寒武纪思元MLU270测试

> MLU：Machine Learning Unit，MLU。

一、寒武纪加速库：

1、寒武纪机器学习库（Cambricon Machine Learning Library，CNML）：

是一个基于寒武纪机器学习单元（Machine Learning Unit，MLU）并针对机器学习以及深度学习的编程库。CNML 为用户提供简洁、高效、通用、灵活并且可扩展的编程接口，用于在 MLU 上加速用户各种机器学习和深度学习算法。CNML 提供了丰富的基本算子。通过组合基本算子可以实现多样的机器学习和深度学习算法。

Cambricon 机器学习库 CNML 提供了一套高效、通用、灵活、可扩展的编程接口，用于在 MLU 上加速各种机器学习和深度学习算法。CNML 主要包含以下几个特性：

支持丰富的基本算子。

– 常见神经网络算子：

* 卷积、反卷积；

* 池化；

* 激活算子，如 ReLU、Sigmoid、TANH 等；

*Local Response Normalization（LRN）、批规范化；

*Softmax；

* 全连接；

– 矩阵、向量、标量算子：

* 矩阵乘；

* 张量加和减；

* 张量逻辑运算；

* 张量变换，如 Crop、Reshape、Slice、Concat 等；

– 循环神经网络算子：

*Long Short-Term Memory（LSTM）；

*BasicRNNPro、RNN；

2、CNPlugin：

在 CNML（Cambricon Machine Learning Library 寒武纪机器学习库）层提供一个接口，将 BANG C 语言生成的算子与 CNML 的执行逻辑统一起来。BANG C 语言的 MLU 异构编程解决方案运行在 CNRT(Cambricon Runtime Library，寒武纪运行时库) 之上。因此为了让 BANG C 实现的算子更好

的与 CNML 协同工作，提供了插件算子 CNPlugin，实现了用 BANG C 语言对 CNML 的操作进行扩展。除此之外，还可以支持 CNML 的特性及多种运行模式，如在线、离线、逐层、融合等。

3、寒武纪 CNNL（寒武纪神经网络计算库）：

CNNL 是一个基于寒武纪机器学习单元（Machine Learning Unit，MLU）并针对 DNN（深度神经网络）的计算库。CNNL 针对 DNN 应用场景，提供了高度优化的常用算子，同时也为用户提供简洁、高效、通用、灵活并且可扩展的编程接口。

4、CNCL（Cambricon Communications Library，寒武纪通信库）：

CNCL 是面向 MLU（Machine Learning Unit）设计的高性能通信库：

- CNCL 帮助应用开发者优化了基于 MLU 进行多机多卡的集合通信（Collective）操作；
- CNCL 支持多种 MLU 处理芯片的互联技术，包括 PCIe、Interlaken、RoCE、Infiniband Verbs 以及Sockets；
- CNCL 能够根据芯片的互联拓扑关系，自动的选择最优的通信算法和数据传输路径，从而最大化利用传输带宽完成不同的通信操作；

二、寒武纪AI框架Cambricon_PyTorch

当前的 Cambricon PyTorch 适配的**原生 PyTorch 版本为 v1.3.0** 。

对原生PyTorch做了进一步扩展，主要修改有：添加MLU设备、实现MLU特有的在线融合模式、部分算子的分发方式，torchision访问权限和Cache拓展包。

为支持寒武纪 MLU（Machine Learning Unit，机器学习处理器），寒武纪定制了开源深度学习编程框架 PyTorch（以下简称 Cambricon PyTorch）。

Cambricon PyTorch 借助 PyTorch 自身提供的设备扩展接口将 MLU 后端库中所包含的算子操作动态注册到 PyTorch 中，MLU 后端库可处理 MLU 上的张量和神经网络算子的运算。Cambricon PyTorch 会基于 CNML 库在 MLU 后端实现一些常用神经网络算子，并完成一些数据拷贝操作。

Cambricon PyTorch 兼容原生 PyTorch 的 Python 编程接口和原生 PyTorch 网络模型，支持在线逐层、在线融合和离线三种方式执行推理，同时还支持在线逐层训练。网络的权重可以从 pth 格式文件中读取，已支持的分类和检测网络结构由 torchvision 管理，可以从 torchvision 中读取。对于推理任务，Cambricon PyTorch 不仅支持 float16、float32 等网络模型，而且在寒武纪机器学习处理器上能高效地支持 int8 和 int16 网络模型。对于训练任务，支持 float32 及自适应量化网络模型。

三、寒武纪CNToolKit

1、Compilers

编译类组件主要支持 BANG C 和 BANG C++ 编程语言的编译，生成可以和 Host 端目标平台 obj 文件链接的对象文件并进行优化，还提供丰富的编译选项配合调试和性能分析工具使用。

1.1 CNCC

CNCC（Cambricon Compiler Collection，寒武纪 BANG C 语言编译器），是基于 Clang 和 LLVM 开发的 BANG C 和 BANG C++ 的编译器主驱动程序，负责将 *.mlu 的 C 或 C++ 源码文件编译为 *.s 的 MLISA 汇编文件。

1.2 CNAS

CNAS（Cambricon Assembler，寒武纪 MLISA 语言编译器），负责将 *.s 的 MLISA 汇编文件编译为 *.cncode （REL-Device 端可重定位）或 *.cnbin （EXE-Device 端链接后）或 *.cnfatbin （集合多个 cnbin 的胖二进制）或 *.o （可与 Host 端目标平台链接的）的 ELF 格式的 obj 文件。

2、Tools

工具类组件主要帮助用户解决开发中遇到的功能和性能调试问题。

2.1 CNGDB

CNGDB（Cambricon GNU Debugger，寒武纪 BANG 语言调试工具）基于 GDB 修改，添加了对 BANG C 和 C++ 语言的支持，可同时调试 Host 端和 Device 端的代码，并遵循 GPL 协议。目前已开放源代码。源代码信息，参见 https://github.com/Cambricon/CN-GDB 。

2.2 CNPerf

CNPerf（Cambricon Performance，寒武纪性能剖析工具）是一款针对寒武纪软件栈全栈设计的性能剖析工具。以性能事件为基础，可对 MLU 异构并行编程查找性能瓶颈和热点函数。

3、Libraries

运行库类组件主要为用户提供部署环境和开发环境中的运行时支持。各个组件可以定制化独立安装在部署环境或开发环境中，但需要注意各组件之间的版本依赖，以及开发部署两种环境的版本要尽量保持一致。

3.1 CNCodec

CNCodec（Cambricon Codec Library，寒武纪硬件编解码库），为带有视频或图片编解码能力的 MLU 设备提供 Host 端的 API 支持。

3.2 CNDev

CNDev（Cambricon Device Interface Library，寒武纪设备接口库），主要为上层库或工具提供公共统一的获取硬件设备信息的 API。

3.3 CNDrv

CNDrv（Cambricon Driver Interface Library，寒武纪驱动接口库），主要为上层库或工具提供异构编程中 Host 和 Device 内存管理、异步执行和控制、设备管理、多卡多机协同等能力。

3.4 CNRT

CNRT（Cambricon Runtime Library，寒武纪运行时库）提供和 CNDrv 类似的 Host 和 Device内存管理、异步执行和控制、设备管理、多卡多机协同等功能 API。二者差异在于 CNRT 将 CNDrv 若干API 做了流程上或逻辑上的友好封装，帮助用户降低编程复杂度。

寒武纪软件栈中在 CNToolkit 之上还有很多高层级的库或框架。例如 CNML（Cambricon Machine Learning Library，寒武纪机器学习库）是一个面向推理场景的引擎算子库，它会依赖 CNRT 提供一些离线模型的解析和执行能力。

3.5 CNRTC

CNRTC（Cambricon Runtime Compilation Library，寒武纪运行时编译库）, 它是 BANG C/C++ 的运行时编译库，通过 API 接收字符串形式的 BANG 语言源码，并创建可在多架构 MLU 上执行的 cnfatbin，然后配合 cndrv 层 Module 解析和 Kernel 执行接口实现 JIT 方式的编译执行。

这个库多用于 AI 框架中的 Runtime 层，用来实现框架级编译器的 JIT 功能。

3.6 CNStudio

CNStudio（Cambricon Studio，寒武纪集成开发环境）为用户开发和调试 BANG C/C++ 语言提供 IDE 级的环境。当前 CNStudio 只提供了 Visual Studio Code 的插件，帮助用户在 VSCode 中快速开发、构建、调试 BANG C/C++ 的工程。

3.7 CNJPU

CNJPU（Cambricon Edge JPEG Processing Unit Library，寒武纪 JPEG 处理单元库）为 Edge 形态的用户提供调用 JPEG 处理单元的运行时支持，在 CNCodec 的编程中会使用或链接此库。

此库只支持 aarch64 平台，提供给 Edge 形态的产品使用，用法为拷贝动态库至 Edge 操作系统对应的目录，或拷贝至 x86_64 的主机测进行交叉编译和链接使用。

3.8 CNION

CNION（Cambricon Edge ION Library，寒武纪 ION 库）为 Edge 形态的用户提供调用 ION 用户态支持，此库是基于 AOSP ION 添加了寒武纪相关函数功能的 ION 库，在 CNCodec 的编程中会使用或链接此库。

此库只支持 aarch64 平台，提供给 Edge 形态的产品使用，用法为拷贝动态库至 Edge 操作系统对应的目录，或拷贝至 x86_64 的主机测进行交叉编译和链接使用。

四、测试整套流程：

环境准备：这部分是后期加的，因为考虑到后人还要用，所以标号可能比较混乱。

（1）pytorch要使用图下面第一个容器，我把这个容器外放了端口，用于网络通信，其他的也都能用，如果使用tensorflow另外研究一下。

```
ict@ict-Precision-3630-Tower:~$ su root
密码:
root@ict-Precision-3630-Tower:/home/ict# docker ps -a
CONTAINER ID   IMAGE                                                          COMMAND       CREATED
STATUS                  PORTS     NAMES
933a2b8c520a   yellow.hub.cambricon.com/pytorch/pytorch:0.15.0-ubuntu16.04_3             "/bin/bash"   12 days ago
Exited (0) 2 days ago             dev_cn
a84d9ef269f9   yellow.hub.cambricon.com/tensorflow/tensorflow:1.4.0-tf1-ubuntu16.04-py2  "/bin/bash"   2 months ago
Exited (0) 2 months ago           keen_franklin
ef3bbec19207   yellow.hub.cambricon.com/tensorflow/tensorflow:1.4.0-tf1-ubuntu16.04-py3  "/bin/bash"   2 months ago
Exited (0) 2 months ago           ecstatic_wozniak
c2e0e359e31b   yellow.hub.cambricon.com/pytorch/pytorch:0.15.0-ubuntu16.04               "/bin/bash"   2 months ago
Exited (0) 12 days ago            pedantic_cannon
```

（2）利用容器的ID启动容器，并且进入他给分配的终端，就是root@933a2b8c520a。

```
root@ict-Precision-3630-Tower:/home/ict# docker start 933a2b8c520a
933a2b8c520a
root@ict-Precision-3630-Tower:/home/ict# docker exec -it 933a2b8c520a /bin/bash
root@933a2b8c520a:/# ls
bin   dep_libs_extract  etc   lib    lib64   media  opt   root  sbin  sys  torch  var
boot  dev               home  lib32  libx32  mnt    proc  run   srv   tmp  usr
```

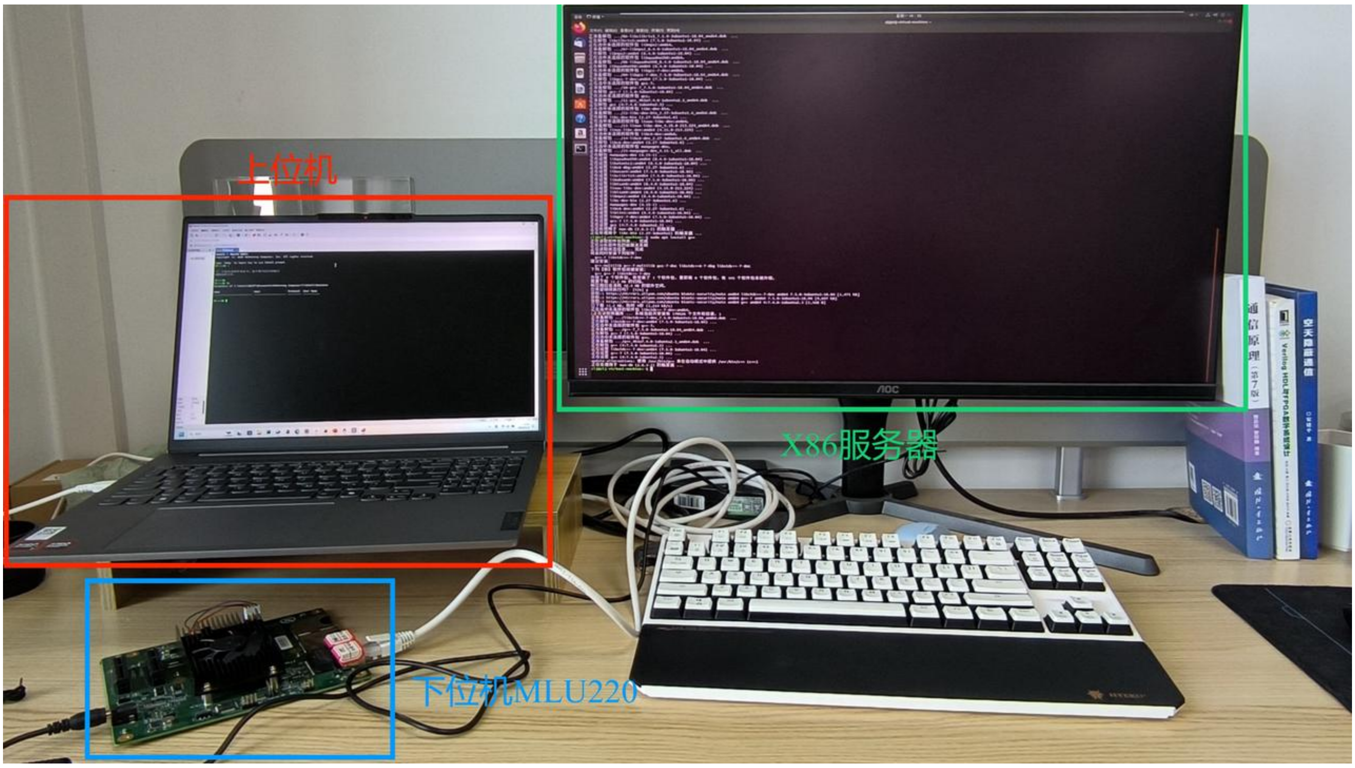
（3）进入pytorch环境，运行：

```
source /torch/venv3/pytorch/bin/activate
```

```
root@933a2b8c520a:/torch/venv3/pytorch/bin# source activate
(pytorch) root@933a2b8c520a:/torch/venv3/pytorch/bin# cd ..
```

可以看到前边多了一个（pytorch）。

> 开发、训练环境：
>
> PyTorch:1.12.1
>
> CUDA: 12.3
>
> Python:3.9



1、在x86服务器 + MLU270上的docker容器里，进行CPU模式的推理、量化、MLU逐层推理，MLU融合推理，并生成MLU270的离线模型。（参考文档：寒武纪PyTorch用户手册Cambricon-PyTorch-User-Guide-CN-v0.15.0.pdf 存放位置：cambricon/1.7.0/PyTorch/Cambricon-PyTorch-User-Guide-CN-v0.15.0.pdf）

导入必要模块：

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch_mlu.core.mlu_quantize as mlu_quantize
import torch_mlu.core.mlu_model as ct
```

1.1 将训练好后的模型保存为.pth权重文件，然后拷贝到x86服务器 + MLU270容器内。

> **注意：PyTorch 1.6.0后的版本中torch.save默认使用zip文件保存权重.pth，因此PyTorch 1.5及以下无法直接加载，会报错。**
>
> ```
> RuntimeError: ./checkpoint_epoch50.pth is a zip archive (did you mean to use torch.jit.load()?)
> ```
>
> 解决方法：最好直接使用与Cambricon_PyTorch适配的**原生PyTorch 1.3.0**，或者在PyTorch 1.6.0及以上版本中修改参数_use_new_zipfile_serialization=False，代码如下（重新加载权重，并修改权重文件保存的格式）：

```
#在torch 1.6以及更高版本版本中重新加载一下网络参数
policy_net = policy_net().to(device) #实例化模型并加载到cpu货GPU中
policy_net.load_state_dict(torch.load(model_cp_pth))  #加载模型参数，model_cp为之前训练好的模型参数（zip格式）
#重新保存网络参数，此时注意改为非zip格式
torch.save(policy_net.state_dict(), model_cp_pth, _use_new_zipfile_serialization=False)
```

1.2 CPU模式模型量化

使用模型量化工具对模型权，并保存量化后的权重，此处命名为 `policyNet_cp_quantization.pth`，代码如下：

```
# CPU模式模型量化
checkpoint = torch.load('./cp_epoch50.pth', map_location='cpu')
# 加载
policy_net.load_state_dict(checkpoint)
# 为了加速推理，量化为int8、int16
net_quantization = mlu_quantize.quantize_dynamic_mlu(policy_net, dtype='int8', gen_quant=True)
# 保存量化模型
torch.save(net_quantization.state_dict(), 'policyNet_cp_quantization.pth')
```

1.3 MLU逐层推理

MLU逐层推理，代码如下：

```
# step 1 使用quantize_dynamic_mlu接口替换网络中需要量化的算子为Cambricon PyTorch对应的自定义算子
net_quantization = mlu_quantize.quantize_dynamic_mlu(policy_net)
# step 2 加载量化权重
net_quantization.load_state_dict(torch.load('policyNet_cp_quantization.pth'), False)
# step 3 模型和数据拷贝到MLU上
net_quantization_mlu = net_quantization.to(ct.mlu_device())
input_h_mlu = input_h.to(ct.mlu_device())
input_o_mlu = input_o.to(ct.mlu_device())
# step 4 逐层推理
x, h = net_quantization_mlu(input_o_mlu, input_h_mlu)
print(type(x), type(h)) #<class 'torch.Tensor'>
print(x.cpu())  #放回当CPU输出
```

结果：出现[warning]。

```
[WARNING][/pytorch/catch/torch_mlu/csrc/aten/operators/op_methods.cpp][line:173][addmm][thread:140419487147776][process:2959]:

addmm Op cannot run on MLU device, start running on CPU!
[WARNING][/pytorch/catch/torch_mlu/csrc/aten/operators/op_methods.cpp][line:173][addmm][thread:140419487147776][process:2959]:

addmm Op cannot run on MLU device, start running on CPU!
[WARNING][/pytorch/catch/torch_mlu/csrc/aten/operators/op_methods.cpp][line:1936][sigmoid_][thread:140419487147776][process:29
59]:
sigmoid_ Op cannot run on MLU device, start running on CPU!
[WARNING][/pytorch/catch/torch_mlu/csrc/aten/operators/op_methods.cpp][line:1936][sigmoid_][thread:140419487147776][process:29
59]:
sigmoid_ Op cannot run on MLU device, start running on CPU!
```

查看文档，MLU暂不支持addmm、sigmoid_算子，因此涉及这种算子的代码会把数据放到CPU上运行，然后把结果拷贝回MLU，因此不影响代码运行。

7.2.1.1 对 MLU 不支持算子的处理

对于 MLU 暂不支持的算子，如果未在 catch/torch_mlu/tools/mlu_functions.yaml 中声明，程序会直接终止，并抛出无法分发到 MLU 设备的异常；如果已经在 catch/torch_mlu/tools/mlu_functions.yaml 中声明但在运行至 wrapper 或 kernel 时失败，输入数据将会拷贝到 CPU 上，然后调用 CPU 相关算子，使其在 CPU 上运行，最后再将输出结果拷回到 MLU 上。具体实现，可以查询 op_methods.cpp，该文件在 catch/torch_mlu/csrc/aten/operators/ 目录下。

1.4 融合推理

```python
# fusion infer
ct.save_as_cambricon("policyNet_offline")
traced_model = torch.jit.trace(net_quantization.forward, x_mlu,
check_trace=False)
print(type(traced_model))
x, h = traced_model(x_mlu)
```

出现问题：RNN中使用了MLU不适配的算子，无法进行融合推理，无法生成寒武纪离线模型，但是在mlu_functions.yaml中这两个算子均有声明。

```
[WARNING][/pytorch/catch/torch_mlu/csrc/jit/passes/segment_graph.cpp][line:41][MLUSupport][thread:140554987390720][process:284
53]:
[Fusion Segment] Please check mlu_functions.yaml && Maybe MLU fusion does NOT supports op: addmm
%33 : Float(*, *) = aten::addmm(%16, %input.5, %30, %31, %31), scope: RnnAgent/GRUCell[rnn] # /torch/venv3/pytorch/lib/python3
.6/site-packages/torch/nn/modules/rnn.py:1023:0

[WARNING][/pytorch/catch/torch_mlu/csrc/jit/passes/segment_graph.cpp][line:41][MLUSupport][thread:140554987390720][process:284
53]:
[Fusion Segment] Please check mlu_functions.yaml && Maybe MLU fusion does NOT supports op: sigmoid_
%52 : Tensor = aten::sigmoid_(%51), scope: RnnAgent/GRUCell[rnn] # /torch/venv3/pytorch/lib/python3.6/site-packages/torch/nn/m
odules/rnn.py:1023:0
```

```
Graph is segmented into 4 subgraphs.
Please check the above WARNING logs which include "[Fusion Segment]" to see whether there are MLU unsupported ops exists in py
torch model.

Traceback (most recent call last):
  File "test_mlu_copy.py", line 87, in <module>
    x, h = traced_model(x_mlu)
  File "/torch/venv3/pytorch/lib/python3.6/site-packages/torch/nn/modules/module.py", line 541, in __call__
    result = self.forward(*input, **kwargs)
RuntimeError: outputs_[i]->uses().empty() INTERNAL ASSERT FAILED at /pytorch/torch/csrc/jit/ir.cpp:1027, please report a bug t
o PyTorch.  (eraseOutput at /pytorch/torch/csrc/jit/ir.cpp:1027)
```

TracedModule打印:

```
TracedModule[RnnAgent](
  original_name=RnnAgent
  (enc): TracedModule[Sequential](
    original_name=Sequential
    (0): TracedModule[MLULinear](original_name=MLULinear)
    (1): TracedModule[ReLU](original_name=ReLU)
    (2): TracedModule[MLULinear](original_name=MLULinear)
    (3): TracedModule[ReLU](original_name=ReLU)
  )
  (rnn): TracedModule[GRUCell](original_name=GRUCell)
  (f_out): TracedModule[MLULinear](original_name=MLULinear)
)
```

解决方案:

- **在Python 3.6、PyTorch 1.3.0、cudatoolkit 9.2环境重新训练，尽可能保证算子适配:**

  训练出现问题:

```
RuntimeError: CUDA error: CUBLAS_STATUS_EXECUTION_FAILED when calling cublasSgemm( handle, opa, opb, m, n, k, &alpha, a, lda, b, ldb, &beta, c, ldc)
```

  将代码放到CPU、高版本CUDA跑，没有出现问题，所以不是代码的原因，升级CUDA版本10.1，可以运行，因此**推荐开发、训练环境：python 3.7、Pytorch 1.3.0、cudatoolkit 10.1。**

  在**python 3.7、Pytorch 1.3.0、cudatoolkit 10.1**该环境中重新训练，报相同的错误。

- 把不适配的算子重新修改一下，拆成几个操作的组合，并且用现有的PyTorch函数的实现，防止无法计算梯度等各种问题，**难度比较大。**

2024.3.12

使用论文Reconfigurable Intelligent Surface Assisted Multiuser MISO Systems Exploiting Deep Reinforcement Learning中模型，**state是传输功率，接收功率，还有t-1时隙的信道矩阵，输入尺寸为(1, 128)，action是传输波束赋形矩阵和相位移动矩阵，尺寸为(1，40)**，更换模型：

```
def __init__(self, state_dim, action_dim, M, N, K, power_t, device, max_action=1):
        super(Actor, self).__init__()
        hidden_dim = 1 if state_dim == 0 else 2 ** (state_dim - 1).bit_length()

        self.device = device

        self.M = M
        self.N = N
        self.K = K
        self.power_t = power_t

        self.l1 = nn.Linear(state_dim, hidden_dim)
        self.l2 = nn.Linear(hidden_dim, hidden_dim)
        self.l3 = nn.Linear(hidden_dim, action_dim) #会被量化为cnq.MLULinear

        self.bn1 = nn.BatchNorm1d(hidden_dim) #不会被量化
        self.bn2 = nn.BatchNorm1d(hidden_dim)

        self.max_action = max_action
```

逐层推理和融合推理都没有出现问题，并逐层推理和融合推理结果一样，但是与cpu推理有一定精度差距（因为量化模型需要将模型权重参数为int8、int16等整形数据），最后生成寒武纪**MLU270离线模型**actor_offline.cambricon。

> 之后一般不会降低多少精度，有的甚至会提高精度。寒武纪软件栈针对卷积、全连接算子等必须要进行量化后才能运行；而其他如激活算子、BN 算子等不需要量化，直接使用浮点型计算。

而且通过文档得知，BN算子BatchNorm1d不会被量化，直接使用浮点型计算。

原生的 PyTorch 算子	替换后的 MLU 算子
nn.Linear	cnq.MLULinear
nn.Conv3d	cnq.MLUConv3d
nn.Conv2d	cnq.MLUConv2d
nn.Conv1d	cnq.MLUConv1d
nn.ConvTranspose3d	cnq.MLUConvTranspose3d
nn.ConvTranspose2d	cnq.MLUConvTranspose2d
nn.LocalResponseNorm	cnq.MLULocalResponseNorm
nn.LSTM	cnq.MLULSTM
nn.GRU	cnq.MLUGRU
nn.InstanceNorm2d	cnq.MLUInstanceNorm2d
nn.MaxUnpool2d	cnq.MLUMaxUnpool2d

文档里原生与模型量化后替换的算子表格中并没有nn.GRUcell，只有nn.GRU，所以尝试使用GRU重新训练（TODO）。

2、编写运行离线模型的CNRT（C++）们在x86服务器+MLU270上的容器中运行正常后，生成MLU220的离线模型。（参考文档：寒武纪运行时库用户手册 Cambricon-CNRT-User-Guide-CN-v4.10.1.pdf 位置 cambricon/1.7.0/CNtoolkit/cnrt/Cambricon-CNRT-User-Guide-CN-v4.10.1.pdf）

> 文档中说的开发样例位于/usr/local/neuware/samples/cnrt/下。

2.1 环境准备

- CNRT驱动、CNRT头文件 `/usr/local/neuware/include` 、动态链接库 `/usr/local/neuware/lib64` 或者 `/usr/local/neuware/lib` 目录下。

- 已经生成的离线模型文件： `model.cambricon` 和 `model.cambricon_twins` （ `cat odel.cambricon_twins` 可以查看离线模型的具体信息），把core_version设置为MLU220，然后利用前边的步骤保存为MLU220离线模型即可。

```
ct.set_core_number(4)
ct.set_core_version("MLU220")
torch.set_grad_enabled(False)
```

- CMake：用于编译
- glog（用于日志记录，可选）

2.2 编写CMakeList.txt

- 文件结构整理如下：

  bin：存放输出文件（可执行文件）

  build：用于存放编译后的文件

  src：存放源文件

```
(pytorch) root@c2e0e359e31b:/home/ict/test_mlu270/newtry/cnrt_test# tree
.
├── CMakeLists.txt
├── bin
├── build
└── src
    ├── CMakeLists.txt
    └── main.cpp
```

- 外层CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8 FATAL_ERROR)
project(demo_cnrt)

add_subdirectory(src)
```

- src/CMakeLists.txt

```
# add cnrt
include_directories(/usr/local/neuware/include) #x86
link_directories(/usr/local/neuware/lib64) #x86
link_libraries(cnrt)
```

```

aux_source_directory(. SRC_LIST)
add_executable(democnrt ${SRC_LIST}) #执行

set (EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin) #将输出文件保存到bin目录
target_link_libraries(democnrt cnrt) # （可选）
```

- 在build文件中：

```
cmake .. && make

"""
-- The C compiler identification is GNU 7.3.0
-- The CXX compiler identification is GNU 7.3.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to:
/home/ict/test_mlu270/newtry/cnrt_test/build
Scanning dependencies of target democnrt
[ 50%] Building CXX object src/CMakeFiles/democnrt.dir/main.cpp.o
[100%] Linking CXX executable ../../bin/democnrt
[100%] Built target democnrt
"""
```

- 运行无误之后在bin文件中生成可执行文件，运行:

```
./democnrt
```

2.3 编写CNRT代码，运行、DEBUG MLU270离线模型

根据官方给的例程修改即可，

```
#include "cnrt.h"
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

using namespace std;

int offline_test(const char *name) {
         // This example is used for MLU270 and MLU220. You need to choose the corresponding offline model.
         // when generating an offline model, u need cnml and cnrt both
         // when running an offline model, u need cnrt only
        //在调用任何 CNRT API 之前需要调用此 API 初始化环境,保证线程安全。
        //1. init
        cnrtInit(0);

        //2. check mlu device, set device
        cnrtDev_t dev;
        cnrtGetDeviceHandle(&dev, 0);
        cnrtSetCurrentDevice(dev);

        //3. load model
        cnrtModel_t model;
        cnrtLoadModel(&model, "../model/actor_mlu270_offline.cambricon");
        cout << "model:" << model << "\n";

        //4. get model propertys
        //get model total memory
        int64_t totalMem;
        cnrtGetModelMemUsed(model, &totalMem);
        printf("total memory used:%ld Bytes\n", totalMem);
        //get model parallelism
        int model_parallelism;
        cnrtQueryModelParallelism(model, &model_parallelism);
        printf("model parallelism:%d.\n", model_parallelism);
        //5. build inference logic flow
        cnrtFunction_t function;
        cnrtCreateFunction(&function);
        char *func_name = NULL;
        int func_name_size = 0;
        cnrtGetFunctionSymbol(model, 0, &func_name, &func_name_size);
        cout << "func_name:" << func_name << "\n";
        //生成的模型推理结构其实被当做一个内核函数，函数名就是这个subnet0
```

```
        cnrtExtractFunction(&function, model, func_name);

        //6. get size of input, output
        int inputNum, outputNum;
        int64_t *inputSizeS, *outputSizeS;
        cnrtGetInputDataSize(&inputSizeS, &inputNum, function);
        cnrtGetOutputDataSize(&outputSizeS, &outputNum, function);
        cout << "inputNum:" << inputNum << " inputSizeS:" << *inputSizeS << "\n";
        cout << "outputNum:" << outputNum << " outputSizeS:" << *outputSizeS << "\n";

        //7. prepare data on cpu
        //申请CPU上的输入输出内存
        void **inputCpuPtrS = (void **)malloc(inputNum * sizeof(void *));
        void **outputCpuPtrS = (void **)malloc(outputNum * sizeof(void *));

        //8. allocate I/O data memory on MLU
        //申请MLU上的输入输出内存，申请数据节点的句柄，没有真正申请内存
        void **inputMluPtrS = (void **)malloc(inputNum * sizeof(void *));
        void **outputMluPtrS = (void **)malloc(outputNum * sizeof(void *));

        //prepare input buffer
        //真正申请内存空间，类比cuda的话就是申请显存
        for (int i = 0; i < inputNum; i++) {
                //converts data format when using new interface model
                inputCpuPtrS[i] = malloc(inputSizeS[i]);
                //malloc mlu memory
                cnrtMalloc(&(inputMluPtrS[i]), inputSizeS[i]);
                cnrtMemcpy(inputMluPtrS[i], inputCpuPtrS[i], inputSizeS[i], CNRT_MEM_TRANS_DIR_HOST2DEV);
        }

        //prepare output buffer
        for (int i = 0; i < outputNum; i++) {
                outputCpuPtrS[i] = malloc(outputSizeS[i]);
                //malloc mlu memory
                cnrtMalloc(&(outputMluPtrS[i]), outputSizeS[i]);
        }

        //prepare parameters for cnrtInvokeRuntimeContext
        void **param = (void **)malloc(sizeof(void *) * (inputNum + outputNum));
        for (int i = 0; i < inputNum; ++i) {
                param[i] = inputMluPtrS[i];
        }
        for (int i = 0; i < outputNum; ++i) {
```

```
                param[inputNum + i] = outputMluPtrS[i];
        }

        //setup runtime ctx
        cnrtRuntimeContext_t ctx;
        cnrtCreateRuntimeContext(&ctx, function, NULL);

        //bind device
        cnrtSetRuntimeContextDeviceId(ctx, 0);
        cnrtInitRuntimeContext(ctx, NULL);

        //compute offline
        cnrtQueue_t queue;
        cnrtRuntimeContextCreateQueue(ctx, &queue);

        // invoke
        cnrtInvokeRuntimeContext(ctx, param, queue, NULL);

        // sync
        cnrtSyncQueue(queue);

        // copy mlu result to cpu
           for (int i = 0; i < outputNum; i++) {
                cnrtMemcpy(outputCpuPtrS[i], outputMluPtrS[i],
outputSizeS[i], CNRT_MEM_TRANS_DIR_DEV2HOST);
        }

        //free memory space
        for (int i = 0; i < inputNum; i++) {
                free(inputCpuPtrS[i]);
                cnrtFree(inputMluPtrS[i]);
        }
        for (int i = 0; i < outputNum; i++) {
                free(outputCpuPtrS[i]);
                cnrtFree(outputMluPtrS[i]);
        }
        free(inputCpuPtrS);
        free(outputCpuPtrS);
        free(param);

        cnrtDestroyQueue(queue);
        cnrtDestroyRuntimeContext(ctx);
        cnrtDestroyFunction(function);
        cnrtUnloadModel(model);
        cnrtDestroy();

        return 0;
```

```
}

int main() {
        printf("offline test\n");
        offline_test("mlp");

        return 0;
}
```

3、上一步运行没问题题之后，在上面代码中把cnrtLoadModel加载的模型改称MLU220的离线模型actor_mlu220_offline.cambricon。准备在X86服务器上交叉编译，生成aarch64可执行文件，将aarch64可执行文件和mlu220离线模型文件通过网络拷贝到MLU220 SOC开发板的硬盘上。

3.1 安装交叉编译工具链gcc-linaro-6.2.1-2016.11-x86_64_aarch64-linux-gnu

```
mkdir /usr/local/arm #创建arm目录
mv ./gcc-linaro-6.2.1-2016.11-x86_64_aarch64-linux-gnu.tar.xz ./usr/local/arm #把交叉编译工具链拷贝过来
cd /usr/local/arm
tar -vxf gcc-linaro-6.2.1-2016.11-x86_64_aarch64-linux-gnu.tar.xz #解压
cd ./gcc-linaro-6.2.1-2016.11-x86_64_aarch64-linux-gnu/bin
export PATH=$PATH:$PWD/ #配置环境变量，推荐写入~/.bashrc，否则关闭终端丢失。
aarch64-linux-gnu-gcc -v #检测是否安装成功

"""打印下面信息即安装成功
Using built-in specs.
COLLECT_GCC=aarch64-linux-gnu-gcc
COLLECT_LTO_WRAPPER=/usr/local/arm/gcc-linaro-6.2.1-2016.11-x86_64_aarch64-linux-gnu/bin/../libexec/gcc/aarch64-linux-gnu/6.2.1/lto-wrapper
Target: aarch64-linux-gnu
Configured with: /home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-amd64-tcwg-build/target/aarch64-linux-gnu/snapshots/gcc-linaro-6.2-2016.11/configure SHELL=/bin/bash --with-mpc=/home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-amd64-tcwg-build/target/aarch64-linux-gnu/_build/builds/destdir/x86_64-unknown-linux-gnu --with-mpfr=/home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-amd64-tcwg-build/target/aarch64-linux-gnu/_build/builds/destdir/x86_64-unknown-linux-gnu --with-gmp=/home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-amd64-tcwg-build/target/aarch64-linux-gnu/_build/builds/destdir/x86_64-unknown-linux-gnu --with-gnu-as --with-gnu-ld --disable-libstdcxx-pch --disable-libmudflap --with-cloog=no --with-ppl=no --with-isl=no --disable-nls --enable-c99 --enable-gnu-indirect-function --disable-multilib --with-arch=armv8-a --enable-fix-
```

```
cortex-a53-835769 --enable-fix-cortex-a53-843419 --enable-multiarch --with-build-sysroot=/home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-amd64-tcwg-build/target/aarch64-linux-gnu/_build/sysroots/aarch64-linux-gnu --enable-lto --enable-linker-build-id --enable-long-long --enable-shared --with-sysroot=/home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-amd64-tcwg-build/target/aarch64-linux-gnu/_build/builds/destdir/x86_64-unknown-linux-gnu/aarch64-linux-gnu/libc --enable-languages=c,c++,fortran,lto --enable-checking=release --disable-bootstrap --build=x86_64-unknown-linux-gnu --host=x86_64-unknown-linux-gnu --target=aarch64-linux-gnu --prefix=/home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-amd64-tcwg-build/target/aarch64-linux-gnu/_build/builds/destdir/x86_64-unknown-linux-gnu
Thread model: posix
gcc version 6.2.1 20161016 (Linaro GCC 6.2-2016.11)
"""
```

3.2 因为交叉编译需要链接MLU220的cnrt动态链接库（aarch64架构），因此通过ssh拷贝MLU220的cnrt动态链接库到X86服务器，在宿主机端输入：

```
scp -r root@192.168.100.50:/neuware C:\Users\Lenovo\neuware #mlu220 -> 宿主机
```

再通过U盘拷贝到X86服务器，在X86服务器创建目录 `/usr/local/neuware220/` ，将动态链接库文件拷贝进去。

3.3 编写CMakeLists文件：

- 文件目录结构如下：

```
root@c2e0e359e31b:/home/ict/test_mlu270/newtry/cnrt_test# tree
.
├── CMakeLists.txt
├── bin
├── build
├── model
│   ├── actor_mlu270_offline.cambricon
│   └── actor_mlu270_offline.cambricon_twins
└── src
    ├── CMakeLists.txt
    └── main.cpp
```

- 外层CMakeLists：

```
cmake_minimum_required(VERSION 2.8 FATAL_ERROR)
project(demo_cnrt)

SET(CMAKE_SYSTEM_NAME Linux)
SET(CMAKE_SYSTEM_PROCESSOR aarch64)

add_compile_options(-std=c++11)
SET(CMAKE_C_COMPILER "aarch64-linux-gnu-gcc")
SET(CMAKE_CXX_COMPILER "aarch64-linux-gnu-g++")

add_subdirectory(src)
```

- src/CMakeLists：

```
# add cnrt
include_directories(/usr/local/neuware/include)
link_directories(/usr/local/neuware220/neuware/lib64/) #mlu220 cnrt动态链接库的目录
link_libraries(cnrt)

# add aarch64-linux-gnu-gcc
include_directories(/usr/local/arm/gcc-linaro-6.2.1-2016.11-x86_64_aarch64-linux-gnu/include/)
link_directories(/usr/local/arm/gcc-linaro-6.2.1-2016.11-x86_64_aarch64-linux-gnu/aarch64-linux-gnu/lib64/)

aux_source_directory(. SRC_LIST)
add_executable(democnrt ${SRC_LIST})

set (EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)

target_link_libraries(democnrt cnrt)
```

3.3 编译、生成arm可执行文件。

```
cd build/
cmake .. && make
"""
-- The C compiler identification is GNU 7.3.0
-- The CXX compiler identification is GNU 7.3.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
```

```
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/ict/test_mlu270/newtry/cnrt_test/build
Scanning dependencies of target democnrt
[ 50%] Building CXX object src/CMakeFiles/democnrt.dir/main.cpp.o
[100%] Linking CXX executable ../../bin/democnrt
[100%] Built target democnrt
"""
file ../bin/democnrt #下边打印ARM aarch64即交叉编译成功
"""
ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV), dynamically linked,
interpreter /lib/ld-linux-aarch64.so.1, for GNU/Linux 3.7.0, BuildID[sha1]=12a9f3ba441d8326d83f030d67d1ce084a0ac019, not stripped
"""
```

3.4 将可执行文件与离线模型拷贝至MLU220

- 压缩aarch64可执行文件与离线模型。

```
tar -czvf mlu220_run.tar.gz model bin
"""
model/
model/actor_mlu220_offline.cambricon_twins
model/actor_mlu220_offline.cambricon
bin/
bin/democnrt
"""
```

- 在宿主机ssh拷贝到MLU220的/cambricon目录下，防止掉电丢失。

```
scp -r G:\mlu220_run.tar.gz root@192.168.100.50:/cambricon #宿主机 -> mlu220
```

- 在MLU220中，解压，运行aarch64可执行文件。

```
cd /cambricon
mv mlu220_run.tar.gz ./mlu220_run
cd ./mlu220_run
tar -xzvf ./mlu220_run.tar.gz
cd bin
./democnrt #运行
```

```
[root@cambricon /cambricon/mlu220_run/bin]# ./democnrt
offline test
CNRT: 4.10.1 a884a9a
model:0x1c8bfd70
total memory used:140448 Bytes
model parallelism:4.
func_name:subnet0
inputNum:1 inputSizeS:448
outputNum:1 outputSizeS:160
[root@cambricon /cambricon/mlu220_run/bin]#
```

流程完成。

五、使用CNRT（CNRT代码：离线推理流程）

主要依据例程：/torch/examples/offline/test_forward_offline/test_forward_offline.cpp，并查阅寒武纪CNRT开发者文档（Cambricon-CNRT-Developer-Guide-EN-v4.10.1.pdf）。

1、初始化：初始化环境，并且获取设备句柄，设置当前使用的设备。

```
cnrtInit(0);
cnrtDev_t dev;
cnrtGetDeviceHandle(&dev, 0);
cnrtSetCurrentDevice(dev);
```

2、加载模型，并且获取一些必要信息

2.1 测试就加载270的离线模型，交叉编译则加载220的离线模型，可以使用一些API获取模型的一些信息，比如占用内存 `cnrtGetModelMemUsed` ，并行度 `cnrtQueryModelParallelism` 等等。

```
cnrtModel_t model;
cnrtLoadModel(&model, "../model/actor_mlu270_offline.cambricon");
cnrtGetModelMemUsed(model, &totalMem);
```

```
cnrtQueryModelParallelism(model, &model_parallelism);
```

2.2 把模型转成一个内核函数，用于调用，这里函数名字就是function。`cnrtCreateRuntimeContext` 在MLU上基于CNRT function创建CNRT function的上下文描述，存放到rt_ctx_中。作用：将计算资源与function本身解除绑定，实现同一个CNRT function可以根据不同的配置，实例化function到不同的MLU硬件设备。

```
cnrtFunction_t function;
cnrtCreateFunction(&function);
cnrtGetFunctionSymbol(model, 0, &func_name, &func_name_size);
cnrtExtractFunction(&function, model, func_name);
cnrtCreateRuntimeContext(&rt_ctx_, function, NULL);
```

2.3 获取模型的输入输出规模（Bytes）、输入输出数据类型

```
int inputNum, outputNum;
int64_t *inputSizeS, *outputSizeS;
cnrtGetInputDataSize(&inputSizeS, &inputNum, function);
cnrtGetOutputDataSize(&outputSizeS, &outputNum, function);
cnrtDataType_t* input_data_type = nullptr;
cnrtDataType_t* output_data_type = nullptr;
cnrtGetInputDataType(&input_data_type, &inputNum, function);
cnrtGetOutputDataType(&output_data_type, &outputNum, function);
```

3、在系统内存上分配 I/O 数据空间并准备输入数据

3.1 分配输入空间

- `cnrtDataTypeSize` 可以获取输入数据类型的大小，这里是4 Bytes，因此输入规模（Bytes）/数据类型大小（Bytes）= 输入尺寸（无量纲），本模型输入尺寸为112。
- 申请输入尺寸大小的databuf，然后把数据读入databuf（这里采用随机数），然后放到申请好的内存空间inputCpuPtrS上。
- `cnrtGetInputDataShape` 获取输入数据的尺寸，正常数据输入的格式是NCHW，这个API返回的是NHWC，注意调整。

```
void **inputCpuPtrS = reinterpret_cast<void **>(malloc(inputNum * sizeof(void *)));
void **outputCpuPtrS = reinterpret_cast<void **>(malloc(outputNum * sizeof(void *)));
```

```
vector<int> out_count;
unsigned int in_n, in_c, in_h, in_w;
for (int i = 0; i < inputNum; i++) {
    int ip = inputSizeS[i] / cnrtDataTypeSize(input_data_type[i]);
    auto databuf = reinterpret_cast<float *>(malloc(sizeof(float) * ip));
    rand_data(databuf, ip); //data in databuf
    inputCpuPtrS[i] = reinterpret_cast<void*>(databuf);  // NCHW
    vector<int> shape(4, 1);
    int dimNum = 4;
    cnrtGetInputDataShape((int**)&shape, &dimNum, i, function);  // NHWC
    in_n = shape[0];
    in_c = (input_data_type[i] == CNRT_UINT8) ? (shape[3] - 1) : shape[3];
    in_h = shape[1];
    in_w = shape[2];
}
```

3.2 分配输出空间

与上面流程类似。

```
vector<float *> output_cpu;
unsigned int out_n, out_c, out_h, out_w;
for (int i = 0; i < outputNum; i++) {
    int op = outputSizeS[i] / cnrtDataTypeSize(output_data_type[i]);
    float* outcpu = reinterpret_cast<float*>(malloc(op * sizeof(float)));
    out_count.push_back(op);
    output_cpu.push_back(outcpu);
    outputCpuPtrS[i] = reinterpret_cast<void*>(outcpu);
    std::vector<int> shape(4, 1);
    int dimNum = 4;
    cnrtGetOutputDataShape((int**)&shape, &dimNum, i, function);  // NHWC
    out_n = shape[0];
    out_c = shape[3];
    out_h = shape[1];
    out_w = shape[2];
    cout << "out_n " << out_n << " out_c " << out_c
                        << " out_h " << out_h << " out_w " << out_w << "\n";
}
```

4、在MLU内存上分配I/O数据空间并且准备数据

- 使用 `cnrtMalloc` 在mlu内存上分配数据空间。
- param后便会用到。

```
void** inputMluPtrS = reinterpret_cast<void**>(malloc(sizeof(void*) *
inputNum));
void** outputMluPtrS = reinterpret_cast<void**>(malloc(sizeof(void*) *
outputNum));
void **param =
    reinterpret_cast<void **>(malloc(sizeof(void *) * (inputNum +
outputNum)));
for (int i = 0; i < inputNum; i++) {
    cnrtMalloc(&inputMluPtrS[i], inputSizeS[i]);
}
for (int i = 0; i < outputNum; i++) {
    cnrtMalloc(&outputMluPtrS[i], outputSizeS[i]);
}

for (int i = 0; i < inputNum; i++) {
    param[i] = inputMluPtrS[i];
}
for (int i = 0; i < outputNum; i++) {
      param[inputNum + i] = outputMluPtrS[i];
}
```

4.1 接下来要把数据从CPU拷贝到MLU进行离线推理了，这里就需要用到上面提到的function的上下文rt_ctx，需要设置使用该function上下文的MLU设备，然后初始化funtion上下文。为了计算任务，还要在此context上创建计算队列。

> 队列queue：在 CNRT 中,一个队列(cnrtQueue_t)是由主机端发布的一系列对 MLU 设备的执行操作。**所有的设备操作都是通过队列进行的，**其中包括计算任务、通讯任务以及事件等。**同一个队列可以容纳多个任务。在一个队列中，操作是按顺序串行执行的,不同的队列中的操作可以并发执行。**队列内部无论是执行Notifier 任务还是计算任务,始终遵循 FIFO(First In First Out,先进先出)调度原则。当没有指定队列的时候，CNRT 会使用默认的队列。
>
> 用法：可以使用cnrtCreateQueue创建一个队列然后cnrtInvokeRuntimeContext调用funtion上下文时调用这个队列，也可以使用cnrtRuntimeContextCreateQueue直接在MLU上为function创建一个队列。

```
//create cnrt_queue
cnrtQueue_t cnrt_queue;
//cnrtCreateQueue(&cnrt_queue);
cnrtSetRuntimeContextDeviceId(rt_ctx_, dev);
cnrtInitRuntimeContext(rt_ctx_, NULL);
cnrtRuntimeContextCreateQueue(rt_ctx_, &cnrt_queue);
void** tempPtrS = reinterpret_cast<void**>(malloc(sizeof(void*) * inputNum));
void* temp_input_cpu_data = nullptr;
for (int i = 0; i < inputNum; i++) {
    int ip = inputSizeS[i] / cnrtDataTypeSize(input_data_type[i]);
```

```
        auto databuf = reinterpret_cast<float*>(malloc(sizeof(float) * ip));
        tempPtrS[i] = reinterpret_cast<void*>(databuf);
        std::vector<int> shape(4, 1);
        int dimNum = 4;
        cnrtGetInputDataShape((int**)&shape, &dimNum, i, function);
        int dim_order[4] = {0, 2, 3, 1};
        int dim_shape[4] = {shape[0], shape[3],
                            shape[1], shape[2]};  // NCHW
        cnrtTransDataOrder(inputCpuPtrS[i], CNRT_FLOAT32, tempPtrS[i],
                                 4, dim_shape, dim_order); //transfer order
        temp_input_cpu_data = (void*)malloc(inputSizeS[i]);
        int input_count = inputSizeS[i] / cnrtDataTypeSize(input_data_type[i]);
        if (input_data_type[i] != CNRT_FLOAT32) {
            cnrtCastDataType(tempPtrS[i],
                            CNRT_FLOAT32,
                            temp_input_cpu_data,
                            input_data_type[i],
                                                          input_count,
                            nullptr);
        } else {
            temp_input_cpu_data = tempPtrS[i];
        }
        cnrtMemcpy(inputMluPtrS[i],
                    temp_input_cpu_data,
                    inputSizeS[i],
                    CNRT_MEM_TRANS_DIR_HOST2DEV);
        if (temp_input_cpu_data) {
            free(temp_input_cpu_data);
            temp_input_cpu_data = nullptr;
        }
    }
```

在把CPU上的数据拷贝到MLU输入数据前，使用`cnrtTransDataOrder`对数据进行了预处理，NCHW->NHWC，并且使用`cnrtCastDataType`对输入数据进行类型转换，转换为CNRT_FLOAT32类型，然后使用`cnrtMemcpy`将数据从CPU拷贝到MLU上，这里temp_input_cpu_data其实就是一个中间变量。

4.2 运行上下文

```
cnrtNotifier_t notifierBeginning, notifierEnd;
cnrtCreateNotifier(&notifierBeginning);
cnrtCreateNotifier(&notifierEnd);
float event_time_use;
// run MLU
// place start_event to cnrt_queue
```

```
cnrtPlaceNotifier(notifierBeginning, cnrt_queue);
CNRT_CHECK(cnrtInvokeRuntimeContext(rt_ctx_, param, cnrt_queue, nullptr));
// place end_event to cnrt_queue
cnrtPlaceNotifier(notifierEnd, cnrt_queue);
```

这里使用 `cnrtInvokeRuntimeContext` 调用funtion上下文执行队列里的运算任务，这里用到了上面提到的param。

- CNRT_CHECK用于打印Debug信息。
- Notifier机制：CNRT使用Notifier机制统计硬件执行时间。Notifier 是一种特殊的任务,它和计算任务一样可以放置到队列中执行。相比计算任务,Notifier 不执行实际的硬件操作。如果相邻的两个或者多个 Notifier 之间包含计算任务,可以通过 Notifier 来实现对计算任务耗时的精确统计。任务执行时间=notifier_end - notifier_begin。详情参考寒武纪CNRT用户手册和开发者手册。

4.3 把计算的结果从MLU拷贝到CPU，并且转换数据格式NHWC->NCHW。

这里数据保存到了outPtrS中。

```
void** outPtrS = reinterpret_cast<void**>(malloc(sizeof(void*) * outputNum));
void* temp_output_cpu_data = nullptr;
for (int i = 0; i < outputNum; i++) {
    temp_output_cpu_data = (void*)malloc(outputSizeS[i]);
    cnrtMemcpy(temp_output_cpu_data,
               outputMluPtrS[i],
               outputSizeS[i],
               CNRT_MEM_TRANS_DIR_DEV2HOST); //MLU->temp_output_cpu
    int output_count = outputSizeS[i] / cnrtDataTypeSize(output_data_type[i]);
    if (output_data_type[i] != CNRT_FLOAT32) {
        cnrtCastDataType(temp_output_cpu_data,
                         output_data_type[i],
                         outputCpuPtrS[i],
                         CNRT_FLOAT32,
                         output_count,
                         nullptr);
    } else {
        memcpy(outputCpuPtrS[i], temp_output_cpu_data, outputSizeS[i]); //temp_output_cpu_data -> outputCpuPtrS
    }
    auto databuf = reinterpret_cast<float*>(malloc(sizeof(float) * output_count));
    outPtrS[i] = reinterpret_cast<void*>(databuf);
    std::vector<int> shape(4, 1);
    int dimNum = 4;
    cnrtGetOutputDataShape((int**)&shape, &dimNum, i, function);
    int dim_order[4] = {0, 3, 1, 2};
```

```
    int dim_shape[4] = {shape[0], shape[1],
                                  shape[2], shape[3]};  // NHWC
    cnrtTransDataOrder(outputCpuPtrS[i], CNRT_FLOAT32, outPtrS[i],
                                 4, dim_shape, dim_order);
    if (temp_output_cpu_data) {
            free(temp_output_cpu_data);
            temp_output_cpu_data = nullptr;
    }
}
```

4.4 打印数据

out_count是数据尺寸。

```
for (int i = 0; i < outputNum; i++) {
    for (int j = 0; j < out_count[i]; ++j) {
        cout << (reinterpret_cast<float*>(outPtrS[i]))[j] << "\n";
    }
}
```

5、释放MLU的内存空间

```
output_cpu.clear();
//free memory space
free(inputCpuPtrS);
free(outputCpuPtrS);
free(outPtrS);
for (int i = 0; i < inputNum; i++) {
        cnrtFree(inputMluPtrS[i]);
}
for (int i = 0; i < outputNum; i++) {
        cnrtFree(outputMluPtrS[i]);
}

cnrtDestroyQueue(cnrt_queue);
cnrtDestroyFunction(function);
cnrtUnloadModel(model);
cnrtDestroy();
```

6、输出结果

尺寸为1x40输出。

```
copying output data of 0th function:out_count:40
-1
0.857186
1
1
-1
-1
1
1
1
-1
1
1
1
1
1
-1
-1
1
-1
1
0.997366
-1
-0.998165
-1
1
1
-1
-1
-1
-1
-1
-1
1
1
1
1
-1
1
-1
-1
```

7、代码：

```
#include "cnrt.h"
#include <iostream>
```

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctime>
#include <vector>
#include "glog/logging.h"
#include <fstream>


using namespace std;

void rand_data(float *data, int length) {
    unsigned int seed = 1024;
    for (int i = 0; i < length; i++) {
        if (i % 5 == 4) {
            data[i] = rand_r(&seed) % 100 / 100. + 0.0625;
        } else if (i % 5 >= 2) {
            data[i] = data[i - 2] + (rand_r(&seed) % 100) / 100.0 + 0.0625;
        } else {
            data[i] = (rand_r(&seed) % 100) / 100. + 0.0625;
        }
    }
}

int offline_test(const char *name) {
         // This example is used for MLU270 and MLU220. You need to choose the corresponding offline model.
         // when generating an offline model, u need cnml and cnrt both
         // when running an offline model, u need cnrt only
        //在调用任何 CNRT API 之前需要调用此 API 初始化环境,保证线程安全。
        //1. init
        cnrtInit(0);
        //2. check mlu device, set device
        cnrtDev_t dev;
        cnrtGetDeviceHandle(&dev, 0);
        cnrtSetCurrentDevice(dev);

        //3. load model
        cnrtModel_t model;
        cnrtLoadModel(&model, "../model/actor_mlu270_offline.cambricon");
        cout << "model:" << model << "\n";

        //4. get model propertys
        //get model total memory
        int64_t totalMem;
        cnrtGetModelMemUsed(model, &totalMem);
        printf("total memory used:%ld Bytes\n", totalMem);
```

```
        //get model parallelism
        int model_parallelism;
        cnrtRuntimeContext_t rt_ctx_;
        cnrtQueryModelParallelism(model, &model_parallelism);
        printf("model parallelism:%d.\n", model_parallelism);
        //init struct variable
        cnrtFunction_t function;
        cnrtCreateFunction(&function);
        char *func_name = NULL;
        int func_name_size = 0;
        cnrtGetFunctionSymbol(model, 0, &func_name, &func_name_size);
        cout << "func_name:" << func_name << "\n";
        //生成的模型推理结构其实被当做一个内核函数，函数名就是这个subnet0
        cnrtExtractFunction(&function, model, func_name);
        cnrtCreateRuntimeContext(&rt_ctx_, function, NULL);

        //6. get size of input, output
        int inputNum, outputNum;
        int64_t *inputSizeS, *outputSizeS;
        cnrtGetInputDataSize(&inputSizeS, &inputNum, function);
        cnrtGetOutputDataSize(&outputSizeS, &outputNum, function);
        cout << "inputNum:" << inputNum << " inputSizeS:" << *inputSizeS << "\n";
        cout << "outputNum:" << outputNum << " outputSizeS:" << *outputSizeS << "\n";
        cnrtDataType_t* input_data_type = nullptr;
        cnrtDataType_t* output_data_type = nullptr;
        cnrtGetInputDataType(&input_data_type, &inputNum, function);
        cnrtGetOutputDataType(&output_data_type, &outputNum, function);

        //7. prepare data on cpu
        //申请CPU上的输入输出内存
        void **inputCpuPtrS = reinterpret_cast<void **>(malloc(inputNum * sizeof(void *)));
        void **outputCpuPtrS = reinterpret_cast<void **>(malloc(outputNum * sizeof(void *)));
        vector<int> out_count;
        unsigned int in_n, in_c, in_h, in_w;
        for (int i = 0; i < inputNum; i++) {
            int ip = inputSizeS[i] / cnrtDataTypeSize(input_data_type[i]);
            cout << "ip:" << ip << " " << "cnrtDataTypeSize:" << cnrtDataTypeSize(input_data_type[i]) << "\n";
            auto databuf = reinterpret_cast<float *>(malloc(sizeof(float) * ip));
            rand_data(databuf, ip); //data in databuf
            inputCpuPtrS[i] = reinterpret_cast<void*>(databuf);   // NCHW
            vector<int> shape(4, 1);
```

```
            int dimNum = 4;
            cnrtGetInputDataShape((int**)&shape, &dimNum, i, function);  // NHWC
            in_n = shape[0];
            in_c = (input_data_type[i] == CNRT_UINT8) ? (shape[3] - 1) : shape[3];
            in_h = shape[1];
            in_w = shape[2];
            cout << "in_n: " << in_n << " in_c: " << in_c
                            << " in_h: " << in_h << " in_w: " << in_w << "\n";
        }
        vector<float *> output_cpu;
        unsigned int out_n, out_c, out_h, out_w;
        for (int i = 0; i < outputNum; i++) {
            int op = outputSizeS[i] / cnrtDataTypeSize(output_data_type[i]);
            float* outcpu = reinterpret_cast<float*>(malloc(op * sizeof(float)));
            out_count.push_back(op);
            output_cpu.push_back(outcpu);
            outputCpuPtrS[i] = reinterpret_cast<void*>(outcpu);
            std::vector<int> shape(4, 1);
            int dimNum = 4;
            cnrtGetOutputDataShape((int**)&shape, &dimNum, i, function);  // NHWC
            out_n = shape[0];
            out_c = shape[3];
            out_h = shape[1];
            out_w = shape[2];
            cout << "out_n " << out_n << " out_c " << out_c
                                << " out_h " << out_h << " out_w " << out_w << "\n";
        }

        // allocate I/O data space on MLU memory and copy Input data
        void** inputMluPtrS = reinterpret_cast<void**>(malloc(sizeof(void*) * inputNum));
        void** outputMluPtrS = reinterpret_cast<void**>(malloc(sizeof(void*) * outputNum));
        void **param =
            reinterpret_cast<void **>(malloc(sizeof(void *) * (inputNum + outputNum)));
        for (int i = 0; i < inputNum; i++) {
            cnrtMalloc(&inputMluPtrS[i], inputSizeS[i]);
        }
        for (int i = 0; i < outputNum; i++) {
            cnrtMalloc(&outputMluPtrS[i], outputSizeS[i]);
        }
```

```

        for (int i = 0; i < inputNum; i++) {
            param[i] = inputMluPtrS[i];
        }
        for (int i = 0; i < outputNum; i++) {
              param[inputNum + i] = outputMluPtrS[i];
        }

        //create cnrt_queue
        cnrtQueue_t cnrt_queue;
        //cnrtCreateQueue(&cnrt_queue);
        cnrtSetRuntimeContextDeviceId(rt_ctx_, dev);
        cnrtInitRuntimeContext(rt_ctx_, NULL);
        cnrtRuntimeContextCreateQueue(rt_ctx_, &cnrt_queue);
        void** tempPtrS = reinterpret_cast<void**>(malloc(sizeof(void*) * inputNum));
        void* temp_input_cpu_data = nullptr;
        for (int i = 0; i < inputNum; i++) {
            int ip = inputSizeS[i] / cnrtDataTypeSize(input_data_type[i]);
            auto databuf = reinterpret_cast<float*>(malloc(sizeof(float) * ip));
            tempPtrS[i] = reinterpret_cast<void*>(databuf);
            std::vector<int> shape(4, 1);
            int dimNum = 4;
            cnrtGetInputDataShape((int**)&shape, &dimNum, i, function);
            int dim_order[4] = {0, 2, 3, 1};
            int dim_shape[4] = {shape[0], shape[3],
                                shape[1], shape[2]};  // NCHW
            cnrtTransDataOrder(inputCpuPtrS[i], CNRT_FLOAT32, tempPtrS[i],
                                     4, dim_shape, dim_order); //transfer order
            temp_input_cpu_data = (void*)malloc(inputSizeS[i]);
            int input_count = inputSizeS[i] / cnrtDataTypeSize(input_data_type[i]);
            if (input_data_type[i] != CNRT_FLOAT32) {
                cnrtCastDataType(tempPtrS[i],
                                CNRT_FLOAT32,
                                temp_input_cpu_data,

                                input_data_type[i],
                                                          input_count,
                                nullptr);
            } else {
                temp_input_cpu_data = tempPtrS[i];
            }
            cnrtMemcpy(inputMluPtrS[i],
                            temp_input_cpu_data,
```

```
                        inputSizeS[i],
                        CNRT_MEM_TRANS_DIR_HOST2DEV);
            if (temp_input_cpu_data) {
                free(temp_input_cpu_data);
                temp_input_cpu_data = nullptr;
            }
        }

        // create start_event and end_event
        cnrtNotifier_t notifierBeginning, notifierEnd;
        cnrtCreateNotifier(&notifierBeginning);
        cnrtCreateNotifier(&notifierEnd);
        float event_time_use;
        // run MLU
        // place start_event to cnrt_queue
        cnrtPlaceNotifier(notifierBeginning, cnrt_queue);
        CNRT_CHECK(cnrtInvokeRuntimeContext(rt_ctx_, param, cnrt_queue, nullptr));
        // place end_event to cnrt_queue
        cnrtPlaceNotifier(notifierEnd, cnrt_queue);
        if (cnrtSyncQueue(cnrt_queue) == CNRT_RET_SUCCESS) { //Waits for a queue operations to be completed.
            //get start_event and end_event elapsed time
            cnrtNotifierDuration(notifierBeginning, notifierEnd, &event_time_use); //Computes the time duration between starting and ending notifiers.
            cout << " hardware time: " << event_time_use << "\n";
        } else {
            cout << " SyncQueue Error " << "\n";
        }
        void** outPtrS = reinterpret_cast<void**>(malloc(sizeof(void*) * outputNum));
        void* temp_output_cpu_data = nullptr;
        for (int i = 0; i < outputNum; i++) {
            temp_output_cpu_data = (void*)malloc(outputSizeS[i]);
            cnrtMemcpy(temp_output_cpu_data,
                        outputMluPtrS[i],
                        outputSizeS[i],
                        CNRT_MEM_TRANS_DIR_DEV2HOST); //MLU->temp_output_cpu
            int output_count = outputSizeS[i] / cnrtDataTypeSize(output_data_type[i]);
            if (output_data_type[i] != CNRT_FLOAT32) {
                cnrtCastDataType(temp_output_cpu_data,
                                output_data_type[i],
                                outputCpuPtrS[i],
                                CNRT_FLOAT32,
                                output_count,
```

```
                                nullptr);
            } else {
                memcpy(outputCpuPtrS[i], temp_output_cpu_data, outputSizeS[i]); //temp_output_cpu_data -> outputCpuPtrS
            }
            auto databuf = reinterpret_cast<float*>(malloc(sizeof(float) * output_count));
            outPtrS[i] = reinterpret_cast<void*>(databuf);
            std::vector<int> shape(4, 1);
            int dimNum = 4;
            cnrtGetOutputDataShape((int**)&shape, &dimNum, i, function);
            int dim_order[4] = {0, 3, 1, 2};
            int dim_shape[4] = {shape[0], shape[1],
                                      shape[2], shape[3]};  // NHWC
            cnrtTransDataOrder(outputCpuPtrS[i], CNRT_FLOAT32, outPtrS[i],
                                     4, dim_shape, dim_order);
            if (temp_output_cpu_data) {
                    free(temp_output_cpu_data);
                    temp_output_cpu_data = nullptr;
            }
        }

        for (int i = 0; i < outputNum; i++) {
            cout << "copying output data of " << i << "th" << " function:";
            stringstream ss;
            ss << "hello.txt";
            string output_name = ss.str();
            std::ofstream fout(output_name, std::ios::out);
            fout << std::flush;
            cout << "out_count:" << out_count[i] << "\n";
            for (int j = 0; j < out_count[i]; ++j) {
                fout <<(reinterpret_cast<float*>(outPtrS[i]))[j] << std::endl;
                cout << (reinterpret_cast<float*>(outPtrS[i]))[j] << "\n";
            }
            fout << std::flush;
            fout.close();
        }
        for (auto flo : output_cpu) {
            free(flo);
        }
        output_cpu.clear();
        //free memory space
        free(inputCpuPtrS);
        free(outputCpuPtrS);
        free(outPtrS);
        for (int i = 0; i < inputNum; i++) {
                cnrtFree(inputMluPtrS[i]);
```

```
        }
        for (int i = 0; i < outputNum; i++) {
                cnrtFree(outputMluPtrS[i]);
        }

        cnrtDestroyQueue(cnrt_queue);
        cnrtDestroyFunction(function);
        cnrtUnloadModel(model);
        cnrtDestroy();

        return 0;
}

int main() {
        printf("offline test\n");
        offline_test("mlp");
        return 0;
}
```

- 在CNRT中，本质上是将模型转成了一个函数function。
- 原模型输入规模是112，输出规模是40，在使用 `cnrtGetInputDataSize` 获取的inputSizeS为448，是因为此函数返回的是字节，使用malloc申请内存空间也是需要单位为字节，一个int4个字节，可以使用 `cnrtGetInputDataType` 获得function的输入输出数据类型datatype，然后使用 `cnrtDataTypeSize` 获得数据类型的大小datatypesize，那么输入规模就是inputSizeS/datatypesize。
- 队列queue：在 CNRT 中,一个队列(cnrtQueue_t)是由主机端发布的一系列对 MLU 设备的执行操作。**所有的设备操作都是通过队列进行的,**其中包括计算任务、通讯任务以及事件等。**同一个队列可以容纳多个任务。在一个队列中,操作是按顺序串行执行的,不同的队列中的操作可以并发执行。**队列内部无论是执行Notifier 任务还是计算任务,始终遵循 FIFO(First In First Out,先进先出)调度原则。当没有指定队列的时候,CNRT 会使用默认的队列。指定队列就使用 `cnrtCreateQueue` 这个API。MLU270 单卡队列的数量上限为 4,096。MLU220 单卡队列的数量上限为 1,024。
- 数据格式转化：cnrtTransDataOrder用于转换输入数据的格式，NHWC->NCHW，把inputCpuPtrS转换格式后的数据放到tempPtrS。

```
cnrtTransDataOrder(inputCpuPtrS[i], CNRT_FLOAT32, tempPtrS[i],
                        4, dim_shape, dim_order);
```

- 数据类型转换：cnrtCastDataType用于转换数据类型，输入到MLU上的数据一般是FP15、FP32类型。
- Notifier机制：

CNRT使用Notifier机制统计硬件执行时间。Notifier 是一种特殊的任务,它和计算任务一样可以放置到队列中执行。相比计算任务,Notifier 不执行实际的硬件操作。如果相邻的两个或者多个 Notifier 之间包含计算任务,可以通过 Notifier 来实现对计算任务耗时的精确统计。任务执行时间=notifier_end - notifier_begin。详情参考寒武纪CNRT用户手册和开发者手册。

- 离线模型不依赖于 PyTorch 框架,只基于 CNRT(Cambricon Neuware Runtime Library,寒武纪运行时库)单独运行。离线模型为.cambricon文件,生成离线模型可使用 Cambricon PyTorch 的 Python 接口将模型转为离线模型。**离线模型的生成以及离线推理教程在寒武纪PyTorch用户手册中写到，离线模型的使用在《寒武纪运行时库用户手册（CNRT开发者手册）》（cambricon/1.7.0/CNToolkit/cnrt）中写到。**

六、上位机与下位机交互

1、方案1：

1.1 x86服务器：

把推理程序编译成一个动态链接库，拷贝到MLU220。

1.2 上位机（作为client）：

使用Python运行强化学习环境（state），向server通过http请求的方式发送数据，接收server返回的结果（action）与环境交互。

1.3 MLU220（server）：

安装Python，编写Python程序接收client发送的数据，然后调用动态链接库，输入数据，得到结果再返回给server。

问题：

- MLU220里只有与业务相关的程序和库，是相当于一个极其精简的Linux系统，python、g++、gcc以及Linux很多命令都没有，所以在上边写程序不容易，除非重新在MLU220装一个完整的系统。

2、方案2：

2.1 x86服务器：

（1）寒武纪开发环境运行在x86服务器的docker容器中，外部网络与docker容器通信需要将docker容器端口映射到主机端口，外部网络才能通过主机端口访问到docker容器。

- 把已经装好开发环境的容器commit为一个新的镜像，c2e0e359e31b是当前可用容器ID，后边是新镜像的名字：

```
docker commit c2e0e359e31b yellow.hub.cambricon.com/pytorch/pytorch:0.15.0-
```

```
ubuntu16.04_3
```

- 基于新镜像启动一个新的容器（把系统家目录挂载到容器家目录，把mlu设备挂载到容器，并把容器的8050端口映射到主机8050端口）

```
docker run --name dev_cn -p 8050:8050 --privileged=true -dit --device=/dev/cambricon_dev0 --device=/dev/cambricon_ctl --user root -v /usr/bin/cnmon:/usr/bin/imagecnmon -v /home/:/home yellow.hub.cambricon.com/pytorch/pytorch:0.15.0-ubuntu16.04_3 /bin/bash
```

新创建的容器ID为933a2b8c520a，现在外部网络可以通过访问主机的8050端口访问容器，容器内需要监听8050端口的请求。

（2）把推理程序编译成一个动态链接库：

- 文件结构：

```
root@933a2b8c520a:/home/ict/test_mlu270/gen_so# tree
.
├── CMakeLists.txt
├── build
├── infer
│   ├── infer.cpp
│   └── infer.h
└── lib
```

- infer.cpp：封装了推理程序，两个参数：databuf是输入，outPtrS用于存储输出并返回。

```cpp
#include "infer.h"
#include "cnrt.h"
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctime>
#include <vector>
#include <fstream>
#include <cstring>

using namespace std;

int offline_infer(float *databuf, void** outPtrS, bool init, int num) {
```

```
         // This example is used for MLU270 and MLU220. You need to choose the corresponding offline model.
         // when generating an offline model, u need cnml and cnrt both
         // when running an offline model, u need cnrt only
        //在调用任何 CNRT API 之前需要调用此 API 初始化环境,保证线程安全。
        //1. init
        cnrtInit(0);
        //2. check mlu device, set device
        cnrtDev_t dev;
        cnrtGetDeviceHandle(&dev, 0);
        cnrtSetCurrentDevice(dev);

        //3. load model
        cnrtModel_t model;
        cnrtLoadModel(&model, "../model/actor_mlu270_offline.cambricon");
        if (init) {
                cout << "model:" << model << "\n";
        }

        //4. get model propertys
        //get model total memory
        int64_t totalMem;
        cnrtGetModelMemUsed(model, &totalMem);
        if (init) {
                printf("model total memory used:%ld Bytes\n", totalMem);

        }
        //get model parallelism
        int model_parallelism;
        cnrtRuntimeContext_t rt_ctx_;
        cnrtQueryModelParallelism(model, &model_parallelism);
        if (init) {
                printf("model parallelism:%d.\n", model_parallelism);
        }
        //init struct variable
        cnrtFunction_t function;
        cnrtCreateFunction(&function);
        char *func_name = NULL;
        int func_name_size = 0;
        cnrtGetFunctionSymbol(model, 0, &func_name, &func_name_size);
        if (init) {
                cout << "model func_name:" << func_name << "\n";
        }
        //生成的模型推理结构其实被当做一个内核函数，函数名就是这个subnet0
        cnrtExtractFunction(&function, model, func_name);
        cnrtCreateRuntimeContext(&rt_ctx_, function, NULL);

```

```
        //6. get size of input, output
        int inputNum, outputNum;
        int64_t *inputSizeS, *outputSizeS;
        cnrtGetInputDataSize(&inputSizeS, &inputNum, function);
        cnrtGetOutputDataSize(&outputSizeS, &outputNum, function);
        if (init) {
                cout << "model inputNum:" << inputNum << " model inputSizeS(Bytes):" << *inputSizeS << "\n";
                cout << "model outputNum:" << outputNum << " model outputSizeS(Bytes):" << *outputSizeS << "\n";
        }
        cnrtDataType_t* input_data_type = nullptr;
        cnrtDataType_t* output_data_type = nullptr;
        cnrtGetInputDataType(&input_data_type, &inputNum, function);
        cnrtGetOutputDataType(&output_data_type, &outputNum, function);

        //7. prepare data on cpu
        //申请CPU上的输入输出内存
        void **inputCpuPtrS = reinterpret_cast<void **>(malloc(inputNum * sizeof(void *)));
        void **outputCpuPtrS = reinterpret_cast<void **>(malloc(outputNum * sizeof(void *)));
        vector<int> out_count;
        unsigned int in_n, in_c, in_h, in_w;
        for (int i = 0; i < inputNum; i++) {
            int ip = inputSizeS[i] / cnrtDataTypeSize(input_data_type[i]);
            if (init) {
                    cout << "model ipsize:" << ip << " " << "cnrtDataTypeSize(Bytes):" << cnrtDataTypeSize(input_data_type[i]) << "\n";
            }
            //auto databuf = reinterpret_cast<float *>(malloc(sizeof(float) * ip));
            //rand_data(databuf, ip); //data in databuf
            inputCpuPtrS[i] = reinterpret_cast<void*>(databuf);  // NCHW
            vector<int> shape(4, 1);
            int dimNum = 4;
            cnrtGetInputDataShape((int**)&shape, &dimNum, i, function);  // NHWC
            in_n = shape[0];
            in_c = (input_data_type[i] == CNRT_UINT8) ? (shape[3] - 1) : shape[3];
            in_h = shape[1];
            in_w = shape[2];
            if (init) {
                    cout << "in_n: " << in_n << " in_c: " << in_c
                            << " in_h: " << in_h << " in_w: " << in_w << "\n";
            }
```

```
        }
        vector<float *> output_cpu;
        unsigned int out_n, out_c, out_h, out_w;
        for (int i = 0; i < outputNum; i++) {
            int op = outputSizeS[i] / cnrtDataTypeSize(output_data_type[i]);
            float* outcpu = reinterpret_cast<float*>(malloc(op * sizeof(float)));
            out_count.push_back(op);
            output_cpu.push_back(outcpu);
            outputCpuPtrS[i] = reinterpret_cast<void*>(outcpu);
            std::vector<int> shape(4, 1);
            int dimNum = 4;
            cnrtGetOutputDataShape((int**)&shape, &dimNum, i, function);  // NHWC
            out_n = shape[0];
            out_c = shape[3];
            out_h = shape[1];
            out_w = shape[2];
            if (init) {
                cout << "out_n " << out_n << " out_c " << out_c
                                << " out_h " << out_h << " out_w " << out_w << "\n";
            }
        }

        // allocate I/O data space on MLU memory and copy Input data
        void** inputMluPtrS = reinterpret_cast<void**>(malloc(sizeof(void*) * inputNum));
        void** outputMluPtrS = reinterpret_cast<void**>(malloc(sizeof(void*) * outputNum));
        void **param =
            reinterpret_cast<void **>(malloc(sizeof(void *) * (inputNum + outputNum)));
        for (int i = 0; i < inputNum; i++) {
            cnrtMalloc(&inputMluPtrS[i], inputSizeS[i]);
        }
        for (int i = 0; i < outputNum; i++) {
            cnrtMalloc(&outputMluPtrS[i], outputSizeS[i]);
        }

        for (int i = 0; i < inputNum; i++) {
            param[i] = inputMluPtrS[i];
        }
        for (int i = 0; i < outputNum; i++) {
              param[inputNum + i] = outputMluPtrS[i];
        }

```

```
        //create cnrt_queue
        cnrtQueue_t cnrt_queue;
        //cnrtCreateQueue(&cnrt_queue);
        cnrtSetRuntimeContextDeviceId(rt_ctx_, dev);
        cnrtInitRuntimeContext(rt_ctx_, NULL);
        cnrtRuntimeContextCreateQueue(rt_ctx_, &cnrt_queue);
        void** tempPtrS = reinterpret_cast<void**>(malloc(sizeof(void*) * inputNum));
        void* temp_input_cpu_data = nullptr;
        for (int i = 0; i < inputNum; i++) {
            int ip = inputSizeS[i] / cnrtDataTypeSize(input_data_type[i]);
            auto databuf = reinterpret_cast<float*>(malloc(sizeof(float) * ip));
            tempPtrS[i] = reinterpret_cast<void*>(databuf);
            std::vector<int> shape(4, 1);
            int dimNum = 4;
            cnrtGetInputDataShape((int**)&shape, &dimNum, i, function);
            int dim_order[4] = {0, 2, 3, 1};
            int dim_shape[4] = {shape[0], shape[3],
                                shape[1], shape[2]};  // NCHW
            cnrtTransDataOrder(inputCpuPtrS[i], CNRT_FLOAT32, tempPtrS[i],
                                     4, dim_shape, dim_order); //transfer order
            temp_input_cpu_data = (void*)malloc(inputSizeS[i]);
            int input_count = inputSizeS[i] / cnrtDataTypeSize(input_data_type[i]);
            if (input_data_type[i] != CNRT_FLOAT32) {
                cnrtCastDataType(tempPtrS[i],
                                CNRT_FLOAT32,
                                temp_input_cpu_data,

                                input_data_type[i],

                                input_count,
                                nullptr);
            } else {
                temp_input_cpu_data = tempPtrS[i];
            }
            cnrtMemcpy(inputMluPtrS[i],
                        temp_input_cpu_data,
                        inputSizeS[i],
                        CNRT_MEM_TRANS_DIR_HOST2DEV);
            if (temp_input_cpu_data) {
                free(temp_input_cpu_data);
                temp_input_cpu_data = nullptr;
            }
        }
```

```

        // create start_event and end_event
        cnrtNotifier_t notifierBeginning, notifierEnd;
        cnrtCreateNotifier(&notifierBeginning);
        cnrtCreateNotifier(&notifierEnd);
        float event_time_use;
        // run MLU
        // place start_event to cnrt_queue
        cnrtPlaceNotifier(notifierBeginning, cnrt_queue);
        CNRT_CHECK(cnrtInvokeRuntimeContext(rt_ctx_, param, cnrt_queue, nullptr));
        // place end_event to cnrt_queue
        cnrtPlaceNotifier(notifierEnd, cnrt_queue);
        if (cnrtSyncQueue(cnrt_queue) == CNRT_RET_SUCCESS) { //Waits for a queue operations to be completed.
            //get start_event and end_event elapsed time
            cnrtNotifierDuration(notifierBeginning, notifierEnd, &event_time_use); //Computes the time duration between starting and ending notifiers.
            cout << " hardware time: " << event_time_use << "\n";
        } else {
            cout << " SyncQueue Error " << "\n";
        }
        cout << num << "th inference." << "\n\n\n";
        //void** outPtrS = reinterpret_cast<void**>(malloc(sizeof(void*) * outputNum));
        void* temp_output_cpu_data = nullptr;
        for (int i = 0; i < outputNum; i++) {
            temp_output_cpu_data = (void*)malloc(outputSizeS[i]);
            cnrtMemcpy(temp_output_cpu_data,
                        outputMluPtrS[i],
                        outputSizeS[i],
                        CNRT_MEM_TRANS_DIR_DEV2HOST); //MLU->temp_output_cpu
            int output_count = outputSizeS[i] / cnrtDataTypeSize(output_data_type[i]);
            if (output_data_type[i] != CNRT_FLOAT32) {
                cnrtCastDataType(temp_output_cpu_data,
                                output_data_type[i],
                                outputCpuPtrS[i],
                                CNRT_FLOAT32,
                                output_count,
                                nullptr);
            } else {
                memcpy(outputCpuPtrS[i], temp_output_cpu_data, outputSizeS[i]); //temp_output_cpu_data -> outputCpuPtrS
            }
```

```
            auto databuf = reinterpret_cast<float*>(malloc(sizeof(float) * output_count));
            outPtrS[i] = reinterpret_cast<void*>(databuf);
            std::vector<int> shape(4, 1);
            int dimNum = 4;
            cnrtGetOutputDataShape((int**)&shape, &dimNum, i, function);
            int dim_order[4] = {0, 3, 1, 2};
            int dim_shape[4] = {shape[0], shape[1],
                                      shape[2], shape[3]};  // NHWC
            cnrtTransDataOrder(outputCpuPtrS[i], CNRT_FLOAT32, outPtrS[i],
                                     4, dim_shape, dim_order);
            if (temp_output_cpu_data) {
                    free(temp_output_cpu_data);
                    temp_output_cpu_data = nullptr;
            }
        }

        for (auto flo : output_cpu) {
          free(flo);
        }

        output_cpu.clear();
        //free memory space
        free(inputCpuPtrS);
        free(outputCpuPtrS);
        //free(outPtrS);
        for (int i = 0; i < inputNum; i++) {
                cnrtFree(inputMluPtrS[i]);
        }
        for (int i = 0; i < outputNum; i++) {
                cnrtFree(outputMluPtrS[i]);
        }

        cnrtDestroyQueue(cnrt_queue);
        cnrtDestroyFunction(function);
           cnrtUnloadModel(model);
          cnrtDestroy();

        return 0;
}
```

- infer.h:

```
/*
infer.h
```

```
*/

#ifndef _INFER_H_
#define _INFER_H_

int offline_infer(float *databuf, void** outPtrS, bool init, int num);

#endif
```

- CMakeLists.txt:

```
cmake_minimum_required (VERSION 2.8)

project (infer)

include_directories(/usr/local/neuware/include)
link_directories(/usr/local/neuware/lib64/)
link_libraries(cnrt)

set (SRC_LIST ${PROJECT_SOURCE_DIR}/infer/infer.cpp)

add_library (infer_shared SHARED ${SRC_LIST})
add_library (infer_static STATIC ${SRC_LIST})

set_target_properties (infer_shared PROPERTIES OUTPUT_NAME "infer")
set_target_properties (infer_static PROPERTIES OUTPUT_NAME "infer")

set (LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)
```

- 在build中执行 `cmake .. && make` 编译生成动态链接库与静态链接库libinfer.so libinfer.a

```
root@933a2b8c520a:/home/ict/test_mlu270/gen_so# tree
.
├── CMakeLists.txt
├── build
├── infer
│   ├── infer.cpp
│   └── infer.h
└── lib
    ├── libinfer.a
    └── libinfer.so
```

(3）socket网络通信、推理调试：

- 文件结构

```
root@933a2b8c520a:/home/ict/test_mlu270/newtry3/cnrt_test# tree
.
├── CMakeLists.txt
├── bin
│   └── democnrt
├── build
├── infer
│   ├── inc
│   │   └── infer.h
│   └── lib
│       ├── libinfer.a
│       └── libinfer.so
├── mlu220_run.tar.gz
├── model
│   ├── actor_mlu220_offline.cambricon
│   ├── actor_mlu220_offline.cambricon_twins
│   ├── actor_mlu270_offline.cambricon
│   └── actor_mlu270_offline.cambricon_twins
└── src
    ├── CMakeLists.txt
    └── main.cpp
```

- CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8 FATAL_ERROR)
project(demo_cnrt)

add_subdirectory(src)
```

- bin：用于存放编译链接后的可执行程序。
- build：用于存放编译后生成的相关内容。
- infer：
  - inc/infer.h：声明推理函数 `offline_infer`

```
/*
** infer.h
*/

#ifndef _INFER_H_
#define _INFER_H_
```

```

int offline_infer(float *databuf, void** outPtrS);

#endif
```

- lib/：存放生成的静态与动态链接库（上面步骤生成的直接拷贝到这里）。

- mlu220_run.tar.gz：最终要拷贝到MLU220中的文件打包。
- model：存放离线模型
- src：
  - CMakeLists.txt：要链接上面生成的动态链接库

```
# add cnrt
include_directories(/usr/local/neuware/include)
link_directories(/usr/local/neuware/lib64/)
link_libraries(cnrt)

# add infer
include_directories(${PROJECT_SOURCE_DIR}/infer/inc)
MESSAGE(STATUS "PROJECT_SOURCE_DIR=${PROJECT_SOURCE_DIR}")
link_directories(${PROJECT_SOURCE_DIR}/infer/lib)
link_libraries(infer)

aux_source_directory(. SRC_LIST)
add_executable(democnrt ${SRC_LIST})

set (EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)

# link
target_link_libraries(democnrt cnrt infer)
```

  - main.cpp：
    - socket_init：初始化网络链接，client链接后返回客户端标识符。
    - deal_recv_str：处理client发送的数据（字符串），处理为float数组，并返回float数组。
    - deal_send_str：处理server要发送的数据（数字），处理为char *字符串，并返回char *数组。
    - inference：推理函数，调用了deal_recv_str、deal_send_str和offline_infer，返回处理好的待发送数据。offline_infer在头文件 `#include "infer.h"` 中声明，因为在上边步骤中将其编译成了动态链接库，所以在这里直接调用（需要在CMakeLists.txt中链接）。

```
#include "cnrt.h"
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctime>
#include <vector>
#include <fstream>
#include <cstring>
#include <sys/fcntl.h>
#include <sys/socket.h>
#include <unistd.h>
#include <netinet/in.h>
#include <errno.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include "infer.h"
#include <string>

using namespace std;

int socket_init(int *fd) {
    int listenfd;
    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    if (listenfd == -1) {
        printf("socket create fail\n");
        return -1;
    }
    struct sockaddr_in serveraddr;
    memset(&serveraddr, 0, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serveraddr.sin_port = htons(8050); //specify port
    if (bind(listenfd, (struct sockaddr *)&serveraddr, sizeof(serveraddr)) != 0) {
        printf("bind failed\n");
        return -1;
    }

    if (listen(listenfd, 5) != 0) {
        printf("Listen failed\n");
        close(listenfd);
        return -1;
    }
    printf("...waiting for client...\n");

    int clientfd;
```

```
    int socklen = sizeof(struct sockaddr_in);
    struct sockaddr_in client_addr;
    clientfd = accept(listenfd, (struct sockaddr*)&client_addr, (socklen_t *)&socklen);
    if (clientfd == -1) {
        printf("connect failed\n");
    } else {
        printf("client %s has connected\n", inet_ntoa(client_addr.sin_addr));
    }
    fd[0] = listenfd;
    fd[1] = clientfd;

    return 0;
}

void deal_recv_str(char *recvBuf, float *data) {
    char *p;
    p = strtok(recvBuf, " ");
    string s;
    int i = 0;
    while (p != NULL) {
        s = p;
        data[i++] = stof(s);
//          cout << data[i] << "\n";
        p = strtok(NULL, " ");
    }
}

void deal_send_str(char *sendBuf, float *data, int length) {
    string sendstream;
    for (int i = 0; i < length; i++) {
        sendstream += to_string(data[i]) + " ";
    }
    strcpy(sendBuf, const_cast<char *>(sendstream.c_str()));
}

void inference(char *recvBuf, int input_len, char *sendBuf, int output_len, bool init, int num) {
    cout << "recvBufLen:" << strlen(recvBuf) << "\n";
    auto databuf = reinterpret_cast<float *>(malloc(sizeof(float) * input_len));
    deal_recv_str(recvBuf, databuf);
    //cout << "input:" << "\n";
    //for (int i = 0; i < input_len; i++) {
    //          cout << databuf[i] << "\n";
    //}
```

```

    int outputNum = 1;
    void** outPtrS = reinterpret_cast<void**>(malloc(sizeof(void*) * outputNum));
    offline_infer(databuf, outPtrS, init, num);
    //cout << "\noutput:\n";
    //for (int i = 0; i < outputNum; i++) {
    //        for (int j = 0; j < output_len; ++j) {
    //            cout << (reinterpret_cast<float*>(outPtrS[i]))[j] << "\n";
    //        }
    //}
    deal_send_str(sendBuf, reinterpret_cast<float*>(outPtrS[0]), 40);
    //cout << "sendBuf: " << sendBuf << "\n";
}

int main() {
        int *fd = (int *)malloc(sizeof(int) * 2);
        int ok_socket = socket_init(fd);
        int clientfd = fd[1], listenfd = fd[0];
        cout << "clienfd = " << clientfd << " listenfd = " << listenfd << "\n";
        if (ok_socket == -1) {
            printf("socket init unsuccessfully!\n");
            return 0;
        }

        int iret;
        char recvBuf[30000], sendBuf[30000];
        char looprecv[1025];
        memset(recvBuf, 0, sizeof(recvBuf));
        memset(sendBuf, 0, sizeof(sendBuf));
        bool init = true;
        int num = 1;
        while (true) {
            while (true) {
                iret = recv(clientfd, looprecv, 1024, 0); //waiting until new info
                strcat(recvBuf, looprecv);
                if (iret < 1024) {
                    break;
                }
                memset(looprecv, 0, sizeof(looprecv));
            }
            //cout << "recvBuf:" << recvBuf << "\n";
            int ip_len = 112, op_len = 40;
            inference(recvBuf, ip_len, sendBuf, op_len, init, num);
```

```
                //cout << "sendBuf:::" << sendBuf << "\n";
                if ((iret = send(clientfd, sendBuf, strlen(sendBuf), 0)) <= 0)
    {
                        perror("send");
                          break;
                }
                //printf("send:%s\n", sendBuf);
                memset(recvBuf, 0, sizeof(recvBuf));
                memset(sendBuf, 0, sizeof(sendBuf));
                memset(looprecv, 0, sizeof(looprecv));
                num++;
                init = false;
            }

            close(listenfd);
            close(clientfd);

            return 0;
    }
```

（4）交叉编译MLU220上运行的动态链接库：

- 修改代码：把infer.cpp中要加载的模型改为220的离线模型

```
cnrtLoadModel(&model, "../model/actor_mlu220_offline.cambricon");
```

- 目录结构

```
root@933a2b8c520a:/home/ict/test_mlu270/gen_220_so# tree
.
├── CMakeLists.txt
├── build
├── infer
│   ├── infer.cpp
│   └── infer.h
└── lib
    ├── libinfer.a
    └── libinfer.so
```

- CMakeLists.txt

```
cmake_minimum_required (VERSION 2.8)

project (infer)


SET(CMAKE_SYSTEM_NAME Linux)
SET(CMAKE_SYSTEM_PROCESSOR aarch64)

add_compile_options(-std=c++11)
SET(CMAKE_C_COMPILER "aarch64-linux-gnu-gcc")
SET(CMAKE_CXX_COMPILER "aarch64-linux-gnu-g++")

# add cnrt
include_directories(/usr/local/neuware/include)
link_directories(/usr/local/neuware220/neuware/lib64/) #mlu220 cnrt动态链接库的目录
link_libraries(cnrt)

# add aarch64-linux-gnu-gcc
include_directories(/usr/local/arm/gcc-linaro-6.2.1-2016.11-x86_64_aarch64-linux-gnu/include/)
link_directories(/usr/local/arm/gcc-linaro-6.2.1-2016.11-x86_64_aarch64-linux-gnu/aarch64-linux-gnu/lib64/)

set (SRC_LIST ${PROJECT_SOURCE_DIR}/infer/infer.cpp)

add_library (infer_shared SHARED ${SRC_LIST})
add_library (infer_static STATIC ${SRC_LIST})

set_target_properties (infer_shared PROPERTIES OUTPUT_NAME "infer")
set_target_properties (infer_static PROPERTIES OUTPUT_NAME "infer")


set (LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)
```

- 编译

```
cd build
cmake .. && make
"""
root@933a2b8c520a:/home/ict/test_mlu270/gen_220_so/build# cmake .. && make
-- The C compiler identification is GNU 7.3.0
-- The CXX compiler identification is GNU 7.3.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
```

```
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/ict/test_mlu270/gen_220_so/build
Scanning dependencies of target infer_shared
[ 25%] Building CXX object CMakeFiles/infer_shared.dir/infer/infer.cpp.o
[ 50%] Linking CXX shared library ../lib/libinfer.so
[ 50%] Built target infer_shared
Scanning dependencies of target infer_static
[ 75%] Building CXX object CMakeFiles/infer_static.dir/infer/infer.cpp.o
[100%] Linking CXX static library ../lib/libinfer.a
[100%] Built target infer_static
"""
```

- 查看编译结果

```
root@933a2b8c520a:/home/ict/test_mlu270/gen_220_so/lib# ls
libinfer.a  libinfer.so
root@933a2b8c520a:/home/ict/test_mlu270/gen_220_so/lib# file libinfer.so
libinfer.so: ELF 64-bit LSB shared object, ARM aarch64, version 1 (SYSV), dynamically linked, BuildID[sha1]=abccef39f928c103f32ee65236a70f7fd2801d82, not stripped
```

（5）交叉编译运行在MLU220上的程序

- 把编译好的动态链接库copy过来，文件目录结构如下：

```
root@933a2b8c520a:/home/ict/test_mlu270/mlu220_debug/cnrt_test# tree
.
├── CMakeLists.txt
├── bin
│   └── democnrt
├── build
├── infer
│   ├── inc
│   │   └── infer.h
│   └── lib
│       ├── libinfer.a
│       └── libinfer.so
├── mlu220_run.tar.gz
├── model
│   ├── actor_mlu220_offline.cambricon
│   ├── actor_mlu220_offline.cambricon_twins
│   ├── actor_mlu270_offline.cambricon
│   └── actor_mlu270_offline.cambricon_twins
└── src
    ├── CMakeLists.txt
    └── main.cpp
```

- CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8 FATAL_ERROR)
project(demo_cnrt)

SET(CMAKE_SYSTEM_NAME Linux)
SET(CMAKE_SYSTEM_PROCESSOR aarch64)

add_compile_options(-std=c++11)
SET(CMAKE_C_COMPILER "aarch64-linux-gnu-gcc")
SET(CMAKE_CXX_COMPILER "aarch64-linux-gnu-g++")

add_subdirectory(src)
```

- src/CMakeLists.txt

```
# add cnrt
include_directories(/usr/local/neuware/include)
link_directories(/usr/local/neuware220/neuware/lib64/) #mlu220 cnrt动态链接库的目录
link_libraries(cnrt)
```

```

# add aarch64-linux-gnu-gcc
include_directories(/usr/local/arm/gcc-linaro-6.2.1-2016.11-x86_64_aarch64-linux-gnu/include/)
link_directories(/usr/local/arm/gcc-linaro-6.2.1-2016.11-x86_64_aarch64-linux-gnu/aarch64-linux-gnu/lib64/)

# add infer
include_directories(${PROJECT_SOURCE_DIR}/infer/inc)
MESSAGE(STATUS "PROJECT_SOURCE_DIR=${PROJECT_SOURCE_DIR}")
link_directories(${PROJECT_SOURCE_DIR}/infer/lib)
link_libraries(infer)

aux_source_directory(. SRC_LIST)
add_executable(democnrt ${SRC_LIST})

set (EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)

# link
target_link_libraries(democnrt cnrt infer)
```

- 编译

```
cd build
cmake .. && make
"""
root@933a2b8c520a:/home/ict/test_mlu270/mlu220_debug/cnrt_test/build# cmake .. && make
-- The C compiler identification is GNU 7.3.0
-- The CXX compiler identification is GNU 7.3.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- PROJECT_SOURCE_DIR=/home/ict/test_mlu270/mlu220_debug/cnrt_test
-- Configuring done
-- Generating done
```

```
-- Build files have been written to:
/home/ict/test_mlu270/mlu220_debug/cnrt_test/build
Scanning dependencies of target democnrt
[ 50%] Building CXX object src/CMakeFiles/democnrt.dir/main.cpp.o
[100%] Linking CXX executable ../../bin/democnrt
[100%] Built target democnrt
"""
```

- 查看结果

```
root@933a2b8c520a:/home/ict/test_mlu270/mlu220_debug/cnrt_test/build# file ../bin/democnrt
../bin/democnrt: ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-aarch64.so.1, for GNU/Linux 3.7.0, BuildID[sha1]=35772f94d197c46a9ad135eaf6dbb4e70288a418, not stripped
```

（6）打包、拷贝

- 在X86服务器上，把生成的可执行文件democnrt、离线模型，以及上面的动态链接库libinfer.so（跟可执行文件一块放到bin目录下即可）打包为mlu220_run.tar.gz，命令如图所示：

```
root@933a2b8c520a:/home/ict/test_mlu270/mlu220_debug/cnrt_test# tar -czvf mlu220_run.tar.gz model bin
model/
model/actor_mlu220_offline.cambricon_twins
model/actor_mlu220_offline.cambricon
bin/
bin/democnrt
```

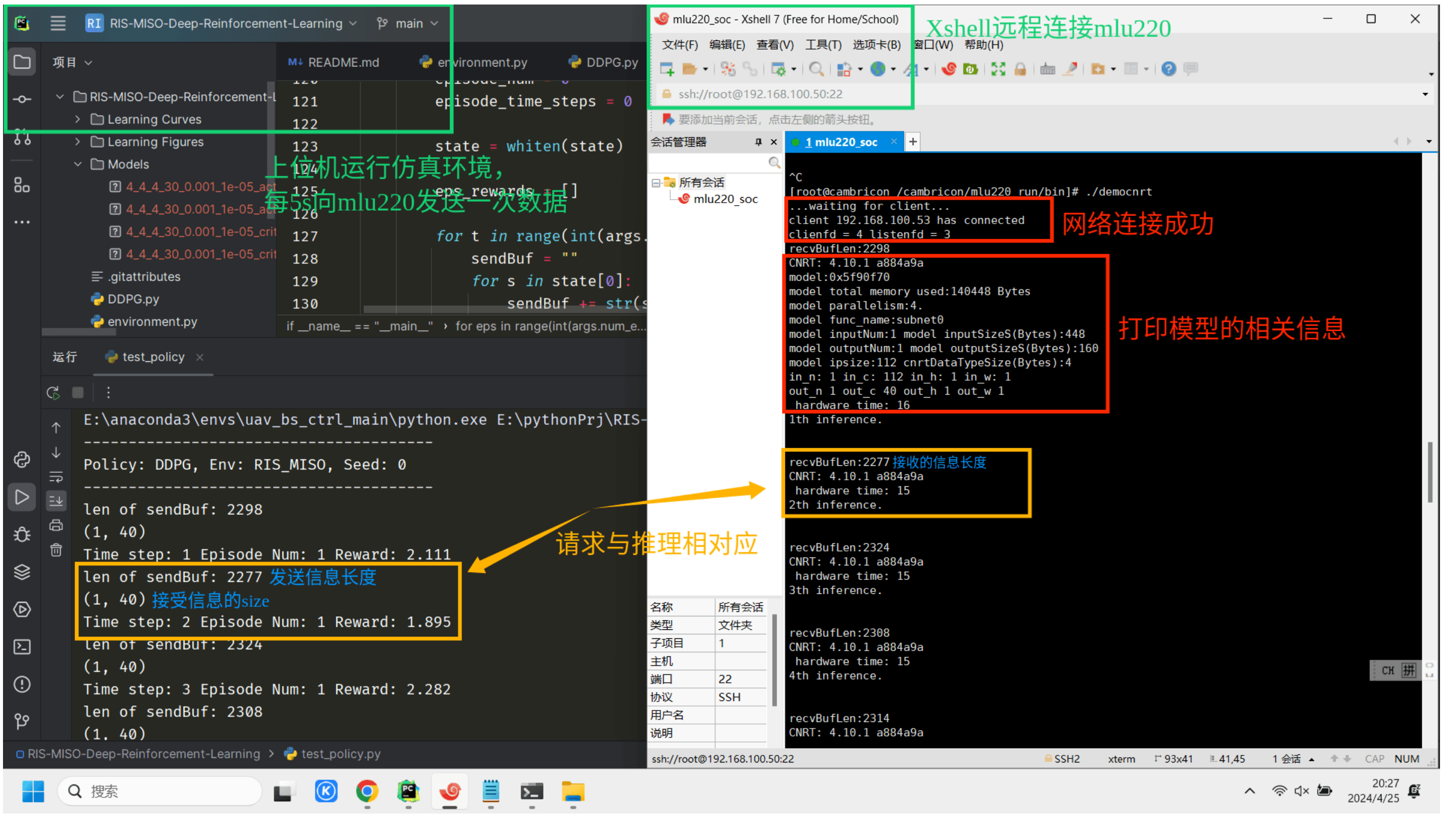
- 在上位机通过ssh拷贝到MLU220的/cambricon目录下，防止掉电丢失，在cmd中运行下面命令。

```
scp -r G:\mlu220_run.tar.gz root@192.168.100.50:/cambricon #宿主机 -> mlu220
```

- 在MLU220中，解压，运行aarch64可执行文件。

```
cd /cambricon
mv mlu220_run.tar.gz ./mlu220_run
cd ./mlu220_run
tar -xzvf ./mlu220_run.tar.gz
cd bin
./democnrt #运行
```

（7）结果展示：



2.2 上位机（作为client）：

使用Python运行强化学习环境（state），向server通过http请求的方式发送数据，接收server返回的结果（action）与环境交互。

- 在于MLU270或者MLU220交互的时候要记得修改为相应的IP。

2.3 MLU220（server）：

MLU220仅作为一个运行程序的平台，不做开发，开发全部在x86服务器完成。

七、完善

1、解决算子不适配的问题。

1.1 问题

因为在调用GRUCell时，报错提示addmm与sigmoid_不支持，但是在mlu_functions.yaml中均已经声明。

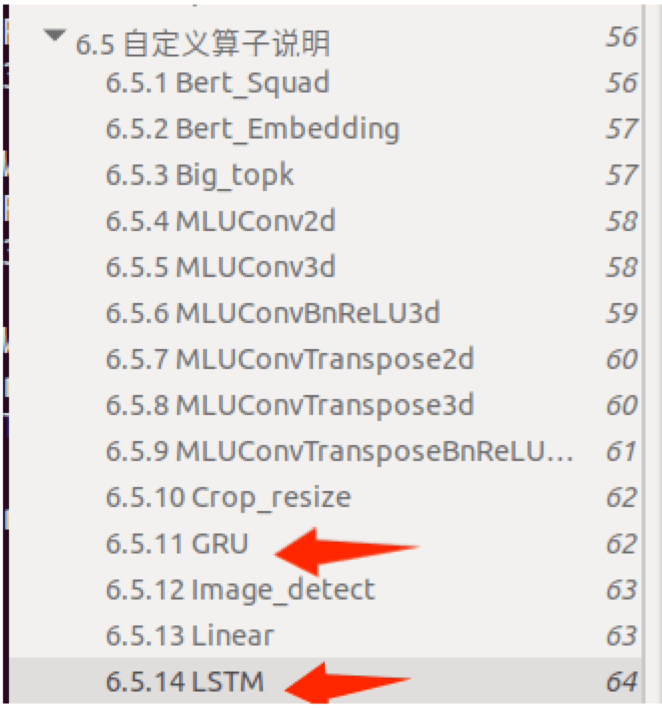
```
[WARNING][/pytorch/catch/torch_mlu/csrc/jit/passes/segment_graph.cpp][line:41][MLUSupport][thread:140363448362752][process:8786]:
[Fusion Segment] Please check mlu_functions.yaml && Maybe MLU fusion does NOT supports op: addmm
%21 : Float(*, *) = aten::addmm(%7, %hx, %18, %9, %9), scope: RnnAgent/GRUCell[grucell] # /torch/venv3/pytorch/lib/python3.6/site-packages/torch/nn/modules/rnn.py:1023:0

[WARNING][/pytorch/catch/torch_mlu/csrc/jit/passes/segment_graph.cpp][line:41][MLUSupport][thread:140363448362752][process:8786]:
[Fusion Segment] Please check mlu_functions.yaml && Maybe MLU fusion does NOT supports op: sigmoid_
%30 : Tensor = aten::sigmoid_(%29), scope: RnnAgent/GRUCell[grucell] # /torch/venv3/pytorch/lib/python3.6/site-packages/torch/nn/modules/rnn.py:1023:0
```

mlu_functions.yaml中两个算子的声明。

```yaml
- name: addmm
  use_mlu_dispatcher: unboxed_only
  derived_type: cnnl
  schema_string: aten::addmm(Tensor self, Tensor mat1, Tensor mat2, *, Scalar beta=1, Scalar alpha=1) -> Tensor
  arguments:
  - name: self
    type: const at::Tensor &
  - name: mat1
    type: const at::Tensor &
  - name: mat2
    type: const at::Tensor &
  - name: beta
    type: at::Scalar
  - name: alpha
    type: at::Scalar
  return_type: at::Tensor
```

```yaml
- name: sigmoid_
  use_mlu_dispatcher: unboxed_only
  derived_type: cnnl
  schema_string: aten::sigmoid_(Tensor(a!) self) -> Tensor(a!)
  arguments:
  - name: self
    type: at::Tensor &
  return_type: at::Tensor &
```

在pytorch中，RNN有非常多的变体，比如RNN、LSTM、LSTMCell、GRU、GRUCell，在文档中明确说了已经自定义了GRU和LSTM：



经过测试了一下GRU、LSTM，发现是可行的，所以原因就是MLU不支持GRUCell，在**算子分发**的时候出现了问题：

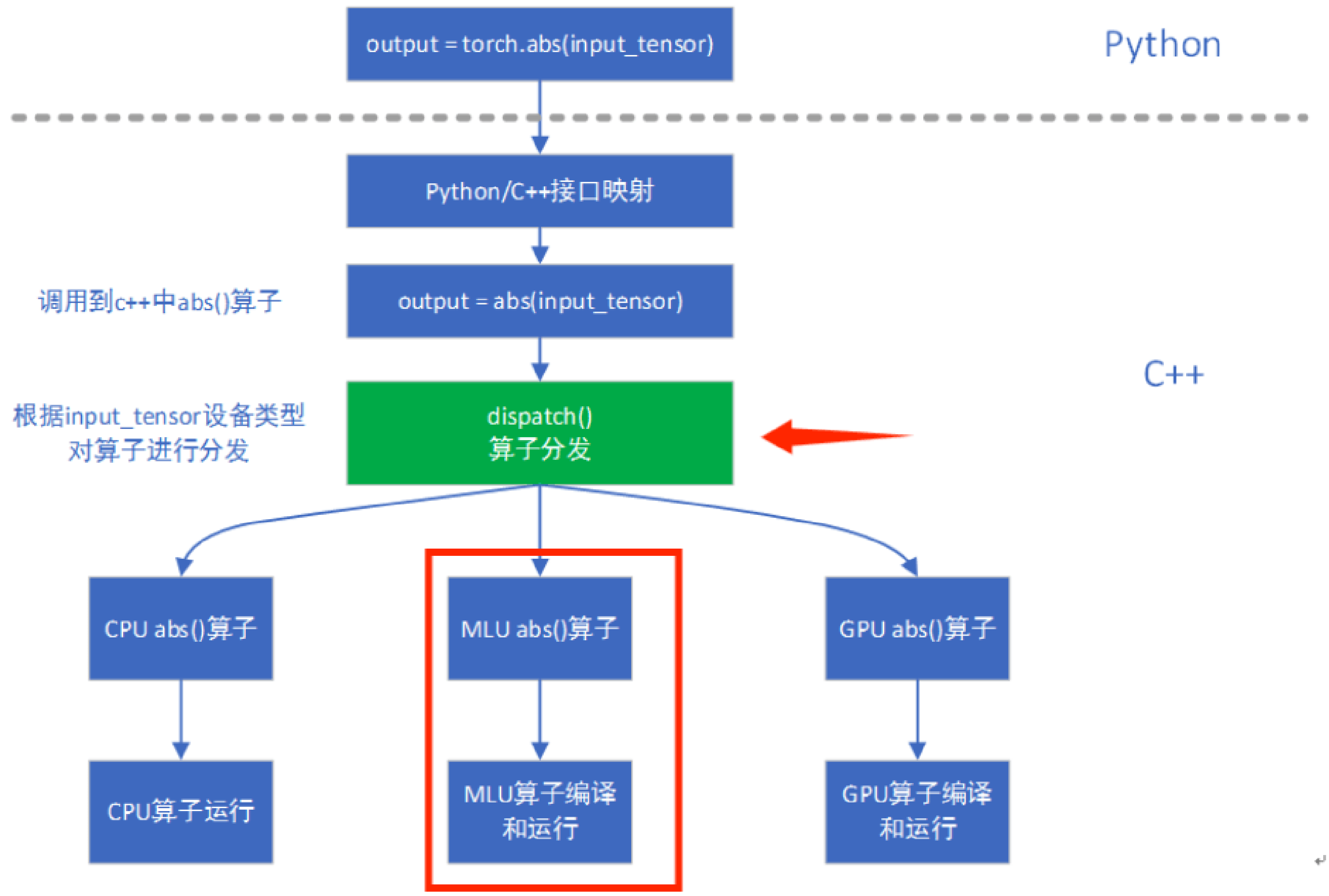


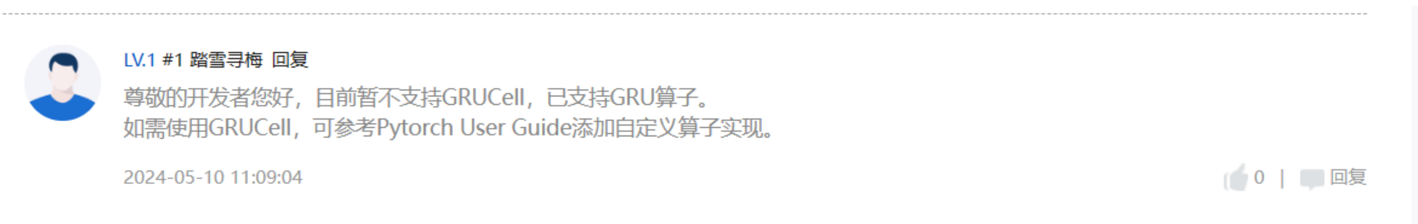
图 7.2: 逐层算子分发图

但是对于mlu不支持算子的处理有两种情况：

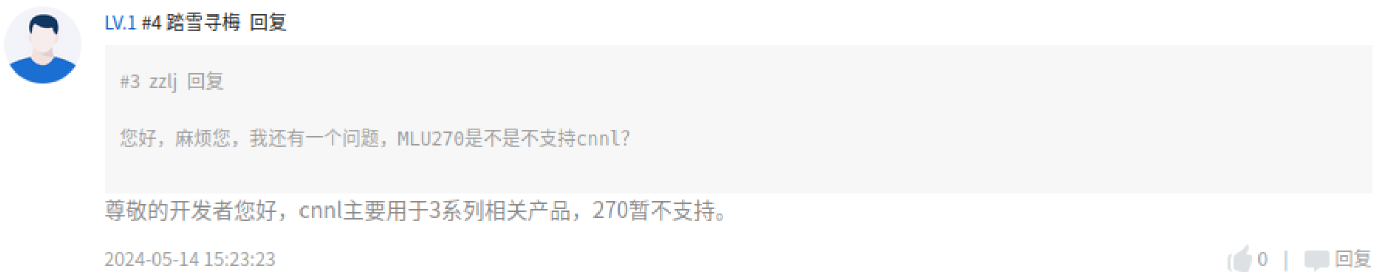
对于 MLU 暂不支持的算子，如果未在 `catch/torch_mlu/tools/mlu_functions.yaml` 中声明，程序会直接终止，并抛出无法分发到 MLU 设备的异常；如果已经在 `catch/torch_mlu/tools/mlu_functions.yaml` 中声明但在运行至 wrapper 或 kernel 时失败，输入数据将会拷贝到 CPU 上，然后调用 CPU 相关算子，使其在 CPU 上运行，最后再将输出结果拷回到 MLU 上。具体实现，可以查询 op_methods.cpp，该文件在 `catch/torch_mlu/csrc/aten/operators/` 目录下。

应该不是第一种，因为已经在mlu_functions.yaml文件中声明了，说明算子成功分发到mlu上，而从mlu_function.yaml中由发现两个算子的派生类型是**CNNL**，偶然在论坛里看到有个帖子(https://forum.cambricon.com/index.php?m=content&c=index&a=show&catid=156&id=2974)说了MLU270不支持CNNL，所以才猜测是不是mlu270不支持CNNL的原因。

在寒武纪论坛询问得到官方人员回复，不支持GRUCell。


LV.1 #1 踏雪寻梅 回复
尊敬的开发者您好，目前暂不支持GRUCell，已支持GRU算子。
如需使用GRUCell，可参考Pytorch User Guide添加自定义算子实现。
2024-05-10 11:09:04
0 | 回复

并且不支持寒武纪神经网络计算库CNNL（3系列才支持），只支持寒武纪机器学习库CNML，因此在所有在mlu_functions.yaml中声明的，派生类型(derived_type)里不包CNML的的算子都无法使用：



1.2 解决办法：

（1）根据GRUCell的计算公式，把GRUCell拆成寒武纪支持的算子，直接在代码文件中利用寒武纪支持的算子实现一个GRUCell。

- 优点：方便、简单
- 缺点：由python实现，效率比较低，但是完全满足科研需求。

a. 根据pytorch 1.3.0 [官方文档](https://pytorch.org/docs/1.3.0/nn.html?highlight=grucell#torch.nn.GRUCell)中的公式，把forward实现：

A gated recurrent unit (GRU) cell

$$
\begin{aligned}
r &= \sigma(W_{ir}x + b_{ir} + W_{hr}h + b_{hr}) \\
z &= \sigma(W_{iz}x + b_{iz} + W_{hz}h + b_{hz}) \\
n &= \tanh(W_{in}x + b_{in} + r * (W_{hn}h + b_{hn})) \\
h' &= (1 - z) * n + z * h
\end{aligned}
$$

where σ is the sigmoid function, and $*$ is the Hadamard product.

b. 根据要求初始化权重和偏置：

Variables

- **~GRUCell.weight_ih** – the learnable input-hidden weights, of shape (3**hidden_size, input_size*)
- **~GRUCell.weight_hh** – the learnable hidden-hidden weights, of shape (3**hidden_size, hidden_size*)
- **~GRUCell.bias_ih** – the learnable input-hidden bias, of shape (3**hidden_size*)
- **~GRUCell.bias_hh** – the learnable hidden-hidden bias, of shape (3**hidden_size*)

• NOTE

All the weights and biases are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{hidden\_size}}$

c. 代码实现：

```
class GRUCell(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(GRUCell1, self).__init__()
        stdv = 1.0 / math.sqrt(hidden_size)
        self.in2hid_w = nn.ParameterList([self.__init(stdv, input_size, hidden_size) for _ in range(3)])
        self.hid2hid_w = nn.ParameterList([self.__init(stdv, hidden_size, hidden_size) for _ in range(3)])
        self.in2hid_b = nn.ParameterList([self.__init(stdv, hidden_size) for _ in range(3)])
        self.hid2hid_b = nn.ParameterList([self.__init(stdv, hidden_size) for _ in range(3)])

    @staticmethod
    def __init(stdv, dim1, dim2=None):
        if dim2 is None:
            return nn.Parameter(nn.init.uniform_(torch.Tensor(dim1), -stdv, stdv))  # 按照官方的初始化方法来初始化网络参数
        else:
            return nn.Parameter(nn.init.uniform_(torch.Tensor(dim1, dim2), -stdv, stdv))

    def forward(self, x, hid):
        r = torch.sigmoid(torch.mm(x, self.in2hid_w[0]) + self.in2hid_b[0] + torch.mm(hid, self.hid2hid_w[0]) + self.hid2hid_b[0])
        z = torch.sigmoid(torch.mm(x, self.in2hid_w[1]) + self.in2hid_b[1] + torch.mm(hid, self.hid2hid_w[1]) + self.hid2hid_b[1])
        n = torch.tanh(torch.mm(x, self.in2hid_w[2]) + self.in2hid_b[2] + torch.mul(r, (torch.mm(hid, self.hid2hid_w[2]) + self.hid2hid_b[2])))
        next_hid = torch.mul(-(z - 1), n) + torch.mul(z, hid)

```

```
        return next_hid
```

除此之外，为了保证算子结果准确性，初始化的时候可以直接把官方算子GRUCell初始化的权重和偏执直接导入到自己写的算子中，代码如下：

```python
def init_rnn_wb(self):
    """load initial weights and bias from official torch.nn.grucell"""
    net = torch.nn.GRUCell(self._hidden_size, self._hidden_size)
    p = self.rnn.state_dict()
    print(net.state_dict()['weight_ih'].shape)
    p['in2hid_w.0'] = net.state_dict()['weight_ih'][0:self._hidden_size, :].transpose(0, 1)
    p['in2hid_w.1'] = net.state_dict()['weight_ih'][self._hidden_size:self._hidden_size*2, :].transpose(0, 1)
    p['in2hid_w.2'] = net.state_dict()['weight_ih'][self._hidden_size*2:self._hidden_size*3, :].transpose(0, 1)

    p['hid2hid_w.0'] = net.state_dict()['weight_hh'][0:self._hidden_size, :].transpose(0, 1)
    p['hid2hid_w.1'] = net.state_dict()['weight_hh'][self._hidden_size:self._hidden_size*2, :].transpose(0, 1)
    p['hid2hid_w.2'] = net.state_dict()['weight_hh'][self._hidden_size*2:self._hidden_size*3, :].transpose(0, 1)

    p['in2hid_b.0'] = net.state_dict()['bias_ih'][0:self._hidden_size]
    p['in2hid_b.1'] = net.state_dict()['bias_ih'][self._hidden_size:self._hidden_size*2]
    p['in2hid_b.2'] = net.state_dict()['bias_ih'][self._hidden_size*2:self._hidden_size*3]

    p['hid2hid_b.0'] = net.state_dict()['bias_hh'][0:self._hidden_size]
    p['hid2hid_b.1'] = net.state_dict()['bias_hh'][self._hidden_size:self._hidden_size*2]
    p['hid2hid_b.2'] = net.state_dict()['bias_hh'][self._hidden_size*2:self._hidden_size*3]
    self.rnn.load_state_dict(p)
```

（2）使用Bang C写一个C++算子，然后修改算子调用过程中的各种文件。

- 优点：效率高，符合工业界要求。
- 缺点：比较麻烦。

1.3 使用上述解决方法1出现新的问题：

- 高维矩阵乘法算子torch.matmul不支持：

```
[WARNING][/pytorch/catch/torch_mlu/csrc/jit/passes/segment_graph.cpp][line:41][MLUSupport][thread:140439080888064][process:2157]:
[Fusion Segment] Please check mlu_functions.yaml && Maybe MLU fusion does NOT supports op: matmul
%28 : Float(*, *) = aten::matmul(%x, %5), scope: GRUCell1 # test_mlu.py:32:0

[WARNING][/pytorch/catch/torch_mlu/csrc/jit/passes/segment_graph.cpp][line:41][MLUSupport][thread:140439080888064][process:2157]:
[Fusion Segment] Please check mlu_functions.yaml && Maybe MLU fusion does NOT supports op: matmul
%37 : Float(*, *) = aten::matmul(%x, %6), scope: GRUCell1 # test_mlu.py:33:0
```

暂时不用解决，不要需要。

- z是一个20*30规模的矩阵，由公式中可以看到有1 - z运算，这样就会调用rsub算子，导致不兼容：

```
[WARNING][/pytorch/catch/torch_mlu/csrc/jit/passes/segment_graph.cpp][line:41][MLUSupport][thread:140439080888064][process:2157]:
[Fusion Segment] Please check mlu_functions.yaml && Maybe MLU fusion does NOT supports op: rsub
%49 : Tensor = aten::rsub(%z, %20, %20), scope: GRUCell1 # /torch/venv3/pytorch/lib/python3.6/site-packages/torch/tensor.py:355:0

[WARNING][/pytorch/catch/torch_mlu/csrc/jit/passes/segment_graph.cpp][line:41][MLUSupport][thread:140439080888064][process:2157]:
[Fusion Segment] Please check mlu_functions.yaml && Maybe MLU fusion does NOT supports op: rsub
%49 : Tensor = aten::rsub(%z, %20, %20), scope: GRUCell1 # /torch/venv3/pytorch/lib/python3.6/site-packages/torch/tensor.py:355:0
```

解决问题：因为1是标量，z是张量，pytorch为了方便处理调用了sub的反向算子rsub算子，所以可以改为 `next_hid = torch.mul(-(z-1), n) + torch.mul(z, hid)` 即可。

1.4 生成并且测试离线模型，代码如下：

```python
# encoding: utf-8
import torch

import torch.nn as nn

import math

import torch_mlu
import torch_mlu.core.mlu_model as ct
import torch_mlu.core.mlu_quantize as mlu_quantize
import torchvision.models as models

ct.set_core_number(4)
ct.set_core_version("MLU270")
torch.set_grad_enabled(False)

class GRUCell(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(GRUCell, self).__init__()
        stdv = 1.0 / math.sqrt(hidden_size)
        self.weight_ih = nn.Parameter(nn.init.uniform_(torch.Tensor(3 * hidden_size, input_size), -stdv, stdv))
        self.in2hid_w = nn.ParameterList([self.__init(stdv, input_size, hidden_size) for _ in range(3)])
```

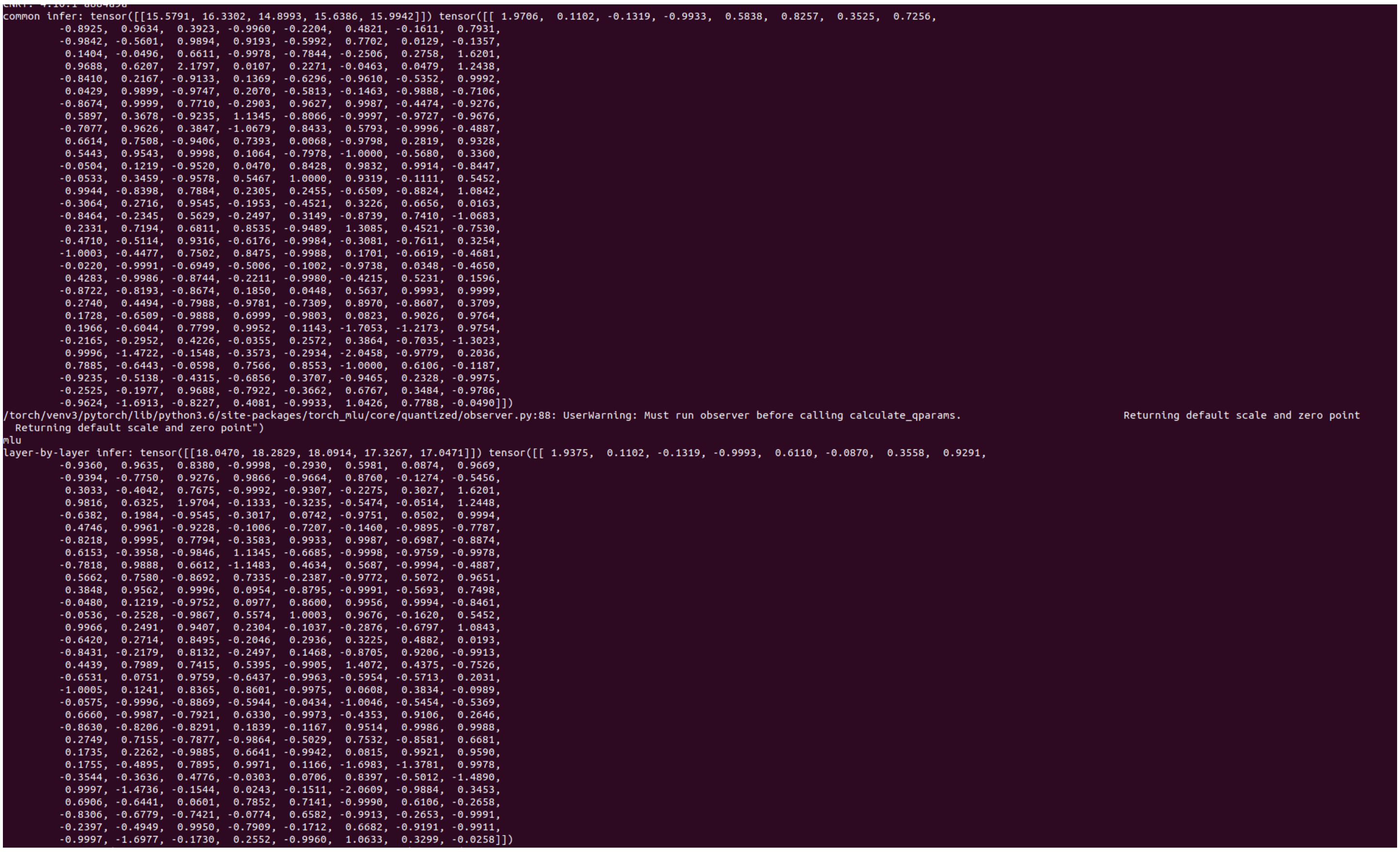
```
        self.hid2hid_w = nn.ParameterList([self.__init(stdv, hidden_size, hidden_size) for _ in range(3)])
        self.in2hid_b = nn.ParameterList([self.__init(stdv, hidden_size) for _ in range(3)])
        self.hid2hid_b = nn.ParameterList([self.__init(stdv, hidden_size) for _ in range(3)])

    @staticmethod
    def __init(stdv, dim1, dim2=None):
        if dim2 is None:
            return nn.Parameter(nn.init.uniform_(torch.Tensor(dim1), -stdv, stdv))  # 按照官方的初始化方法来初始化网络参数
        else:
            return nn.Parameter(nn.init.uniform_(torch.Tensor(dim1, dim2), -stdv, stdv))

    def forward(self, x, hid):
        r = torch.sigmoid(torch.mm(x, self.in2hid_w[0]) + self.in2hid_b[0] + torch.mm(hid, self.hid2hid_w[0]) + self.hid2hid_b[0])
        z = torch.sigmoid(torch.mm(x, self.in2hid_w[1]) + self.in2hid_b[1] + torch.mm(hid, self.hid2hid_w[1]) + self.hid2hid_b[1])
        n = torch.tanh(torch.mm(x, self.in2hid_w[2]) + self.in2hid_b[2] + torch.mul(r, (torch.mm(hid, self.hid2hid_w[2]) + self.hid2hid_b[2])))
        next_hid = torch.mul(-(z - 1), n) + torch.mul(z, hid)

        return next_hid

class RnnAgent(nn.Module):
    def __init__(self, obs_shape, n_actions, hidden_size, n_layers):
        super(RnnAgent, self).__init__()

        self._n_layers = n_layers
        self._hidden_size = hidden_size

        layers = [nn.Linear(obs_shape, self._hidden_size), nn.ReLU()]
        for l in range(self._n_layers - 1):
            layers += [nn.Linear(self._hidden_size, self._hidden_size), nn.ReLU()]

        self.enc = nn.Sequential(*layers)
        self.rnn = GRUCell(self._hidden_size, self._hidden_size)
        self.init_rnn_wb()
        self.f_out = nn.Linear(self._hidden_size, n_actions)

    def init_hidden(self):
        return torch.zeros(1, self._hidden_size)

```

```python
    def init_rnn_wb(self):
        """load initial weights and bias from official torch.nn.grucell"""
        net = torch.nn.GRUCell(self._hidden_size, self._hidden_size)
        p = self.rnn.state_dict()
        # print(net.state_dict()['weight_ih'].shape)
        p['in2hid_w.0'] = net.state_dict()['weight_ih'][0:self._hidden_size, :].transpose(0, 1)
        p['in2hid_w.1'] = net.state_dict()['weight_ih'][self._hidden_size:self._hidden_size*2, :].transpose(0, 1)
        p['in2hid_w.2'] = net.state_dict()['weight_ih'][self._hidden_size*2:self._hidden_size*3, :].transpose(0, 1)

        p['hid2hid_w.0'] = net.state_dict()['weight_hh'][0:self._hidden_size, :].transpose(0, 1)
        p['hid2hid_w.1'] = net.state_dict()['weight_hh'][self._hidden_size:self._hidden_size*2, :].transpose(0, 1)
        p['hid2hid_w.2'] = net.state_dict()['weight_hh'][self._hidden_size*2:self._hidden_size*3, :].transpose(0, 1)

        p['in2hid_b.0'] = net.state_dict()['bias_ih'][0:self._hidden_size]
        p['in2hid_b.1'] = net.state_dict()['bias_ih'][self._hidden_size:self._hidden_size*2]
        p['in2hid_b.2'] = net.state_dict()['bias_ih'][self._hidden_size*2:self._hidden_size*3]

        p['hid2hid_b.0'] = net.state_dict()['bias_hh'][0:self._hidden_size]
        p['hid2hid_b.1'] = net.state_dict()['bias_hh'][self._hidden_size:self._hidden_size*2]
        p['hid2hid_b.2'] = net.state_dict()['bias_hh'][self._hidden_size*2:self._hidden_size*3]
        self.rnn.load_state_dict(p)

    def forward(self, obs, h):
        x = self.enc(obs)
        h = self.rnn(x, h)
        x = self.f_out(h)
        return x, h

def setup_seed(seed):
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

if __name__ == '__main__':
    # 设置随机数种子
```

```
    setup_seed(4)

    rnn_agent = RnnAgent(obs_shape=82, n_actions=5, hidden_size=256, n_layers=2)
    model = torch.load('./checkpoint_ep50_unzip.pth',
                       map_location='cpu')
    rnn_agent.load_state_dict(model, False)
    rnn_agent.eval().float()
    input_obs = torch.randn(1, 82)
    input_h = torch.randn(1, 256)
    # common infer
    x, h = rnn_agent(input_obs, input_h)
    # print(x.shape)
    # print(type(x))
    # print(h.shape)
    # print(type(h))
    print("common infer:", x, h)
    # quantization and save quantization model
    torch.set_grad_enabled(False)
    mean = [0]
    std = [1/255]
    rnn_agent_quantization = mlu_quantize.quantize_dynamic_mlu(rnn_agent, {'mean':mean, 'std':std, 'firstconv':True}, dtype='int8', gen_quant=True)
    torch.save(rnn_agent_quantization.state_dict(), 'rnn_agent_quantization.pth')
    print(ct.mlu_device())
    # load quantization model
    rnn_agent_quantization = mlu_quantize.quantize_dynamic_mlu(rnn_agent)
    rnn_agent_quantization.load_state_dict(torch.load('rnn_agent_quantization.pth'), False)
    # print(rnn_agent_quantization.state_dict())
    # send data and model to mlu_device
    rnn_agent_quantization.to(ct.mlu_device())
    input_obs_mlu = input_obs.to(ct.mlu_device())
    input_h_mlu = input_h.to(ct.mlu_device())
    # layer-by-layer infer:
    x, h = rnn_agent_quantization(input_obs_mlu, input_h_mlu)
    print("layer-by-layer infer:", x.cpu(), h.cpu())
    # generate offline model
    ct.save_as_cambricon('rnn_agent270')
    input_data = (input_obs_mlu, input_h_mlu)
    trace_model = torch.jit.trace(rnn_agent_quantization, input_data, check_trace=False)
    # fus infer:
    x, h = trace_model(input_obs_mlu, input_h_mlu)
    print('fus infer:', x.cpu(), h.cpu())
```

```python

    # sim quant infer
    from torch_mlu.core.utils import sim_quant_utils
    rnn_agent = RnnAgent(obs_shape=82, n_actions=5, hidden_size=256, n_layers=2)
    rnn_agent_quantization = mlu_quantize.quantize_dynamic_mlu(rnn_agent, {'mean':mean, 'std':std, 'firstconv':True}, dtype='int8', gen_quant=True)
    sim_quant_utils.register_quant_hook(rnn_agent_quantization)
    rnn_agent_quantization.load_state_dict(torch.load('rnn_agent_quantization.pth'), False)
    x, h = rnn_agent_quantization(input_obs, input_h)
    print("sim quantization infer:", x, h)
```

经过测试发现，**模型量化后的CPU推理与MLU逐层推理差异较大，图如下（上边是CPU推理，下边是MLU逐层推理）：**



1.5

2、使用http传输数据

2.1 客户端使用http

客户端用python实现，可以直接安装requests库即可。

2.2 服务端使用http

> 使用httplib开源库，github地址：https://github.com/yhirose/cpp-httplib?tab=readme-ov-file#server-multi-threaded

这是一个header-only library，因此可以直接下载头文件放到项目中使用即可。

3、使用json

3.1 服务端使用json

> cpp没有原生的json库，所以需要使用开源的库，这里选用了jsoncpp，github地址：https://github.com/open-source-parsers/jsoncpp

（1）下载源码然后按照如下步骤编译生成静态和动态连接库即可：

```
mkdir jsoncpp-Sandbox
cd jsoncpp-sandbox
git clone https://github.com/open-source-parsers/jsoncpp.git # or download & unpack the source tarball
mkdir jsoncpp-build
# this will create the following directory structure:
# jsoncpp-Sandbox
#   +--jsoncpp
#   +--jsoncpp-build

Then you can proceed to configure and build
by using the following commands

cd jsoncpp-build
cmake ../jsoncpp # or ccmake, or cmake-gui
make
```

（2）结果：用的就是libjsoncpp.a和libjsoncpp.so：