

并行算法期末大作业报告

研究主题：基于动态计算框架的神经网络算子并行加速库——mytorch

团队成员（无先后之分）：陈兴平、刘华壹、罗弘杰

Contents

1. 研究背景

1.1 相关工作——pytorch

1.2 前沿发展——高性能算子的设计

2. mytorch框架设计

2.1 项目结构图

2.2 框架介绍

3. mytorch算法实现

3.1 Tensor相关实现

3.1.1 核心类结构与设备支持

3.1.2 张量的数据管理

3.1.3 张量的属性与操作

3.1.4 自动微分与计算图

3.1.5 运算符与函数接口

3.1.6 设备管理与迁移

3.2 Function相关实现

3.2.1 Function类的核心设计

3.2.2 前向/反向传播接口

3.2.3 典型算子运算的Function子类

3.2.4. 与Tensor的关系

3.2.5 反向传播的依赖管理

3.2.6 灵活性与可扩展性

3.3 nn模块说明

3.3.1 核心基类设计

3.3.2 典型层的实现与特点

3.3.3 结构组合与复用

3.3.4 参数与设备统一管理

3.3.5 其它说明

3.4 并行算法

3.4.1 OpenMP并行 (CPU端)

3.4.2 CUDA并行 (GPU端)

3.5 to方法的实现——CPU和GPU的统一

3.5.1 Tensor::to 方法实现 (核心代码)

3.5.2 to 方法的使用例子

3.5.3 设备属性与数据分配

3.5.4 相关辅助/底层实现

4. 加速算法-FlashAttention介绍

4.1 online-softmax的历史演变

4.2 分块访存前向计算

4.3 反向计算优化

5. FlashAttention算法实现及测试

5.1 算法实现

5.2 实验测试

6. mytorch使用示例与效果展示

6.1 以mnist数据集的训练为例

6.2 测试结果展示

6.3 并行算法加速情况

6.4 FlashAttention + mytorch


附录：个人报告

参考资料

摘要:

mytorch是一个基于动态计算框架的轻量级神经网络算子并行加速库，旨在通过多设备支持（CPU/GPU）和高效并行算法（如OpenMP和CUDA）优化深度学习模型的训练与推理性能。该框架参考PyTorch的核心设计，实现了张量计算、自动微分、计算图追踪等基础功能，并提供了模块化的神经网络层（如Linear、Conv2D）和优化器（如SGD）。此外，我们团队在mytorch基础上集成了FlashAttention算法，通过online-softmax和分块访存技术显著提升了Attention算子的计算效率。实验表明，mytorch能够成功在MNIST数据集上训练，且CPU/GPU并行加速效果显著，为轻量化深度学习框架的开发提供了实践参考。通过几周的努力以及上千行代码的实践，我们成功完成了我们预设的目标。

关键词： mytorch; pytorch; flashattention; OpenMP; CUDA; 自动微分; 并行计算; 神经网络加速

相关代码已上传到团队仓库：[Legend717/mytorch](#) 

1. 研究背景

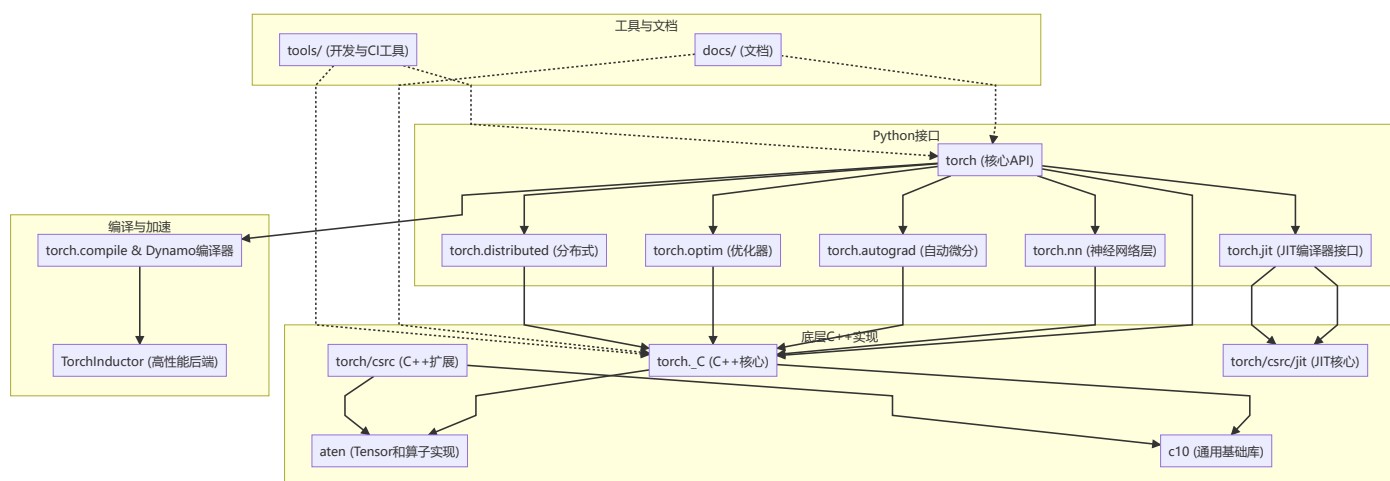
“

项目的详细背景以及规划可以见我们的开题报告。在此仅结合我们所作工作做简单的介绍。

1.1 相关工作——pytorch

在当今神经网络研究领域，pytorch已成为不可或缺的核心工具。作为Meta（原Facebook）开发的开源框架，它凭借动态图优先的设计哲学脱颖而出，通过 `torch.autograd` 在Python运行时动态构建计算图，这种机制相比TensorFlow等静态图框架更能满足科研场景的快速迭代需求。PyTorch不仅提供直观的Pythonic编程体验和高效的GPU加速能力，还集成了完整的深度学习工具链（如TorchVision、TorchText），并与Python生态无缝对接，同时通过TorchScript和ONNX支持实现便捷的模型部署。随着PyTorch 2.0引入编译优化技术并持续强化分布式训练与大模型支持（如Llama），该框架在保持科研灵活性的同时不断提升工业级性能，已成为贯穿算法探索到生产落地的首选平台。

以下是 pytorch 的框架图（使用 Mermaid 代码编写）：



我们将参考pytorch的底层实现，将其化繁为简，设计一个相对轻量化的框架mytorch来训练我们的神经网络。

1.2 前沿发展——高性能算子的设计

Attention算子是Transformer模型的核心组件之一，主要用于处理序列数据。它通过计算输入序列中各个位置之间的相关性来生成输出序列。

其集体的计算公式是

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

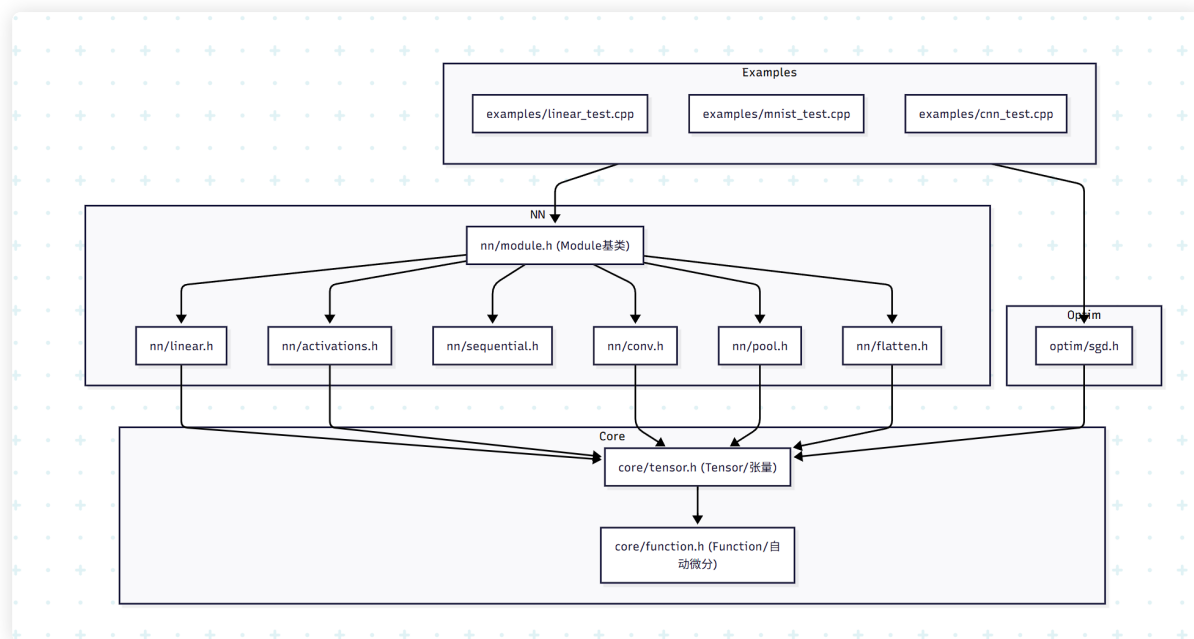
Q, K, V 是 x 输入经过三个线性层得到的查询（Query）、键（Key）和值（Value）矩阵， d_k 是键的维度。

不同于以往的并行算法，Attention算子目前不仅受制于运算速度，还受制于内存带宽，以及空间复杂度，算力对高速显存的依赖需要我们改进算法，在计算速度和占用内存之间取得平衡。

我们将在自己设计的 `mytorch` 框架基础上，实现对Attention算子的加速。

2. mytorch框架设计

2.1 项目结构图



2.2 框架介绍

- **Core 层**：负责张量（Tensor）定义、数据存储与基本运算、动态计算图的构建和自动微分机制。
- **nn 层**：面向用户，封装常见神经网络层，每个模块都继承自 Module，组合/嵌套灵活。
- **optim 层**：如 SGD 优化器，用于实现反向传播后的参数更新。
- **examples 层**：提供端到端模型训练/推理示例，便于开发者快速上手。
- **构建/接口**：CPU/GPU 兼容，CMake 工程，预留 Python 绑定。

3. mytorch算法实现

我们实现了CPU和GPU两个版本的代码。其中，GPU版本代码兼容CPU的计算，所以在此以GPU版本代码为准进行介绍。（因为部分代码为了方便写CUDA做了重构）

3.1 Tensor相关实现

`Tensor` 实现兼顾了多设备支持、自动微分、常见算子重载和计算图追踪等深度学习框架核心要素。设计思路高度参考主流框架（PyTorch），并通过 C++ 智能指针、设备枚举等机制，保证内存与计算的安全与灵活性。

3.1.1 核心类结构与设备支持

- `Tensor` 类定义于 `gpu/include/core/tensor.h`
- 支持多设备（CPU/CUDA），通过枚举 `Device` 区分，Tensor 内部变量 `_device` 标识张量当前所在设备。
- 禁止直接拷贝构造和赋值，强制使用静态工厂函数（如 `create`，`randn`，`ones`，`zeros`）进行构造，确保管理一致性和设备感知。
- 构造函数示例：

```
1 Tensor(std::vector<size_t> shape, bool requires_grad = false, Device device = Device::CPU);
```

C++

工厂函数示例（可用来创建 CPU 张量并初始化数据）：

```
1 static std::shared_ptr<Tensor> create(const std::vector<float>& data, std::vector<size_t> shape, bool requires_grad = false);
```

C++

3.1.2 张量的数据管理

- 内部数据指针 `_data`，存储张量的实际数据（实现支持不同设备的数据分配）。
- 提供 `data_ptr()`、`mutable_data_ptr()` 用于获得底层数据指针，实现与设备无关的调用。
- `data_cpu()` 方法支持将数据从任意设备拷贝回 CPU 并以 `std::vector<float>` 返回，方便调试和跨设备操作。

3.1.3 张量的属性与操作

- 支持张量形状（`shape()`）、元素个数（`size()`）、单元素访问（`item()`）。
- 步幅（stride）通过 `compute_stride()` 计算，支持高维张量的存储与遍历。
- 典型代码片段（步幅计算）：

```

1 void Tensor::compute_stride() {
2     _stride.resize(_shape.size());
3     size_t stride = 1;
4     for (int i = _shape.size() - 1; i >= 0; i--) {
5         _stride[i] = stride;
6         stride *= _shape[i];
7     }
8 }

```

3.1.4 自动微分与计算图

- 自动微分属性：`requires_grad` 标志、`grad()` 获取梯度、`set_grad()` 设置梯度，确保梯度与数据在同一设备。
- 通过 `_ctx` (`std::shared_ptr<Function>`) 记录产生当前张量的运算上下文，实现反向传播时的计算依赖追踪。
- `backward()` 方法实现反向传播，递归拓扑排序所有依赖的节点，自动计算梯度。核心思想与 PyTorch 类似。

3.1.5 运算符与函数接口

- 支持常见算子 (`add`、`sub`、`mul`、`div`、`matmul`、`sum`、`relu` 等) 作为成员函数存在，返回新张量并自动构建计算图。
- 相关代码接口示例 (部分):

```

1 std::shared_ptr<Tensor> add(const std::shared_ptr<Tensor>& other);
2 std::shared_ptr<Tensor> matmul(const std::shared_ptr<Tensor>& other);
3 std::shared_ptr<Tensor> relu();

```

3.1.6 设备管理与迁移

- `to(Device device)` 支持张量在 CPU/GPU 之间迁移，保证设备一致性。
- 在梯度设置等操作中显式检查设备一致性，防止跨设备错误，提升健壮性。

3.2 Function 相关实现

`Function` 实现了基于计算图的自动微分核心框架，类设计高度模块化，便于扩展和维护（不然难以 debug）。每个运算都作为 `Function` 子类存在，统一接口支持前向与反向传播，灵活高效。`Function` 与 `Tensor` 紧密协作，实现链式自动微分，支持深度学习常见运算与自定义扩展。

3.2.1 Function类的核心设计

- `Function` 类定义于 `gpu/include/core/function.h`
- 核心作用：作为所有具体算子（如加法、乘法等）自动微分操作的基类，负责正向与反向传播的统一接口。
- 继承自 `std::enable_shared_from_this<Function>`，方便在运算图构建和反向传播中安全管理智能指针引用。

3.2.2 前向/反向传播接口

- 提供统一的 `apply`（正向）、`backward`（反向）接口，自动保存输入用于反向传播追溯。
- 通过纯虚函数 `_forward` 和 `_backward`，要求所有具体算子必须实现自身的前/反向逻辑。

```
1  virtual std::shared_ptr<Tensor> _forward(const
    std::vector<std::shared_ptr<Tensor>>& inputs) = 0;
2  virtual std::vector<std::shared_ptr<Tensor>> _backward(const
    std::shared_ptr<Tensor>& grad_output) = 0;
```

C++

- `_saved_inputs` 保存本次前向传播涉及的输入张量，便于自动微分时恢复依赖。

3.2.3 典型算子运算的Function子类

- 针对常见张量操作，每个操作都实现为 `Function` 的子类。例如：
 - `Add`：加法算子
 - `Sub`：减法算子
 - `Mul`：乘法算子
 - `MatMul`：矩阵乘法
 - `Sum`：求和
 - `ReLUFunc`：ReLU激活
 - `Conv2DFunc`、`MaxPool2DFunc` 等卷积、池化相关操作
- 每个子类都重写 `_forward` 和 `_backward` 以实现各自的运算与梯度计算。
- 例如，`Add` 的反向传播将上游梯度直接传递给两个输入，`Mul` 的反向传播则需要乘以另一个输入的值。

3.2.4. 与Tensor的关系

- `Function` 与 `Tensor` 通过 `Tensor::_ctx` 建立联系，每个由算子生成的新张量都保存了对应的 `Function` 实例指针，实现了计算图的自动追踪。

- 在 `Tensor::backward()` 时，会自动遍历 `_ctx` 链条递归回溯，依次调用各 `Function` 子类的 `backward` 方法，完成全自动链式反向传播。
- 支持复杂网络结构和运算图拓扑。

3.2.5 反向传播的依赖管理

- `Function` 保存前向输入 (`_saved_inputs`)，能精确还原每个操作的依赖链。
- 支持释放已保存输入以节省内存 (`release_saved_inputs()`)，便于大规模训练和推理场景应用。

3.2.6 灵活性与可扩展性

- 任何新的算子都可以通过继承 `Function` 并实现 `_forward` / `_backward` 两个方法来扩展。
- 这样保证了所有算子都可以被无缝集成到自动微分系统中，且与设备无关（具体运算留给子类或后续实现）。

3.3 nn模块说明

`mytorch` 的 `nn` 模块为深度学习模型的各类层 (Layer) 与常用结构提供了抽象与实现，核心设计理念高度参考 PyTorch 的 `torch.nn`，实现灵活组合和参数管理，并支持多设备 (CPU/GPU) 训练。

3.3.1 核心基类设计

- **Module基类**

所有神经网络层都继承自 `nn::Module` 抽象基类（定义见 `nn/module.h`，未在本次检索结果中直接列出）。

每个子类都需实现如下接口：

- `forward(std::shared_ptr<Tensor> input)`：前向传播，返回输出张量。
- `parameters()`：返回本层可学习参数的张量列表，便于优化器统一管理。
- `to(Device device)`：将本层参数搬移到指定设备（如GPU），支持多设备训练。

3.3.2 典型层的实现与特点

- **线性层 Linear**

参考 `gpu/include/nn/linear.h` 和 `gpu/src/nn/linear.cpp`

- 构造时可指定输入输出特征数、是否带bias。
- 权重采用 Kaiming He 初始化（对ReLU函数友好）。
- 前向传播为 $Y = XW + b$ ，支持自动广播 bias。
- 参数管理和设备迁移实现见 `parameters()` 和 `to()`。

- **卷积层 Conv2D**

见 `gpu/include/nn/conv.h` 与 `gpu/src/nn/conv.cpp`

- 支持设置输入/输出通道数、卷积核尺寸、步幅、padding。
- 权重同样采用 Kaiming 初始化。
- 前向传播通过 `Conv2DFunc` 实现，自动支持参数迁移和收集。

- **池化层 MaxPool2D**

见 `gpu/include/nn/pool.h` 与 `gpu/src/nn/pool.cpp`

- 支持池化核大小、步幅设置。
- 前向调用 `MaxPool2DFunc` 完成实际操作。
- 池化层无可学习参数。

- **激活层 ReLU**

见 `gpu/include/nn/activations.h`

- 实现简单，无可学习参数，前向传播直接调用 `relu`。

- **Flatten层**

见 `gpu/include/nn/flatten.h`

- 用于展平输入张量形状，常用于卷积->全连接的连接部位。

3.3.3 结构组合与复用

- **Sequential容器**

见 `gpu/src/nn/sequential.cpp`

- 支持按顺序组合多个 `Module` 层，自动递归前向传播、参数收集、设备迁移。
- 方便搭建常见的多层感知机/卷积网络等结构。

3.3.4 参数与设备统一管理

- 每个 `Module` 子类均实现 `parameters()`，递归收集所有可学习参数，便于优化器如 SGD 实现统一管理。
- 通过 `to(Device device)` 支持参数（如权重、偏置）一键搬移，多设备切换灵活。

3.3.5 其它说明

- 所有层均兼容自动微分与反向传播（通过 `Tensor/Function` 构建的计算图），使用时只需调用 `backward()` 即可自动求导。
- 代码整体风格简洁清晰，易于扩展自定义层或结构。（当然这得感谢pytorch，pytorch的设计比我们复杂但更加精妙）

3.4 并行算法

考虑到我们这节课是并行算法。所以专门开一个环节进行介绍我们是怎么从CPU和GPU两个维度进行并行的。

- **CPU端并行**：采用OpenMP，主要通过 `#pragma omp parallel for` 指令，把向量/矩阵操作自动分发到多个CPU核心，提高吞吐量。
- **GPU端并行**：采用CUDA，将大规模数据操作映射为CUDA kernel，利用成百上千的GPU线程进行大规模数据并行。
- **接口无缝切换**：同一套高层API（如Tensor、Function等）内部自动判断设备，透明切换CPU/GPU后端，对使用者友好。

3.4.1 OpenMP并行（CPU端）

在OMP（OpenMP）版本实现中，核心并行策略是利用 `#pragma omp parallel for` 等指令对常见的数值运算（如张量加法、乘法、矩阵乘法、ReLU激活、SGD优化器）进行多线程加速。

例：SGD优化器并行更新参数

代码片段（见 `omp/src/optim/sgd.cpp`）：

```
C++
1 void SGD::step() {
2     for (auto& p : _params) {
3         if (p->grad()) {
4             auto p_data = p->get_shared_data();
5             auto g_data = p->grad()->get_shared_data();
6             #pragma omp parallel for schedule(guided)
7             for (size_t i = 0; i < p_data->size(); ++i) {
8                 (*p_data)[i] -= _lr * (*g_data)[i];
9             }
10        }
11    }
12 }
```

- 这里对每个参数的每一个元素进行并行的梯度更新，大大提升了大规模参数量网络的更新效率。
- 其它如ReLU等操作，也广泛用 `#pragma omp parallel for` 进行矢量化加速。

例：ReLU激活并行实现

```

1  std::shared_ptr<Tensor> ReLUFunc::_forward(const
    std::vector<std::shared_ptr<Tensor>>& inputs) {
2      const auto& x = inputs[0]->data();
3      std::vector<float> result_data(x.size());
4      #pragma omp parallel for
5      for(size_t i=0; i<x.size(); ++i) {
6          result_data[i] = std::max(0.0f, x[i]);
7      }
8      return Tensor::create(result_data, inputs[0]->shape());
9  }

```

- 这种写法保证了CPU多核资源的高效利用，适合在传统服务器或本地多核环境下部署。

3.4.2 CUDA并行（GPU端）

在CUDA版本实现中，核心思路是将大规模矢量/矩阵计算任务分发到成百上千的GPU线程上，通过CUDA kernel函数实现数据并行。

例：im2col卷积前向传播

im2col_kernel 核函数提供了对卷积的并行，使用的方法是im2col，能够在gpu中显著提升运算速度

```

1  __global__ void im2col_kernel(const float* data_im, float* data_col,
2                                int N, int C, int H, int W,
3                                int K, int S, int P,
4                                int H_out, int W_out) {
5      int index = blockIdx.x * blockDim.x + threadIdx.x;
6      int col_size = C * K * K;
7      int num_kernels = N * H_out * W_out;
8
9      if (index < num_kernels * col_size) {
10         // 计算在输出列矩阵中的位置
11         int col_idx = index % col_size; // 列索引(0到C*K*K-1)
12         int row_idx = index / col_size; // 行索引(0到N*H_out*W_out-1)
13
14         // 分解列索引找到核位置
15         int k_w = col_idx % K;           // 核宽度坐标
16         int k_h = (col_idx / K) % K;     // 核高度坐标
17         int c_in = col_idx / (K * K);    // 输入通道
18
19         // 分解行索引找到输出像素位置
20         int w_out = row_idx % W_out;     // 输出宽度坐标
21         int h_out = (row_idx / W_out) % H_out; // 输出高度坐标
22         int n = row_idx / (H_out * W_out); // 批次索引

```

```

23
24     // 计算对应的输入坐标
25     int h_in = h_out * S - P + k_h;
26     int w_in = w_out * S - P + k_w;
27
28     // 如果在边界内则复制，否则填充0
29     if (h_in >= 0 && h_in < H && w_in >= 0 && w_in < W) {
30         data_col[index] = data_im[(n * C + c_in) * H * W + h_in * W +
w_in];
31     } else {
32         data_col[index] = 0.0f;
33     }
34 }
35 }

```

`im2col_cuda` 函数提供了方便的C++接口：

1. 计算输出尺寸： $H_{out} = (H + 2 * P - K) / S + 1$
2. 创建输出张量：使用 `Tensor::zeros` 在GPU上分配空间
3. 启动内核并检查错误

```

1  std::shared_ptr<Tensor> im2col_cuda(const std::shared_ptr<Tensor>& input,
2                                     size_t K, size_t S, size_t P) {
3      const auto& shape = input->shape();
4      int N = shape[0];
5      int C = shape[1];
6      int H = shape[2];
7      int W = shape[3];
8
9      int H_out = (H + 2 * P - K) / S + 1;
10     int W_out = (W + 2 * P - K) / S + 1;
11
12     // 在GPU上创建输出列Tensor
13     auto col_tensor = Tensor::zeros({(size_t)C * K * K, (size_t)N * H_out *
W_out}, false, Device::CUDA);
14     size_t n = col_tensor->size();
15     if (n == 0) return col_tensor;
16
17     int threads = 256;
18     int blocks = (n + threads - 1) / threads;
19
20     im2col_kernel<<<blocks, threads>>>(
21         static_cast<const float*>(input->data_ptr()),
22         static_cast<float*>(col_tensor->mutable_data_ptr()),
23         N, C, H, W, K, S, P, H_out, W_out
24     );

```

```

25     CUDA_CHECK(cudaPeekAtLastError());
26
27     return col_tensor;
28 }

```

这种实现充分利用了GPU的并行能力，将im2col操作高效地映射到CUDA架构上，是卷积神经网络前向传播的重要优化步骤。

3.5 to方法的实现——CPU和GPU的统一

以下是结合 mytorch 仓库 gpu2 目录下相关代码，对 to 方法及其相关设备迁移实现的详细源码分析：

3.5.1 Tensor::to 方法实现（核心代码）

在 gpu2/src/core/tensor.cu 中，`Tensor::to(Device device)` 实现了张量的设备迁移：

```

1  // 将张量移动到另一个设备
2  std::shared_ptr<Tensor> Tensor::to(Device device) {
3      if (this->_device == device) {
4          return shared_from_this();
5      }
6      auto new_tensor = std::make_shared<Tensor>(_shape, _requires_grad,
device);
7      size_t data_size = this->size() * sizeof(float);
8      if (data_size == 0) {
9          return new_tensor;
10     }
11     // 根据数据转移的方向，选择正确的指针和拷贝方式
12     if (device == Device::CUDA) { // 方向: CPU -> CUDA
13         // 源(this)在CPU上, _data 是 std::vector<float>*
14         // 目标(new_tensor)在GPU上, mutable_data_ptr() 返回 float* (GPU地址)
15         // 1. 从源CPU张量中获取 std::vector<float> 对象
16         // ...
17     }
18     // 反之亦然, CUDA -> CPU
19 }

```

- 首先判断目标设备是否与当前一致，不一致则新建目标设备的张量，并进行数据内存的复制。
- 针对 CPU->CUDA、CUDA->CPU，各自调用 `cudaMemcpy` 或直接内存拷贝，保证正确的数据迁移。
- 迁移时 `shape`、`requires_grad` 属性全部保留。

3.5.2 to 方法的使用例子

nn::Linear 的 to 方法

在 gpu2/src/nn/linear.cpp:

```
1 void Linear::to(Device device) {
2     if (_weight) _weight = _weight->to(device);
3     if (_bias) _bias = _bias->to(device);
4 }
```

C++

- 将权重和偏置 (Tensor) 分别迁移到目标设备。

nn::Sequential 的 to 方法

在 gpu2/src/nn/sequential.cpp:

```
1 void Sequential::to(Device device) {
2     for(auto& layer : _layers) {
3         layer->to(device); // 递归调用每一层
4     }
5 }
```

C++

- 对所有子模块递归调用 to, 实现整个网络的设备统一。

3.5.3 设备属性与数据分配

在 gpu2/include/core/tensor.h:

```
1 enum class Device {
2     CPU,
3     CUDA
4 };
5
6 class Tensor {
7     Device _device;
8     // ...
9     Device device() const { return _device; }
10    std::shared_ptr<Tensor> to(Device device);
11    // ...
12 };
```

C++

- 每个 Tensor 都附带 device 信息。
- 数据分配时 `allocate_data()` 会根据 device 类型选择分配 CPU 内存 (`std::vector`) 或 GPU 内存 (`cudaMalloc`)。

3.5.4 相关辅助/底层实现

- `allocate_data()`、`data_cpu()`、`item()` 等函数对 device 做专门分支处理，保证数据访问和迁移一致性。

这里以 `allocate_data()` 为例

```
C++
1 // 这个函数由构造函数调用，负责根据设备分配内存
2 void Tensor::allocate_data() {
3     size_t total_size = this->size();
4     if (total_size == 0) {
5         _data = nullptr;
6         return;
7     }
8
9     if (_device == Device::CPU) {
10         _data = new std::vector<float>(total_size, 0.0f);
11     } else { // _device == Device::CUDA
12         CUDA_CHECK(cudaMalloc(&_amp;data, total_size * sizeof(float)));
13         // 确保新分配的GPU内存被清零，这对于zeros()等操作很重要
14         CUDA_CHECK(cudaMemset(_data, 0, total_size * sizeof(float)));
15     }
16 }
```

4. 加速算法-FlashAttention介绍

FlashAttention是一种高效的Attention计算方法，主要通过以下方式加速：

1. online-softmax
2. 分块访存利用内存层次

4.1 online-softmax的历史演变

softmax的伪代码如下：

Algorithm 1 Naive softmax

```
1:  $d_0 \leftarrow 0$ 
2: for  $j \leftarrow 1, V$  do
3:    $d_j \leftarrow d_{j-1} + e^{x_j}$ 
4: end for
5: for  $i \leftarrow 1, V$  do
6:    $y_i \leftarrow \frac{e^{x_i}}{d_V}$ 
7: end for
```

知乎 @TaurusMoon

1. 从访存的角度考虑，原始的softmax对每个元素需要2次load,1次store操作；
2. 从数值稳定性上考虑，原始的naive softmax在计算过程中可能会出现数值溢出的问题，因此需要对其进行改进。

safe-softmax

safe-softmax通过减去输入向量的最大值来避免这种问题。

safe-softmax的伪代码如下：

Algorithm 2 Safe softmax

```
1:  $m_0 \leftarrow -\infty$ 
2: for  $k \leftarrow 1, V$  do
3:    $m_k \leftarrow \max(m_{k-1}, x_k)$ 
4: end for
5:  $d_0 \leftarrow 0$ 
6: for  $j \leftarrow 1, V$  do
7:    $d_j \leftarrow d_{j-1} + e^{x_j - m_V}$ 
8: end for
9: for  $i \leftarrow 1, V$  do
10:   $y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$ 
11: end for
```

知乎 @TaurusMoon

这次改进由于需要计算max,所以每个元素需要3次load,1次store操作。

online-softmax最早由nvidia在2018年提出，主要利用了softmax中指数运算法则的特性，将softmax的前后依赖关系打破，允许在计算softmax的过程中进行并行计算。伪代码如下：

Algorithm 3 Safe softmax with online normalizer calculation

```
1:  $m_0 \leftarrow -\infty$ 
2:  $d_0 \leftarrow 0$ 
3: for  $j \leftarrow 1, V$  do
4:    $m_j \leftarrow \max(m_{j-1}, x_j)$ 
5:    $d_j \leftarrow d_{j-1} \times e^{m_{j-1} - m_j} + e^{x_j - m_j}$ 
6: end for
7: for  $i \leftarrow 1, V$  do
8:    $y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$ 
9: end for
```

知乎 @TaurusMoon

4.2 分块访存前向计算

GPU的访存根据访问速度从高速到低速有层次之分，从register, L1-cache/shared memory, L2-cache, L3-cache, HBM等。为了充分利用GPU的高速缓存，FlashAttention采用了分块访存的方式进行前向计算。

Algorithm 1 FLASHATTENTION

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size M .

- 1: Set block sizes $B_c = \lceil \frac{M}{4d} \rceil, B_r = \min(\lceil \frac{M}{4d} \rceil, d)$.
- 2: Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}, \ell = (0)_N \in \mathbb{R}^N, m = (-\infty)_N \in \mathbb{R}^N$ in HBM.
- 3: Divide \mathbf{Q} into $T_r = \lceil \frac{N}{B_r} \rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide \mathbf{K}, \mathbf{V} in to $T_c = \lceil \frac{N}{B_c} \rceil$ blocks $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
- 4: Divide \mathbf{O} into T_r blocks $\mathbf{O}_1, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_1, \dots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks m_1, \dots, m_{T_r} of size B_r each.
- 5: **for** $1 \leq j \leq T_c$ **do**
- 6: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM. 先load K, V
- 7: **for** $1 \leq i \leq T_r$ **do**
- 8: Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM. 再load Q
- 9: On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
- 10: On chip, compute $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}, \tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
- 11: On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}, \ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
- 12: Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1} (\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.
- 13: Write $\ell_i \leftarrow \ell_i^{\text{new}}, m_i \leftarrow m_i^{\text{new}}$ to HBM.
- 14: **end for**
- 15: **end for**
- 16: Return \mathbf{O} .

知乎 @DefTruth

通过分块，可以将Q,K,V矩阵分成多个小块，每次只计算一个小块的Attention，具体是将小块载入共享内存，然后在共享内存中进行计算。这样可以减少对HBM的访问次数，提高计算速度。

4.3 反向计算优化

从访存上分析，反向计算根本不需要从HBM获取中间变量，而是直接利用sram的分块Q,K进行recompute。因为工程上速度慢一点没有很大影响，但是recompute可以大大节省显存开销，所以可以设计更大的神经网络，最终的网络效果会更好。

5. FlashAttention算法实现及测试

5.1 算法实现

使用RTX4090-24GB显卡作为实验平台，使用Triton编译器，以python语言实现算子，Jit动态编译将triton算子转化为cuda代码。Triton编译器使得python开发高性能算子成为可能，只需要对分块进行说明，就可以执行高效的分块并行计算。虽然在性能上不一定比得上C++实现，但在开发效率上有很大提升。

```
1 # 前向传播内核: FlashAttention 前向
2 @triton.jit
3 def _fwd_kernel(
```

Python

```

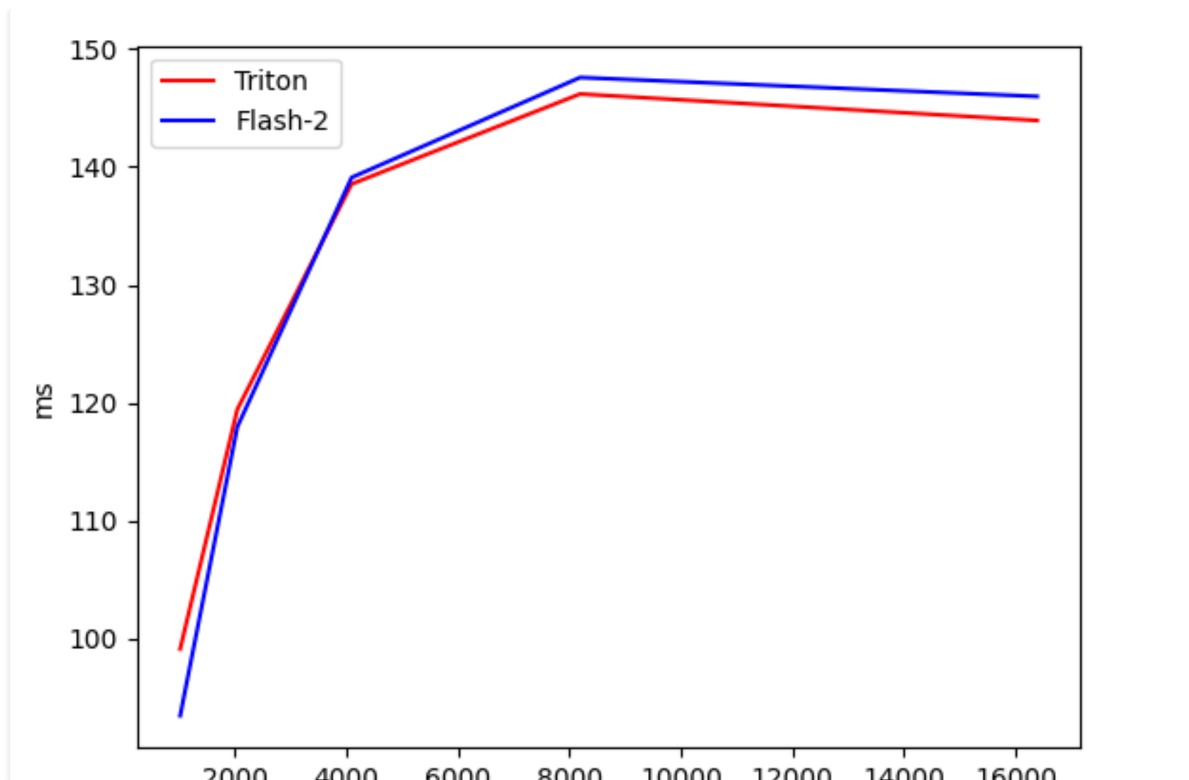
4      Q, K, V, sm_scale,          # 输入: q, k, v 和 softmax 缩放因子
5      L, Out,                    # 输出: 行和 log(sum(exp)), 以及最终输出
6      stride_qz, stride_qh, stride_qm, stride_qk,
7      stride_kz, stride_kh, stride_kn, stride_kk,
8      stride_vz, stride_vh, stride_vk, stride_vn,
9      stride_oz, stride_oh, stride_om, stride_on,
10     Z, H, N_CTX,              # 批量、头数、序列长度
11     BLOCK_M: tl.constexpr, BLOCK_DMODEL: tl.constexpr,
12     BLOCK_N: tl.constexpr,
13     IS_CAUSAL: tl.constexpr,
14 ):

```

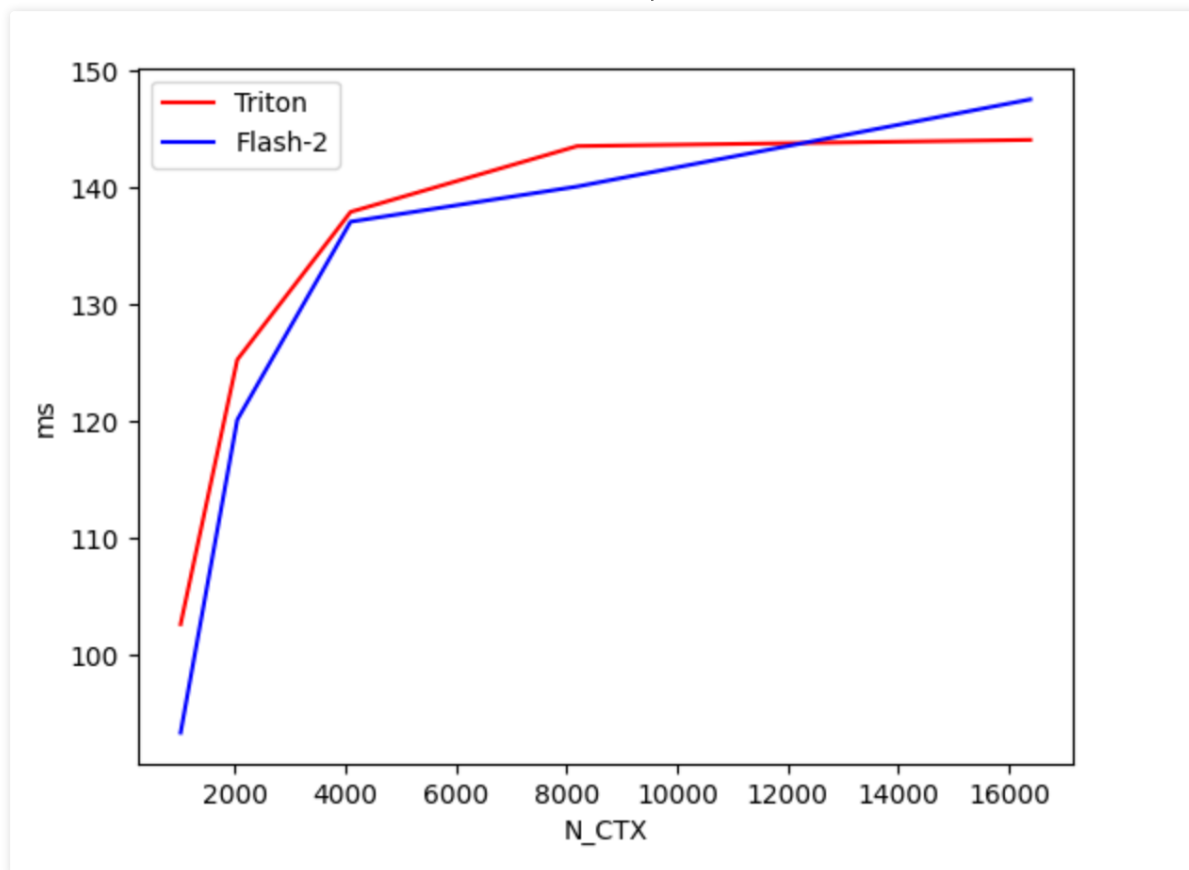
5.2 实验测试

首先是以4, 48, 4096, 32作为测试:

可以发现和cuda实现性能上没有什么区别, 稍微落后于高性能的Flash-2 cuda实现。



backward的实现和flash-2的cuda实现更相近，在部分测试中甚至超过了Flash-2的cuda实现。



6. mytorch使用示例与效果展示

6.1 以mnist数据集的训练为例

完整代码请参考 `gpu/examples/mnist_test.cpp`

头文件引入

```
1  #include "core/tensor.h"
2  #include "nn/module.h"
3  #include "nn/linear.h"
4  #include "nn/activations.h"
5  #include "nn/sequential.h"
6  #include "nn/flatten.h"
7  #include "optim/sgd.h"
8  #include "loader/mnist_loader.h"
```

C++

这些头文件引入了mytorch框架的核心组件：

- `tensor.h`：定义了张量(Tensor)类，是框架的基础数据结构
- 各种神经网络模块：线性层、激活函数、序列容器等
- 优化器：SGD优化器
- 数据加载器：MNIST数据集加载器

设备设置

```
1  Device device = Device::CPU;
2  if (cuda_err == cudaSuccess && device_count > 0) {
3      device = Device::CUDA;
4  }
```

C++

我们的mytorch框架支持CPU和CUDA设备，这里自动检测并选择可用的设备。

模型定义

```
1  auto model = std::make_shared<nn::Sequential>(
2      std::vector<std::shared_ptr<nn::Module>>{
3          std::make_shared<nn::Linear>(INPUT_FEATURES, HIDDEN_FEATURES),
4          std::make_shared<nn::ReLU>(),
5          std::make_shared<nn::Linear>(HIDDEN_FEATURES, OUTPUT_CLASSES)
6      }
7  );
8  model->to(device);
```

C++

这里使用了mytorch框架的 `Sequential` 容器来构建一个简单的全连接网络：

1. 第一个线性层(784->32)
2. ReLU激活函数
3. 第二个线性层(32->10)

`model->to(device)` 将模型参数移动到指定设备(CPU或CUDA)。

优化器设置

```
1 optim::SGD optimizer(model->parameters(), LEARNING_RATE);
```

C++

使用mytorch框架的SGD优化器，传入模型参数和学习率。

数据加载

```
1 MnistLoader::load(MNIST_DATA_PATH, X_train_cpu, y_train_cpu, X_test_cpu,  
  y_test_cpu);  
2 auto X_train = X_train_cpu->to(device);
```

C++

使用mytorch的MNIST加载器加载数据，然后使用 `to(device)` 方法将数据移动到指定设备。

训练循环

训练循环展示了mytorch框架的核心API调用：

- 前向传播

```
1 auto y_pred = model->forward(x_batch);
```

C++

- 损失计算

```
1 auto loss = mse_loss(y_pred, y_batch);
```

C++

- 反向传播

```
1 loss->backward();
```

C++

- 参数更新

```
1 optimizer.step();
```

C++

辅助函数

- 准确率计算

```
1 float calculate_accuracy(const std::shared_ptr<Tensor>& pred, const
  std::shared_ptr<Tensor>& target) {
2     auto pred_data = pred->data_cpu();
3     auto target_data = target->data_cpu();
4     // ... 计算逻辑 ...
5 }
```

C++

- MSE损失

```
1 std::shared_ptr<Tensor> mse_loss(const std::shared_ptr<Tensor>& pred, const
  std::shared_ptr<Tensor>& target) {
2     auto diff = pred->sub(target);
3     auto sq_diff = diff->mul(diff);
4     auto sum_loss = sq_diff->sum();
5     // ... 缩放处理 ...
6 }
```

C++

6.2 测试结果展示

以下两张图是CPU跑出来的结果：

```
模型成功加载到设备
正在从路径加载 MNIST 数据集: ../data
图像: 60000, 尺寸: 28x28
标签数量: 60000
图像: 10000, 尺寸: 28x28
标签数量: 10000
MNIST 数据加载完成。
数据成功加载到设备
开始训练 10 个周期...
周期 [1/10], 平均损失: 0.596403, 测试集准确率: 82.04%
周期 [2/10], 平均损失: 0.382702, 测试集准确率: 86.7%
周期 [3/10], 平均损失: 0.310609, 测试集准确率: 88.19%
周期 [4/10], 平均损失: 0.274372, 测试集准确率: 89.01%
周期 [5/10], 平均损失: 0.254003, 测试集准确率: 89.54%
周期 [6/10], 平均损失: 0.240676, 测试集准确率: 90.05%
周期 [7/10], 平均损失: 0.230941, 测试集准确率: 90.32%
周期 [8/10], 平均损失: 0.2233, 测试集准确率: 90.52%
周期 [9/10], 平均损失: 0.216978, 测试集准确率: 90.76%
周期 [10/10], 平均损失: 0.211551, 测试集准确率: 90.91%
--- 训练完成 ---
总用时: 391426 ms
```

```
模型成功加载到设备
正在从路径加载 MNIST 数据集: ../data
图像: 60000, 尺寸: 28x28
标签数量: 60000
图像: 10000, 尺寸: 28x28
标签数量: 10000
MNIST 数据加载完成。
数据成功加载到设备
开始训练 10 个周期...
周期 [1/10], 平均损失: 0.743275, 测试集准确率: 73.25%
周期 [2/10], 平均损失: 0.500152, 测试集准确率: 83.88%
周期 [3/10], 平均损失: 0.398438, 测试集准确率: 86.63%
周期 [4/10], 平均损失: 0.345616, 测试集准确率: 88.6%
周期 [5/10], 平均损失: 0.308023, 测试集准确率: 89.32%
周期 [6/10], 平均损失: 0.282616, 测试集准确率: 89.91%
周期 [7/10], 平均损失: 0.266052, 测试集准确率: 90.17%
周期 [8/10], 平均损失: 0.254117, 测试集准确率: 90.5%
周期 [9/10], 平均损失: 0.244829, 测试集准确率: 90.64%
周期 [10/10], 平均损失: 0.237316, 测试集准确率: 90.84%
--- 训练完成 ---
总用时: 1203234 ms
```

以下是GPU的结果：

```
模型成功加载到设备
正在从路径加载 MNIST 数据集: ../data
图像: 60000, 尺寸: 28x28
标签数量: 60000
图像: 10000, 尺寸: 28x28
标签数量: 10000
MNIST 数据加载完成。
数据成功加载到设备
开始训练 10 个周期...
周期 [1/10], 平均损失: 1.49408, 测试集准确率: 19.65%
周期 [2/10], 平均损失: 1.1621, 测试集准确率: 27.23%
周期 [3/10], 平均损失: 1.02109, 测试集准确率: 33.32%
周期 [4/10], 平均损失: 0.936113, 测试集准确率: 38.17%
周期 [5/10], 平均损失: 0.878662, 测试集准确率: 42.21%
周期 [6/10], 平均损失: 0.837216, 测试集准确率: 45.14%
周期 [7/10], 平均损失: 0.80587, 测试集准确率: 47.38%
周期 [8/10], 平均损失: 0.781241, 测试集准确率: 49.18%
周期 [9/10], 平均损失: 0.761277, 测试集准确率: 50.63%
周期 [10/10], 平均损失: 0.744681, 测试集准确率: 51.78%
--- 训练完成 ---
总用时: 29316 ms
```

可以看到，CPU和GPU设备均能正常执行。

6.3 并行算法加速情况

手写数字集（MNIST）的训练速度对比（所测试CPU为6核心）：

设备	线程数	训练时间（s）	加速比
CPU	1	1200	1.0
CPU	4	390	3.08
CPU	8	260	4.62
GPU	-	30	40

从上表可以看出，我们的mytorch框架并行效果显著，CPU的加速比达到预期目标。

从中也可以看出，在没有对GPU的CUDA相关代码做很深度的优化的情况下，GPU在机器学习上的潜力远远大于CPU。

6.4 FlashAttention + mytorch

最后，我们在 `cpu/examples/flash_test.cpp` 中尝试了我们mytorch架构对高级优化算子的支持。

```
1  int main() {
2      py::scoped_interpreter guard{}; // 只初始化一次解释器
3      std::cout << "Python interpreter initialized" << std::endl;
4
5      // 创建输入
6      auto Q = Tensor::randn({16, 8, 64, 16}, true);
7      auto K = Tensor::randn({16, 8, 64, 16}, true);
8      auto V = Tensor::randn({16, 8, 64, 16}, true);
9      show_tensor(Q);
10
11     // 调用 flash attention 前向
12     auto attn_layer = std::make_shared<nn::FlashAttn>(false, 1.0f);
13     printf("atten layer created address: %p\n", attn_layer.get());
14     auto o = attn_layer->forward({Q, K, V});
15     std::cout << "Output shape: " << o->shape()[0] << ", " << o->shape()[1] <<
16     ", " << o->shape()[2] << ", " << o->shape()[3] << std::endl;
17     o->backward(); // 触发反向传播
18     show_tensor(o);
19     //测试反向传播
20     auto label = Tensor::randn({64, 8, 128, 64}); // 假设标签与输出形状一致
21     auto loss = mse_loss(Q, label);
22     loss->backward(); // 触发反向传播
23
24     return 0; // 解释器由 guard 自动清理
25 }
```

附录：个人报告

• 陈兴平 22336037

◦ 具体工作

- 与刘华壹同学共同完成了整个mytorch框架的设计
- 实现了大多数tensor的前向传播以及反向传播
- 参与了Function的实现以及nn的构建

- 通过omp实现了CPU版本的并行，并写了 `linear_test.cpp` 测试函数
- 为了兼容GPU，在CPU代码的基础上重构了部分函数，并增加了to方法等方法
- 制造并修复了很多bug，最终让项目成功运行
- 撰写了实验报告相关部分

◦ 感想

- 这次大工程让我深深认识到了框架设计的重要性。同时，在写代码过程中借鉴了许多pytorch的设计和实现思路，让我对pytorch有了更加深刻的理解。
- 本次项目我手写了可能上千行代码(当然也制造了不少bug)，在代码成功运行的那一刻，非常的有成就感，感谢老师和助教给了我们这样尝试自己、突破自己的机会。

• 刘华壹 22336149

◦ 具体工作

- 与陈兴平同学一起完成了mytorch框架的设计，以及其在cpu和gpu上的部署实现
- 完善Function、Tensor的各个函数，
- 写了 `mnist_test.cpp` 和 `cnn_test` 测试函数，编写mnist数据集的loader函数
- 在mytorch中实现卷积算子、池化算子相应的forward、backwards，使用im2col方法加速卷积运算
- 修复了to方法的一些bug，修复了一些backward未能gpu并行的问题，修正在backward中未进行内存释放问题，使得backward完后内存得以释放
- 撰写了实验报告相关部分

◦ 感想

- 在实验过程中，我们从头开始学习pytorch的框架，通过学习模仿pytorch框架实现自己的框架，对于深度学习框架有了进一步的理解，对于如何加速深度神经网络有了进一步的认识，对于数据是如何在神经网络中流动和计算有了进一步的感悟。在实验过程中，搭建并测试神经网络是个巨大的难题，在设计框架时，如果设计不好，推倒重来的代价是很大的，因此一个好的设计非常重要。同时，在训练神经网络过程中，有一个很大的难题就是内存释放，在实验过程中，内存释放问题一度严重影响我们的进度。同时，各个层之间forward，backward设计也是很大的难题，特别是卷积算子，在实现过程中遇到了很大挑战，虽然最后效果不是很理想，但也学到了很多。整个实验推动我学会了很多，让我明白，虽然仅仅是将数据放到gpu上进行计算比较简单的并行，在作为框架进行实现过程中仍然有很多的工程的细节需要考虑。

• 罗弘杰 22336173

◦ 具体工作:

- 我主要负责FlashAttention算子的实现和测试，使用Triton编译器用python语言实现FlashAttention算子，并与论文作者的开源实现进行比较。
- 我还负责将Python算子嵌入到C++框架，使用Pybind11将Python算子与C++代码进行集成，确保算子可以在C++环境中调用。
- 负责撰写FlashAttention算子部分的报告。

◦ 感想：

- 当前深度学习依赖的算子并行目标不仅仅是加快运行速度，还要考虑到访存效率以及空间复杂度的限制。

- 数值运算的某些特性可以赋予并行加速的可能性，比如Online-softmax就依赖了指数运算的可分解性。
- 在实现算子时，需要充分利用GPU的并行计算能力，还需要考虑数值运算的稳定性，比如safe-softmax就是考虑了运算中的数值稳定性才加入求max这一步。
- 在实现并行算法时，还要考虑硬件架构，通过计算机体系结构的角度充分挖掘计算机的计算能力，比如FlashAttention算子就充分利用了计算卡的多层次内存，尽量将数据访存放置到离运算单元更近的地方。

参考资料

- <https://zhuanlan.zhihu.com/p/668888063> 
- <https://github.com/pytorch/pytorch> 