task3 report

张艺洋

January 2026

1 简介

任务 3 报告我将介绍 Mytorch 中卷积算子的实现，与 pytorch 的对比分析。并且呈现在 Pytorch 和 Mytorch 上面进行 Cifar10 数据集的训练和测试的效果对比。

本报告中所有实验使用的 GPU 都是笔记本自带 GPU：NVIDIA GeForce RTX 4070。

2 卷积算子实现

底层代码在 `mytorch/src/operators.cu` 文件中。

2.1 前向传播

1. 实现 `im2col` 来实现对 input 的转换，以把卷积操作化为矩阵形式。这里我对每个 batch 的数据调用了 `im2col_kernel` 来对数据进行处理。在 kernel 内部，首先根据 index 算出线程处理的点在 output 和 input 中的位置，随后进行数据从 input 向 output 的存储，对于越界的坐标则根据 padding 原理置为 0.

2. 首先计算 `im2col(input)`, 然后使用矩阵乘法计算出对应结果，再调用 kernel 在每个 batch 计算出的结果上加上 bias 即可。

2.2 反向传播

1. 与 `im2col` 同理，实现 `col2im` ，把求出的 $\frac{\partial L}{\partial \tilde{x}}$ 转化到和原先 input 维度一致的 $\frac{\partial L}{\partial x}$.

2. 通过矩阵乘法和 kernel launch 计算剩余几个参数的梯度。

3 卷积算子对比

通过测试 Pytorch 与 Mytorch 的卷积算子的运行时间来进行对比。

3.1 测试方法

使用 `python comparison.py` 即可运行代码并且看到对应的参数下的结果。

牢记王鹏帅老师上课讲的学生单开 `time.time()` 的故事,在计算运行时间的时候,使用 `torch.cuda.synchronize()` 保证所有 GPU 运行结束之后 CPU 才会读取时间。

而且考虑到 GPU 的启动时间，我在每个算子测试时都先进行 10 个回合的 warm up，然后测量 50 个回合 forward+backward 运行的平均时间作为结果。

3.2 对比数据

表 1: 不同 Batch Size 下 PyTorch 与 MyTorch 卷积性能对比

Batch Size	PyTorch (ms)	MyTorch (ms)	Speedup
1	0.316	0.154	2.06×
4	0.315	0.313	1.01×
8	0.275	1.161	0.24×
16	0.275	2.016	0.14×
32	0.313	3.946	0.08×

表 2: 不同输入分辨率下 PyTorch 与 MyTorch 卷积性能对比

Input Size ($H \times W$)	PyTorch (ms)	MyTorch (ms)	Speedup
8×8	0.148	0.313	0.47×
16×16	0.314	0.684	0.46×
32×32	0.275	1.161	0.24×
64×64	0.256	2.743	0.09×

表 3: 不同通道配置下 PyTorch 与 MyTorch 卷积性能对比

C_{in}	C_{out}	PyTorch (ms)	MyTorch (ms)	Speedup
3	8	0.275	1.161	0.24×
8	8	0.241	1.894	0.13×
8	16	0.308	1.867	0.17×

3.3 对比分析

1. 膜拜 PyTorch：在不同卷积核参数下，PyTorch 的运行时间几乎没有明显变化，而 MyTorch 则呈现近似线性的时间增长。这主要是因为 PyTorch 的卷积算子底层调用了 `cuDNN` 库，并利用了高度优化的 GPU kernel，包括 Tensor Core 加速和 kernel fusion。即使卷积核参数变化，cuDNN 会根据输入尺寸、通道数和卷积参数选择最优算法，从而保证运行时间相对稳定。[1][4]

2. MyTorch 在并行运行上存在限制,其底层实现并没有真正意义上的 batch 并行处理,这意味着随着 `batch_size` 增大，所有样本只能顺序计算，GPU 并行利用率低，从而导致性能快速下降。相比之下，PyTorch 会将整个 batch 作为一个大张量输入，充分利用 GPU 的并行能力和 memory coalescing，从而显著提升大 batch 下的性能。

3. 然而，当 `batch_size` 较小时，MyTorch 在运行时间上仍然与 PyTorch 相当甚至略快。这是因为在小 batch 的情况下, GPU kernel launch 的开销占主导, 而 MyTorch 的自定义 kernel 较轻量, 没有 cuDNN 的调度和 autotune 开销，因此在短小任务中反而略有优势。[2]

4. 总结来说，PyTorch 的优势在于：

- 自动选择最优卷积算法（算法 autotune）
- 高度优化的 GPU 并行计算（Tensor Core、SIMD 并行）
- kernel fusion 和 memory reuse 减少访存开销

4 Cifar10 训练对比

5 网络结构

按照自己之前在底层 CUDA 实现的算子设计了一个相对简单的神经网络模型结构。

表 4: PyTorch CNN 网络结构（用于 CIFAR-10）

Layer	Output Shape	Kernel / Stride / Padding	Notes
Input	$3 \times 32 \times 32$	-	CIFAR-10 image
Conv1	$8 \times 32 \times 32$	3×3, stride=1, padding=1	ReLU, He 初始化
MaxPool1	$8 \times 16 \times 16$	2×2, stride=2	
Conv2	$16 \times 16 \times 16$	3×3, stride=1, padding=1	ReLU, He 初始化
MaxPool2	$16 \times 8 \times 8$	2×2, stride=2	
Conv3	$32 \times 8 \times 8$	3×3, stride=1, padding=1	ReLU, He 初始化
Flatten	$32 \cdot 8 \cdot 8 = 2048$	-	
FC1	256	-	ReLU, He 初始化
FC2	10	-	输出 CIFAR-10 分类概率

由于网络的参数量较大且层数较多，在后面 Mytorch 测试的时候出现了梯度消失的问题, 所以使用 He 初始化 [3]。

在 Mytorch 框架中，Tensor 以及 Conv，FC，RELU，MAXPOOL 算子都是我自己使用 CUDA 编程底层实现的。在 Python 端我把各个算子封装成类，随后把网络封装进 MyNetwork 类，在类内实现了整个网络的初始化、前向传播和反向传播、参数更新及数据测试。本实验使用的优化器是 SGD。

5.1 实验结果

命令行写下 `python pytorchTest.py`，`python network.py` 可分别运行 Pytorch 框架下和 Mytorch 框架下的训练和测试代码。

训练过程中的损失函数曲线分别如图1,2。由于 SGD 优化器以及较小的 batch_size，曲线的震荡程度比较大。

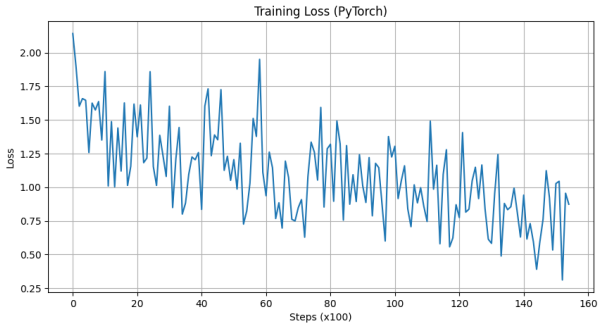

图 1: pytorch loss

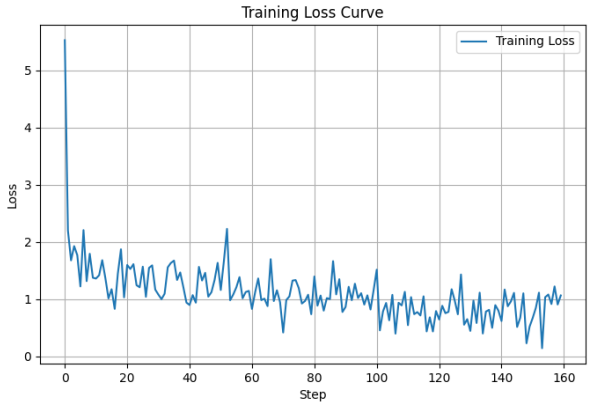

图 2: mytorch loss

由于本模型参数量大且缺少 Batch_normalization 和 Dropout，因此为了避免过拟合情况的出现，仅训练 5 个 epoch。对比数据如下表5。

表 5: Pytorch 与 Mytorch 对比

框架	平均 epoch 用时 (s)	测试准确率
pytorch	10.1	65.3%
mytorch	30.5	66.7%

可以看到，使用 Mytorch 算子，训练速度约为 Pytorch 的三分之一，但是最终在测试准确率上和 pytorch 的效果是相当的。

参考文献

[1] Sharan Chetlur, Cliff Woolley, Pierre Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning, 2014. NVIDIA Technical Report.

[2] Mark Harris. Cuda programming: A developer's guide to parallel computing with gpus, 2010.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, 2015.

[4] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.