《智能计算系统》第三次实验

本次实验分为三部分：基于 Pytorch 的实时图像风格迁移的推断，CPU 上自定义 Pytorch 算子并与图像风格迁移集成，DLP 上自定义 Pytorch 算子，分别对应实验指导书的 4.2，4.4 和 5.1 节。完成前两个实验时，创建开发镜像选择 v4，完成第三个实验时，创建开发镜像选择 v5_mul。

一、基于 Pytorch 的实时图像风格迁移推断

本实验对应指导书的 4.2 节，实验代码位于/opt/code_chap_4_student/exp_4_2_fast_style_transfer_infer_student，其中 stu_upload/evaluate_cpu.py 和 evaluate_cnnl_mfus.py 分别是需要补充的 CPU 和 DLP 平台上推断的代码。请补全代码，然后执行 bash run_cpu.sh 和 bash run_mlu.sh 分别运行实验。
其中 COCODataSet 数据集读取部分如下：

```
# 使用 cv2.imdecode()函数从指定的内存缓存中读取数据，并把数据转换(解码)成彩色图像格式。
image = cv2.imdecode(image, cv2.IMREAD_COLOR)
# 使用 cv2.resize()将图像缩放为 512*512 大小，其中所采用的插值方式为：区域插值
image = cv2.resize(image, (512, 512), interpolation=cv2.INTER_AREA)
# 使用 cv2.cvtColor 将图片从 BGR 格式转换成 RGB 格式
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
# 将 image 从 numpy 形式转换为 torch.float32,并将其归一化为[0,1]
image = torch.from_numpy(image).float() / 255
# 用 permute 函数将 tensor 从 HxWxC 转换为 CxHxW
image = image.permute(2, 0, 1)
```

主函数的推断过程部分如下：

```
# 使用 cpu 生成图像转换网络模型并保存在 g_net 中
g_net = TransNet().cpu()
# 从/models 文件夹下加载网络参数到 g_net 中
g_net.load_state_dict(torch.load('./models/fst.pth'))
# (dlp)将 g_net 模型转化为 eval,并转化为浮点类型，输出得到 net
net = g_net.eval().float()
# (dlp)使用 JIT 对 net 模型进行 trace，得到 net_trace
net_traced = torch.jit.trace(net,example_forward_input)
# (dlp)将 image_c 图片拷贝到 MLU 设备，得到 input_image_c
input_image_c = image_c.to("mlu")
# (dlp)将 net_trace 模型拷贝到 MLU 设备，得到 net_mlu
net_mlu = net_traced.to("mlu")
# (dlp)对 input_image_c 计算 net_mlu,得到 image_g_mlu
image_g_mlu = net_mlu(input_image_c)
# (cpu)计算 g_net,得到 image_g
image_g = g_net(image_c)
# 利用 save_image 函数将 tensor 形式的生成图像 image_g_mlu 以及输入图像 image_c 以 jpg 格式左右拼接的形式保存在/out/mlu_cnnl_mfus/文件夹下
save_image([image_g_mlu[0],image_c[0]], f'./out/mlu_cnnl_mfus/{i}.jpg', padding=0,
normalize=True, range=(0, 1))
```

推断的主体部分，关于神经网络的组成以及各个结构的参数，请参考实验指导书第 4.2.2.1 节的介绍，以及 Pytorch 的官方文档：https://pytorch.org/docs/stable/index.html
神经网络中可能会用到的各个模块对应的 torch.nn 中的基本模块分别为：

nn.Conv2d	二维卷积层
nn.InstanceNorm2d	二维实例归一化层
nn.ReLU	ReLU 激活函数层
nn.Upsample	上采样层
nn.Sigmoid	Sigmoid 激活函数层

二、CPU 上自定义 Pytorch 算子实验
本实验对应指导书的 4.4 节，实验代码位于/opt/code_chap_4_student/exp_4_4_custom_pytorch_op_student。实验步骤如下：

1. 算子 C++实现
补充 stu_upload/op_hsigmoid/hsigmoid.cpp 中的内容，关于 torch::Tensor 的使用，可参考 Pytorch 的 C++文档中关于 at::Tensor 的部分：https://pytorch.org/cppdocs/api/classat_1_1_tensor.html
pybind11 的基本使用方法见下：

```
PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
    m.def("function_name", &function_implementation, "Function description");
}
```

其中三个参数分别代表函数在 python 中的名称、函数的 C++实现、函数描述。

2.算子编译
补全 op_hsigmoid/setup.py，内容如下：

```
setup(
    name='op_exp', # 这里填写包的名称
    ext_modules=[
        cpp_extension.CppExtension(
            'op_exp', # python 模块的名称
            ['hsigmoid.cpp'] # c++源文件
        )
    ],
    cmdclass={
        'build_ext': cpp_extension.BuildExtension
    }
)
```

然后在 op_hsigmoid 目录下执行 python setup.py build_ext --inplace 编译算子。

3.算子测试

补全 test_hsigmoid.py 中的内容，执行 python test_hsigmoid.py 进行算子测试。

4.模型推断
补全 evaluate_cpu.py 中的内容，其中大部分内容参照上一个实验，只需要导入自己实现的 op_exp 模块，并在执行 sigmoid 的步骤中，将原本的 sigmoid 算子替换为自定义的 hsigmoid 算子即可。执行 python evaluate_cpu.py 进行模型推断。

三、DLP 上自定义算子实验
本实验对应指导书的 5.1 节，需要使用 BANG 语言开发能够在 DLP 上运行的自定义算子。BANG C 语言是 C/C++的一个变体（类似 cuda/hip），用于开发在异构硬件上的任务。实验的代码位于/opt/code_chap_5_student / exp_5_1_custom_pytorch_mlu_op
实验步骤如下：

1. 实现核函数
补充 BANG C 代码 mlu_custom_ext/mlu/src/bang_sigmoid_sample.mlu。
VSCode 插件 CNStudio 可以为.mlu 文件实现解析，可参考指导书或手册安装。BANG C 开发者手册的 5.1 节提供了异构编程的介绍，第 6 章提供了 BANG C 的详细介绍，可参考文档补充代码。

2. 实现 Sigmoid 主程序
补充代码 mlu_custom_ext/mlu/src/bang_sigmoid.cpp。这个 C++程序用来将调用 BANG C 的程序，并使用 pybind 绑定进 python 库。

3. 实现 python 封装
补充代码 mlu_custom_ext/mlu_functions/mlu_functions.py。这个文件实现了对自定义 sigmoid 的 pytorch 封装。

4. 算子编译
执行 python setup.py install 编译算子。

5. 算子测试
补全 tests/test_sigmoid.py 和 tests/test_sigmoid_benchmark.py。
在 tests 中执行 python test_sigmoid.py 和 python test_sigmoid_benchmark.py 分别执行精度对比测试和性能测试。