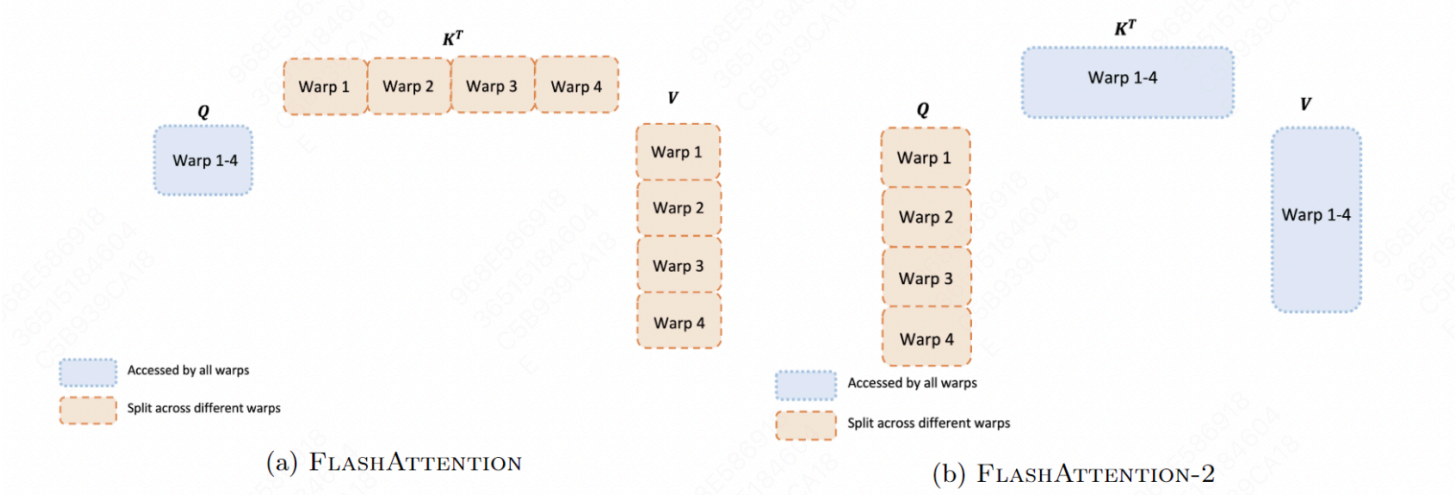
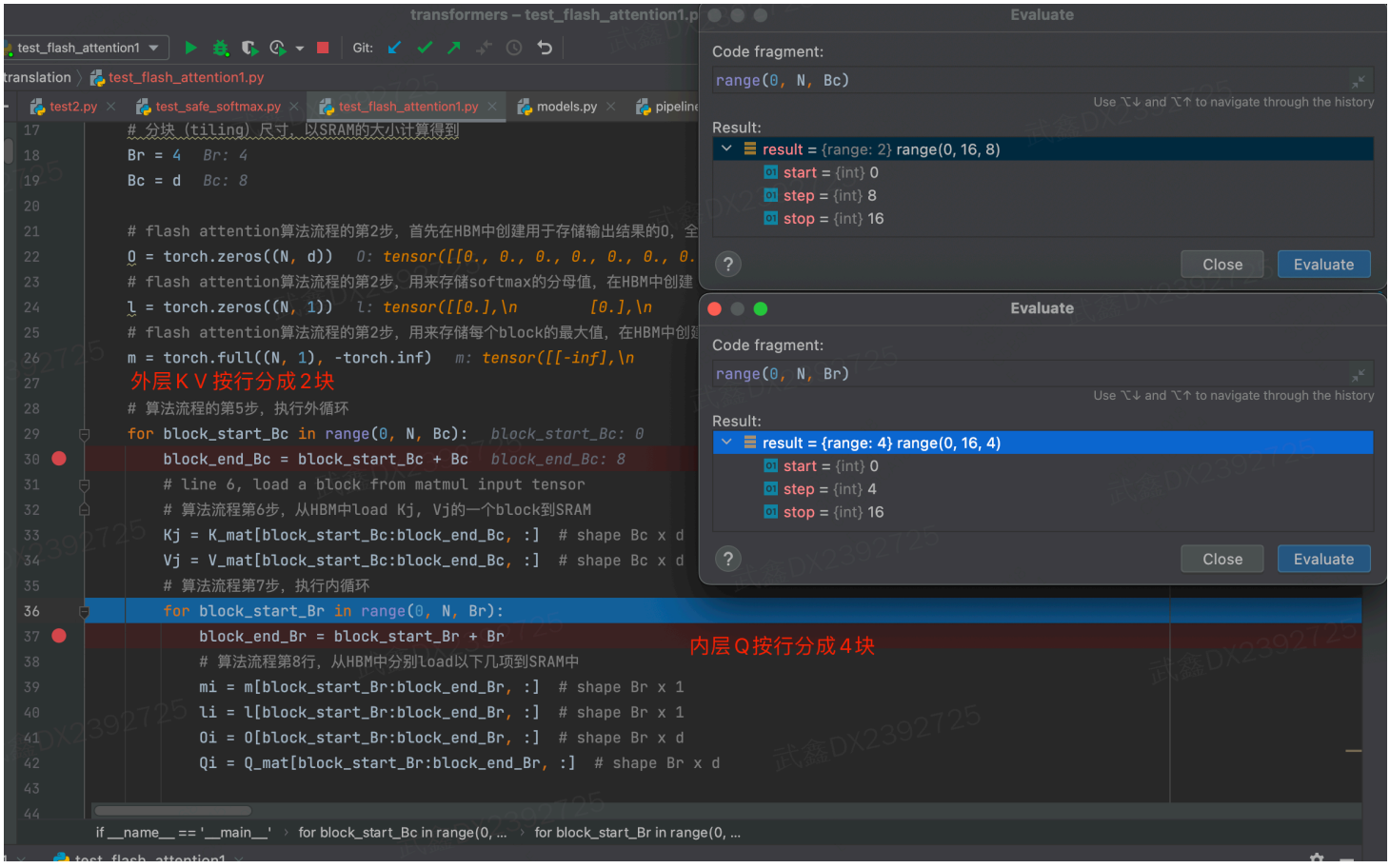
http://admin.guyuehome.com/44894

http://admin.guyuehome.com/44906

https://openai.com/research/triton




flash attention1的forward计算中，对于每一个block，是将K,V切分到4个不同的warps（warps 是NVIDIA GPU并行计算的基本单元。一个Warp通常包含32个线程，它们同时执行相同的指令，但对不同的数据进行操作。在GPU执行指令时，通常以Warps为单位进行调度，这可以充分利用GPU的并行处理能力）上，但是将Q保持为对所有的warps是可见的。关于这样修改为什么会减少shared memory的读写以提高性能，paper的原文是这么说的：



### 4. 示例代码

python 复制代码

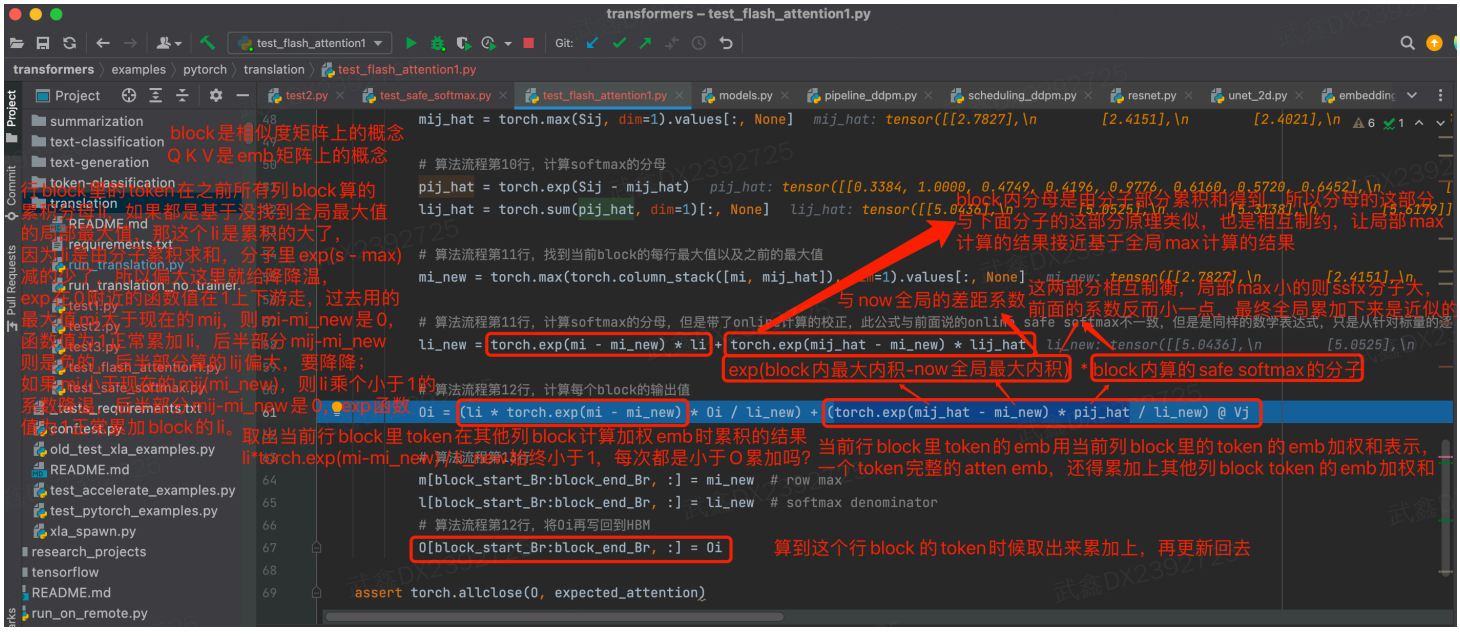
```python
import torch

# 创建一些一维张量
t1 = torch.tensor([1, 2, 3])
t2 = torch.tensor([4, 5, 6])
t3 = torch.tensor([7, 8, 9])

# 使用 column_stack 将它们按列堆叠成一个二维张量
result = torch.column_stack((t1, t2, t3))

print(result)
```

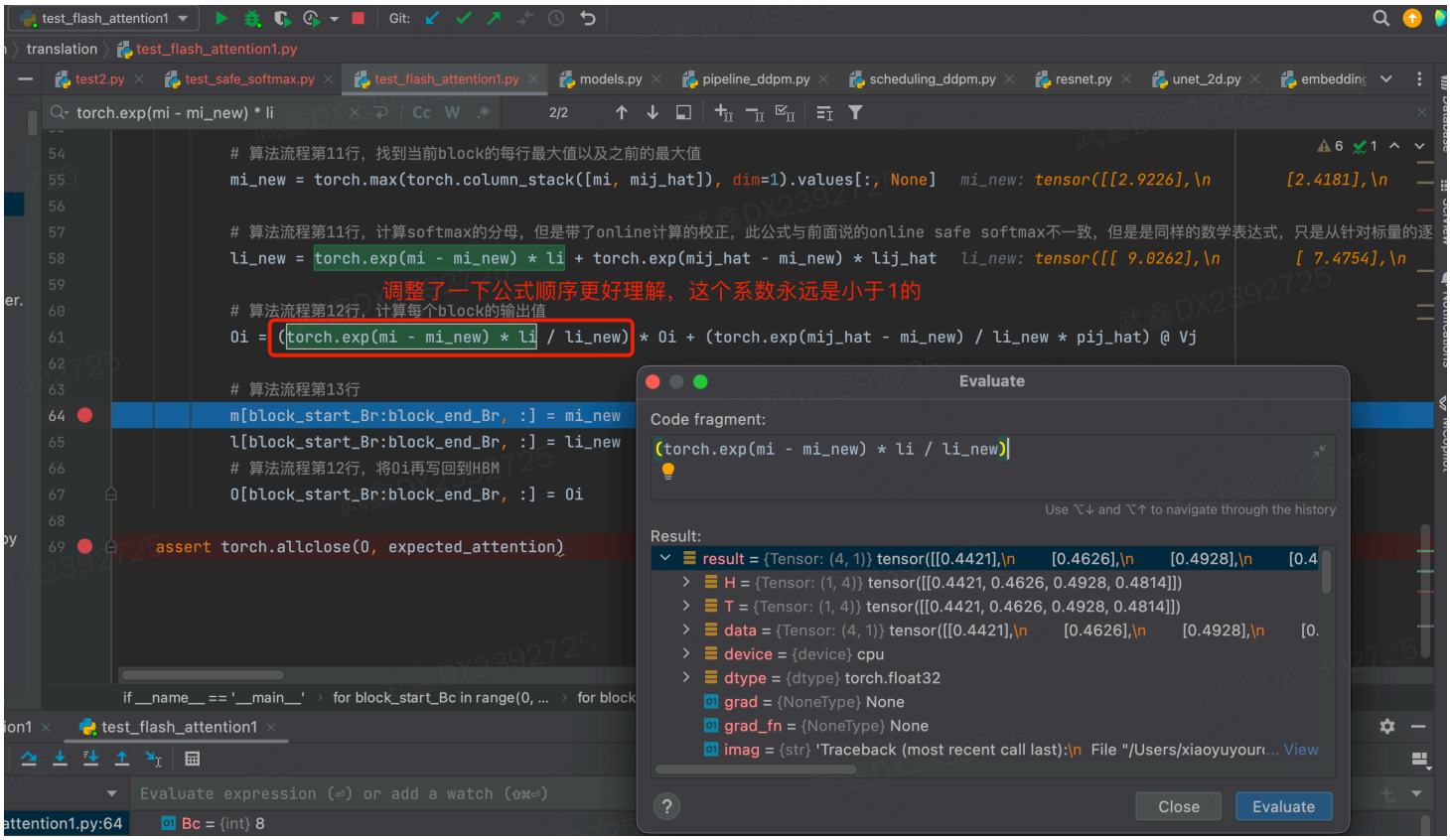
### 5. 输出结果

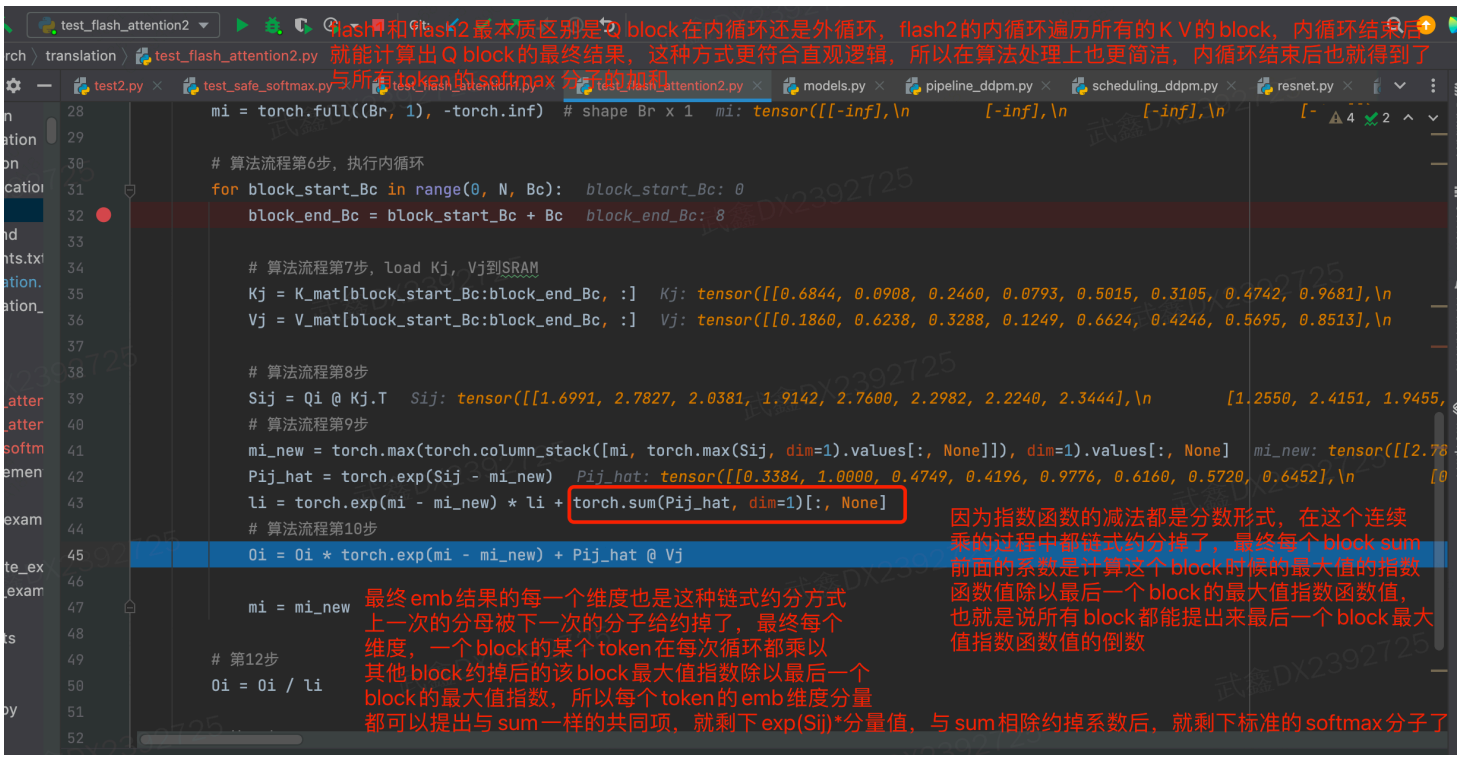
执行上述代码后，`result` 将是一个形状为 `(3, 3)` 的二维张量，内容如下：

复制代码

```
tensor([[1, 4, 7],
        [2, 5, 8],
        [3, 6, 9]])
```






gpu并行训练优化

https://huggingface.co/blog/zh/bloom-megatron-deepspeed

https://huggingface.co/blog/zh/megatron-training

### 实现流水线并行的关键步骤

#### 1. 微批次划分

即使模型的层是相互依赖的，流水线并行仍然可以通过将大的输入批次分割成多个较小的微批次来实现并行。每个微批次可以独立地在流水线的不同阶段进行处理。

#### 2. 流水线调度

在流水线并行中，每个GPU在任何给定时间点都在处理不同的微批次的不同阶段。例如，当第一个微批次在GPU1上处理第一组层时，第二个微批次可以进入流水线并在GPU1上开始处理。一旦第一个微批次完成GPU1上的处理并移动到GPU2，GPU1就可以开始处理第三个微批次。

#### 3. 层间依赖的处理

为了处理层间依赖，每个GPU完成一个微批次的计算后，必须将计算结果（即激活值）传递给下一个GPU。这样，下一个GPU可以使用这些激活值作为其输入并继续计算下一组层。

#### 4. 前向和反向传播

在前向传播期间，激活值按照层的顺序从第一个GPU传递到最后一个GPU。在反向传播期间，梯度按相反的顺序传递回去，每个GPU在计算完自己负责的层的梯度后将梯度传递给前一个GPU。

#### 5. 同步和通信

为了确保数据在GPU之间正确传递，必须在GPU之间进行同步。这通常涉及到使用特定的通信原语（例如，通过NVIDIA NCCL库）来进行高效的数据交换。

#### 6. 管理流水线气泡

在流水线并行中，"气泡"（即空闲时间）可能会出现，因为某些GPU在等待上游GPU完成工作时可能不会执行任何计算。为了最小化这些气泡，可以通过精心设计微批次的大小和数量来优化流水线的效率。

### 并行实现的挑战

- **通信开销**：GPU之间的通信可能会成为瓶颈，特别是当模型层产生大量数据时。
- **负载平衡**：不同的模型层可能有不同的计算需求，因此在分配层到GPU时需要考虑如何平衡负载以避免某些GPU过载而其他GPU空闲。
- **资源限制**：每个GPU的内存大小可能限制了可以并行处理的微批次的数量。

流水线并行要求开发者仔细地考虑如何将模型分割成多个部分，以及如何安排微批次的流动，以确保整个流水线的高效运行。尽管存在这些挑战，流水线并行仍然是一种强大的工具，可以显著提高大型模型训练的效率。