摩尔线程
MOORE THREADS

摩尔线程 Torch_MUSA 开发者手册

版本 ***1.1.0***

2024 年 01 月 15 日

摩尔线程
MOORE THREADS

目录

1 版权声明

版权章节待完善。

- 版权声明
- © 2023

摩尔线程
MOORE THREADS

2 前言

2.1 版本记录

表 2.1: 版本记录

文档名称	摩尔线程 Torch_MUSA 开发者手册
版本号	V 1.1.0
作者	Moore Threads
修改日期	2024 年 01 月 15 日

2.2 更新历史

- **V0.1.0**

  更新时间: 2023.05.29

  更新内容:

  – 完成初版 Torch_MUSA 开发者手册。

- **V1.1.0**

  更新时间: 2024 年 01 月 15 日

  更新内容:

  – 增加调试工具章节。

  – 增加 MUSAExtension 章节。

  – 增加性能分析章节。

  – 快速入门章节增加更多示例代码。

  – 完善 FAQ 章节。

摩尔线程
MOORE THREADS

3 简介

3.1 MUSA 概述

MUSA (Metaverse Unified System Architecture) 是摩尔线程公司为摩尔线程 GPU 推出的一种通用并行计算平台和编程模型。它提供了 GPU 编程的简易接口，用 MUSA 编程可以构建基于 GPU 计算的应用程序，利用 GPUs 的并行计算引擎来更加高效地解决比较复杂的计算难题。同时摩尔线程还推出了 MUSA 工具箱（MUSAToolkits），工具箱中包括 GPU 加速库，运行时库，编译器，调试和优化工具等。MUSAToolkits 为开发人员在摩尔线程 GPU 上开发和部署高性能异构计算程序提供软件环境。

关于 MUSA 软件栈的更多内容，请参见 MUSA 官方文档。

3.2 PyTorch 概述

PyTorch 是一款开源的深度学习编程框架，可以用于计算机视觉，自然语言处理，语音处理等领域。PyTorch 使用动态计算，这在构建复杂架构时提供了更大的灵活性。PyTorch 使用核心 Python 概念，如类、结构和条件循环，因此理解起来更直观，编程更容易。此外，PyTorch 还具有可以轻松扩展、快速实现、生产部署稳定性强等优点。

关于 PyTorch 的更多内容，请参见 PyTorch 官方文档。

3.3 torch_musa 概述

为了摩尔线程 GPU 能支持开源框架 PyTorch，摩尔线程公司开发了 torch_musa。在 PyTorch v2.0.0 基础上，torch_musa 以插件的形式来支持摩尔线程 GPU，最大程度与 PyTorch 代码解耦，便于代码维护与升级。torch_musa 利用 PyTorch 提供的第三方后端扩展接口，将摩尔线程高性能计算库动态注册到 PyTorch 上，从而使得 PyTorch 框架能够利用摩尔线程显卡的高性能计算单元。利用摩尔线程显卡 CUDA 兼容的特性，torch_musa 内部引入了 cuda 兼容模块，使得 PyTorch 社区的 CUDA kernels 经过 porting 后就可以运行在摩尔线程显卡上，而且 CUDA Porting 的工作是在编译 torch_musa 的过程中自动进行，这大幅降低了 torch_musa 算子适配的成本，提高模型开发效率。同时，torch_musa 在 Python 前端接口与 PyTorch 社区 CUDA 接口形式上基本保持一致，这极大地降低了用户的学习成本和模型的迁移成本。

本手册主要介绍了基于 MUSA 软件栈的 torch_musa 开发指南。

3.4 torch_musa 核心代码目录概述

- torch_musa/tests 测试文件。
- torch_musa/core 主要包含 Python module，提供 amp/device/memory/stream/even 等模块的 Python 前端接口。
- torch_musa/csrc c++ 侧实现代码；
  - csrc/amp 提供混合精度模块的 C++ 实现。
  - csrc/aten 提供 C++ Tensor 库，包括 MUDNN 算子适配，CUDA-Porting 算子适配等等。
  - csrc/core 提供核心功能库，包括设备管理，内存分配管理，Stream 管理，Events 管理等。
  - csrc/distributed 提供分布式模块的 C++ 实现。

4 编译安装

注意：

编译安装前，需要安装 MUSAToolkits 软件包，MUDNN 库，MCCL 库，muThrust 库，muAlg 库，muRAND 库，muSPARSE 库。具体安装步骤，请参见相应组件的安装手册。

4.1 依赖环境

- Python == 3.8/3.9/3.10。
- 摩尔线程 MUSA 软件包，推荐版本如下：
  - MUSA 驱动 musa_2.6.0
  - MUSAToolkits rc2.0.0
  - MUDNN rc2.4.0
  - MCCL rc1.4.0
  - muAlg_dev-0.1.1-Linux.deb
  - muRAND_dev1.0.0.tar.gz
  - muSPARSE_dev0.1.0.tar.gz
  - muThrust_dev-0.1.1-Linux.deb
  - Docker Container Toolkits[1]

4.2 编译流程

1. 向 PyTorch 源码打 patch
2. 编译 PyTorch
3. 编译 torch_musa

torch_musa 是在 PyTorch v2.0.0 基础上以插件的方式来支持摩尔线程显卡。开发时涉及到对 PyTorch 源码的修改，目前是以打 patch 的方式实现的。PyTorch 社区正在积极支持第三方后端接入，这个 issue[2] 下有相关 PR。我们也在积极向 PyTorch 社区提交 PR，避免在编译过程中向 PyTorch 打 patch。

[1] https://mcconline.mthreads.com/software
[2] https://github.com/pytorch/pytorch/issues/98406

4.3 开发 Docker

为了方便开发者开发 torch_musa，我们提供了开发用的 docker image，参考命令：

```
docker run -it --name=torch_musa_dev --env MTHREADS_VISIBLE_DEVICES=all --shm-size=80g sh-
↪harbor.mthreads.com/mt-ai/musa-pytorch-dev:latest /bin/bash
```

注意：

使用 docker 时，请务必提前安装 mt-container-toolkit[3] ，并且在启动 docker container 时添加选项“--env MTHREADS_VISIBLE_DEVICES=all”，否则在 docker container 内部无法使用 torch_musa。

[3] https://mcconline.mthreads.com/software/1?id=1

4.4 编译步骤

4.4.1 设置环境变量

```
export MUSA_HOME=path/to/musa_libraries(including musa_toolkits, mudnn and so on) # defalut␣
↪value is /usr/local/musa/
export LD_LIBRARY_PATH=$MUSA_HOME/lib:$LD_LIBRARY_PATH
export PYTORCH_REPO_PATH=path/to/PyTorch source code
# if PYTORCH_REPO_PATH is not set, PyTorch-v2.0.0 will be downloaded outside this directory␣
↪automatically when building with build.sh
```

4.4.2 使用脚本一键编译（推荐）

```
cd torch_musa
bash docker/common/daily/update_daily_mudnn.sh # update daily mudnn lib if needed
bash build.sh   # build original PyTorch and torch_musa from scratch

# Some important parameters are as follows:
bash build.sh --torch  # build original PyTorch only
bash build.sh --musa   # build torch_musa only
bash build.sh --fp64   # compile fp64 in kernels using mcc in torch_musa
bash build.sh --debug  # build in debug mode
bash build.sh --asan   # build in asan mode
bash build.sh --clean  # clean everything built
```

在初次编译时，需要执行 `bash build.sh`（先编译 PyTorch，再编译 torch_musa）。在后续开发过程中，如果不涉及对 PyTorch 源码的修改，那么执行 `bash build.sh -m`（仅编译 torch_musa）即可。

4.4.3 分步骤编译

如果不想使用脚本编译，那么可以按照如下步骤逐步编译。

1. 在 PyTorch 打 patch

```
# 请保证 PyTorch 源码和 torch_musa 源码在同级目录或者 export PYTORCH_REPO_PATH=path/to/PyTorch 指向 PyTorch 源码
bash build.sh --only-patch
```

2. 编译 PyTorch

```
cd pytorch
pip install -r requirements.txt
python setup.py install
# debug mode: DEBUG=1 python setup.py install
# asan mode:  USE_ASAN=1 python setup.py install
```

3. 编译 torch_musa

```
cd torch_musa
pip install -r requirements.txt
python setup.py install
# debug mode: DEBUG=1 python setup.py install
# asan mode:  USE_ASAN=1 python setup.py install
```


摩尔线程 MOORE THREADS

5 快速入门

注解：

使用 torch_musa 时，需要先导入 torch 包（import torch）和 torch_musa 包 (import torch_musa)。

5.1 常用环境变量

开发 torch_musa 过程中常用环境变量如下表所示：

表 5.1: 常用环境变量

环境变量示例	所属组件	功能说明
export TORCH_SHOW_CPP_STACKTRACES=1	PyTorch	当 python 程序发生错误时显示 PyTorch 中 C++ 调用栈
export MUDNN_LOG_LEVEL=INFO	MUDNN	使能 MUDNN 算子库调用的 log
export MUSA_VISIBLE_DEVICES=0,1,2,3	Driver	控制当前可见的显卡序号
export MUSA_LAUNCH_BLOCKING=1	Driver	驱动以同步模式下发 kernel，即当前 kernel 执行结束后再下发下一个 kernel

5.2 常用 api 示例代码

```
import torch
import torch_musa

torch.musa.is_available()
torch.musa.device_count()

a = torch.tensor([1.2, 2.3], dtype=torch.float32, device='musa')
b = torch.tensor([1.8, 1.2], dtype=torch.float32, device='cpu').to('musa')
c = torch.tensor([1.8, 1.3], dtype=torch.float32).musa()
```

```
d = a + b + c

torch.musa.synchronize()

with torch.musa.device(0):
    assert torch.musa.current_device() == 0

if torch.musa.device_count() > 1:
    torch.musa.set_device(1)
    assert torch.musa.current_device() == 1
    torch.musa.synchronize("musa:1")
```

torch_musa 中 python api 基本与 PyTorch 原生 api 接口保持一致，极大降低了新用户的学习成本。

5.3 推理示例代码

```
import torch
import torch_musa
import torchvision.models as models

model = models.resnet50().eval()
x = torch.rand((1, 3, 224, 224), device="musa")
model = model.to("musa")
# Perform the inference
y = model(x)
```

5.4 训练示例代码

```
import torch
import torch_musa
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

## 1. prepare dataset
transform = transforms.Compose(
    [transforms.ToTensor(),
```



```
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

batch_size = 4
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)
testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)
classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
device = torch.device("musa")

## 2. build network
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

net = Net().to(device)

## 3. define loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

## 4. train
```



```
for epoch in range(2):  # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs.to(device))
        loss = criterion(outputs, labels.to(device))
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 2000:.3f}')
            running_loss = 0.0

print('Finished Training')

PATH = './cifar_net.pth'
torch.save(net.state_dict(), PATH)

net.load_state_dict(torch.load(PATH))

## 5. test
correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for our outputs
with torch.no_grad():
    for data in testloader:
        images, labels = data
        # calculate outputs by running images through the network
        outputs = net(images.to(device))
        # the class with the highest energy is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels.to(device)).sum().item()
```



```
print(f'Accuracy of the network on the 10000 test images: {100 * correct // total} %')
```

5.5 混合精度 AMP 训练示例代码

```
import torch
import torch_musa
import torch.nn as nn

class SimpleModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(5, 4)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(4, 3)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x
    def __call__(self, x):
        return self.forward(x)

DEVICE = "musa"

def train_in_amp(low_dtype=torch.float16):
    model = SimpleModel().to(DEVICE)
    criterion = nn.MSELoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=0.1)

    # create the scaler object
    scaler = torch.musa.amp.GradScaler()

    inputs = torch.randn(6, 5).to(DEVICE)  # 将数据移至 GPU
    targets = torch.randn(6, 3).to(DEVICE)
    for step in range(20):
        optimizer.zero_grad()
        # create autocast environment
        with torch.musa.amp.autocast(dtype=low_dtype):
            outputs = model(inputs)
```



```
            assert outputs.dtype == low_dtype
            loss = criterion(outputs, targets)

        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()
    return loss

if __name__ == "__main__":
    train_in_amp(torch.float16)
```

5.6 分布式训练示例代码

```
"""Demo of DistributedDataParall"""
import os
import torch
from torch import nn
from torch import optim
from torch.nn.parallel import DistributedDataParallel as DDP
import torch.distributed as dist
import torch.multiprocessing as mp
import torch_musa


class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(5,5)
    def forward(self, x):
        return self.linear(x)

def start(rank, world_size):
    if os.getenv("MASTER_ADDR") is None:
        os.environ["MASTER_ADDR"]= ip # IP must be specified here
    if os.getenv("MASTER_PORT") is None:
        os.environ["MASTER_PORT"]= port # port must be specified here
    dist.init_process_group("mccl", rank=rank, world_size=world_size)

def clean():
    dist.destroy_process_group()
```



```
def runner(rank, world_size):
    torch_musa.set_device(rank)
    start(rank, world_size)
    model = Model().to('musa')
    ddp_model = DDP(model, device_ids=[rank])
    optimizer = optim.SGD(ddp_model.parameters(), lr=0.001)
    for _ in range(5):
        input_tensor = torch.randn(5, dtype=torch.float, requires_grad=True).to('musa')
        target_tensor = torch.zeros(5, dtype=torch.float).to('musa')
        output_tensor = ddp_model(input_tensor)
        loss_f = nn.MSELoss()
        loss = loss_f(output_tensor, target_tensor)
        loss.backward()
        optimizer.step()
    clean()

if __name__ == "__main__":
    mp.spawn(runner, args=(2,), nprocs=2, join=True)
```

5.7 使能 TensorCore 示例代码

```
import torch
import torch_musa
# 在 s4000 或者更新的设备上，可以使能 TensorCore，来加速计算过程
with torch.backends.mudnn.flags(allow_tf32=True):
    assert torch.backends.mudnn.allow_tf32
    a = torch.randn(10240, 10240, dtype=torch.float, device='musa')
    b = torch.randn(10240, 10240, dtype=torch.float, device='musa')
    result_tf32 = a @ b

torch.backends.mudnn.allow_tf32 = True
assert torch_musa._MUSAC._get_allow_tf32()
a = torch.randn(10240, 10240, dtype=torch.float, device='musa')
b = torch.randn(10240, 10240, dtype=torch.float, device='musa')
result_tf32 = a @ b
```


5.8 C++ 部署示例代码

```cpp
#include <torch/script.h>
#include <torch_musa/csrc/core/Device.h>
#include <iostream>
#include <memory>

int main(int argc, const char* argv[]) {
  // Register 'musa' for PrivateUse1 as we save model with 'musa'.
  c10::register_privateuse1_backend("musa");

  torch::jit::script::Module module;
  // Load model which saved with torch jit.trace or jit.script.
  module = torch::jit::load(argv[1]);

  std::vector<torch::jit::IValue> inputs;
  // Ready for input data.
  torch::Tensor input = torch::rand({1, 3, 224, 224}).to("musa");
  inputs.push_back(input);

  // Model execute.
  at::Tensor output = module.forward(inputs).toTensor();

  return 0;
}
```

详细用法请参考 examples/cpp[4] 下内容。

[4] https://github.mthreads.com/mthreads/torch_musa/tree/main/examples/cpp

6 算子支持

本节主要介绍如何在 torch_musa 中适配一个新算子，算子实现后端包括 MUDNN 算子库和 CUDA-Porting kernels。

6.1 何时需要适配新算子?

以“tril”算子为例，当我们的测试代码有如下报错 log，则说明 torch_musa 中没有适配“tril”算子。

```
import torch
import torch_musa
input_data = torch.randn(3, 3, device="musa")
result = torch.tril(input_data)
```

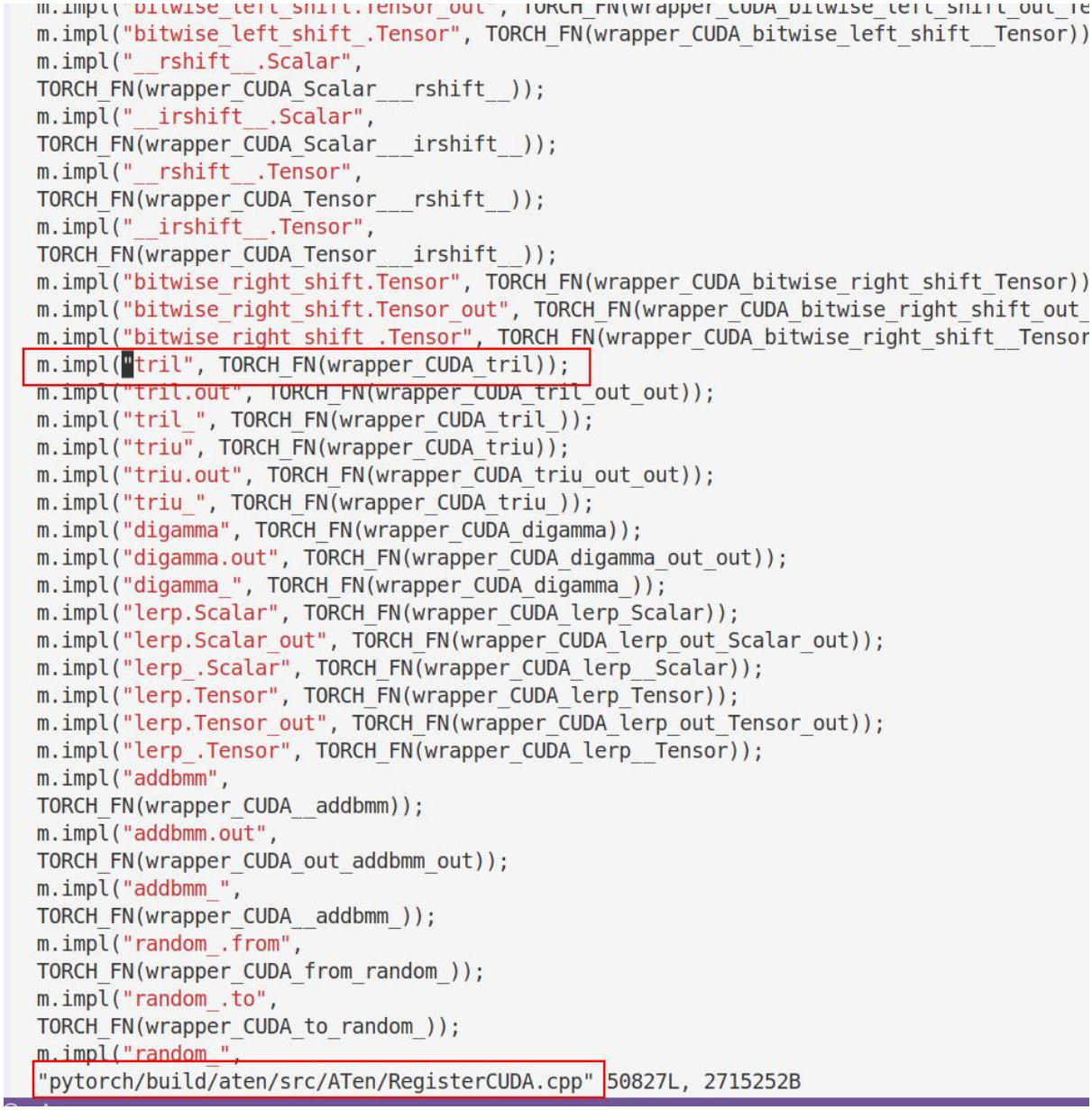
```
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    result = torch.tril(input_data)
NotImplementedError: Could not run 'aten::tril.out' with arguments from the 'musa' backend. This could be because the operator doesn't exist for this backend, or was omitted during the selective/custom build process (if using custom build). If you are a Facebook employee using PyTorch on mobile, please visit https://fburl.com/ptmfixes for possible resolutions. 'aten::tril.out' is only available for these backends: [CPU, Meta, BackendSelect, Python, FuncTorchDynamicLayerBackMode, Functionalize, Named, Conjugate, Negative, ZeroTensor, ADInplaceOrView, AutogradOther, AutogradCPU, AutogradCUDA, AutogradHIP, AutogradXLA, AutogradMPS, AutogradIPU, AutogradXPU, AutogradHPU, AutogradVE, AutogradLazy, AutogradMeta, AutogradMTIA, AutogradPrivateUse1, AutogradPrivateUse2, AutogradPrivateUse3, AutogradNestedTensor, Tracer, AutocastCPU, AutocastCUDA, FuncTorchBatched, FuncTorchVmapMode, Batched, VmapMode, FuncTorchGradWrapper, PythonTLSSnapshot, FuncTorchDynamicLayerFrontMode, PythonDispatcher].

CPU: registered at /home/pytorch/build/aten/src/ATen/RegisterCPU.cpp:31034 [kernel]
Meta: registered at /dev/null:219 [kernel]
BackendSelect: fallthrough registered at /home/pytorch/aten/src/ATen/core/BackendSelectFallbackKernel.cpp:3 [backend fallback]
Python: registered at /home/pytorch/aten/src/ATen/core/PythonFallbackKernel.cpp:144 [backend fallback]
FuncTorchDynamicLayerBackMode: registered at /home/pytorch/aten/src/ATen/functorch/DynamicLayer.cpp:491 [backend fallback]
Functionalize: registered at /home/pytorch/build/aten/src/ATen/RegisterFunctionalization_1.cpp:23013 [kernel]
Named: registered at /home/pytorch/aten/src/ATen/core/NamedRegistrations.cpp:7 [backend fallback]
Conjugate: registered at /home/pytorch/aten/src/ATen/ConjugateFallback.cpp:17 [backend fallback]
```

6.2 如何适配新算子

6.2.1 先注册新算子

算子实现的注册可以参考 PyTorch 官方文档 https://pytorch.org/tutorials/advanced/dispatcher.html，也可以参考 PyTorch 框架中 CUDA 后端的注册代码。在编译完 PyTorch 代码后会生成下图中的文件，PyTorch 在该文件中实现了 CUDA 后端实现的注册。



同理，我们也需要给“tril”算子为 MUSA 后端实现注册。部分代码如下所示：

```
#include <torch/library.h>

namespace at {
namespace musa {

TORCH_LIBRARY_IMPL(aten, PrivateUse1, m) {
  m.impl("tril", Tril);  // 'Tril' is a function implemented somewhere
}

} // namespace musa
} // namespace at
```


PyTorch 社区推荐使用 PrivateUse1 作为第三方扩展后端的 key，所以我们这里复用了 PrivateUse1。

6.2.2 利用 MUDNN 实现算子

如果 MUDNN 算子库支持了该算子，那么需要以 MUDNN 算子库作为后端来适配该新算子。使用 MUDNN 适配新算子的主要步骤如下：

1. tensor 的数据类型和 device 类型检查;
2. 添加 DeviceGuard;
3. 目前大部分 MUDNN 算子只支持连续的 tensor，因此在 `createMUTensor` 前需要将其转换为连续 tensor（如果该 tensor 不连续，此操作将会产生 copy 耗时）;
4. 创建 muTensor;
5. 调用 MUDNN 的 op 实现接口;

以 `addcdiv.out` 算子为例，部分代码如下：

```
#include <mudnn.h>

Tensor& AddcDivOut(const Tensor& base, const Tensor& tensor1,
                   const Tensor& tensor2, const Scalar& value, Tensor& out) {
  //1). check Dtype & device
  TORCH_CHECK(self.device().type() == kMUSA,
              "Device of input tensor of addcdiv must be MUSA, but now it is ",
              self.device());
  TORCH_CHECK(
      self.scalar_type() == at::ScalarType::Float,
      "Dtype of input tensor of addcdiv only support Float32, but now it is ",
      self.scalar_type());
     ....
  // 2).convert it to contiguous tensor
  Tensor tensor_cong = tensor1.contiguous();
     ...
  // 3). create muTensor, which binds the two variables by address.
  muTensor musa_tensor1 = CreateMUTensor(tensor_cong);
  muTensor mu_out = CreateMUTensor(tensor_cong);     ....
  // 4). call musa op to implement the calculation.
  ::musa::dnn::Handle h;
  ::musa::dnn::Ternary mop;

  if (!alpha_scalar.equal(1)) {
    if (self.is_floating_point()) {
      CHECK_MUDNN_STATUS(mop.SetAlpha(alpha_scalar.toDouble()), "SetAlpha");
    } else {
```



```
      CHECK_MUDNN_STATUS(mop.SetAlpha(alpha_scalar.toLong()), "SetAlpha");
    }
  }
  CHECK_MUDNN_STATUS(mop.SetMode(TERNARY_MODE::ADDCDIV_ALPHA), "SetMode");
  CHECK_MUDNN_STATUS(mop.Run(h, om_mt, musa_base, musa_tensor1, musa_tensor2), "Run");

}

TORCH_LIBRARY_IMPL(aten, PrivateUse1, m){
   ...
   m.impl("addcdiv.out", &AddcDivOut);
}
```

通过 mudnn*.h 头文件可以查看到 MUDNN 算子库函数接口。默认 MUDNN 算子库的头文件会在 /usr/local/musa/include 目录下。

6.2.3 利用 CUDA-Porting 实现算子

如果该算子 MUDNN 算子库不支持，那么我们需要通过 CUDA-Porting kernels 作为后端来适配新算子。

首先介绍一下 CUDA-Porting 的流程：

1. 在 torch_musa/build 下新建目录（默认目录名是 torch_musa/build/generated_cuda_compatible）用来保存 CUDA-Porting 过程需要用到的文件。
2. 从 PyTorch 仓库中将 kernels 相关的 cu/cuh 文件以及 include 头文件复制到上一步新建目录中去。这些文件需要经过 CUDA-Porting 脚本的处理（torch_musa/torch_musa/tools/cuda_porting/cuda_porting.py）。
3. 运行 porting 工具。主要是一些字符串替换处理，如将 cudaMalloc 替换成 musaMalloc，cuda_fp16.h 替换成 musa_fp16.h 等。
4. 经过上述操作后，build/generated_cuda_compatible/aten/src/ATen/native/musa/会有很多****.mu 文件，这些 mu 文件就是我们适配时会用到的 kernels 文件。
5. 适配 CUDA-Porting 工具处理过的 kernels。

上述步骤 1，2，3，4 会在编译过程中自动完成，适配新算子关心的步骤 5 即可。有一点需要注意的是，在开发过程中引用的 PyTorch 头文件来自于 torch_musa/build/generated_cuda_compatible/include 目录，而不是系统下 PyTorch 安装目录下的头文件。

下面以两种典型算子为例，介绍如何利用 CUDA-Porting kernels 适配新算子。在开始适配之前，可以在 pytorch/build/aten/src/ATen/RegisterCUDA.cpp 文件中查看该算子在 CUDA 中的实现方式。

6.2.3.1 以 abs 算子为例

CUDA 中 abs 算子的部分适配代码如下：

```
at::Tensor & wrapper_CUDA_out_abs_out(const at::Tensor & self, at::Tensor & out) {
  // No device check
const OptionalDeviceGuard device_guard(device_of(self));
return at::native::abs_out(self, out);
}

******
m.impl("abs.out", TORCH_FN(wrapper_CUDA_out_abs_out));
```

如果该算子直接调用了 at::native 下面的函数接口，那么我们也这么做就可以了：

```
#include "torch_musa/csrc/core/MUSAGuard.h"
at::Tensor& MusaAbsout(const at::Tensor& self, at::Tensor& out) {
c10::musa::MUSAGuard device_gaurd(self.device());
return at::native::abs_out(self, out);
}

TORCH_LIBRARY_IMPL(aten, PrivateUse1, m) {
  m.impl("abs.out", &MusaAbsout);
}
```

这里的关键是 PyTorch 仓库提供了 DispatchStub 机制。我们在 CUDA-Porting 时，将 `REGISTER_CUDA_DISPATCH` 替换成 `REGISTER_MUSA_DISPATCH`，从而能实现根据 device 类型调用到 porting 后的 kernels。对这背后机制感兴趣的话，可以查看一下如下几个文件：

- abs_out 函数实现：https://github.com/pytorch/pytorch/blob/v2.0.0/aten/src/ATen/native/UnaryOps.cpp#L546
- abs_stub 注册：https://github.com/pytorch/pytorch/blob/v2.0.0/aten/src/ATen/native/cuda/AbsKernel.cu#L49
- DispatchStub 定义：https://github.com/pytorch/pytorch/blob/v2.0.0/aten/src/ATen/native/DispatchStub.h

6.2.3.2 以 tril 算子为例

CUDA 中 tril 算子的部分适配代码如下：

```
struct structured_tril_cuda_functional final : public at::native::structured_tril_cuda {
    void set_output_strided(
        int64_t output_idx, IntArrayRef sizes, IntArrayRef strides,
        TensorOptions options, DimnameList names
    ) override {
        auto current_device = guard_.current_device();
        if (C10_UNLIKELY(current_device.has_value())) {
          TORCH_INTERNAL_ASSERT(*current_device == options.device(),
            "structured kernels don't support multi-device outputs");
        } else {
          guard_.reset_device(options.device());
        }
        outputs_[output_idx] = create_out(sizes, strides, options);
        if (!names.empty()) {
          namedinference::propagate_names(*outputs_[output_idx], names);
        }
        // super must happen after, so that downstream can use maybe_get_output
        // to retrieve the output
    }
    void set_output_raw_strided(
        int64_t output_idx, IntArrayRef sizes, IntArrayRef strides,
        TensorOptions options, DimnameList names
    ) override {
        auto current_device = guard_.current_device();
        if (C10_UNLIKELY(current_device.has_value())) {
          TORCH_INTERNAL_ASSERT(*current_device == options.device(),
            "structured kernels don't support multi-device outputs");
        } else {
          guard_.reset_device(options.device());
        }
        outputs_[output_idx] = create_out(sizes, strides, options);
        if (!names.empty()) {
          namedinference::propagate_names(*outputs_[output_idx], names);
        }
        // super must happen after, so that downstream can use maybe_get_output
        // to retrieve the output
    }
    const Tensor& maybe_get_output(int64_t output_idx) override {
```

```
      return *outputs_[output_idx];
    }
    std::array<c10::ExclusivelyOwned<Tensor>, 1> outputs_;
    c10::cuda::OptionalCUDAGuard guard_;
};
at::Tensor wrapper_CUDA_tril(const at::Tensor & self, int64_t diagonal) {
c10::optional<Device> common_device = nullopt;
(void)common_device; // Suppress unused variable warning
  c10::impl::check_and_update_common_device(common_device, self, "wrapper_CUDA_tril", "self");
  structured_tril_cuda_functional op;
op.meta(self, diagonal);
op.impl(self, diagonal, *op.outputs_[0]);
return std::move(op.outputs_[0]).take();
}

******
m.impl("tril", TORCH_FN(wrapper_CUDA_tril));
```

该算子在实现时继承了基类 `at::native::structured_tril_cuda`，那么我们也需要这么实现：

```
#include <ATen/ops/tril_native.h>

#include "torch_musa/csrc/aten/utils/Utils.h"
#include "torch_musa/csrc/core/MUSAGuard.h"

namespace at {
namespace musa {

namespace {
struct structured_tril_musa_functional final
    : public at::native::structured_tril_cuda {
  void set_output_strided(
      int64_t output_idx,
      IntArrayRef sizes,
      IntArrayRef strides,
      TensorOptions options,
      DimnameList names) override {
    auto current_device = guard_.current_device();
    if (C10_UNLIKELY(current_device.has_value())) {
      TORCH_INTERNAL_ASSERT(
          *current_device == options.device(),
          "structured kernels don't support multi-device outputs");
```

```
    } else {
      guard_.reset_device(options.device());
    }
    outputs_[output_idx] = create_out(sizes, strides, options);
  }
  void set_output_raw_strided(
      int64_t output_idx,
      IntArrayRef sizes,
      IntArrayRef strides,
      TensorOptions options,
      DimnameList names) override {
    auto current_device = guard_.current_device();
    if (C10_UNLIKELY(current_device.has_value())) {
      TORCH_INTERNAL_ASSERT(
          *current_device == options.device(),
          "structured kernels don't support multi-device outputs");
    } else {
      guard_.reset_device(options.device());
    }
    outputs_[output_idx] = create_out(sizes, strides, options);
  }
  const Tensor& maybe_get_output(int64_t output_idx) override {
    return *outputs_[output_idx];
  }
  std::array<c10::ExclusivelyOwned<Tensor>, 1> outputs_;
  c10::musa::OptionalMUSAGuard guard_;
};
} // namespace

Tensor Tril(const Tensor& self, int64_t diagonal) {
  structured_tril_musa_functional op;
  op.meta(self, diagonal);
  op.impl(self, diagonal, *op.outputs_[0]);
  return std::move(op.outputs_[0]).take();
}
} // namespace musa
} // namespace at
```

至此，我们已经完成了通过 CUDA-Porting kernels 来适配新算子。

6.2.4 利用 CPU 实现算子

对于部分算子，如果 MUDNN 不支持，CUDA-Porting 也无法支持，可以临时中利用 CPU 后端实现该算子。主要逻辑是，先把 tensor 拷贝到 CPU 侧，在 CPU 完成计算，再将结果拷贝到 GPU 侧。可以参考下述代码：

```
Tensor& AddcDivOut(const Tensor& base, const Tensor& tensor1,
                   const Tensor& tensor2, const Scalar& value, Tensor& out) {
  auto cpu_base =
      at::empty(base.sizes(), base.options().device(DeviceType::CPU));
  auto cpu_factor1 =
      at::empty(tensor1.sizes(), tensor1.options().device(DeviceType::CPU));
  auto cpu_factor2 =
      at::empty(tensor2.sizes(), tensor2.options().device(DeviceType::CPU));
  auto cpu_out =
      at::empty(out.sizes(), out.options().device(DeviceType::CPU));
  cpu_base.copy_(base);
  cpu_factor1.copy_(tensor1);
  cpu_factor2.copy_(tensor2);
  auto result = addcdiv_out(cpu_out, cpu_base, cpu_factor1, cpu_factor2);
  out.copy_(cpu_out);
  return out;
}
```

6.2.5 添加算子单元测试

如果已经完成了新算子的适配，那么还需要添加算子单元测试，保证算子适配结果的正确性。算子测试文件在 `torch_musa/tests/unittest/operator` 目录下，参考已有算子测试添加即可，在此不展开描述。

算子测试命令如下：

```
pytest -s torch_musa/tree/main/tests/unittest/operator/xxxx.py
```

6.2.6 使用调试注册接口注册算子

如果希望对算子的调用记录和输入输出值加以记录或统计，可以使用调试注册接口进行注册。

调试注册接口是一组宏，提供对算子的额外调试封装。目前提供三个宏作为注册接口使用：

1. ADVANCED_REGISTER

ADVANCED_REGISTER(lib, key, yaml, func) 是对原生注册机制 TORCH_LIBRARY_IMPL 的改进版本。

默认情况下，它将原算子封装为一个包含了额外调试组件的新函数，并附加上额外的隐藏前缀注册。

使用时，将原生注册时指定的 lib（如 aten）、key（如 PrivateUse1）、算子的注册名称（如"Abs"）和算子实现函数作为参数传入即可。

下面是一个示例：

原注册方式：

```
TORCH_LIBRARY_IMPL(aten, PrivateUse1, m) {
  m.impl("Abs", &CustomAbs);
}
```

新的注册方式：

```
ADVANCED_REGISTER(aten, PrivateUse1, "Abs", CustomAbs)
```

在上述示例中，ADVANCED_REGISTER 会首先将 CustomAbs 封装为一个别名函数，并在实际注册时，使用这个封装后的函数进行注册。

默认情况下，这个别名函数的名称为 wrapper_CustomAbs。

如果函数名称中存在额外的名字空间，或该函数为类成员函数等情况时，函数名称可能包含特殊符号（如:: 等）。此时需要使用下述的 REGISTER_IMPL 指定封装名称。

2. REGISTER_IMPL

REGISTER_IMPL(lib, key, yaml, func, name) 的用法与 ADVANCED_REGISTER 相似。

但与之不同的时，REGISTER_IMPL 额外接受一个 name 参数，以指定封装后的别名函数的命名：该别名函数将被命名为 wrapper_##name。

如果 name 和 func 参数相同时，REGISTER_IMPL 即为 ADVANCED_REGISTER。

下面给出一个示例：

```
REGISTER_IMPL(aten, PrivateUse1, "view", at::native::view, at_native_view)
```

示例中，view 算子的实现为 at::native::view，包含了名字空间，因此使用 REGISTER_IMPL 将其名称指定为 at_native_view。

其实际注册的封装函数名称为 wrapper_at_native_view。

3. REDEFINE_REGISTER

部分情况下，同一个算子可能会被注册为多个不同的别名，例如，not_equal 和 ne 都使用相同的实现。

在已经使用 ADVANCED_REGISTER 或 REGISTER_IMPL 将其封装为别名函数之后，不能再进行重复封装，而应直接复用封装后的函数。

因此，使用 REDEFINE_REGISTER(lib, key, yaml, name) 进行重复注册。

如果之前的封装注册使用 ADVANCED_REGISTER，则 name 即为 ADVANCED_REGISTER 中的 func 参数。

如果之前的封装注册使用 REGISTER_IMPL，则 name 为 REGSITER_IMPL 中的 name 参数。

示例如下：

```
ADVANCED_REGISTER(aten, PrivateUse1, "ne.Tensor", NotEqualTensor)
ADVANCED_REGISTER(aten, PrivateUse1, "ne_.Tensor", NotEqual_Tensor)
ADVANCED_REGISTER(aten, PrivateUse1, "ne.Tensor_out", NotEqual_out)
// not_equal, alias for torch.ne
REDEFINE_REGISTER(aten, PrivateUse1, "not_equal.Tensor", NotEqualTensor)
REDEFINE_REGISTER(aten, PrivateUse1, "not_equal_.Tensor", NotEqual_Tensor)
REDEFINE_REGISTER(aten, PrivateUse1, "not_equal.Tensor_out", NotEqual_out)
```

示例中，首先使用 ADVANCED_REGISTER 将 NotEqual 系列的三个实现分别注册为 ne 类接口，之后的 not_equal 类接口使用了相同的实现，因此，改用 REDEFINE_REGISTER 注册。

6.2.7 即将支持的特性

调试注册接口将在未来支持 torch 原生的函数封装宏，如 TORCH_FN 等，可以适应更多的算子注册方式。

引入 codegen 模块，实现算子的注册代码和实现代码的生成，能进一步简化算子适配的工作量。

请关注这部分工作。

7 自定义算子支持

7.1 支持自定义算子

待完善。

8 第三方库 MUSA 扩展支持

本节主要介绍如何对 PyTorch 生态的第三方库进行 MUSA 扩展的构建 (MUSAExtension)，对应于 CUDAExtension。

8.1 为什么要对第三方库进行 MUSA 扩展的构建?

以 mmcv 库 (commit id 为 0a2f60ba0198f8d567b536313bfba329588f9c3f) 为例，当我们的测试代码有如下报错 log，则说明 mmcv 中没有构建 MUSA 扩展。此时，我们需要对 mmcv 库进行 MUSA 扩展，从而使得 mmcv 库运行在摩尔线程显卡上。

```
import numpy as np
import torch
import torch_musa
from mmcv.ops import nms
np_boxes = np.array([[6.0, 3.0, 8.0, 7.0], [3.0, 6.0, 9.0, 11.0],
                     [3.0, 7.0, 10.0, 12.0], [1.0, 4.0, 13.0, 7.0]],
                     dtype=np.float32)
np_scores = np.array([0.6, 0.9, 0.7, 0.2], dtype=np.float32)
np_inds = np.array([1, 0, 3])
np_dets = np.array([[3.0, 6.0, 9.0, 11.0, 0.9],
                    [6.0, 3.0, 8.0, 7.0, 0.6],
                    [1.0, 4.0, 13.0, 7.0, 0.2]])
boxes = torch.from_numpy(np_boxes)
scores = torch.from_numpy(np_scores)
# check if cpu can work
dets, inds = nms(boxes, scores, iou_threshold=0.3, offset=0)
# check if musa can work
dets, inds = nms(boxes.musa(), scores.musa(), iou_threshold=0.3, offset=0)
```

```
>>> scores = torch.from_numpy(np_scores)
>>> dets, inds = nms(boxes, scores, iou_threshold=0.3, offset=0)
>>> dets
tensor([[ 3.0000,  6.0000,  9.0000, 11.0000,  0.9000],
        [ 6.0000,  3.0000,  8.0000,  7.0000,  0.6000],
        [ 1.0000,  4.0000, 13.0000,  7.0000,  0.2000]])
>>> inds
tensor([1, 0, 3])
>>> dets, inds = nms(boxes.musa(), scores.musa(), iou_threshold=0.3, offset=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/opt/conda/envs/py38/lib/python3.8/site-packages/mmengine-0.10.2-py3.8.egg/mmengine/utils/misc.py", line 395, in new_func
    output = old_func(*args, **kwargs)
  File "/opt/conda/envs/py38/lib/python3.8/site-packages/mmcv-2.0.1-py3.8-linux-x86_64.egg/mmcv/ops/nms.py", line 127, in nms
    inds = NMSop.apply(boxes, scores, iou_threshold, offset, score_threshold,
  File "/opt/conda/envs/py38/lib/python3.8/site-packages/torch/autograd/function.py", line 506, in apply
    return super().apply(*args, **kwargs)  # type: ignore[misc]
  File "/opt/conda/envs/py38/lib/python3.8/site-packages/mmcv-2.0.1-py3.8-linux-x86_64.egg/mmcv/ops/nms.py", line 27, in forward
    inds = ext_module.nms(
RuntimeError: nms_impl: implementation for device musa:0 not found.
```

注意以上测试不要在 mmcv 根目录下进行，以免将当前目录下的 mmcv 包导入。

8.2 如何对第三方库进行 MUSA 扩展?

8.2.1 了解 MUSAExtension 这个 API

阅读 torch_musa/utils/README.md 中关于 MUSAExtension 的介绍。

8.2.2 CUDA-Porting

我们需要先找到与 CUDA 相关文件所在的位置，在 mmcv 中，有如下几处：

- mmcv/mmcv/ops/csrc/common/cuda/
- mmcv/mmcv/ops/csrc/pytorch/cuda/

为了方便我们将对 mmcv/mmcv/ops/csrc 这个目录进行 CUDA-Porting，将会生成 mmcv/mmcv/ops/csrc_musa 目录。

同时也为了减少不必要的 Porting，我们将如下几个目录进行忽略：

- mmcv/ops/csrc/common/mlu
- mmcv/ops/csrc/common/mps
- mmcv/mmcv/ops/csrc/parrots
- mmcv/mmcv/ops/csrc/pytorch/mlu
- mmcv/mmcv/ops/csrc/pytorch/mps
- mmcv/mmcv/ops/csrc/pytorch/npu

之后从 mmcv 根目录全局搜索 cu、nv、cuda 和对应的大写关键词。搜索关键词的目的在于梳理自定义的映射规则，本次对搜索结果的映射规则提取如下：

- _CU_H_ -> _MU_H_
- _CUH -> _MUH

- __NVCC__ -> __MUSACC__
- MMCV_WITH_CUDA -> MMCV_WITH_MUSA
- AT_DISPATCH_FLOATING_TYPES_AND_HALF -> AT_DISPATCH_FLOATING_TYPES
- #include <ATen/cuda/CUDAContext.h> -> #include "torch_musa/csrc/aten/musa/MUSAContext.h"
- #include <c10/cuda/CUDAGuard.h> -> #include "torch_musa/csrc/core/MUSAGuard.h"
- ::cuda:: -> ::musa:
- /cuda/ -> /musa/
- , CUDA, -> , PrivateUse1,
- .cuh -> .muh
- .is_cuda() -> .is_privateuseone()

大多数情况下，有一些基本的映射规则即 cu->mu、nv->mt、cuda->musa、cuh->muh 及对应的大写映射。如果在编译过程中遇到 HALF 相关的编译报错，可以如上所示将 HALF 相关的宏取消掉。然后我们将搜索出来的关键词拓展，形成单词边界然后进行 映射，如果直接 cu->mu 那么就会产生 Accumulate->Acmumulate 这样的不期望的结果。第 3、6、7、10、12 个规则是一些固定的转换，其中 PrivateUse1 是 PyTorch 中对于扩展的自定义 backend 默认名字，is_privateuseone 也是属于自定义 backend 相关的 API。

因此由上述分析我们得到如下 CUDA-porting 脚本：

```
SimplePorting(cuda_dir_path="./mmcv/ops/csrc", ignore_dir_paths=[
              "./mmcv/ops/csrc/common/mlu",
              "./mmcv/ops/csrc/common/mps",
              "./mmcv/ops/csrc/parrots",
              "./mmcv/ops/csrc/pytorch/mlu",
              "./mmcv/ops/csrc/pytorch/mps",
              "./mmcv/ops/csrc/pytorch/npu"
              ],
              mapping_rule={
                  "_CU_H_": "_MU_H_",
                  "_CUH": "_MUH",
                  "__NVCC__": "__MUSACC__",
                  "MMCV_WITH_CUDA": "MMCV_WITH_MUSA",
                  "AT_DISPATCH_FLOATING_TYPES_AND_HALF": "AT_DISPATCH_FLOATING_TYPES",
                  "#include <ATen/cuda/CUDAContext.h>": "#include \"torch_musa/csrc/aten/musa/
↪MUSAContext.h\"",
                  "#include <c10/cuda/CUDAGuard.h>": "#include \"torch_musa/csrc/core/
↪MUSAGuard.h\"",
                  "::cuda::": "::musa::",
                  "/cuda/": "/musa/",
                  ", CUDA,": ", PrivateUse1,",
```

```
                    ".cuh": ".muh",
                    ".is_cuda()": ".is_privateuseone()",
                    }
                    ).run()
```

需要注意的是尽管我们自定义了映射规则，但是我们没有传入 drop_default_mapping 参数，因此在 CUDA-porting 时还会使用默认的映射规则，见 torch_musa/utils/mapping 文件夹。由于文件夹下的 general.json 条目过多，并且基本上不会被用到，所以默认的映射规则里只包含除了 它之外的其他映射规则（mapping 文件夹中除了 general.json 之外的其他 json 文件），general.json 可作为自定义映射规则的参考。如果不 想在代码里添加映射规则，也可以在 extra.json 文件中添加条目或者自行添加新的 json 文件。

8.2.3 分析 mmcv 的构建脚本 setup.py

```
...
elif is_rocm_pytorch or torch.cuda.is_available() or os.getenv(
    'FORCE_CUDA', '0') == '1':
if is_rocm_pytorch:
    define_macros += [('MMCV_WITH_HIP', None)]
define_macros += [('MMCV_WITH_CUDA', None)]
cuda_args = os.getenv('MMCV_CUDA_ARGS')
extra_compile_args['nvcc'] = [cuda_args] if cuda_args else []
op_files = glob.glob('./mmcv/ops/csrc/pytorch/*.cpp') + \
    glob.glob('./mmcv/ops/csrc/pytorch/cpu/*.cpp') + \
    glob.glob('./mmcv/ops/csrc/pytorch/cuda/*.cu') + \
    glob.glob('./mmcv/ops/csrc/pytorch/cuda/*.cpp')
extension = CUDAExtension
include_dirs.append(os.path.abspath('./mmcv/ops/csrc/pytorch'))
include_dirs.append(os.path.abspath('./mmcv/ops/csrc/common'))
include_dirs.append(os.path.abspath('./mmcv/ops/csrc/common/cuda'))
elif (hasattr(torch, 'is_mlu_available') and
...
```

在 CUDA 扩展的构建逻辑中，我们可以看到有环境变量'FORCE_CUDA' 来控制是否构建，也可以看到有 CUDA 相关的宏定义'MMCV_WITH_CUDA'，赋值 extension 为 CUDAExtension，然后就是源文件以及头文件的设置。因此我们也可以加一个 elif 分支并利用环境变量'FORCE_MUSA' 来控制是否构建，然后添加宏定义'MMCV_WITH_MUSA'。为了方便，我们直接对 mmcv/mmcv/ops/csrc 这个目录进行 CUDA-porting，会生成 mmcv/mmcv/ops/csrc_musa。所以我们在设置源 文件以及头文件的路径时只需将 csrc 改为 csrc_musa，最后将 extension 赋值为 MUSAExtension 就可以了。增加的分支如下所示：


```
...
elif os.getenv('FORCE_MUSA', '0') == '1':
  from torch_musa.utils.simple_porting import SimplePorting
  from torch_musa.utils.musa_extension import MUSAExtension
  SimplePorting(cuda_dir_path="./mmcv/ops/csrc", ignore_dir_paths=[
  "./mmcv/ops/csrc/common/mlu",
  "./mmcv/ops/csrc/common/mps",
  "./mmcv/ops/csrc/parrots",
  "./mmcv/ops/csrc/pytorch/mlu",
  "./mmcv/ops/csrc/pytorch/mps",
  "./mmcv/ops/csrc/pytorch/npu"
  ],
  mapping_rule={
      "_CU_H_": "_MU_H_",
      "_CUH": "_MUH",
      "__NVCC__": "__MUSACC__",
      "MMCV_WITH_CUDA": "MMCV_WITH_MUSA",
      "AT_DISPATCH_FLOATING_TYPES_AND_HALF": "AT_DISPATCH_FLOATING_TYPES",
      "#include <ATen/cuda/CUDAContext.h>": "#include \"torch_musa/csrc/aten/musa/MUSAContext.
↪h\"",
      "#include <c10/cuda/CUDAGuard.h>": "#include \"torch_musa/csrc/core/MUSAGuard.h\"",
      "::cuda::": "::musa::",
      "/cuda/": "/musa/",
      ", CUDA,": ", PrivateUse1,",
      ".cuh": ".muh",
      ".is_cuda()": ".is_privateuseone()",
      }
      ).run()
  op_files = glob.glob('./mmcv/ops/csrc_musa/pytorch/*.cpp') + \
      glob.glob('./mmcv/ops/csrc_musa/pytorch/cpu/*.cpp') + \
      glob.glob('./mmcv/ops/csrc_musa/pytorch/cuda/*.mu') + \
      glob.glob('./mmcv/ops/csrc_musa/pytorch/cuda/*.cpp')
  define_macros += [
      ('MMCV_WITH_MUSA', None)
      ]

  extension = MUSAExtension
  include_dirs.append(os.path.abspath('./mmcv/ops/csrc_musa/pytorch'))
  include_dirs.append(os.path.abspath('./mmcv/ops/csrc_musa/common'))
  include_dirs.append(os.path.abspath('./mmcv/ops/csrc_musa/common/cuda'))
elif (hasattr(torch, 'is_mlu_available') and
...
```

由于构建 MUSA 扩展时生成了额外的共享库，然后它会被链接到最终的目标库，因此在安装包时它需要被一起打包，具体示例见 torch_musa/utils/README.md。mmcv 中是以设置'include_package_data=True' 和配置 MANIFEST.in 文件来控制需要打包的数据的，原始 MANIFEST.in 文件如下所示：

```
include requirements/runtime.txt
include mmcv/ops/csrc/common/cuda/*.cuh mmcv/ops/csrc/common/cuda/*.hpp mmcv/ops/csrc/common/*.
↪hpp
include mmcv/ops/csrc/pytorch/*.cpp mmcv/ops/csrc/pytorch/cuda/*.cu mmcv/ops/csrc/pytorch/cuda/
↪*.cpp mmcv/ops/csrc/pytorch/cpu/*.cpp
include mmcv/ops/csrc/parrots/*.h mmcv/ops/csrc/parrots/*.cpp
include mmcv/ops/csrc/pytorch/mps/*.mm mmcv/ops/csrc/common/mps/*.h mmcv/ops/csrc/common/mps/*.
↪mm
recursive-include mmcv/ops/csrc/ *.h *.hpp *.cpp *.cuh *.cu *.mm
```

修改之后如下所示：

```
include requirements/runtime.txt
include mmcv/ops/csrc_musa/common/cuda/*.cuh mmcv/ops/csrc_musa/common/cuda/*.hpp mmcv/ops/
↪csrc_musa/common/*.hpp
include mmcv/ops/csrc_musa/pytorch/*.cpp mmcv/ops/csrc_musa/pytorch/cuda/*.cu mmcv/ops/csrc_
↪musa/pytorch/cuda/*.cpp mmcv/ops/csrc_musa/pytorch/cpu/*.cpp
include mmcv/ops/csrc_musa/parrots/*.h mmcv/ops/csrc_musa/parrots/*.cpp
include mmcv/ops/csrc_musa/pytorch/mps/*.mm mmcv/ops/csrc_musa/common/mps/*.h mmcv/ops/csrc_
↪musa/common/mps/*.mm
recursive-include mmcv/ops/csrc_musa/ *.h *.hpp *.cpp *.cuh *.cu *.mm
```

仅仅将 csrc 批量替换成 csrc_musa 即可。

8.2.4 尝试构建并测试

由于本次实验是在 MTT S3000 上进行，对应的 MUSA 架构版本为 21，而 mmcv 中涉及到 fp64 的使用，所以我们也要打开这个选项。对于这些额外的环境变量，可以参 考 torch_musa 根目录下的 CMakeLists.txt 和 build.sh。

接下来，我们尝试执行'MUSA_ARCH=21 ENABLE_COMPILE_FP64=1 FORCE_MUSA=1 python setup.py install > build.log' 构建 mmcv 并记录构建日志。很不幸，在第一次构建时遇到了一些编译错误，其中一个如下图所示：


```
/usr/local/musa/bin/mcc /home/mmcv/mmcv/ops/csrc_musa/pytorch/cuda/upfirdn2d_kernel.mu -c -o /home/mmcv/build/MMCV/CMakeFiles/mmcv__ext.dir/mmcv/ops/csrc_musa/pytorch/cuda/./mmcv__ext_generated_upfirdn2d_kernel.mu.o -fPIC -m64 -Dmmcv__ext_EXPORTS -DMMCV_WITH_MUSA -O2 -fPIC -Wall -Wextra -Wno-unused-parameter -Wno-unused-variable -Wno-unused-function -Wno-sign-compare -Wno-missing-field-initializers -fno-math-errno -fno-trapping-math -Werror=format -Werror=cast-function-type -Wno-unused-parameter -Wno-unused-variable -Wno-sign-compare -w -fPIC -O3 -DNDEBUG -DMMCV_WITH_MUSA -U__CUDA__ -I/usr/local/musa-1.5.1/include -I/opt/conda/envs/py38/lib/python3.8/site-packages/torch_musa/share/generated_cuda_compatible/aten/src -I/opt/conda/envs/py38/lib/python3.8/site-packages/torch_musa/share/generated_cuda_compatible/include -I/opt/conda/envs/py38/lib/python3.8/site-packages/torch_musa/share/generated_cuda_compatible/include/torch/csrc/api/include -I/opt/conda/envs/py38/lib/python3.8/site-packages -I/home/mmcv/mmcv/ops/csrc_musa/pytorch -I/home/mmcv/mmcv/ops/csrc_musa/common -I/home/mmcv/mmcv/ops/csrc_musa/common/cuda -I/usr/local/musa/include -I/opt/conda/envs/py38/include/python3.8
error: shared memory (42904) exceeds limit (28672) in function '_ZL22upfirdn2d_kernel_smallIdLi1ELi1ELi1ELi1ELi24ELi24ELi64ELi32ELi1EEv23upfirdn2d_kernel_params'
error: shared memory (31768) exceeds limit (28672) in function '_ZL22upfirdn2d_kernel_smallIdLi1ELi1ELi1ELi1ELi16ELi16ELi64ELi32ELi1EEv23upfirdn2d_kernel_params'
error: shared memory (28824) exceeds limit (28672) in function '_ZL22upfirdn2d_kernel_smallIdLi1ELi1ELi1ELi1ELi24ELi24ELi32ELi32ELi1EEv23upfirdn2d_kernel_params'
error: shared memory (31384) exceeds limit (28672) in function '_ZL22upfirdn2d_kernel_smallIdLi1ELi1ELi1ELi1ELi7ELi7ELi16ELi16ELi8EEv23upfirdn2d_kernel_params'
error: shared memory (41776) exceeds limit (28672) in function '_ZL22upfirdn2d_kernel_smallIdLi1ELi1ELi2ELi2ELi24ELi24ELi32ELi16ELi1EEv23upfirdn2d_kernel_params'
error: shared memory (30768) exceeds limit (28672) in function '_ZL22upfirdn2d_kernel_smallIdLi1ELi1ELi2ELi2ELi16ELi16ELi32ELi16ELi1EEv23upfirdn2d_kernel_params'
error: shared memory (31504) exceeds limit (28672) in function '_ZL22upfirdn2d_kernel_smallIdLi1ELi1ELi2ELi2ELi8ELi8ELi8ELi8EEv23upfirdn2d_kernel_params'
7 errors generated when compiling for mp_10.
-- Removing /home/mmcv/build/MMCV/CMakeFiles/mmcv__ext.dir/mmcv/ops/csrc_musa/pytorch/cuda/./mmcv__ext_generated_upfirdn2d_kernel.mu.o
/opt/conda/envs/py38/lib/python3.8/site-packages/cmake/data/bin/cmake -E remove /home/mmcv/build/MMCV/CMakeFiles/mmcv__ext.dir/mmcv/ops/csrc_musa/pytorch/cuda/./mmcv__ext_generated_upfirdn2d_kernel.mu.o
CMake Error at mmcv__ext_generated_upfirdn2d_kernel.mu.o.Release.cmake:283 (message):
  Error generating file
  /home/mmcv/build/MMCV/CMakeFiles/mmcv__ext.dir/mmcv/ops/csrc_musa/pytorch/cuda/./mmcv__ext_generated_upfirdn2d_kernel.mu.o
```

这是由于定义的结构体（upfirdn2d_kernel_params）要使用的 shared memory 过大，超过了硬件（此次编译是在 MTT S3000 上进行的）规格的限制，因此我们尝试避免构建该 kernel 的 musa 扩展（mmcv/mmcv/ops/csrc_musa/pytorch/cuda/upfirdn2d_kernel.mu）。如果您的模型中没有真实用到该 kernel，那么可以将其注释起来，临时绕过该算子，保证模型的正常运行。如果您的模型确认需要使用该 kernel，那么请联系摩尔线程 AI 研发中心，反馈该问题（在外网提 issue），我们及时修复。同理，对于其他的编译错误也是可以进行类似的修改。

汇总一下，我们对 mmcv 进行 MUSA 适配需要修改如下文件：

- MANIFEST.in
- mmcv/ops/csrc/common/cuda/carafe_cuda_kernel.cuh
- mmcv/ops/csrc/common/cuda/chamfer_distance_cuda_kernel.cuh
- mmcv/ops/csrc/common/cuda/scatter_points_cuda_kernel.cuh
- mmcv/ops/csrc/pytorch/cuda/upfirdn2d_kernel.cu
- setup.py

再次测试本节开头的例子，我们得到结果如下：


```
>>> import torch_musa
>>> from mmcv.ops import nms
>>> np_boxes = np.array([[6.0, 3.0, 8.0, 7.0], [3.0, 6.0, 9.0, 11.0],
...                      [3.0, 7.0, 10.0, 12.0], [1.0, 4.0, 13.0, 7.0]],
...                      dtype=np.float32)
>>> np_scores = np.array([0.6, 0.9, 0.7, 0.2], dtype=np.float32)
>>> np_inds = np.array([1, 0, 3])
>>> np_dets = np.array([[3.0, 6.0, 9.0, 11.0, 0.9],
...                     [6.0, 3.0, 8.0, 7.0, 0.6],
...                     [1.0, 4.0, 13.0, 7.0, 0.2]])
>>> boxes = torch.from_numpy(np_boxes)
>>> scores = torch.from_numpy(np_scores)
>>> # check if cpu can work
>>> dets, inds = nms(boxes, scores, iou_threshold=0.3, offset=0)
>>> # check if musa can work
>>> dets, inds = nms(boxes.musa(), scores.musa(), iou_threshold=0.3, offset=0)
>>> dets
tensor([[ 3.0000,  6.0000,  9.0000, 11.0000,  0.9000],
        [ 6.0000,  3.0000,  8.0000,  7.0000,  0.6000],
        [ 1.0000,  4.0000, 13.0000,  7.0000,  0.2000]], device='musa:0')
>>>
```

当然这并不能证明适配的 mmcv 的功能完全，我们可以对 mmcv 自带的单元测试进行简单的改动就可以进行测试了。如 tests/test_ops/test_box_iou_quadri.py：

```python
# Copyright (c) OpenMMLab. All rights reserved.
import numpy as np
import pytest
import torch
import torch_musa

# from mmcv.utils import IS_CUDA_AVAILABLE


class TestBoxIoUQuadri:

    @pytest.mark.parametrize('device', [
        'cpu',
        pytest.param(
            'musa',
            marks=pytest.mark.skipif(
                not True, reason='requires MUSA support')),
    ])
    def test_box_iou_quadri_musa(self, device):
        from mmcv.ops import box_iou_quadri
        np_boxes1 = np.asarray([[1.0, 1.0, 3.0, 4.0, 4.0, 4.0, 4.0, 1.0],
                                [2.0, 2.0, 3.0, 4.0, 4.0, 2.0, 3.0, 1.0],
```

```
                            [7.0, 7.0, 8.0, 8.0, 9.0, 7.0, 8.0, 6.0]],
                           dtype=np.float32)
        np_boxes2 = np.asarray([[0.0, 0.0, 0.0, 2.0, 2.0, 2.0, 2.0, 0.0],
                            [2.0, 1.0, 2.0, 4.0, 4.0, 4.0, 4.0, 1.0],
                            [7.0, 6.0, 7.0, 8.0, 9.0, 8.0, 9.0, 6.0]],
                           dtype=np.float32)
        np_expect_ious = np.asarray(
            [[0.0714, 1.0000, 0.0000], [0.0000, 0.5000, 0.0000],
            [0.0000, 0.0000, 0.5000]],
            dtype=np.float32)
        np_expect_ious_aligned = np.asarray([0.0714, 0.5000, 0.5000],
                                            dtype=np.float32)

        boxes1 = torch.from_numpy(np_boxes1).to(device)
        boxes2 = torch.from_numpy(np_boxes2).to(device)

        ious = box_iou_quadri(boxes1, boxes2)
        assert np.allclose(ious.cpu().numpy(), np_expect_ious, atol=1e-4)

        ious = box_iou_quadri(boxes1, boxes2, aligned=True)
        assert np.allclose(
            ious.cpu().numpy(), np_expect_ious_aligned, atol=1e-4)

    @pytest.mark.parametrize('device', [
        'cpu',
        pytest.param(
            'musa',
            marks=pytest.mark.skipif(
                not True, reason='requires MUSA support')),
    ])
    def test_box_iou_quadri_iof_musa(self, device):
        from mmcv.ops import box_iou_quadri
        np_boxes1 = np.asarray([[1.0, 1.0, 3.0, 4.0, 4.0, 4.0, 4.0, 1.0],
                            [2.0, 2.0, 3.0, 4.0, 4.0, 2.0, 3.0, 1.0],
                            [7.0, 7.0, 8.0, 8.0, 9.0, 7.0, 8.0, 6.0]],
                           dtype=np.float32)
        np_boxes2 = np.asarray([[0.0, 0.0, 0.0, 2.0, 2.0, 2.0, 2.0, 0.0],
                            [2.0, 1.0, 2.0, 4.0, 4.0, 4.0, 4.0, 1.0],
                            [7.0, 6.0, 7.0, 8.0, 9.0, 8.0, 9.0, 6.0]],
                           dtype=np.float32)
        np_expect_ious = np.asarray(
            [[0.1111, 1.0000, 0.0000], [0.0000, 1.0000, 0.0000],
```

```
            [0.0000, 0.0000, 1.0000]],
            dtype=np.float32)
        np_expect_ious_aligned = np.asarray([0.1111, 1.0000, 1.0000],
                                            dtype=np.float32)

        boxes1 = torch.from_numpy(np_boxes1).to(device)
        boxes2 = torch.from_numpy(np_boxes2).to(device)

        ious = box_iou_quadri(boxes1, boxes2, mode='iof')
        assert np.allclose(ious.cpu().numpy(), np_expect_ious, atol=1e-4)

        ious = box_iou_quadri(boxes1, boxes2, mode='iof', aligned=True)
        assert np.allclose(
            ious.cpu().numpy(), np_expect_ious_aligned, atol=1e-4)
```

我们进入到 mmcv/tests/test_ops 目录下，然后执行'pytest -s test_box_iou_quadri.py' 就可以测试该单元测试用例了，测试结果如下所示：

```
(py38) root@mccx:/home/mmcv/tests/test_ops# pytest -s test_box_iou_quadri.py
============================= test session starts ==============================
platform linux -- Python 3.8.18, pytest-7.2.2, pluggy-1.3.0
rootdir: /home/mmcv
plugins: hypothesis-6.91.0
collected 4 items

test_box_iou_quadri.py ....

============================== 4 passed in 4.47s ===============================
```

8.3 基于 MUSAExtension 构建定制化算子

本小节以构建 swintransformer 中的 window_process_kernel 算子[5] 为例介绍如何通过 cuda-porting 和 MUSAExtension 完成 CUDA 代码到 MUSA 代码的转译及构建。

原始目录结构如下：

```
window_process
   setup.py
   swin_window_process.cpp
   swin_window_process_kernel.cu
   unit_test.py
   window_process.py
```

首先在 window_process 同级目录下执行 cuda-porting 命令将 CUDA 代码转化为 MUSA 代码：`python -m torch_musa.utils.simple_porting --cuda-dir-path ./window_process --mapping-rule "{\"cuda\":\"musa\",\"torch::kCUDA\":\"torch::kPrivateUse1\",`

[5] https://github.com/microsoft/Swin-Transformer/tree/main/kernels/window_process

\"type().is_cuda\":\"is_privateuseone\"}", 命令执行完毕后会在当前目录下生成名为 window_process_musa 的新目录，该目录结构如下：

```
window_process_musa
  swin_window_process.cpp
  swin_window_process_kernel.mu
```

之后基于 MUSAExtension 编写 setup.py 文件，该文件写法与 CUDAExtension 类似，其代码如下：

```
from setuptools import setup
from torch.utils.cpp_extension import BuildExtension
from torch_musa.utils.musa_extension import MUSAExtension



setup(
    name="swin_window_process",
    ext_modules=[
        MUSAExtension(
            "swin_window_process",
            [
                "swin_window_process.cpp",
                "swin_window_process_kernel.mu",
            ],
        )
    ],
    cmdclass={"build_ext": BuildExtension},
)
```

在编译并安装 MUSA 扩展时需要通过 MUSA_ARCH 环境变量指定架构号，在 S80 或 S3000 机器上可执行 `MUSA_ARCH=21 python setup.py install`，在 S4000 机器上可执行 `MUSA_ARCH=22 python setup.py install` 完成 window_process_kernel 扩展的安装，之后在 python 解释器中尝试 `import swin_window_process` 若无错误出现则表示定制化算子构建并安装成功。

摩尔线程
MOORE THREADS

9 性能优化

9.1 profiler 工具

可以使用 PyTorch 官方性能分析工具 **torch.profiler** 来对 torch_musa 进行性能分析，使用方法可以参考 官方示例[6] 。需要注意的是，当前 torch_musa 不支持 **ProfilerActivity.MUSA** ，我们需要开启同步模式 `export MUSA_LAUNCH_BLOCKING=1` , 通过 CPU 耗时信息做性能分析。**ProfilerActivity.MUSA** 相关特性正在开发中，在不久的新版本中 torch_musa 将会对其进行支持。

9.2 使能 TensorCore 优化

s4000 支持 **TensorFloat32(TF32)** tensor cores。利用 tensor cores，可以使得 **矩阵乘** 和 **卷积** 计算得到加速。torch_musa 中 TF32 的使用方式和 CUDA 中一致，可以参考 PyTorch 官方示例[7] 。在快速入门章节中，我们也提供了示例代码，见*使能 TensorCore 示例代码* 。需要注意的是，CUDA 中矩阵乘和卷积计算是分开控制的，而在 torch_musa 中是统一控制的，代码差异如下所示。在 torch_musa 中 **allow_tf32** 默认值是 False。

```
import torch
# The flag below controls whether to allow TF32 on matmul. This flag defaults to False
# in PyTorch 1.12 and later.
torch.backends.cuda.matmul.allow_tf32 = True

# The flag below controls whether to allow TF32 on cuDNN. This flag defaults to True.
torch.backends.cudnn.allow_tf32 = True

import torch_musa
# The flag below controls whether to allow TF32 on muDNN. This flag defaults to False.
torch.backends.mudnn.allow_tf32 = True
```

在 **s4000** 上，对于具有卷积算子的模型，使能 **NHWC layout 优化**，可以使得性能得到提升，示例代码如下：

```
import torch
import torch_musa

torch.backends.mudnn.allow_tf32 = True
```

[6] https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html
[7] https://pytorch.org/docs/stable/notes/cuda.html#tensorfloat-32-tf32-on-ampere-devices

```
model = Model() # define model here
model = model.to(memory_format=torch.channels_last) # transform layout to NHWC
```


摩尔线程 MOORE THREADS

10 调试工具

10.1 CompareTool 使用指南

如果发现模型准确率不对，可以通过算子精度对比工具来快速找出异常算子并进行修复。对比工具基于 PyTorch 的 Dispatcher[8] 和 DispatchMode[9] 实现。在算子被 dispatch 到 MUSA device 之前捕获到调用函数，然后分别调用 CPU 算子函数和 MUSA 算子函数进行计算，得出结果进行对比。所有算子（包括前向算子和反向算子）都会被依次 dispatch 并输出对比结果到日志，通过查询日志可以快速定位出第一个出现计算异常的算子。该工具在训练和推理中都能被使用。

10.1.1 基本用法

```
from torch_musa.utils.compare_tool import CompareWithCPU

with CompareWithCPU(atol=0.001, rtol=0.001, verbose=True):
    ...original code...
```

在这段代码中，`CompareWithCPU` 工具被用来比较在 MUSA 设备上执行算子作和在 CPU 上执行的相同算子的结果。这里，`atol` 和 `rtol` 分别代表绝对公差和相对公差，用于在比较时确定允许的误差范围。当 `verbose` 设置为 `True` 时，它会输出更详细的信息，包括输入输出 tensor 的 shape、dtype、stride、nan_num 等信息。

10.1.2 输出示例

输出内容大致如下：

```
============================
============================
torch._ops.aten...div.Tensor
....... input .........
0:  Tensor <shape=torch.Size([16, 3, 640, 640]), stride=(1228800, 409600, 640, 1), dtype=torch.
↪float32, device=musa:0, size=19660800,>,
1: 255,
...... output ........
Tensor <shape=torch.Size([16, 3, 640, 640]), stride=(1228800, 409600, 640, 1), dtype=torch.
↪float32, device=musa:0, size=19660800,>
```

[8] http://blog.ezyang.com/2020/09/lets-talk-about-the-pytorch-dispatcher/
[9] https://dev-discuss.pytorch.org/t/torchdispatchmode-for-debugging-testing-and-more/717

```
...... compare with cpu .......
torch._ops.aten...div.Tensor succeeds to pass CompareWithCPU test

============================
torch._ops.aten..._to_copy.default
....... input .........
0:  Tensor <shape=torch.Size([48, 3, 6, 6]), stride=(108, 1, 18, 3), dtype=torch.float32,␣
↪device=musa:0, size=5184,>,

...... output ........
Tensor <shape=torch.Size([48, 3, 6, 6]), stride=(108, 1, 18, 3), dtype=torch.float16,␣
↪device=musa:0, size=5184,>

...... compare with cpu .......
torch._ops.aten..._to_copy.default succeeds to pass CompareWithCPU test


============================
torch._ops.aten..._to_copy.default
....... input .........
0:  Tensor <shape=torch.Size([16, 3, 640, 640]), stride=(1228800, 409600, 640, 1), dtype=torch.
↪float32, device=musa:0, size=19660800,>,

...... output ........
Tensor <shape=torch.Size([16, 3, 640, 640]), stride=(1228800, 409600, 640, 1), dtype=torch.
↪float16, device=musa:0, size=19660800,>

...... compare with cpu .......
torch._ops.aten..._to_copy.default succeeds to pass CompareWithCPU test


============================
torch._ops.aten...convolution.default
....... input .........
0:  Tensor <shape=torch.Size([16, 3, 640, 640]), stride=(1228800, 409600, 640, 1), dtype=torch.
↪float16, device=musa:0, size=19660800,>,
1:  Tensor <shape=torch.Size([48, 3, 6, 6]), stride=(108, 1, 18, 3), dtype=torch.float16,␣
↪device=musa:0, size=5184,>,
2: None,
3: [2, 2, ],
4: [2, 2, ],
```



```
5: [1, 1, ],
6: False,
7: [0, 0, ],
8: 1,

...... output ........
Tensor <shape=torch.Size([16, 48, 320, 320]), stride=(4915200, 1, 15360, 48), dtype=torch.
↪float16, device=musa:0, size=78643200,>

...... compare with cpu .......
"slow_conv2d_cpu" not implemented for 'Half'
Convert to float32 ...
.......... output 0 is not close ........

Too many indices (total 2388336) to print
...
Element at index (0, 3, 1, 75) is not close: 0.5458984375 vs 0.5460812449455261
Element at index (0, 3, 1, 78) is not close: 0.5498046875 vs 0.5499854683876038
Element at index (0, 3, 1, 86) is not close: 0.55029296875 vs 0.5501253008842468
Element at index (0, 3, 1, 88) is not close: 0.55126953125 vs 0.5510903000831604
Element at index (0, 3, 1, 91) is not close: 0.55029296875 vs 0.5504764914512634
Element at index (0, 3, 1, 94) is not close: 0.54296875 vs 0.5427778959274292
Element at index (0, 3, 1, 101) is not close: 0.5361328125 vs 0.5359049439430237
Element at index (0, 3, 1, 103) is not close: 0.54638671875 vs 0.5466215014457703
Element at index (0, 3, 1, 104) is not close: 0.54296875 vs 0.5431610941886902
Element at index (0, 3, 1, 108) is not close: 0.54296875 vs 0.5427677631378174
Element at index (0, 3, 1, 110) is not close: 0.5390625 vs 0.5392988920211792
Element at index (0, 3, 1, 112) is not close: 0.5009765625 vs 0.5012078881263733
Element at index (0, 3, 1, 114) is not close: 0.54052734375 vs 0.5403239130973816
Element at index (0, 3, 1, 115) is not close: 0.5361328125 vs 0.5363231897354126
Element at index (0, 3, 1, 117) is not close: 0.5234375 vs 0.5236586332321167
Element at index (0, 3, 1, 118) is not close: 0.5029296875 vs 0.5027626156806946
Element at index (0, 3, 1, 133) is not close: 0.537109375 vs 0.5373141765594482
Element at index (0, 3, 1, 136) is not close: 0.513671875 vs 0.5134815573692322
Element at index (0, 3, 1, 143) is not close: 0.5029296875 vs 0.5031570196151733
Element at index (0, 3, 1, 144) is not close: 0.51953125 vs 0.519349217414856



tensor 1: shape=torch.Size([16, 48, 320, 320]), numbers of nan = 0 of 78643200, numbers of inf␣
↪= 0 of 78643200
tensor([[[[-1.47583e-01, -1.81030e-01, -1.80420e-01,  ..., -1.57349e-01, -1.55396e-01, -1.
↪71204e-02],
```

```
        [-1.83838e-01, -1.55762e-01, -1.54785e-01,  ..., -1.35498e-01, -1.33667e-01,  3.33252e-
↪02],
        [-1.81885e-01, -1.56128e-01, -1.56738e-01,  ..., -1.50024e-01, -1.37939e-01,  3.04413e-
↪02],
        ...,

        ..., ], device='musa:0', dtype=torch.float16)


tensor 2 (golden): shape=torch.Size([16, 48, 320, 320]), numbers of nan = 0 of 78643200,␣
↪numbers of inf = 0 of 78643200
tensor([[[[-1.47546e-01, -1.81061e-01, -1.80469e-01,  ..., -1.57394e-01, -1.55356e-01, -1.
↪71164e-02],
        [-1.83810e-01, -1.55728e-01, -1.54838e-01,  ..., -1.35546e-01, -1.33724e-01,  3.33278e-
↪02],
        [-1.81920e-01, -1.56185e-01, -1.56751e-01,  ..., -1.49999e-01, -1.37939e-01,  3.04426e-
↪02],
        ...,
        [-1.57577e-01, -1.27439e-01, -1.29771e-01,  ..., -1.27387e-01, -1.08608e-01,  1.68391e-
↪02],
        [-1.57745e-01, -1.29170e-01, -1.31028e-01,  ..., -1.31670e-01, -1.23129e-01,  1.17836e-
↪02],
        [ 1.78030e-02, -5.00134e-03, -7.76099e-03,  ..., -2.74925e-02, -2.82936e-02,  2.69481e-
↪02]],
        ...
        ])
all_resuls=[False]
[ERROR] torch._ops.aten...convolution.default fails to pass CompareWithCPU test
```

这段输出显示了在 MUSA 设备上执行的 `torch._ops.aten.div.Tensor`，`torch._ops.aten._to_copy.default`，`torch._ops.aten.convolution.default` 算子和在 CPU 上执行的相同算子的输入、输出对比结果，以及它们是否成功通过了比较测试。

10.1.3 错误追踪与调试

如果在测试中发现错误或不一致，您可以在日志中搜索"[WARNING]" 来追踪产生 nan/inf 的位置，搜索"[ERROR]" 来追踪与 CPU 结果不一致的算子。然后，您可以调整 `atol` 和 `rtol` 的值再次尝试。或者设置 `target_list` 来快速重现异常算子行为，并设置 `dump_error_data` 来保存异常算子的输入数据：

```
from torch_musa.utils.compare_tool import CompareWithCPU
```

```
with CompareWithCPU(atol=0.001, rtol=0.001, target_list=['convolution.default'], dump_error_
↪data=True):
    ...original code...
```

这段代码将会把输入参数保存到 convolution.default_args.pkl 文件中，之后您可以用这个文件来生成单个错误用例进行调试：

```
import torch
import torch_musa
from torch_musa.utils.compare_tool import compare_single_op

compare_single_op('convolution.default_args.pkl', torch.ops.aten.convolution.default, atol=0.
↪0001, rtol=0.0001)
```

使用 compare_single_op 函数，您可以对特定的算子和输入进行更细致的比较和调试。

10.2 调试注册接口使用指南：

目前，torch-musa 中的部分算子已经使用调试注册接口进行了封装，可以在运行时通过环境变量启用或关闭调试功能。

10.2.1 使用说明：

在运行模型前，使用下列环境变量启用和配置调试功能：

TORCH_MUSA_OP_DEBUG=on # 默认为关闭。指定为 on 时启用调试功能，off 关闭。

TORCH_MUSA_OP_DEBUG_LEVEL=1 # 默认为 1，可选 1、2、3、4、5、6 个等级。具体等级对应的功能见下文。注意：部分等级会严重影响性能，并可能占用大量磁盘空间。

TORCH_MUSA_OP_DEBUG_DIR=~/debug # 指定调试日志的输出目录，默认为./DEBUG_DIR，推荐使用绝对路径。调试工具会自动创建该目录，如果目录已经存在，则会创建添加有数字编号后缀的新目录以避免重复。

TORCH_MUSA_OP_DEBUG_LIST="" # 指定调试算子的名单，默认为未配置。使用英文逗号 (,) 分隔，不区分大小写，支持部分匹配。指定该环境变量时，只有算子名称或算子别名中包含列出的名称之一时，才会记录该算子的信息。当该环境变量未配置（unset）时，将默认包含所有算子。

TORCH_MUSA_OP_DEBUG_BLACK_LIST="" # 指定调试算子的黑名单，默认为未配置。使用英文逗号 (,) 分隔，不区分大小写，支持部分匹配。指定该环境变量时，只有算子名称和算子别名中不包含任意列出的名称时，才会记录该算子的信息。该环境变量不能和 TORCH_MUSA_OP_DEBUG_LIST 同时使用。

TORCH_MUSA_OP_DEBUG_LENGTH=50 # 指定调试张量数据类型时记录其部分值的最大长度 N。默认为 50。增加该数值将可能影响性能，并占用额外磁盘空间。

10.2.2 模式说明：

指定 TORCH_MUSA_OP_DEBUG_LEVEL 可以调整运行模式。目前支持 6 个不同等级的模式，使用 1-6 标识。

具体模式功能说明如下：（推荐使用 1、2、3 或 5 等级。4 和 6 等级耗时极长并可能占用极大的磁盘空间！）

1：将所有调用到的算子，以及其数据规模、张量大小和标量值等信息记录到文件中。

2：在 1 的基础上，统计算子的张量信息，即记录其极大值、极小值、均值、方差等信息，并记录到文件中。（耗时较长）

3：在 1 的基础上，将算子张量的部分值（长度为 N，可以使用 TORCH_MUSA_OP_DEBUG_LENGTH 进行配置）记录到文件中。

4：在 1 的基础上，将算子张量的全部值记录到文件中。（警告，该模式将可能记录大量数据，且耗时极长）。

5：2 和 3 模式的组合，即统计张量信息的同时，也记录其部分值（长度为 N）。（耗时较长）。

6：2 和 4 模式的组合，即统计张量信息的同时，也记录其全部值。（警告，该模式将可能记录大量数据，且耗时极长）

使用本工具时，会在指定的目录下首先创建一个名为 full_log.txt 的文本文件，该文件记录了算子的基础信息。同时，每个算子将会创建一个以其算子名称命名，并添加数字编号前缀的目录，目录中记录了算子内的数据信息。

使用调试注册接口可以在运行时记录模型中各个算子的数据，因此能够发现计算故障（如 INF、NaN 等）。

摩尔线程 MOORE THREADS

11 FAQ

11.1 设备查看问题

Q：如果在安装完驱动后，普通用户（非 root 用户）无法查看显卡，使用 sudo 权限才可以查看显卡?

```
--------------------------------------------------------------------------------
test_clinfo@mccx:/home/mccxadmin$ clinfo
Number of platforms                                   0
test_clinfo@mccx:/home/mccxadmin$ mthreads-gmi
Error: failed to initialize mtml.
test_clinfo@mccx:/home/mccxadmin$
```

将该用户添加进 render group 后即可。

1. 如果环境中已经有 render group，执行下述命令即可：

```
sudo usermod -aG render `whoami`
```

2. 如果环境中没有 render group，执行下述命令即可：

```
sudo groupadd -o -g 109 render
sudo usermod -aG render `whoami`
```

11.2 计算库无法找到

Q：如果在安装 torch_musa wheel 包后，`import torch_musa` 时报错无法找到 mudnn.so 库?

```
(test) ➜  tmp python
Python 3.9.17 (main, Jul  5 2023, 20:41:20)
[GCC 11.2.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> import torch_musa
Traceback (most recent call last):
  File "/home/mt_developer/miniconda3/envs/test/lib/python3.9/site-packages/torch_musa/__init__.py", line
 27, in <module>
    import torch_musa._MUSAC
ImportError: libmudnn.so.1: cannot open shared object file: No such file or directory

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/mt_developer/miniconda3/envs/test/lib/python3.9/site-packages/torch_musa/__init__.py", line
 29, in <module>
    raise ImportError("Please try running Python from a different directory!") from err
ImportError: Please try running Python from a different directory!
```

1. 请确认 /usr/local/musa/lib/ 目录下是否有该库，如果没有的话，需要安装该数学库；如果有的话，需要执行：

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/musa/lib
```

11.3 编译安装

Q：如果在更新过 torch_musa 最新代码后，编译报错?

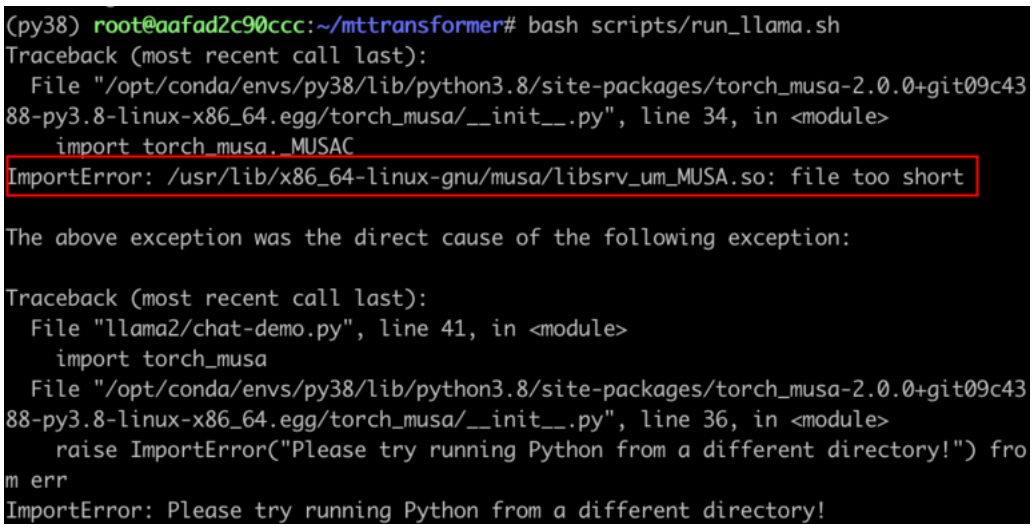
1. 请尝试

```
python setup.py clean
bash build.sh # 整体重新编译
```

如果还报错，可能是因为需要更新 MUSA 软件栈中某个底层软件包。

11.4 Docker 容器

Q：如果在 docker container 内部使用 torch_musa 时，报错 ImportError: libsrv_um_MUSA.so: cannot open shared object file: No such file or directory 或者 ImportError: /usr/lib/x86_64-linux-gnu/musa/libsrv_um_MUSA.so: file too short？



1. 请确保成功安装 mt-container-toolkit，安装步骤可以参考 mt-container-toolkit 文档[10]；请务必注意下面两点：

- 绑定摩尔线程容器运行时到 Docker，执行下图中红框中命令：

[10] https://mcconline.mthreads.com/software/1?id=1

安装 Docker CE

```
sudo apt install docker.io
```

更多安装信息请参考官方文档。

安装摩尔线程容器运行时套件

使用 dpkg 包管理工具进行安装：

```
sudo dpkg -i mtml_1.5.0.deb sgpu-dkms_1.1.1.deb mt-container-toolkit_1.5.0.deb
```

绑定摩尔线程容器运行时到 Docker，设置默认的容器运行时为 `mthreads` 并重启 Docker daemon：

```
$ (cd /usr/bin/musa && sudo ./docker setup $PWD)
```

您可以通过如下命令验证上述步骤是否成功：

```
❯ docker run --rm --env MTHREADS_VISIBLE_DEVICES=all ubuntu:20.04 mthreads-gmi
Mon Jul 17 06:42:48 2023
---------------------------------------------------------------
```

- 在启动 docker container 时请添加 `--env MTHREADS_VISIBLE_DEVICES=all` 。

11.5 适配算子

Q：如果在 CUDA-Porting 适配新算子时，编译可以通过，在 `import torch;import torch_musa` 时报错找不到符号？

```
[GCC 7.5.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> import torch_musa
Traceback (most recent call last):
  File "/opt/conda/envs/test_environment/lib/python3.8/site-packages/torch_musa-
2.0.0-py3.8-linux-x86_64.egg/torch_musa/__init__.py", line 23, in <module>
    import torch_musa._MUSAC
ImportError: /opt/conda/envs/test_environment/lib/python3.8/site-packages/torch_
musa-2.0.0-py3.8-linux-x86_64.egg/torch_musa/lib/libmusa_kernels.so: undefined s
ymbol: _ZN2at4musa3cub28exclusive_sum_in_common_typeIiiEEvPKT_PT0_l

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/opt/conda/envs/test_environment/lib/python3.8/site-packages/torch_musa-
2.0.0-py3.8-linux-x86_64.egg/torch_musa/__init__.py", line 25, in <module>
    raise ImportError("Please try running Python from a different directory!") f
rom err
ImportError: Please try running Python from a different directory!
>>> exit()
(test_environment) root@77f6215babda:/home# c++filt _ZN2at4musa3cub28exclusive_s
um_in_common_typeIiiEEvPKT_PT0_l
void at::musa::cub::exclusive_sum_in_common_type<int, int>(int const*, int*, lon
g)
(test_environment) root@77f6215babda:/home# cd -
/home/torch_musa
(test_environment) root@77f6215babda:/home/torch_musa# vim torch_musa/csrc/CMake
```


1. 先用 c++filt 查看符号名称
2. 在 PyTorch 源码中 grep 搜索这个符号：

- 如果该符号定义在 cu 文件中，那么把该 cu 文件对应的 ****.mu 文件加入到 torch_musa/csrc/CMakeList.txt 中即可。
- 如果该符号定义在 cpp 文件中，那么这是一个 bug，请向 torch_musa 提交一个 issue。
- 如果 PyTorch 中也没有这个符号，那么请在 `/usr/local/musa/` 中去 grep 搜索这个符号，这个符号可能是底层库定义的。
  - 如果找到这个符号，那么请检查是否没有链接这个底层库，查看命令可参考 `ldd path/to/site-packages/torch_musa-2.0.0-py3.8-linux-x86_64.egg/torch_musa/lib/libmusa_kernels.so`
  - 如果找到这个符号，且已经链接了对应底层库，可能底层库只暴露了这个符号，但是还未给出定义。如果是底层库暴露符号，但是 porting 的算子实际运行时没有调用符号，那么我们可以在 torch_musa 中定义一个空的实现，参考 `torch_musa/csrc/aten/ops/musa/unimplemented_functions.cpp` 。如果底层库暴露符号，porting 的算子实际运行需要调用这个符号，那么可以给底层库提交需求。

3. 如果上面都没有找到上述符号，可以在 CUDA PyTorch 环境中下 grep 搜索一下，看看 CUDA 环境中这个符号定义在哪里，再和对应 MUSA 软件模块提交需求。

11.6 问题与反馈

如果在开发或者使用 torch_musa 的过程中，遇到任何 bug 或者没支持的特性，请积极向 torch_musa (https://github.com/MooreThreads/torch_musa/issues) 提交 issue，我们会及时作出反馈。提交 issue 时，请给出复现问题的代码，报错 log，并且打上对应的标签，如下面例子所示：