocator));
}
```

3.1.3 构建模型输入 Tensors

操作步骤

1. 准备输入数据。假设提供的模型输入数据和数据长度是通过以下变量名称提供，并且输入数据处于可分页内存上。

```
void* input_data[inputNames.size()];   // input data ptr
size_t input_data_len[inputNames.size()]; // input data len
```

2. 创建 `MemoryInfo`，用于标识输入数据所在的设备信息。

```
Ort::MemoryInfo memoryInfo =
    Ort::MemoryInfo::CreateCpu(OrtAllocatorType::OrtArenaAllocator,OrtMem
    Type::OrtMemTypeDefault);
```

3. 创建输入 Tensors。假设模型的输入形状是固定的。

```
std::vector<Ort::Value> inputTensors;
for(size_t i=0; i<inputNames.size(); i++){
// get input node data type
  Ort::TypeInfo inputTypeInfo = session.GetInputTypeInfo(i);
  auto inputTensorInfo = inputTypeInfo.GetTensorTypeAndShapeInfo();
  ONNXTensorElementDataType inputType = inputTensorInfo.GetElementType();
  // get input node data length
  std::vector<int64_t> inputDims=inputTensorInfo.GetShape();
  inputTensors.push_back(Ort::Value::CreateTensor(memoryInfo,input_data[
i],input_data_len[i],inputDims.data(),inputDims.size(),inputType));
}
```

3.1.4 获取模型输出 Tensors

操作步骤

1. 执行以下命令，通过同步执行的方式，获取模型的输出 Tensors。在不指定设备信息的情况下，输出 Tensors 中的数据默认位于可分页内存上。

```
auto ouput = session.Run(Ort::RunOptions{nullptr},inputNames.data(),
                         inputTensors.data(),inputNames.size(),
                         ouputNames.data(), outputNames.size());
mcStreamSynchronize(stream);
```

3.2 绑定输入输出设备

有多个模型且模型间存在数据拷贝时，绑定输入输出内存信息，可帮助减少模型之间不必要的输入输出数据拷贝。MacaRT 支持将模型的输入或输出绑定到：

- 两种 Host 端的内存页面：可分页内存（Pageable Memory）和固页内存（Pinned Memory）
- 一种 Device 端的内存信息：曦云系列 GPU 显存（Video Memory）

推荐使用固页内存存储模型输入输出数据，可以提高数据传输效率。

3.2.1 绑定输入输出数据到可分页内存上

具体操作步骤，参见 3.1 部署 ONNX 模型。

3.2.2 绑定输入输出数据到固页内存上

操作步骤

1. 准备输入数据。假设提供的模型输入数据和数据长度是通过以下变量名称提供，并且输入数据处于固页内存上。

```
void* input_data[inputNames.size()];   // input data ptr
size_t input_data_len[inputNames.size()]; // input data len
```

2. 创建 `MemoryInfo`，用于标识输入数据所在的设备信息。

```
constexpr const char* MACA_PINNED_STR= "MacaPinned";
Ort:MemoryInfo memoryInfo = Ort::MemoryInfo(MACA_PINNED_STR,
OrtAllocatorType::OrtDeviceAllocator,0, OrtMemType::OrtMemTypeCPUOutput);
```

3. 创建输入 Tensors。假设模型的输入形状是固定的。

```
std::vector<Ort::Value> inputTensors;
for(size_t i=0; i<inputNames.size(); i++){
// get input node data type
  Ort::TypeInfo inputTypeInfo = session.GetInputTypeInfo(i);
  auto inputTensorInfo = inputTypeInfo.GetTensorTypeAndShapeInfo();
  ONNXTensorElementDataType inputType = inputTensorInfo.GetElementType();
  // get input node data length
  std::vector<int64_t> inputDims=inputTensorInfo.GetShape();
  inputTensors.push_back(Ort::Value::CreateTensor(memoryInfo,input_data[
i],input_data_len[i],inputDims.data(),inputDims.size(),inputType));
}
```

4. 执行以下命令，使用 IOBinding 来获取模型输出。

```
Ort::IoBinding ioBinding(session);
for(size_t i=0; i< inputNames.size(), i++){
  ioBinding.BindInput(inputNames[i],inputTensors[i]);
}
for(size_t i=0; i< outputNames.size(), i++){
  ioBinding.BindOutput(outputNames[i],memoryInfo);
}
session.Run(Ort::RunOptions{nullptr},ioBinding);
auto outputs=ioBinding.GetOutputValues();
```

3.2.3 绑定输入输出数据到显存上

操作步骤

1. 准备输入数据。假设提供的模型输入数据和数据长度是通过以下变量名称提供，并且输入数据处于曦云系列 GPU 显存上。

```
void* input_batch_data[inputNames.size()];   // input data ptr
size_t input_batch_data_len[inputNames.size()]; // input data len
```

2. 创建 `MemoryInfo`，用于标识输入数据所在的内存信息。

```
constexpr const char* MACA_STR= "Maca";
Ort:MemoryInfo memoryInfo = Ort::MemoryInfo(MACA_STR,
OrtAllocatorType::OrtDeviceAllocator, gpu_id,
OrtMemType::OrtMemTypeDefault);
```

3. 创建输入 Tensors。假设模型的输入形状是固定的。

```
std::vector<Ort::Value> inputTensors;
for(size_t i=0; i<inputNames.size(); i++){
// get input node data type
  Ort::TypeInfo inputTypeInfo = session.GetInputTypeInfo(i);
  auto inputTensorInfo = inputTypeInfo.GetTensorTypeAndShapeInfo();
  ONNXTensorElementDataType inputType = inputTensorInfo.GetElementType();
  // get input node data length
  std::vector<int64_t> inputDims=inputTensorInfo.GetShape();
  inputTensors.push_back(Ort::Value::CreateTensor(memoryInfo,input_data[
i], input_data_len[i], inputDims.data(), inputDims.size(),inputType));
}
```

4. 执行以下命令，使用 IOBinding 来获取模型输出。

```
Ort::IoBinding ioBinding(session);
for(size_t i=0; i< inputNames.size(), i++){
  ioBinding.BindInput(inputNames[i],inputTensors[i]);
}
for(size_t i=0; i< outputNames.size(), i++){
  ioBinding.BindOutput(outputNames[i],memoryInfo);
}
session.Run(Ort::RunOptions{nullptr},ioBinding);
auto outputs=ioBinding.GetOutputValues();
```

3.3 动态 Batch 推理

MacaRT 支持 ONNX Runtime 的动态 Batch 推理功能，在指定推理后端为 MacaEP 时，使用 `BatchSize` 值为`-1` 的模型。

推理原理

推理原理如图 3-1 所示，其中 input_i_j 代表第 i 组 Batch 数据，模型的第 j 个输入，M 为模型结构需要的输入个数，N 为输入的 Batch 数目，K 为模型结构的输出个数。input_i 包含了 input_0_i 到 input_N_i 的所有输入数据。

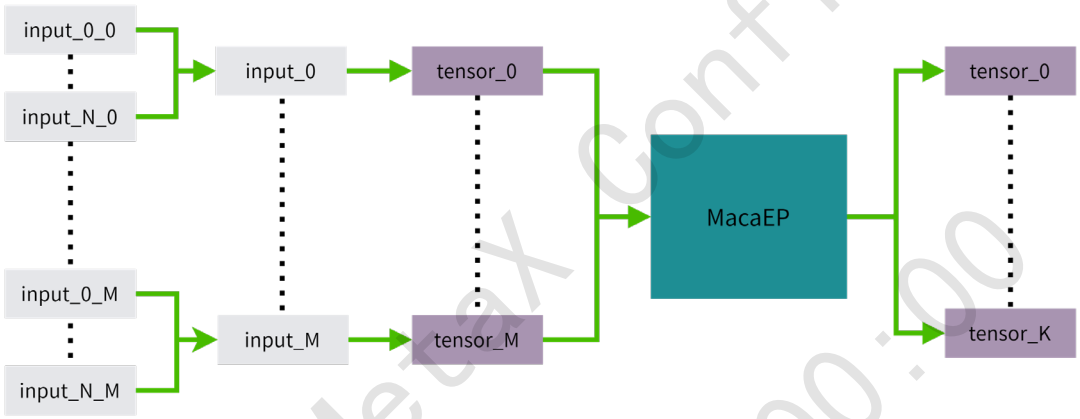

图 3-1 动态 Batch 推理原理图

操作步骤

1. 准备输入数据。假设提供的模型输入数据和数据长度是通过以下变量名称提供，input_data 包含了 Batch 为 N 的输入数据，且输入数据位于可分页内存上。

```
void* input_data[inputNames.size()];
size_t input_data_len[inputNames.size()];
```

2. 创建 `MemoryInfo`，用于标识输入数据所在的设备信息。

```
Ort::MemoryInfo memoryInfo =Ort::MemoryInfo::CreateCpu(
          OrtAllocatorType::OrtArenaAllocator,
OrtMemType::OrtMemTypeDefault);
```

3. 创建输入 Tensors。

```
std::vector<Ort::Value> inputTensors;
for(size_t i=0; i<inputNames.size(); i++){
// get input node data type
  Ort::TypeInfo inputTypeInfo = session.GetInputTypeInfo(i);
  auto inputTensorInfo = inputTypeInfo.GetTensorTypeAndShapeInfo();
  ONNXTensorElementDataType inputType = inputTensorInfo.GetElementType();
  // get input node data length
  std::vector<int64_t> inputDims=inputTensorInfo.GetShape();
```

```
  inputDims[0]=N;
  inputTensors.push_back(Ort::Value::CreateTensor(memoryInfo,input_data[
i], input_data_len[i], inputDims.data(), inputDims.size(),inputType));
}
}
```

4. 支持使用以下两种方式获取模型输出。

```
// session run
auto ouput = session.Run(Ort::RunOptions{nullptr},inputNames.data(),
                         inputTensors.data(),inputNames.size(),
                         ouputNames.data(),outputNames.size());
```

或

```
// iobinding
Ort::IoBinding ioBinding(session);
for(size_t i=0; i< inputNames.size(); i++){
  ioBinding.BindInput(inputNames[i],inputTensors[i]);
}
for(size_t i=0; i< outputNames.size(); i++){
  ioBinding.BindOutput(outputNames[i],memoryInfo);
}
session.Run(Ort::RunOptions{nullptr},ioBinding);
auto outputs = ioBinding.GetOutputValues();
```

3.4 提升曦云系列 GPU 推理性能

使用 MacaRT 将 ONNX 模型部署到曦云系列 GPU 上时，MacaEP 配置信息 `OrtMACAProviderOptions` 会直接影响 MacaRT 的推理速度，`OrtMACAProviderOptions` 支持的配置选项参见表 3-1。

表 3-1 OrtMACAProviderOptions 支持的配置选项

选项	说明
device_id	配置所使用的设备号
gpu_mem_limit	MacaRT 使用的最大显存量
arena_extend_strategy	显存增长策略
default_memory_arena_cfg	内存管理配置

3.4.1 设备 ID

当有多个 GPU 时，需通过配置 `gpu_id` 来管理。通过在 `OrtMACAProviderOptions` 中配置 `device_id`，可以指定所使用的 `gpu_id`，默认为 0。

```
OrtMACAProviderOptions maca_options;
```

```
maca_options.device_id = your_gpu_id;
```

3.4.2 MacaRT 显存管理

MacaRT 提供 BFCarena 显存管理策略，可以有效地使用显存，避免显存重复申请和显存碎片。有关 BFCarena 的更多信息，可参考 ONNX Runtime 的相关文档。

有以下两种方法可以配置 MacaEP 显存管理策略：

- 直接使用 `OrtMACAProviderOptions` 配置。

```
OrtMACAProviderOptions maca_options;
maca_options.gpu_mem_limit=SIZE_MAX;
maca_options.arena_extend_strategy=0;
```

- 创建一个 `OrtArenaCfg` 对象。

```
auto cfg = Ort::ArenaCfg(SIZE_MAX,0,-1,-1);
OrtMACAProviderOptions maca_options;
maca_options.default_memory_arena_cfg=cfg.release();
```

3.5 自定义算子

MacaRT 支持用户定义非官方的 ONNX 算子进行推理。

操作步骤

1. 使用 C++的 API 构建自定义算子库。
2. 通过使用 C++ API 或 Python API 将自定义算子库注册到 `SessionOptions` 对应的 Maca EP。
3. 加载包含自定义算子的模型进行推理。

上述步骤的具体实现细节，可以参考 ONNX Runtime 官方文档中关于 Cuda EP 自定义算子的实现步骤，Maca EP 保持基本一致，我们也在 **/opt/maca-ai/onnxruntime-maca/custom_op_test** 中提供了完整的实现样例。

以下介绍 Maca EP 自定义算子的不同之处。

3.5.1 Domain 的定义

将自定义算子注册到 Maca EP 的后端时，Domain 必须设置为 `custom.op`，即：

```
static const char* c_OpDomain="custom.op";
```

3.5.2 Kernel 输入输出的数据排布

在构建自定义算子库时，在 MacaRT 中新增接口用于指定 Kernel 需要的输入输出数据排布。

```
OrtDataLayout GetInputDataLayout(size_t index); //指定输入的数据排布
OrtDataLayout GetOutputDataLayout(size_t index); //指定输出的数据排布
```

当前 Maca EP 的自定义算子支持以下三种数据排布：

```
typedef enum OrtDataLayout{
      NCHW,
      NHWC8,
      NHWC16,
}OrtDataLayout;
```

需要注意的是：

- 在默认情况下，所有自定义算子 Kernel 的输入输出都是 NCHW 格式。
- 模型的输入数据排布必须是 NCHW 格式，Maca EP 会根据 Kernel 需求自动进行数据排布的转换。
- 模型的输出会被 Maca EP 自动转换为 NCHW 格式的排布。

4. MacaRT Python API

4.1 部署 ONNX 模型

MacaRT 支持使用 Python API 将 ONNX 模型部署到曦云系列 GPU 上，并完成推理。

4.1.1 创建 Session

操作步骤

1. 执行以下命令，创建 Session。

```
import onnxruntime as ort
provider_options=[{'device_id':0}]
session = ort.InferenceSession(your_onnx_model_path,
          providers=["MACAExecutionProvider"],provider_options=provider_o
          ptions)
```

4.1.2 获取模型图的输入输出信息

操作步骤

1. 执行以下命令，获取模型图的输入输出信息。

```
input_nodes = session.get_inputs()
input_names = [i_n.name for i_n in input_nodes]
output_nodes = session.get_outputs()
output_names = [o_n.name for o_n in output_nodes]
```

4.1.3 构建模型输入字典

操作步骤

1. 准备输入数据。假设已创建一个包含所有模型输入数据的 List，List 中的元素为模型的每个输入的 `numpy` 数据，且输入 List 的变量名为 `input_data_list`。
2. 构建输入字典。

```
input_dict ={}
for i_d, i_n in zip(input_data_list, input_names):
  input_dict[i_n] = i_d
```

4.1.4 获取模型输出

操作步骤

1. 执行以下命令，获取模型输出。

```
output_data = session.run([], input_dict )
```

4.2 绑定输入输出设备

有多个模型且模型间存在数据拷贝时，绑定输入输出内存信息，可帮助减少模型之间不必要的输入输出数据拷贝。

MacaRT Python API 也支持将模型的输入或输出绑定到可分页内存、固页内存和显存上。

4.2.1 绑定输入输出数据到可分页内存上

操作步骤

1. 操作步骤同 4.1.1 创建 Session。
2. 操作步骤同 4.1.2 获取模型图的输入输出信息。
3. 操作步骤同 4.1.3 构建模型输入字典。
4. 执行以下命令，使用 IOBinding 获取模型输出。

```
io_binding = session.io_binding()
for key in input_dict.keys():
  io_binding.bind_ortvalue_input(key,ort.OrtValue.ortvalue_from_numpy(in
put_dict[key], "cpu",0))
for o_n in output_names:
  io_binding.bind_output(o_n,"cpu")
session.run_with_iobinding(io_binding)
output = io_binding.get_outputs()
```

4.2.2 绑定输入输出数据到固页内存上

操作步骤

1. 操作步骤同 4.1.1 创建 Session。
2. 操作步骤同 4.1.2 获取模型图的输入输出信息。
3. 操作步骤同 4.1.3 构建模型输入字典。
4. 执行以下命令，使用 IOBinding 获取模型输出。

```
io_binding = session.io_binding()
for key in input_dict.keys():
  io_binding.bind_ortvalue_input(key,ort.OrtValue.ortvalue_from_numpy(in
put_dict[key], "maca_pinned",gpu_id))
for o_n in ouput_names:
  io_binding.bind_output(o_n,"maca_pinned")
session.run_with_iobinding(io_binding)
output = io_binding.get_outputs()
```

4.2.3 绑定输入输出数据到显存上

操作步骤

1. 操作步骤同 4.1.1 创建 Session。
2. 操作步骤同 4.1.2 获取模型图的输入输出信息。
3. 操作步骤同 4.1.3 构建模型输入字典。
4. 执行以下命令，使用 IOBinding 获取模型输出。

```
io_binding = session.io_binding()
for key in input_dict.keys():
  io_binding.bind_ortvalue_input(key,ort.OrtValue.ortvalue_from_numpy(in
put_dict[key], "maca",gpu_id))
for o_n in output_names:
  io_binding.bind_output(o_n,"maca",device_id = gpu_id)
session.run_with_iobinding(io_binding)
output = io_binding.get_outputs()
```

4.3 动态 Batch 推理

操作步骤

1. 准备输入数据。如下所示，其中 input_*代表由模型图的其中一个输入，且每一个输入数据的 BatchSize 都为 N。

```
input_data = [input_0,input_1,...,input_N]
```

2. 操作步骤同 4.1.1 创建 Session。
3. 操作步骤同 4.1.2 获取模型图的输入输出信息。
4. 使用 IOBinding 获取输出。

```
io_binding = session.io_binding()
for i_n,b_d in zip(input_names,input_data):
      io_binding.bind_ortvalue_input(i_n,ort.OrtValue.ortvalue_from_numpy
(b_d, "cpu",0))
for o_n in ouput_names:
```

```
      io_binding.bind_output(o_n,"cpu")
session.run_with_iobinding(io_binding)
output = io_binding.get_outputs()
```

4.4 提升曦云系列 GPU 推理性能

使用 MacaRT 将 ONNX 模型部署到曦云系列 GPU 上时，MacaEP 的配置信息 `provider_options` 会直接影响 MacaRT 的推理速度。相关配置信息的介绍，参见 3.4 提升曦云系列 GPU 推理性能。

4.4.1 设备 ID 设置

操作步骤

1. 执行以下命令，配置 `device_id` 来指定 `gpu_id`，以管理多个 GPU。

```
provider_options=[{'device_id': your_gpu_id }]
```

5. MacaRT 工具链

MacaRT 提供以下工具：

- 模型转换工具 MacaConverter。针对各种深度学习框架（如 Pytorch，Caffe，Tensorflow）训练出来的模型，可以使用 MacaConverter 将训练的模型转换为 ONNX 模型。
- 精度分析工具 MacaPrecision。通过比较 CPU 和曦云系列 GPU 在各个层的输出值，更好地调试模型。
- 模型量化工具 MacaQuantizer。相对于 MacaRT 提供的模型量化功能，MacaQuantizer 可以在模型量化后达到更高的精度和更快的速度。

MacaRT 模型部署工具链如下图所示：

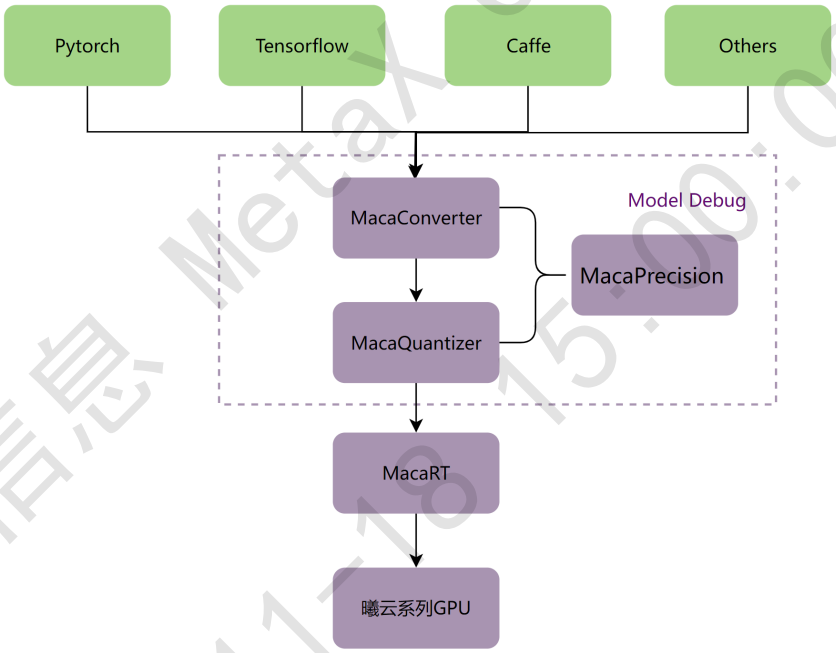

图 5-1 MacaRT 模型部署工具链

5.1 MacaConverter

ONNX 是一种便于在各个主流深度学习框架中迁移模型的中间表达格式，可用于存储训练好的模型。它使得不同的深度学习框架（如 Pytorch，Caffe，TensorFlow）可以采用相同格式存储模型数据，从而进行推理任务的执行。

为了将不同深度学习框架的模型转换为 ONNX 格式的模型，MacaConverter 的内部封装了常用的开源转换工具，对外提供统一的转换接口（命令行），同时提供了部分算子优化功能。

5.1.1 安装

MacaConverter 以 wheel 安装包的形式对外发布，建议在 Conda 环境中安装（建议 Python 版本 3.8）。安装 MacaRT 后，可以在/opt/maca-ai/onnxruntime-maca/python/tools 目录下找到 MacaConverter 对应的 Python 安装包。

操作步骤

1. 执行以下命令，安装 MacaConverter。

```
conda create -n maca_test python=3.8
source activate maca_test
pip install onnxruntime_gpu-1.12.0+mc*.whl
pip install maca_converter-*.whl
```

说明

若运行时报以下错误：

ImportError: libpython3.8.so.1.0: cannot open shared object file: No such file or directory

可执行以下命令：

```
export
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/xxx/anaconda3/envs/maca_test/lib/
```

wheel 包安装过程中会自动下载相关依赖库，主要有：

- numpy
- onnx
- onnxruntime-gpu
- TensorFlow==2.4.0
- torch
- torchvision
- onnxoptimizer==0.2.6
- rich==12.0.0
- packaging
- pyyaml
- paddlepaddle
- paddle2onnx

5.1.2 功能列表

MacaConverter 支持将多种类型的模型转换为 ONNX 模型，参见表 5-1。

表 5-1 支持的模型转换类型

源模型框架	目标模型框架	说明
Caffe	ONNX	支持将 Caffe 模型转换为 ONNX 模型
TensorFlow	ONNX	支持将 TensorFlow 模型转换为 ONNX 模型（包括 CheckPoint，SavedModel，pb 和 H5 四种格式）
Pytorch	ONNX	支持将 Pytorch 模型转换为 ONNX 模型
PaddlePaddle	ONNX	支持将 PaddlePaddle 模型转换为 ONNX 模型
Darknet	ONNX	支持将 Darknet 模型转换为 ONNX 模型（目前只支持 yolov3 和 yolov4）

MacaConverter 还支持其他功能，参见表 5-2。

表 5-2 扩展功能

功能	说明
动态 batch 转换	支持将 BatchSize 非动态的模型修改为动态 BatchSize
ONNX 优化	支持常规的优化操作（如常量折叠、Conv+BN 融合）
Float32 转 Float16	支持将模型中相关算子的输入类型由 Float32 改为 Float16
子图提取	按照给定的输入/输出提取模型的子图
融合 Pad+Pool	将相邻的 Pad 算子融合进 Pool 算子中
op_set 版本转换	支持将原 ONNX 模型的 op_set 版本转换为目标版本
Float32 转 Uint8	支持将模型的输入类型由 Float32 改为 Uint8（仅针对模型的 input 数据，非所有算子）

5.1.3 使用说明

5.1.3.1 Caffe 模型转 ONNX 模型

操作步骤

1. 执行以下命令，将 Caffe 模型转为 ONNX 模型。

```
python -m maca_converter --model_path ./caffe_model  --model_type caffe
--output ./output.onnx
```

参数说明：

`model_path`：Caffe 模型路径（文件夹，需包含.prototxt 和.caffemodel）

`model_type`：模型类型（此处固定为 **`caffe`**）

output：目标模型的路径

5.1.3.2 TensorFlow（H5）模型转 ONNX 模型

操作步骤

1. 执行以下命令，将 TensorFlow H5 格式的模型转为 ONNX 模型。

```
python -m maca_converter --model_path ./test.h5  --model_type tf-h5  --
output ./output.onnx
```

参数说明：

model_path：H5 模型路径

model_type：模型类型（此处固定为 **tf-h5**）

output：目标模型的路径

5.1.3.3 TensorFlow（SavedModel）模型转 ONNX 模型

操作步骤

1. 执行以下命令，将 TensorFlow SavedModel 格式的模型转为 ONNX 模型。

```
python -m maca_converter --model_path ./tfsm  --model_type tf-sm  --
output ./output.onnx
```

参数说明：

model_path：SavedModel 模型路径（文件夹，需包含 assets/saved_model.pb/variables）

model_type：模型类型（此处固定为 **tf-sm**）

output：目标模型的路径

5.1.3.4 TensorFlow（CheckPoint）模型转 ONNX 模型

操作步骤

1. 执行以下命令，将 TensorFlow CheckPoint 格式的模型转为 ONNX 模型。

```
python -m maca_converter --model_path ./ckpt/test.meta --model_type tf-
ckpt  --output ./output.onnx --inputs x:0,y:0  --outputs op_to_store:0 --
inputs_as_nchw x:0,y:0
```

参数说明：

model_path：CheckPoint 模型路径（文件夹，需包含 checkpoint、.meta、.index、.data 文件，此参数需指向.meta 文件的路径）

model_type：模型类型（此处固定为 **tf-ckpt**）

output：目标模型的路径

inputs：TensorFlow 模型的输入（tensor 变量的名称，一般后面需加:0）

outputs：TensorFlow 模型的输出（tensor 变量的名称，一般后面需加:0）

inputs_as_nchw：可选项，如果原始输入维度是 **nhwc**，可使用此选项转变为 **nchw**

5.1.3.5 TensorFlow（pb）模型转 ONNX 模型

操作步骤

1. 执行以下命令，将 TensorFlow pb 格式的模型转为 ONNX 模型。

```
python -m maca_converter --model_path ./test.pb --model_type tf-graph  --
output ./output.onnx  --inputs x:0,y:0 --outputs op_to_store:0  --
inputs_as_nchw x:0,y:0
```

参数说明：

model_path：pb 模型路径

model_type：模型类型（此处固定为 **tf-graph**）

output：目标模型的路径

inputs：TensorFlow 模型的输入（tensor 变量的名称，一般后面需加:0）

outputs：TensorFlow 模型的输出（tensor 变量的名称，一般后面需加:0）

inputs_as_nchw：可选项，如果原始输入维度是 **nhwc**，可使用此选项转变为 **nchw**

5.1.3.6 Pytorch 模型转 ONNX 模型（输入包含模型定义和权重）

操作步骤

1. 执行以下命令，将 Pytorch 模型转为 ONNX 模型（输入包含模型定义和权重）。

```
python -m maca_converter --model_path ./mnist_model.pkl  --model_type
pytorch  --output ./torch.onnx  --model_def_file ./CNN.py  --
model_class_name CNN  --input_shape [1,1,28,28]
```

参数说明：

model_path：Pytorch 模型路径（模型中包含了权重和定义）

model_type：模型类型（此处固定为 **pytorch**）

output：目标模型的路径

model_def_file：模型定义文件（必须和执行命令的路径在同一目录下）

model_class_name：模型中定义的类名

input_shape：模型的输入形状

5.1.3.7 Pytorch 模型转 ONNX 模型（输入仅包含权重）

操作步骤

1. 执行以下命令，将 Pytorch 模型转为 ONNX 模型（输入仅包含权重）。

```
python -m maca_converter --model_path ./xxx  --model_type pytorch  --
output ./unet.onnx  --model_def_file  ./unet.py   --model_class_name Net
--model_weights_file ./9_epoch_0.56322173.pth  --input_shape [64,3,32,32]
```

参数说明：

model_path：Pytorch 模型路径（此处可任意指定，因为这种场景下只有权重文件，没有模型）

model_type：模型类型（此处固定为 **pytorch**）

output：目标模型的路径

model_def_file：模型定义文件（必须和执行命令的路径在同一目录下）

model_class_name：模型中定义的类别名称

model_weights_file：模型权重文件的路径

input_shape：模型的输入形状

5.1.3.8 Pytorch 模型转 ONNX 模型（输入仅包含权重，且模型定义在 Torchvision 中）

操作步骤

1. 执行以下命令，将 Pytorch 模型转为 ONNX 模型（输入仅包含权重，且模型定义在 Torchvision 中）。

```
python -m maca_converter --model_path ./xxx --model_type pytorch  --
output ./output.onnx  --model_class_name torchvision.models.resnet50  --
model_weights_file ./0.9696.pth   --input_shape [16,3,256,256]
```

参数说明：

model_path：Pytorch 模型路径（此处可任意指定，因为这种场景下只有权重文件，没有模型）

model_type：模型类型（此处固定为 **pytorch**）

output：目标模型的路径

model_class_name：模型中定义的类别名称

input_shape：模型的输入形状

5.1.3.9 Darknet 模型转 ONNX 模型

操作步骤

1. 执行以下命令，将 Darket 模型转为 ONNX 模型。

```
python -m maca_converter --model_path ./my_model --model_type darknet --
output ./output.onnx
```

参数说明：

model_path：Darknet 模型路径（文件夹，需包含.cfg 和.weights 文件）

model_type：模型类型（此处固定为 **darknet**）

output：目标模型的路径

5.1.3.10 PaddlePaddle 模型转 ONNX 模型（输入包含权重和定义）

操作步骤

1. 执行以下命令，将 PaddlePaddle 模型转为 ONNX 模型（输入包含权重和定义）。

```
python -m maca_converter --model_path ./paddle_model  --model_type paddle
--output ./output.onnx
```

参数说明：

model_path：PaddlePaddle 模型路径（文件夹，需包含.pdmodel，.pdiparams.info 和.pdiparams 文件）

model_type：模型类型（此处固定为 **paddle**）

output：目标模型的路径

5.1.3.11 PaddlePaddle 模型转 ONNX 模型（输入仅包含权重）

操作步骤

1. 执行以下命令，将 PaddlePaddle 模型转为 ONNX 模型（输入仅包含权重）。

```
python -m maca_converter --model_path ./xxx  --model_type paddle  --
output ./paddle.onnx  --model_def_file  ./mnist.py --model_class_name
LeNet  --model_weights_file ./paddle_checkpoint/final.pdparams --
input_shape [1,1,28,28]
```

参数说明：

model_path：PaddlePaddle 模型路径（可任意指定，因为此场景下只有权重文件，没有模型）

model_type：模型类型（此处固定为 **paddle**）

output：目标模型的路径

model_def_file：模型定义文件（必须和执行命令的路径在同一目录下）

model_class_name：模型中定义的类别名称

model_weights_file：模型权重文件的路径

input_shape：模型的输入形状

5.1.3.12 PaddlePaddle 模型转 ONNX 模型（输入仅包含权重，模型在 paddle.vision 中定义）

操作步骤

1. 执行以下命令，将 PaddlePaddle 模型转为 ONNX 模型（输入仅包含权重，模型在 paddle.vision 中定义）。

```
python -m maca_converter --model_path ./xxx  --model_type paddle   --
output ./paddle.onnx  --model_class_name paddle.vision.models.LeNet  --
model_weights_file ./final.pdparams --input_shape [1,1,28,28]
```

参数说明：

model_path：PaddlePaddle 模型路径（此处可任意指定，因为这种场景下只有权重文件，没有模型）

model_type：模型类型（此处固定为 **paddle**）

output：目标模型的路径

model_class_name：模型中定义的类别名称

model_weights_file：模型权重文件的路径

input_shape：模型的输入形状

5.1.3.13 动态 Batch 转换

操作步骤

1. 执行以下命令，将 BatchSize 非动态的模型转换为动态模型。

```
python -m maca_converter --model_path ./caffe_model  --model_type caffe -
-output ./output.onnx  --dynamic_batch 1
```

5.1.3.14 ONNX 简化

操作步骤

1. 执行以下命令，对输入的模型进行常规优化（如常量折叠，Conv+BN 融合等）。

```
python -m maca_converter --model_path ./test.onnx  --model_type onnx  --
output ./output.onnx
```

说明

该功能默认打开。如需关闭，可加参数-simplify 0。

5.1.3.15 FP32 转 FP16

操作步骤

1. 执行以下命令，将模型中相关算子的输入类型由 Float32 改为 Float16。

```
python -m maca_converter --model_path ./test.onnx --model_type onnx --
output ./output.onnx  --fp32_to_fp16 1
```

5.1.3.16 子图提取

操作步骤

1. 执行以下命令，按照给定的 input/output，提取子图，保存模型。

```
python -m maca_converter --model_path ./test.onnx  --model_type onnx  --
output ./output.onnx --extract_sub 1 --inputs input_1 --outputs
functional_1/concatenate/concat
```

5.1.3.17 op_set 版本转换

操作步骤

1. 执行以下命令，转换 ONNX 模型的 op_set 版本。

```
python -m maca_converter --model_path ./test.onnx --model_type onnx  --
output ./output.onnx  --op_set 13
```

5.1.3.18 Pad 融合

操作步骤

1. 执行以下命令，将模型中相邻的 Pad+Pool 组合，融合进 Pool 算子里。

```
python -m maca_converter  --model_path ./test.onnx --model_type onnx --
output ./output.onnx  --fuse_pad_pool 1
```

5.1.3.19 Float32 转 Uint8（仅针对模型的 input 数据，非所有算子）

操作步骤

1. 执行以下命令，将模型的输入类型由 Float32 改为 Uint8。

```
python -m maca_converter --model_path ./test.onnx --model_type onnx --
output ./output.onnx  --fp32_to_u8 1
```

5.2 MacaPrecision

GPU 开发过程中，针对 ONNX 模型中每一个算子，需要验证其在 GPU 硬件上的计算结果是否正确，以保证相关算子软硬件实现的可靠性。MacaPrecision 是用于保证整个流程完备性的精度对比工具。

MacaPrecision 主要实现以下三个方面的功能：

- 逐层比较 AI 模型在 GPU 与 CPU 上的运行输出结果。
- 支持全量比较与指定节点比较两种方式。
- 单层计算结果的精度对比，支持 SNR/Cosine/MSE 三种方式。

5.2.1 安装

MacaPrecision 以 wheel 安装包的形式对外发布，建议在 Conda 环境中安装（建议 Python 版本为 3.8）。安装 MacaRT 后，可以在**/opt/maca-ai/onnxruntime-maca/python/tools** 目录下找到 MacaPrecision 对应的 Python 安装包。

操作步骤

1. 执行以下命令，安装 MacaPrecision。

```
conda create -n maca_test python=3.8
source activate maca_test
pip install onnxruntime_gpu-1.12.0+mc*.whl
pip install maca_precision-*.whl
```

wheel 包安装过程中会自动下载相关依赖库，主要有：

- onnx
- onnxruntime-gpu
- numpy

5.2.2 使用说明

MacaPrecision 以 CPU 计算结果为基准，对比每一层的输出结果，误差超过一定范围，认为计算精度有误。具体流程如下：

1. 加载 ONNX 模型。
2. 解析 ONNX 模型，分析 input 变量，生成随机输入数据。
3. 在 CPU 上运行模型，保存输出结果（output_cpu）。
4. 在 GPU 上运行模型，保存输出结果（output_gpu）。

5. 逐层比较 output_cpu 与 output_gpu，如某一层的差值超过预设的阈值，则认为 GPU 计算精度存在问题，需要排查。

由于常规模型与量化模型在内部处理方式上存在差异，所以执行操作时需要区分常规模型与量化模型。

常规模型（FP32/FP16）

全量模式

执行以下命令，在全量模式下进行常规模型推理。

```
python -m maca_precision -i ./test.onnx  -c snr  -t common -b 6
```

- 参数-i 指定需要测试的模型路径；
- 参数-c 指定用于逐层对比的计算方法，支持 snr/mse/cosine 三种方式(可不指定，默认 snr)；
- 参数-t 指定模型类型，此处固定为 **common**；
- 参数-b 指定每次比较的中间节点的个数，可不指定，默认为 5(此处需要分批比较的原因是中间结果的参数量可能会比较大，如果直接全部比较会占用大量显存，如指定-1，则一次进行全量比较)。

全量模式下，会首先进行最后一层的比较，如果结果一致，则认为模型推理无异常，不会进行中间节点的比较。

指定模式

执行以下命令，在指定模式下进行常规模型推理。

```
python -m maca_precision -i ./test.onnx  -c snr  -t common -m Conv_0,Relu_1
```

参数-m 指定需要比较的中间节点的名称，多个名称以逗号分隔。

指定模式下，无论最终的结果是否一致，都会进行指定的中间节点的比较。

量化模型（经过 MacaQuantizer 量化后的模型）

量化模型的操作与常规模型基本一致，也分为全量模式和指定模式。指定模式下的节点，只能为量化节点。

唯一不同的是，-t 参数需要指定为 quantize。

例如：

```
python -m maca_precision -i  ./test.onnx -t quantize -b 3
```

或

```
python -m maca_precision -i  ./test.onnx  -t quantize -m
PPQ_Operation_152,PPQ_Operation_160
```

5.3 MacaQuantizer

MacaQuantizer 是一个高效的神经网络量化工具，通过自定义的量化算子库、网络执行器、神经网络调度器、量化计算图等设计，即便在网络极度复杂的情况下，依然能够得到正确的网络量化结果。量化过程中会通过读取配置参数文件，自动搜索最优的量化方案，同时量化中严格控制硬件模拟误差，保证硬件推理时的精度。

目前 MacaQuantizer 使用 ONNX（opset 11~13）模型文件作为输入，覆盖常用的 80 余种 ONNX 算子类型。如果是 Pytorch，TensorFlow 等其他模型，可使用 MacaConverter 将模型转换为 ONNX 模型。

5.3.1 安装

操作步骤

1. 安装 Python（建议版本为 3.8）。

   安装 MacaRT 后，可以在 /opt/maca-ai/onnxruntime-maca/python/tools 目录下找到 MacaQuantizer 对应的 Python 安装包。

2. 执行以下命令，安装 MacaQuantizer。

```
pip install maca_quantizer-xxxx-py3-none-any.whl
```

5.3.2 使用说明

为了提高量化速度，MacaQuantizer 量化工具可以使用曦云系列 GPU 进行量化加速。使用曦云系列 GPU 可以加速量化过程中的重训练过程，提高量化速度。

默认采用曦云系列 GPU 进行加速计算。如果没有曦云系列 GPU 环境，可以使用 CPU 版本进行量化工具，通过设置以下环境变量来进行 GPU 版和 CPU 版切换。环境变量设置为 0 时表示使用 CPU 版本进行加速计算，为 1 时表示使用曦云系列 GPU 进行加速计算。

```
export MACA_QUANTIZER_USING_MXGPU=0
```

运行 MacaQuantizer 之前，需要配置一个 yaml 格式的参数文件。格式如下所示：

```
import_model: /path/to/mode.onnx
export_model: /path/to/export_model.onnx
export_type: onnx
export_batch: 1
quant_algorithm: percentile
output_threshold: 0.1
without_bs: False
force_advance_quant: False
collecting_device: gpu
dataset:
```

```
  calib_dir: /path/to/dataset/calibrate/
  calib_num: 200
  batch_size: 8
  preprocessing:
     enable: True
     attributes:
       isreverse: False
       mean: [123.76, 116,28, 103,53]
       std: [58.4, 57.12, 57.37]
       resize:
          keep_ratio: False
          to: [3, 256, 256]
          centercrop: [224, 224]
```

其中每个参数的说明，参见表 5-3。

表 5-3 参数说明

参数	值	默认值	说明
import_model			输入模型路径，目前只支持 ONNX 模型
export_model			输出模型路径，目前只支持 ONNX 模型
export_type	[maca, native]	maca	导出模型的类型 onnx：导出 onnx qdq 格式 native：导出 qdq 格式外，额外导出二进制的模型
quant_algorithm	[percentile, minmax,mse, kl]	percentile	量化算法选择
export_batch	[True, False]	1	导出模型的 BatchSize，默认为 1
output_threshold	0-1	0.1	量化后 SNR 误差最大允许范围
without_bs	[True, False]	False	模型输入是否带有 BatchSize 维度
force_advance_quant	[True, False]	False	是否量化过程中开启强制优化
calib_dir			校准数据集的目录或 dataset.txt 路径。如果模型为多输入，仅支持 dataset.txt 文本格式，且数据为 npy 或二进制数据
calib_num	[50-500]		校准数据集的数量
batch_size			量化计算时的 BatchSize
enable	[True, False]		是否对数据做预处理 False：校准数据需要是 npy 或二进制数据
isreverse	[True, False]		是否将图像通道 RGB 转为 BGR
mean			预处理均值
std			预处理方差。

参数	值	默认值	说明
keep_ratio	[True, False]		Resize 是否保持等比例。
to			Resize 大小[chw]。
pad_value		0	Resize 边缘填充的值。
centercrop	[True, False]	False	centercrop 大小[h, w]。 如不需要 centercrop，可删除该参数。

操作步骤

1. 执行以下命令，运行模型量化。

```
python  -m  maca_quantizer  -c  quantize.yaml
```

参数说明：

`-c/--config`：yaml 参数文件

预处理说明

预处理流程如图 5-2 所示。

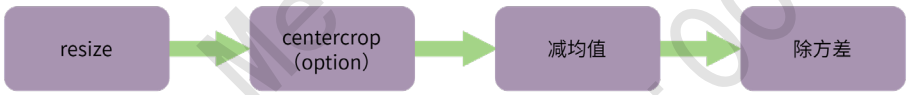

图 5-2 预处理流程

如果提供的预处理方案与模型的预处理方式不一致或无法满足要求时，可将预处理后的数据离线保存为 npy 或二进制格式，关闭数据读取时的预处理，直接读取预处理后的数据。

yaml 示例如下所示：

```
import_model: /path/to/mode.onnx
export_model: /path/to/export_model.onnx
export_type: onnx
quant_algorithm: percentile
export_batch: 1
without_bs: False
force_advance_quant: False
dataset:
   calib_dir: /path/to/dataset/calibrate/
   calib_num: 200
   batch_size: 8
   preprocessing:
      enable: False
```

dataset.txt 文本格式说明

若输入 `calib_dir` 为 dataset.txt 文本格式，请参考如下格式。

- 单输入模型：只读取每行第一列参数（空格分割）；

```
val/ILSVRC2012_val_00000001.JPEG 66
val/ILSVRC2012_val_00000002.JPEG 971
val/ILSVRC2012_val_00000003.JPEG 231
val/ILSVRC2012_val_00000004.JPEG 810
val/ILSVRC2012_val_00000005.JPEG 517
val/ILSVRC2012_val_00000006.JPEG 58
val/ILSVRC2012_val_00000007.JPEG 335
val/ILSVRC2012_val_00000008.JPEG 416
val/ILSVRC2012_val_00000009.JPEG 675
```

- 多输入模型：每一行为一个数据样本，输入空格分割。

```
1000000000_input_ids.npy 1000000000_input_mask.npy
1000000000_segment_ids.npy
1000000001_input_ids.npy 1000000001_input_mask.npy
1000000001_segment_ids.npy
1000000002_input_ids.npy 1000000002_input_mask.npy
1000000002_segment_ids.npy
1000000003_input_ids.npy 1000000003_input_mask.npy
1000000003_segment_ids.npy
1000000004_input_ids.npy 1000000004_input_mask.npy
1000000004_segment_ids.npy
1000000005_input_ids.npy 1000000005_input_mask.npy
1000000005_segment_ids.npy
```

5.3.3 注意事项

- 目前只支持 ONNX 模型的量化，如果不是，建议量化前使用 MacaConverter 对模型进行转换。
- 目前只支持 ONNX（opset 11~13）模型文件作为输入，如果不是，建议量化前使用 MacaConverter 对模型转换。
- 目前 MacaQuantizer 支持的算子参见表 5-4：

表 5-4 MacaQuantizer 支持的 ONNX 算子

Gemm	ReduceMean	Sqrt
grid_sampler	ReduceSum	Log
GlobalAveragePool	Relu	Floor
GlobalMaxPool	Reshape	RoiAlign
Greater	Resize	MMCVRoiAlign
LeakyRelu	ScatterElements	SpaceToDepth

Less	ScatterND	DepthToSpace
MatMul	Shape	Tanh
Max	Sigmoid	Pow
MaxPool	Slice	InstanceNormalization
Min	Softmax	HardSigmoid
Mul	Softplus	HardSwish
NonMaxSuppression	Split	Neg
NonZero	Squeeze	GRU
Not	Sub	Swish
Pad	Tile	Identity
PRelu	TopK	OneHot
Range	Transpose	Reciprocal
ReduceL2	Unsqueeze	Mish
ReduceMax	Where	Elu
Sum	Erf	

6. MacaRT-LLM

6.1 MacaRT-LLM 介绍

MacaRT-LLM 是在曦云系列 GPU 上进行大模型部署和执行的推理引擎，是在 Maca 后端适配了 OpenPPL-LLM。使用 MacaRT-LLM 在曦云系列 GPU 上进行大模型推理，其方法和功能与 OpenPPL-LLM 完全兼容。

OpenPPL-LLM 进行大模型推理的整体流程，如图 6-1 所示，可以分为以下部分：

1. LLM 模型转换，使用 MacaPMX 将 LLM 原始模型转换成 OpenPPL-LLM 支持的 ONNX 模型。
2. 在 PPL.NN.LLM 上对模型图进行处理，包括图优化、图拆分、图编译等。
3. 适配 PPL.LLM Serving，支持 LLM 云端服务。
4. PPL.LLM Kernel Library 通过 MXMACA Driver 在曦云系列 GPU 上执行模型。

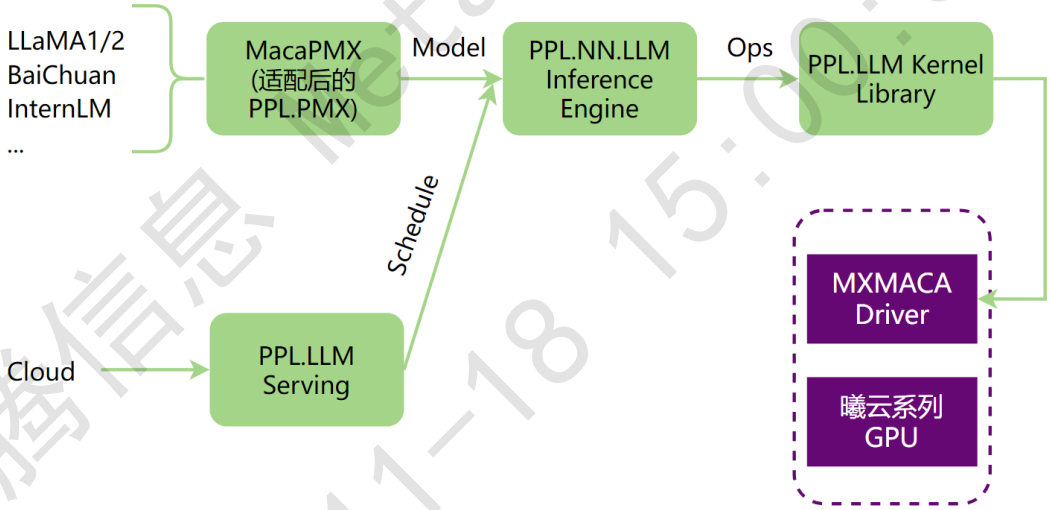

图 6-1 OpenPPL-LLM 大模型推理流程图

6.2 MacaRT-LLM 功能

MacaRT-LLM 完全适配了 OpenPPL-LLM，包含了以下功能和特性：

- 完全适配 PPL.PMX，提供 MacaPMX 支持多种主流大模型进行模型转换。
- 完全适配 PPL.NN.LLM，支持多个主流大模型的推理和模型切分多卡并行推理。
- 完全适配 PPL.LLM.Serving，支持大模型服务化部署。

6.3 MacaRT-LLM 使用流程

本章节介绍 MacaRT-LLM 的使用步骤，主要分为模型转换、本地模型部署验证及服务化部署三个部分。

6.3.1 模型转换

使用 MacaPMX 进行模型转换，MacaPMX 与 PPL.PMX 完全适配，为每个 OP 提供了一组标准的操作符规范文档和相应的函数 Python API，使用户可以轻松地在 Python 中使用 PMX 自定义的 OP 构建需要的模型。

目前，MacaPMX 主要负责 LLM 模型结构的表达。除了 PPL.PMX 提供的开箱即用的 LLM model zoo，用户还可使用 MacaPMX 提供的模型转换工具（从不同社区的模型到 PMX 模型）、Tensor 并行分割工具和 PMX 模型合并工具。

6.3.1.1 安装

MacaPMX 以 wheel 安装包的形式对外发布，建议在 Conda 环境中安装（建议 Python 版本 3.8）。

操作步骤

1. 下载解压相应发行版的 **ppl.llm.serving** 发布包（例如：**mxc500-ppl.llm.serving-2.17.3-1-ubuntu18.04-x86_64.tar.xz**）后，在解压路径**/wheel** 下找到 MacaPMX 对应的 Python 安装包。
2. 执行以下命令，安装 MacaPMX。

```
conda create -n maca_test python=3.8
source activate maca_test
pip install macaPMX-*-py3-none-any.whl
```

wheel 包的依赖库主要有：

- onnx
- torch
- fire
- sentencepiece
- safetensors

其中，torch 应为安装相应的曦云 GPU 软件发布包后得到的版本，其余依赖库会在安装 MacaPMX 时自动安装。

6.3.1.2 转换为 PMX 模型

操作步骤（以 llama-7b 为例）

1. 解压 ppl.llm.serving 发布包后，在解压路径/ai_deb 中可以看到适配后的 ppl.llm.serving 安装包（ppl.llm.serving_*.deb），执行以下命令进行安装。

```
sudo dpkg -i ppl.llm.serving_*.deb
```

2. 安装完成后，在/opt/maca-ai/ppl.llm.serving/samples/samples/model_converter/路径下获取 Llama 系列 PMX 模型转换示例脚本 ConvertWeightToPmx.py。

   根据需要，修改第 4 行导入相应模型的转换函数。当前支持的模型及对应导入代码参见表 6-1。

   表 6-1 MacaPMX 支持的大模型及对应转换函数

LLM Model	ConvertWeightToPmx.py 导入代码
Llama 全系/其它类 Llama 大模型	`from macaPMX.model_zoo.llama.huggingface import write_pmx_model`
chatglm2_6B/chatglm3_6B	`from macaPMX.model_zoo.chatglm2.huggingface import write_pmx_model`
internlm_7B	`from macaPMX.model_zoo.internlm.huggingface import write_pmx_model`
BaiChuan2_7B	`from macaPMX.model_zoo.baichuan.huggingface import write_pmx_model`
Mixtral8x7B	`from macaPMX.model_zoo.mixtral.huggingface import write_pmx_model`
QWen 系列模型	`from macaPMX.model_zoo.qwen.huggingface import write_pmx_model`

说明

Llama 系列对应的 PMX 转换函数，支持的参数与其它模型不同，支持转换 `safetensors` 格式的原始权重，若使用 `use_safetensors` 参数，需要取消示例代码中 16-21 行及 26 行的注释。该参数仅在 Llama 系列模型上生效，其余模型不能设置该参数，否则程序会执行失败。

3. 执行以下命令，将 LLM 原始权重转换为 PMX 模型。

```
python ConvertWeightToPmx.py --input_dir <hf_model_dir> --output_dir <pmx_model_dir> --use_safetensors True
```

参数说明：

`input_dir`：LLM 原始权重模型路径（文件夹）

`output_dir`：输出 PMX 模型目标路径（文件夹）

`use_safetensors`：原始权重文件格式是否为 `safetensors`，仅 Llama 系列支持该参数

6.3.1.3 模型切分

当模型需要在多卡环境上进行并行推理时，需要将生成的 PMX 模型进行切分，以适应多卡并行推理。

操作步骤

1. 在/opt/maca-ai/ppl.llm.serving/samples/samples/model_converter/路径下获取 Llama 系列 PMX 模型切分示例脚本 Split.py。

   根据需要，修改第 2 行导入相应模型的切分函数。当前支持的模型及对应导入代码参见表 6-2。

   表 6-2 MacaPMX 支持的大模型及对应 PMX 切分函数

LLM Model	Split.py 导入代码
Llama/Qwen/其它类 Llama 大模型	`import macaPMX.model_zoo.llama.modeling.SplitModel as SplitModel`
Mixtral8x7B	`import macaPMX.model_zoo.mixtral.modeling.SplitModel as SplitModel`

2. 执行以下命令，进行模型拆分。

```
python Split.py --input_dir <input_directory_path> --num_shards <number_of_shards> --output_dir <output_directory_path>
```

   参数说明：

   `input_dir`：待切分 PMX 模型路径（文件夹）

   `num_shards`：PMX 模型切分份数

   `output_dir`：切分后 PMX 模型目标路径（文件夹）

6.3.1.4 模型合并（可选）

该操作与模型切分相反，将多个切分后的 PMX 模型合并成一个。

操作步骤

1. 在/opt/maca-ai/ppl.llm.serving/samples/samples/model_converter/路径下获取 Llama 系列 PMX 模型合并示例脚本 Merge.py。

   根据需要，修改第 2 行导入相应模型的合并函数。当前支持的模型及对应导入代码参见表 6-3。

   表 6-3 MacaPMX 支持的大模型及对应 PMX 合并函数

LLM Model	Merge.py 导入代码
Llama/Qwen/其它类 Llama 大模型	`import macaPMX.model_zoo.llama.modeling.MergeModel as MergeModel`
Mixtral8x7B	`import macaPMX.model_zoo.mixtral.modeling.MergeModel as MergeModel`

2. 执行以下命令，对切分后 PMX 模型进行合并。

```
python merge.py --input_dir <input_directory_path> --num_shards
<number_of_shards> --output_dir <output_directory_path>
```

参数说明：

input_dir：待合并 PMX 模型路径（文件夹）

num_shards：PMX 子模型的份数

output_dir：合并后 PMX 模型目标路径（文件夹）

6.3.1.5 PMX 模型测试

当生成 PMX 模型后，使用 torch 加载 PMX 模型并执行 LLM 推理，根据输出结果验证 PMX 模型转换的正确性。同时，若设置相应 dump 参数，可以将模型对应 step 的输入、输出保存下来，作为后续本地部署精度验证的输入及输出参考值。

操作步骤

1. 在/opt/maca-ai/ppl.llm.serving/samples/samples/model_converter/路径下获取 Llama 系列 PMX 模型测试示例脚本 Demo.py。

   根据需要，修改第 2 行导入相应模型的 PMX 运行函数。当前支持的模型及对应导入代码见表 6-4。

   表 6-4 MacaPMX 支持的大模型及对应 PMX 模型运行函数

LLM Model	Demo.py 导入代码
Llama 全系/其它类 Llama 大模型	from macaPMX.model_zoo.llama.huggingface import run_demo
chatglm2_6B/chatglm3_6B	from macaPMX.model_zoo.chatglm2.huggingface import run_demo
internlm_7B	from macaPMX.model_zoo.internlm.huggingface import run_demo
BaiChuan2_7B	from macaPMX.model_zoo.baichuan.huggingface import run_7B
Mixtral8x7B	from macaPMX.model_zoo.mixtral.huggingface import run_demo
QWen 系列模型	from macaPMX.model_zoo.qwen.huggingface import run_demo

   说明

   BaiChuan2_7B 对应的 PMX 运行函数与其余模型不同，需要同步修改第 5 行中的函数名。

2. 执行以下命令，验证 PMX 模型精度。

```
OMP_NUM_THREADS=1 torchrun --nproc_per_node $num_gpu Demo.py --ckpt_dir
<llama_dir> --tokenizer_path <llama_tokenizer_dir>/tokenizer.model --
fused_qkv 1 --fused_kvcache 1 --auto_causal 1 --quantized_cache 1 --
dynamic_batching 1 --seqlen_scale_up 1 --max_gen_len 256 --dump_steps
0,1,255 --dump_tensor_path <dump_dir> --batch 1 --cache_layout 3
```

参数说明：

num_gpu：模型推理需要的 GPU 数量

ckpt_dir：PMX 模型路径

tokenizer_path：tokenizer 模型路径

当需要保存测试数据时，设置以下参数：

seqlen_scale_up：输入字节大小的比例因子（序列长度按 8 放大）

max_gen_len：指定生成的最大输出长度（以字节为单位）

dump_steps：保存测试数据的 step，可以指定多个 step，以“,”分隔。若只保存单个 step，必须用“,”结尾，否则会执行失败。

dump_tensor_path：保存测试数据的路径

batch：指定数据处理的批大小

cache_layout：cacheAttention 中 cache 存储 layout，当前仅支持 0 和 3。0：layout 为[MaxT, L, 2, H, Dh]；3：layout 为[L, 2, H, MaxT, Dh]。建议设置为 3，性能更佳。

其余参数与命令中保持一致即可。

说明

cache_layout 的设置需要与 6.3.1.6 导出为 ONNX 模型中的设置一致，否则本地模型部署精度验证会失败。

6.3.1.6 导出为 ONNX 模型

在验证 PMX 模型精度无误后，可以执行最终步骤：将 PMX 导出至 ONNX 模型。

操作步骤

1. 在/opt/maca-ai/ppl.llm.serving/samples/samples/model_converter/路径下获取 Llama 系列 ONNX 模型导出示例脚本 Export.py。

   根据需要，修改第 2 行导入相应模型的 ONNX 导出函数。当前支持的模型及对应导入代码参见表 6-5。

表 6-5 MacaPMX 支持的大模型及对应 ONNX 导出函数

LLM Model	Export.py 导入代码
Llama 全系/其它类 Llama 大模型	from macaPMX.model_zoo.llama.huggingface import run_export
chatglm2_6B/chatglm3_6B	from macaPMX.model_zoo.chatglm2.huggingface import run_export
internlm_7B	from macaPMX.model_zoo.internlm.huggingface import run_export
BaiChuan2_7B	from macaPMX.model_zoo.baichuan.huggingface import export_7B

LLM Model	Export.py 导入代码
Mixtral8x7B	`from macaPMX.model_zoo.mixtral.huggingface import run_export`
QWen 系列模型	`from macaPMX.model_zoo.qwen.huggingface import run_export`

说明

BaiChuan2_7B 对应的 ONNX 导出函数与其余模型不同，需要同步修改第 5 行中的函数名。

2. 执行以下命令，将 PMX 模型导出为 ONNX 模型。

```
OMP_NUM_THREADS=1 torchrun --nproc_per_node $num_gpu Export.py --ckpt_dir
<llama_dir> --tokenizer_path <llama_tokenizer_dir>/tokenizer.model --
fused_qkv 1 --fused_kvcache 1 --auto_causal 1 --quantized_cache 1 --
dynamic_batching 1 --export_path <export_dir> --cache_layout 3
```

参数说明：

`num_gpu`：模型推理需要的 GPU 数量

`ckpt_dir`：PMX 模型路径

`tokenizer_path`：tokenizer 模型路径；chatglm 系列模型不支持此参数，无需设置。

`export_path`：导出 ONNX 模型目标路径

`cache_layout`：cacheAttention 中 cache 存储 layout，当前仅支持 `0` 和 `3`。`0`：layout 为[MaxT, L, 2, H, Dh]；`3`：layout 为[L, 2, H, MaxT, Dh]。建议设置为 `3`，性能更佳。

其余参数与命令中保持一致即可。

6.3.2 本地模型部署精度验证

大多数情况下，大模型会依托服务端部署提供服务端接口供客户端调用，但在服务化部署前，需要依托本地模型部署进行推理验证，以确认模型精度是否符合预期。本章节介绍模型本地部署及精度验证的方法。

本地部署精度验证依赖工具如下：

pplnn_llm：本地部署可执行文件，安装 **ppl.llm.serving_*_amd64.deb** 后在 **/opt/maca-ai/ppl.llm.serving/bin/**下获取。

操作步骤

1. 创建一个测试 bash 脚本 **benchmark.sh**，内容如下：

```
if [ -n "$MPI_LOCALRANKID" ]; then
    MPI_LOCALRANKID=$MPI_LOCALRANKID
elif [ -n "$OMPI_COMM_WORLD_RANK" ]; then
    MPI_LOCALRANKID=$OMPI_COMM_WORLD_RANK
elif [ -n "$PMI_RANK" ]; then
```

```
    MPI_LOCALRANKID=$PMI_RANK
else
    echo "[WARNING] MPI_LOCALRANKID not found, set to 0"
    MPI_LOCALRANKID=0
fi
DEVICE_ID=$MPI_LOCALRANKID

STEP=$1
if [ ! -n "$STEP" ]; then
    STEP=0
fi

MODEL_PATH="/path/to/exported/model/model_slice_${MPI_LOCALRANKID}/model.
onnx"
OUTPUT_DIR="/path/to/output_dir/rank_${MPI_LOCALRANKID}" # we should make
the rank_* directories first
TEST_DATA_DIR="/path/to/dumped/tensor/data/rank_${MPI_LOCALRANKID}"

# we should rearrange the input tensors if the model exporting parameters
has been changed.
TOKEN_IDS=`ls ${TEST_DATA_DIR}/step${STEP}_token_ids-*`
ATTN_MASK=`ls ${TEST_DATA_DIR}/step${STEP}_attn_mask-*`
SEQSTARTS=`ls ${TEST_DATA_DIR}/step${STEP}_seqstarts-*`
KVSTARTS=`ls ${TEST_DATA_DIR}/step${STEP}_kvstarts-*`
CACHESTARTS=`ls ${TEST_DATA_DIR}/step${STEP}_cachestarts-*`
DECODING_BATCHES=`ls ${TEST_DATA_DIR}/step${STEP}_decoding_batches-*`
START_POS=`ls ${TEST_DATA_DIR}/step${STEP}_start_pos-*`
MAX_SEQLEN=`ls ${TEST_DATA_DIR}/step${STEP}_max_seqlen-*`
MAX_KVLEN=`ls ${TEST_DATA_DIR}/step${STEP}_max_kvlen-*`
KV_CAHCE=`ls ${TEST_DATA_DIR}/step${STEP}_kv_cache-*`
KV_SCALE=`ls ${TEST_DATA_DIR}/step${STEP}_kv_scale-*`

TEST_INPUTS="$TOKEN_IDS,$ATTN_MASK,$SEQSTARTS,$KVSTARTS,$CACHESTARTS,$DEC
ODING_BATCHES,$START_POS,$MAX_SEQLEN,$MAX_KVLEN,$KV_CAHCE,$KV_SCALE"
INPUT_DEVICES="device,device,device,device,device,host,device,host,host,d
evice,device"

CMD="/opt/maca-ai/ppl.llm.serving/bin/pplnn_llm --use-llm-cuda \
--onnx-model $MODEL_PATH \
--shaped-input-files $TEST_INPUTS \
--save-outputs \
--device-id $DEVICE_ID \
--save-data-dir $OUTPUT_DIR \
--in-devices $INPUT_DEVICES \
--enable-profiling \
--min-profiling-seconds 3 \
--warmup-iterations 10"

echo "RUN RANK${MPI_LOCALRANKID} STEP${STEP} -> $CMD"
```

```
eval "$CMD"
```

参数说明：

MODEL_PATH：6.3.1.6 导出为 ONNX 模型中导出的 ONNX 模型路径

OUTPUT_DIR：模型输出文件保存路径

TEST_DATA_DIR：模型输入文件保存路径，模型输入数据的操作步骤参见 6.3.1.5 PMX 模型测试

说明

需要将 CMD 中对应 pplnn_llm 设置为安装后放置的路径或者自定义路径，这里默认设置为安装路径。

2. 设置环境变量并运行测试脚本生成输出数据。

```
export MACA_PATH=your_maca_path
export PATH=${MACA_PATH}/bin:${PATH}
export
LD_LIBRARY_PATH=${MACA_PATH}/lib:${MACA_PATH}/mxgpu_llvm/lib:${MACA_PATH}
/ompi/lib:${LD_LIBRARY_PATH}
export USE_GEMM_NN=true
benchmark.sh <STEP> #单卡推理测试命令
mpirun -np 4 benchmark.sh <STEP> #四卡推理测试命令，-np 参数设置与推理卡数一致
```

执行完上述命令后，会在设置的 OUTPUT_DIR 保存对应 step 的输出数据。后续可以根据需求与 6.3.1.5 PMX 模型测试中保存的输出参考数据进行比对。输出数据格式为 float32，使用 np.fromfile(data_path, np.float32)即可加载。

说明

- 执行测试脚本前，需要激活测试环境变量，否则会测试失败。
- <STEP>需要与 6.3.1.5 PMX 模型测试中保存的 step 相对应，否则会导致输入数据加载失败。

6.3.3 本地模型部署性能测试

为了探测特定大模型的极限性能，可以在本地进行性能测试，为后续服务端部署性能做一个参照，进而优化服务端部署参数设置，最大限度发挥出曦云系列 GPU 的硬件性能。

本地部署性能测试依赖工具如下：

benchmark_llama：性能测试可执行文件，安装 ppl.llm.serving_*_amd64.deb 后在/opt/maca-ai/ppl.llm.serving/bin/下获取。当前仅支持 llama 系列及类 llama 模型的测试。

操作步骤（以 llama-7b 为例）

1. 创建一个测试 bash 脚本 benchmark_7b.sh，内容如下：

```
#!/bin/bash

MODEL_TYPE="llama"
MODEL_DIR="/mnt/hpc/shengyunrui/model_card/llama_7b_ppl"
MODEL_PARAM_PATH="/mnt/hpc/shengyunrui/model_card/llama_7b_ppl/params.jso
n"
TENSOR_PARALLEL_SIZE=1
TOP_P=0.0
TOP_K=1
TEMPERATURE=1.0
WARMUP_LOOPS=2
BENCHMARK_LOOPS=2
INPUT_FILE_BASE="tokens_input"


INPUT_LEN=8
GENERATION_LEN=256
BATCH_SIZE_LIST=(1 2 4 8 16 32 64 128 256)

for BATCH_SIZE in ${BATCH_SIZE_LIST[@]}; do
    INPUT_FILE=${INPUT_FILE_BASE}_${INPUT_LEN}

    your_path_of_benchmark_llama \
        --model-type $MODEL_TYPE \
        --model-dir $MODEL_DIR \
        --model-param-path $MODEL_PARAM_PATH \
        --tensor-parallel-size $TENSOR_PARALLEL_SIZE \
        --top-p $TOP_P \
        --top-k $TOP_K \
        --temperature $TEMPERATURE \
        --warmup-loops $WARMUP_LOOPS \
        --generation-len $GENERATION_LEN \
        --benchmark-loops $BENCHMARK_LOOPS \
        --input-file $INPUT_FILE \
        --batch-size $BATCH_SIZE

done
```

参数说明：

`MODEL_TYPE`：LLM 模型类型，当前仅支持 Llama

`MODEL_DIR`：6.3.1.6 导出为 ONNX 模型中导出的 ONNX 模型路径

`MODEL_PARAM_PATH`：6.3.1.6 导出为 ONNX 模型中导出 ONNX 模型的 **params.json** 文件路径

`TENSOR_PARALLEL_SIZE`：Tensor 并行大小，需要与 6.3.1.3 模型切分中模型切分数量保持一致，即导出模型为 2 卡并行，此处设置为 2；4 卡并行则设置为 4

`WARMUP_LOOPS`：warmup 执行次数，在正式测试前执行该次数的 warmup

BENCHMARK_LOOPS：性能测试执行次数

INPUT_FILE_BASE：模型输入文件基础文件名，默认为 **tokens_input**，测试输入的文件名称为 **${INPUT_FILE_BASE}_${INPUT_LEN}**，即通过基础文件名和输出长度拼接。例如，输入长度为 8 的输入文件名称为 **tokens_input_8**

INPUT_LEN：模型输入 token 长度

GENERATION_LEN：模型生成 token 长度

BATCH_SIZE_LIST：性能测试 batchsize 列表，测试会遍历该列表中所有设置

输入文件为文本文件，每一行代表一条输入 token，各 token 之间用空格分隔，以 INPUT_LEN=8 为例，该文件的构成如下。篇幅限制，这里只展示前 8 条输入 token。该文件的行数（lines_of_token）必须大于等于测试设置的最大 batchsize，否则该 batchsize 设置无效，测试程序只能执行最大 batchsize = lines_of_token 的测试，详细可参考相关文件。

```
1, 306, 4658, 278, 6593, 310, 2834, 338
1, 306, 4658, 278, 6593, 310, 2834, 338
1, 306, 4658, 278, 6593, 310, 2834, 338
1, 306, 4658, 278, 6593, 310, 2834, 338
1, 306, 4658, 278, 6593, 310, 2834, 338
1, 306, 4658, 278, 6593, 310, 2834, 338
1, 306, 4658, 278, 6593, 310, 2834, 338
1, 306, 4658, 278, 6593, 310, 2834, 338
```

2. 设置环境变量并运行测试脚本生成输出数据。

```
export MACA_PATH=your_maca_path
export PATH=${MACA_PATH}/bin:${PATH}
export
LD_LIBRARY_PATH=${MACA_PATH}/lib:${MACA_PATH}/mxgpu_llvm/lib:${LD_LIBRARY
_PATH}
export USE_GEMM_NN=true
benchmark_7b.sh
```

执行完上述命令后，会输出对应设置下的性能测试数据，如下所示：

```
CSV format header:prefill(ms),decode(ms),avg(ms),tps(ms),mem(gib)
CSV format output:40.16,14.2576,14.4599,69.1567,14.3461
```

分别对应 prefill 耗时、平均 decode 延迟、平均 step 延迟、tokens 吞吐（tokens per second）以及显存使用。

说明

执行测试脚本前，需要激活测试环境变量，否则会测试失败。

6.3.4 服务化部署

大模型服务化部署是一种将大模型应用于实际场景的有效方式。它具有以下优势：

- 大模型服务化部署能够提供更高的性能和更准确的结果。由于大模型具备更深层次的理解和更复杂的参数配置，其在各种任务上表现更出色。通过将大模型部署为服务，可以利用其强大的计算能力和学习能力，为用户提供更高质量的预测、推荐和决策。
- 大模型服务化部署可以实现灵活的扩展性和定制化需求。采用服务化架构，可以将模型与其他组件解耦，使得系统更易于扩展和维护。同时，根据不同业务需求，可以对大模型进行个性化的调整和优化，以满足特定任务的要求。
- 大模型服务化部署还能提升数据安全和隐私保护。通过将模型部署在云端或私有环境中，可以有效保护敏感数据的安全性，避免将数据传输到公共网络或设备中。这种集中化的方式可以通过严格的权限控制和加密技术来保护用户数据的隐私。

总的来说，大模型服务化部署具有性能优越、灵活扩展和安全保护等诸多优势，可为各行各业提供更高水平的智能服务。

本节将从服务端部署、C++客户端部署、Python 客户端部署三个部分介绍服务化部署。

6.3.4.1 服务端部署

服务端部署依赖文件如下：

- ppl_llm_server：服务端部署可执行文件，安装 ppl.llm.serving_*_amd64.deb 后在/opt/maca-ai/ppl.llm.serving/bin/下获取。
- xxx_config.json：特定大模型配置文件，ppl_llm_server 通过解析该配置文件加载并运行对应的模型。

操作步骤（以 llama-7b 为例）

1. 创建 llama_7b_config.json，主要内容如下：

```
{
    "model_type": "llama",
    "model_dir":  "/path/to/model/dir",
    "model_param_path": "/path/to/model/dir/params.json",

    "tokenizer_path": "/path/to/tokenizer/tokenizer.model",

    "tensor_parallel_size": 1,

    "top_p": 0.0,
    "top_k": 1,

    "quant_method": "none",

    "max_tokens_scale": 0.6,
    "max_tokens_per_request": 4096,
    "max_running_batch": 1024,
```

```
    "max_tokens_per_step": 8192,

    "host": "0.0.0.0",
    "port": 23333
}
```

参数说明：

model_type：LLM 模型类型，可参考/opt/maca-ai/ppl.llm.serving/samples/samples/ppl_server_client/model_config 中已支持模型的配置示例进行配置

model_dir：6.3.1.6 导出为 ONNX 模型中导出的 ONNX 模型路径

model_param_path：6.3.1.6 导出为 ONNX 模型中导出 ONNX 模型的 params.json 文件路径

tokenizer_path：tokenizer model 路径

tensor_parallel_size：tensor 并行大小，需要与 6.3.1.3 模型切分中模型切分数量保持一致，即导出模型为 2 卡并行，此处设置为 2；4 卡并行则设置为 4

max_tokens_scale：设置范围[0.1，0.9]，该参数会影响服务端对显存的使用。简单的说，代表模型加载完成后，额外占用剩余显存的比例，受制于当前显存碎片化管理不够完善，随着请求次数增加，显存会出现溢出，建议设置一个较小的值。若测试大 batchsize，建议设置大一些（0.9），否则由于预分配显存不够，无法按照设定的 batchsize 执行大模型推理，但此时请求的次数要比较少，否则也会出现显存溢出

max_tokens_per_request：单次请求的最大 token 数，建议设置 4096

max_running_batch：执行推理最大 batchsize

max_tokens_per_step：单个 step 处理最大 token 数

2. 设置环境变量并运行服务端程序，启动 llama_7b 服务。

```
export MACA_PATH=your_maca_path
export PATH=${MACA_PATH}/bin:${PATH}
export
LD_LIBRARY_PATH=${MACA_PATH}/lib:${MACA_PATH}/mxgpu_llvm/lib:${LD_LIBRARY
_PATH}
export USE_GEMM_NN=true
export SHOW_PREFILL=true #若需要输出 prefill 性能统计数据，设置该环境变量
./ppl_llm_server llama_7b_config.json
```

当输出下列日志时，表示 llama_7b 服务启动完成：

```
[INFO][2023-12-07 16:05:29.304][llama_worker.cc:962] waiting for
request ...
```

6.3.4.2 C++ 客户端

C++ sample code 获取方式如下：在安装 ppl.llm.serving_*_amd64.deb 后，可在/opt/maca-ai/ppl.

llm.serving/samples/samples/ppl_server_client/cpp_client/路径查看相应代码示例。安装路径中有该代码示例已编译生成的可执行文件：/opt/maca-ai/ppl.llm.serving/bin/cpp_client。工程文件目录如图 6-2 所示：

- cmake：文件夹，包含三方依赖库 cmake 配置。
- CMakeLists.txt：CMake 配置文件。
- proto：文件夹，包含与服务端交互时 grpc 依赖的 proto 文件。
- src：文件夹，包含 client_sample.cc，是 C++客户端的简单使用示例程序。

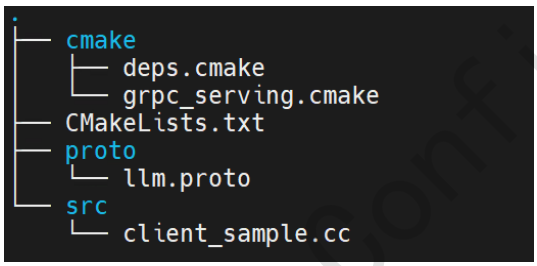

图 6-2 C++客户端目录结构图

参考 client_sample.cc 示例代码，根据业务需求定制相应的 C++客户端即可。

编译时需要将 cmake/deps.cmake 中 37-39 行、41-43 行中 grpc、absl 依赖库地址修改为：

```
hpcc_declare_git_dep(grpc
    https://github.com/grpc/grpc.git
    v1.56.2)
hpcc_declare_git_dep(absl
    https://github.com/abseil/abseil-cpp.git
    lts_2023_01_25)
```

6.3.4.3 Python 客户端

Python sample code 获取方式如下：在安装 ppl.llm.serving_*_amd64.deb 后，可在/opt/maca-ai/ppl.llm.serving/samples/samples/ppl_server_client/python_client/路径查看相应代码示例。工程文件目录如图 6-3 所示：

- llm_pb2.py：与服务端交互时 grpc 依赖 proto 定义文件。
- llm_pb2_grpc.py：与服务端交互时 grpc 依赖 proto 定义文件。
- client.py：Python 客户端示例脚本。

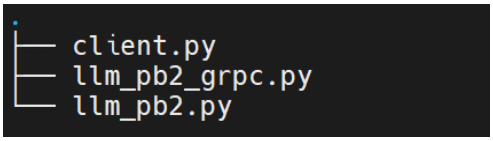

图 6-3 Python 客户端目录结构图

参考 **client.py** 示例代码，根据业务需求定制相应的 Python 客户端即可。

python 客户端依赖 grpc，使用时需安装 grpcio、grpcio-tools 两个依赖项。

```
pip install grpcio
pip install grpcio-tools
```

6.3.4.4 服务端输出性能数据解析

服务端完成部署，客户端成功发送请求，服务端完成请求的处理并将结果发送至客户端后，服务端会输出当前部署大模型的性能数据。示例如图 6-4 所示：

```
[PERF] --- step 1 ------------------------------------------------
[PERF]  |- memory usage: (63.59 - 44.43) -> 19.16 GiB
[PERF]  |- kv cache usage: 1.65 %
[PERF]  |- pending task number: 3
[PERF]  |- running batch: 2, max running batch: 2
[PERF]  |- finished query count: 0, QPS: 0.00
[PERF]  |- gen token count: 2, avg gen len: 0.00, TPS: 3.64
[PERF]  |- pipeline          | cur: 549.33 ms, | avg: 549.33 ms, | total: 549.33 ms
[PERF]  |-- batching         | cur: 0.02 ms, | avg: 0.02 ms, | total: 0.02 ms
[PERF]  |-- copy inputs      | cur: 0.53 ms, | avg: 0.53 ms, | total: 0.53 ms
[PERF]  |-- model inference  | cur: 547.81 ms, | avg: 547.81 ms, | total: 547.81 ms
[PERF]  |-- sampling         | cur: 0.89 ms, | avg: 0.89 ms, | total: 0.89 ms
[PERF]  |-- send response    | cur: 0.08 ms, | avg: 0.08 ms, | total: 0.08 ms
[PERF]  |-- early finish     | cur: 0.00 ms, | avg: 0.00 ms, | total: 0.00 ms
[PERF]  |- schedule cost: 0.28 %
[PERF] --- step 100 ----------------------------------------------
[PERF]  |- memory usage: (63.59 - 44.43) -> 19.16 GiB
[PERF]  |- kv cache usage: 4.22 %
[PERF]  |- pending task number: 0
[PERF]  |- running batch: 5, max running batch: 5
[PERF]  |- finished query count: 0, QPS: 0.00
[PERF]  |- gen token count: 497, avg gen len: 0.00, TPS: 266.31
[PERF]  |- pipeline          | cur: 13.29 ms, | avg: 18.66 ms, | total: 1866.22 ms
[PERF]  |-- batching         | cur: 0.00 ms, | avg: 0.00 ms, | total: 0.04 ms
[PERF]  |-- copy inputs      | cur: 0.14 ms, | avg: 0.13 ms, | total: 12.90 ms
[PERF]  |-- model inference  | cur: 12.96 ms, | avg: 18.38 ms, | total: 1837.81 ms
[PERF]  |-- sampling         | cur: 0.08 ms, | avg: 0.09 ms, | total: 9.05 ms
[PERF]  |-- send response    | cur: 0.10 ms, | avg: 0.06 ms, | total: 6.16 ms
[PERF]  |-- early finish     | cur: 0.00 ms, | avg: 0.00 ms, | total: 0.00 ms
[PERF]  |- schedule cost: 1.52 %
```

图 6-4 服务端输出性能数据示例（llama-7b）

若在启动服务前设置 SHOW_PREFILL 环境变量，服务端会输出 prefill 阶段的性能数据，即图 6-4 中 step1 的性能数据；若不设置，则输出 100 整数倍及最后一个 step 的性能数据。

重点性能参数包括：

memory usage：当前 step 显存使用情况

running batch：当前 step 正在推理的 batchsize

max running batch：从当前 step 回溯的历史最大推理 batchsize

finished query count：当前 step 已经结束生成的请求数量

pipeline：整体流程耗时，包含当前 step 耗时，平均耗时及总耗时

model inference：大模型推理的耗时，包含当前 step 耗时，平均耗时及总耗时

7. MacaRT-vLLM

7.1 MacaRT-vLLM 介绍

MacaRT-vLLM 是在曦云系列 GPU 上适配官方 vLLM 的推理工具，基于 Maca 后端对 vLLM 方法进行了兼容适配和 Kernel 优化。使用 MacaRT-vLLM 在曦云系列 GPU 上进行大模型推理，其方法和功能与官方 vLLM 兼容。当前兼容版本为 vLLM-0.4.0。

7.2 MacaRT-vLLM 功能与局限性

MacaRT-vLLM 兼容适配了 vLLM-0.4.0，包含了以下功能和特性，以及局限性：

功能和特性：

- 兼容 vLLM-0.4.0 支持的所有模型 FLOAT16 和 BFLOAT16 推理
- 兼容原生 LLM、Engine、Kernel 的 API 接口
- 兼容原生 server 和 OpenAI server 接口

局限性：

- 当前暂不支持 Lora，后续完善支持
- 支持 GPTQ 量化方式，暂不支持其他量化方式
- 暂不支持 `enforce_eager=False` 方式，内部关闭
- 暂不支持 FP8 类型的 KV Cache
- 当前仅包含 Ubuntu 20 系统版本，后续完善支持其他系统

7.3 MacaRT-vLLM 使用流程

本章节介绍 MacaRT-vLLM 的使用步骤，主要分为离线推理、吞吐测试和 Sever 服务。

7.3.1 环境准备

使用 MacaRT-vLLM 进行推理需要准备好环境。目前提供两种方式：镜像和 wheel 包。

7.3.1.1 使用 vLLM 镜像

从发布的软件包中获取 vLLM 镜像并启动，参见《曦云®系列通用计算 GPU 用户指南》中“容器相关场景支持”章节。

7.3.1.2 安装 wheel 包

操作步骤

1. 预先安装沐曦平台的 torch、flash attn、xformer（可选）环境。
2. 获取 MacaRT-vLLM 压缩包 **mxc500-vllm-XXXX-ubuntu20.04-x86_64.tar.xz** 并解压。
3. pip 安装以下 wheel 包：
   - **vllm-0.4.0+cu116-cp38-cp38_${OS_Version}.whl**
   - **ray-2.9.3-cp38-cp38_${OS_Version}.whl**

除了上述 wheel 包，其他依赖环境可以通过外部镜像源进行安装。安装过程中有其他依赖，需要提前配置好 Python 的 pip 源。

7.3.2 离线推理

离线推理代码示例如下：

```
from vllm import LLM, SamplingParams
# Sample prompts.
prompts = [
    "Hello, my name is",
    "The president of the United States is",
    "The capital of France is",
    "The future of AI is",
]
# Create a sampling params object.
sampling_params = SamplingParams(temperature=0.8, top_p=0.95)
# Create an LLM.
llm = LLM(model="facebook/opt-125m", tensor_parallel_size=1, dtype="float16",
max_model_len=2048)
# Generate texts from the prompts. The output is a list of RequestOutput
objects
# that contain the prompt, generated text, and other information.
outputs = llm.generate(prompts, sampling_params)
# Print the outputs.
for output in outputs:
    prompt = output.prompt
    generated_text = output.outputs[0].text
    print(f"Prompt: {prompt!r}, Generated text: {generated_text!r}")
```

其中主要配置为：

- LLM 初始化类：指定模型路径（可配置为本地路径，如果不是本地路径，会根据网络下载外网模型）、tensor 切分数量、模型数据类型（可不配置，如果没指定，将通过 `model config` 读取）、模型最大处理长度（可不配置，如果没设置，将从 `model config` 读取）
- `SamplingParams`：设置采样算法方式。参数配置可参照官方 vLLM-0.4.0 版本进行配置。

7.3.3 吞吐测试

吞吐测试代码可参考官方 vLLm-0.4.0 benchmark_throughput.py。

吞吐测试代码的运行方式示例 1：

```
python benchmark_throughput.py --model XXX --tensor-parallel-size 1 --num-prompts 8 --trust-remote-code --input-len 1024 --output-len 512
```

模型为 XXX，tensor 并行为 1，8 条请求数量，信任模型路径的 tokenizer 方式，每条输入长度为 1024 个 token，输出为 512 个 token。

吞吐测试代码的运行方式示例 2：

```
python benchmark_throughput.py --model XXX --tensor-parallel-size 1 --num-prompts 8 --trust-remote-code --dataset XXX.json
```

模型为 XXX，tensor 并行为 1，8 条请求数量，信任模型路径的 tokenizer 方式，通过从 dataset 选取语料进行真实模拟。json 格式可参考 ShareGPT_V3_unfiltered_cleaned_split.json。

运行程序后，结果打印如下：

```
Throughout: XXX request/s, XXXX token/s
```

打印结果指示每秒能接受多少请求，以及每秒处理的 token 数量（包括输入 token）。

7.3.4 Server 服务

Server 请求服务参数设置可参考官方 vLLM-0.4.0 版本：

API sever 参数配置可参考 api_server.py。

OpenAI sever 参数配置可参考 api_server.py。

参考示例如下：

```
python -m vllm.entrypoints.api_server --model XXX #普通 server 方式
python -m vllm.entrypoints.openai.api_server --model XXX --host localhost --chat-template XXX.jinja ## openai 方式的请求
```

8. 附录

8.1 术语/缩略语

术语/缩略语	全称	说明
Batch		全部样本里的一批数据
BFC	Best-Fit with Coalescing	一种内存管理策略
CpuEP	CPU Execution Provider	以 CPU 作为 ONNX Runtime 的后端进行模型推理
LLM	Large Language Model	大语言模型
MacaConverter		沐曦研发，将训练的模型转换为 ONNX 模型的工具
MacaPrecision		沐曦研发，精度对比工具
MacaQuantizer		沐曦研发，模型量化工具
MacaEP	MXMACA Execution Provider	以曦云系列 GPU 作为 ONNX Runtime 的后端进行模型推理
MacaRT	MXMACA Runtime	沐曦研发，曦云系列 GPU 的推理引擎
ONNX	Open Neural Network Exchange	开放神经网络交换，表示深度学习模型的开放格式，可将训练好的模型存储为此格式
ONNX Runtime		一个开源的跨平台推理框架
OpenPPL-LLM		OpenPPL 推出的大语言模型（LLM）推理引擎
PMX	PPL Model Exchage	OpenPPL 模型转换工具
Tensor		张量，是一种特殊的数据结构

声明

本文档中呈现的信息属于沐曦集成电路（上海）有限公司和/或其附属公司（以下统称为“沐曦”），非经沐曦事先书面许可，任何实体或个人均不得获得本文档的副本，且无权以任何方式处理本文档，包括但不限于使用、复制、修改、合并、出版、发行、销售或传播本文档的部分或全部。

本文档内容仅供参考，不提供任何形式的、明示或暗示的保证，包括但不限于对适销性、适用于任何目的和/或不侵权的保证。在任何情况下，沐曦均不对因本文档引起的、由本文档造成的、或与之相关的任何索赔、损害或其他责任负责。

沐曦保留自行决定随时更改、修改、添加或删除本文档的部分或全部的权利。沐曦保留最终解释权。

沐曦、MetaX 和其他沐曦图标是沐曦的商标。本文档中提及的所有其他商标和商品名称均为其各自所有者的财产。