7

Assignment 2 系统篇:Systems and Parallelism

7.1. Assignment 2 Overview

实验目标: 学习如何从**系统底层**优化提升**单 GPU 的训练速度**以及如何将训练扩展到**多 GPU**。

每一部分的实验任务

1. 基准测试与性能分析框架
2. Flash Attention 2 Triton 内核编写
3. 分布式数据并行训练
4. 优化器状态分片

8

性能分析与基准测试 Profiling and Benchmarking

在我们进行优化之前，我们首先就得清楚程序或模型哪个地方还不够好，哪个地方还有可提升的空间。一般来讲可优化的点就是时间和空间 (对应我们老生常谈的时间复杂度和空间复杂度)。那么我们该如何观察到某一个部分的时间和空间的消耗程度呢?

讲义提供了三种性能评估方式：

1. 使用 Python 标准库进行简单的**端到端基准测试**，以测量前向和后向传递的时间；
2. 使用**NVIDIA Nsight Systems**工具进行计算分析，以了解这些时间在 CPU 和 GPU 操作上的分布；
3. 分析**显存使用情况**。

8.1. 端到端简单性能评估测试

参考资料 8.1

- *https://blog.csdn.net/gitblog_00002/article/details/148993434*
- *https://blog.csdn.net/weixin_40198079/article/details/129726302*

为了找到程序的瓶颈点，实现快速迭代优化。我们最好以**脚本化**的方式进行性能评估测试，例如我们可以把需要调节的超参数设置成命令行参数，然后通过脚本自动化地进行性能评估测试。讲义强烈推荐使用 Sbatch 或 Slurm 平台上的**submitit**工具来实现批量扫描。

8.1.1. Slurm 平台上的 submitit 工具简要介绍

submitit 工具也是一种批量调度脚本化的工具，和使用 shell 脚本不同，submitit 和 python 绑定程度更高，它甚至是直接无缝衔接在 python 代码中的。

举个例子，比方说我们要实现一个简单的加法函数。

```
import sys
import time
def add(a, b):
    time.sleep(10) # 模拟耗时
```

```
    return a + b
if __name__ == '__main__':
    # 从命令行读取参数
    a = int(sys.argv[1])
    b = int(sys.argv[2])

    result = add(a, b)

    # 打印结果，Slurm 会把这个输出保存到文件中
    print(f"计算结果是: {result}")
```

之后我们配合 shell 脚本实现。但如果我们使用了 submitit 工具，我们就可以直接在 python 代码中实现。

```
import submitit
import time
# 我们的“菜谱”函数，和之前完全一样，甚至更纯粹
def add(a, b):
    time.sleep(10) # 模拟耗时
    return a + b
# 1. 实例化一个“助理”(Executor)
# AutoExecutor 会自动检测到你正在 Slurm 环境下，并进行配置
executor = submitit.AutoExecutor(folder="slurm_logs")
# 2. 设置“厨房”要求（等同于 #SBATCH 参数）
executor.update_parameters(
    timeout_min=5,  # 任务最长运行 5 分钟
    cpus_per_task=1,
    mem_gb=1
)
print("准备向集群提交任务...")
# 3. 直接让“助理”把你的 Python 函数和参数送去做
job = executor.submit(add, 5, 7)
print(f"任务已提交，任务 ID 是: {job.job_id}")
# 4. 直接在 Python 里等待并获取结果
result = job.result() # 这行代码会等待任务完成，然后把返回值给你
print(f"从集群拿回了结果: {result}")
```

在需要做**大量重复实验**时（比如机器学习调参）。

假设我们要计算三组不同的加法：(5,7), (10,20), (100,200)。

```
params = [(5, 7), (10, 20), (100, 200)]# 准备多组参数
# 一句话提交一个任务数组!
jobs = executor.map_array(add, *zip(*params)) # zip(*params) 会把 [(5,7),...] 变成
↪  ([5,10,100], [7,20,200])
print(f"一次性提交了 {len(jobs)} 个任务! ")
# 等待所有任务完成，并一次性取回所有结果
results = [job.result() for job in jobs]
print(f"所有任务都完成了，结果是: {results}")
# 输出将会是: 所有任务都完成了，结果是: [12, 30, 300]
```

submitit 的一大核心优势是**控制器与工作任务的解耦**，当执行 job = executor.submit(...) 时，发生了两件独立的事情：

- **控制器（你的本地脚本）**：它的任务是打包你的函数和参数，生成一个 Slurm 脚本，通过 sbatch 命令把它发送给 Slurm 调度器。一旦 Slurm 确认收到（返回一个任务 ID），控制器的主要任务就完成了。之后它做的 job.result() 只是一个可选的“等待和接收”动作。
- **工作任务（集群上的脚本）**：Slurm 收到请求后，会在某个计算节点上启动一个全新的、独立的进程。这个进程负责执行你的深度学习函数。它的生死只由 Slurm 和它自己决定，和你的“控制器”脚本没有任何关系。

8.1.2. 前向和反向传播性能评估

首先，通过计时前向和反向传递来对模型进行最基础的性能分析。由于仅需测量时间和内存消耗，因此将采用随机权重和数据。

Keynote 8.1

***CUDA* 异步调用**

*当 GPU 去进行矩阵计算的时候，由于 CUDA 默认是**异步执行**，CPU 也会继续执行下面的任务，不会等待 GPU 执行完毕之后再去。这样做充分利用了计算资源，但在测量的时候由于是异步，所以不能够通过 time.time() 这种方式直接测量 GPU 计算时间 (因为 CPU 会直接跳过 GPU 运算的步骤)，要测量需要**torch.cuda.synchronize()** 强制同步。*

评估流程:

- 给定**超参数**(例如层数)，初始化模型。
- 生成随机数据批次。
- 运行 w 个**预热步骤**(在开始测量时间之前)，然后记录 n 个步骤的执行时间 (根据参数选择仅前向或前向和反向传递)。

Keynote 8.2

端到端基准测试里面的预热并非学习率预热，而是希望机器在训练了几个步骤达到稳定之后再去测试时间，否则由于刚开始的随机权重和数据，导致损失值非常大，梯度也非常大，导致训练不稳定，时间测量不准确。

- 对于计时，可以使用 Python 的 timeit 模块 (例如使用 timeit 函数，或者使用 timeit.default_timer()，它提供系统最高分辨率的时钟，因此比 time() 更适合作为基准测试的默认选项)。
- 每个步骤后调用 torch.cuda.synchronize()。

8.2. Nsight Systems Profiler

参考资料 8.2

https://github.com/NVIDIA/NVTX
https://nvidia.github.io/NVTX/python/reference.html#nvtx.start_range

端到端基准测试无法揭示模型在前向传播和反向传播过程中具体消耗时间与内存的环节，因而难以发现特定组件的优化空间。为准确掌握程序**各组件（如函数）**的运行耗时，可采用性能分析工具。执行分析器通过在函数开始和结束时插入监控点来检测代码，从而提供函数级别的详细执行统计（包括调用次数、该函数累计耗时等指标）。

像 CProfile 这样的 Python 性能分析器无法对 CUDA 内核进行性能分析，因为 CUDA 内核在 GPU 上是异步执行的。因此我们将利用 NVIDIA 提供的一个可以通过命令行工具 nsys 使用的性能分析器。

8.2.1. nsys 简介

nsys 全称**NVIDIA Nsight Systems**，是一个专门记录 CPU、GPU 具体工作的工具，输出的结果类似甘特图、时序图这种，会配合 nvtx 使用清楚标注每一个事件的起始时间。

nsys 的工作流程（The Workflow）

1. **Step 1: 命令行分析 (Profile on the Command Line)**
   讲义中给出的命令是：
   uv run nsys profile -o result.nsys.rep python benchmark.py
   - nsys profile: 这是核心命令，告诉 Nsight Systems “开始录制！”。
   - -o result.nsys.rep: 这个参数非常重要。-o 代表 output。它告诉 nsys 将所有录制下来的性能数据保存到一个名为 result.nsys.rep 的文件中。.rep 是报告（report）文件的后缀。
   - python benchmark.py: 这是你想要分析的目标程序。
2. **Step 2: 查看报告文件 (The .nsys-rep File)**
3. **Step 3: 图形界面分析 (Analyze in the GUI)**

8.2.2. nvtx

NVTX API 本身是一套**“空接口”或“占位符”**。默认情况下，在你的代码中调用 NVTX 函数**不会产生任何实际操作，也不会带来性能开销**。

它的作用只在当你的程序被一个专门的**“开发者工具”（如性能分析器）**启动时才会被激活。这时，这些 NVTX 调用就会被该工具拦截，并转交（重定向）给工具内部的相应功能来处理。

比如一个运动员在跑道上跑步，而教练在赛道上某个位置（比如 100 米、400 米）打个标记方便后续分析，但这些标记不会对运动员跑步产生直接影响。

由于 NVTX 只定义了调用的规范（比如“标记一个时间范围的开始”），而没有规定具体如何响应，因此不同的开发者工具可以根据自己的需要，自由决定如何利用这些调用信息。例如，有的工具可能用它来计时，有的则用它来触发日志记录。

一些常见工具配合 NVTX 使用的例子

- **打印一条消息到控制台**
- **工具精确记录下每一次 NVTX 调用发生的具体时间点**。当程序运行结束后，工具会将所有这些带有时间戳的事件收集起来，并以时间轴的形式图形化地展示出来。这样可以直观地看到代码中不同标记阶段的执行顺序、持续时间以及它们之间是否存在重叠。
- **统计 NVTX 记录下的时间**
- **作为开关，在 NVTX 调用限定的范围内启用/禁用工具功能**
- **作为一个中转站，将数据转发到其他工具出处理**

NVTX 提供的注解类型

- **Markers**

  基础注解类型，主要传递消息，提供了一些选择**颜色、类别**的参数 (在 Ranges 里也有这些可选参数)，配合 Nsight System 这种可视化的性能分析工具使用。

```
nvtx.**mark**(
    message: str | None = None,
    color: str | int | None = 'blue',
    domain: str | None = None,
    category: str | int | None = None,
    payload: int | float | None = None,)
```

- **Ranges**

  Ranges 用于标记程序在**一段时间内**的活动，就像一对相关联的标记点（一个开始，一个结束）。它主要用来衡量一个函数或一段代码块的执行耗时。
  这也是 cs336 希望我们使用的。
  NVTX 提供了两种不同机制的范围，以适应不同的应用场景：

  1. **推入/弹出式范围 (Push/Pop Ranges):**
     - **工作机制：** 这种范围像一个**栈（Stack）**一样工作，遵循“后进先出”（Last-In, First-Out）的原则。当你 ‘Push‘ 一个新范围时，它就被推入栈顶；当你 ‘Pop‘ 时，最顶端的范围被弹出。
     - **核心特点：** 它们必须是**严格嵌套**的，不能交叉重叠。‘Pop‘ 操作总是自动与同一线程上最近一次的 ‘Push‘ 操作配对。
     - **适用场景：** 非常适合标记结构清晰、层层调用的函数或代码块。例如，‘main‘ 函数调用 ‘function_A‘，‘function_A‘ 又调用 ‘function_B‘。

```
nvtx.push_range("数据处理总流程")
            // 步骤 1: 加载数据
            nvtx.push_range("加载数据")
            ...加载数据的代码...
            nvtx.pop_range() // "加载数据"范围结束

            // 步骤 2: 处理数据
            nvtx.push_range("处理数据")
            ...处理数据的代码...
            nvtx.pop_range() // "处理数据"范围结束

            // 步骤 3: 保存结果
            nvtx.push_range("保存结果")
            ...保存结果的代码...
            nvtx.pop_range() // "保存结果"范围结束
        nvtx.pop_range() // "数据处理总流程"范围结束
        主程序结束
```

在 Nsight system 上的结果就是:

- 一个最长的、名为“数据处理总流程”的范围。- 在它**内部**，依次排列着三个互不重叠的短范围：“加载数据”、“处理数据”和“保存结果”。- 你永远不会看到“加载数据”还没结束，“处理数据”就开始的情况。这就是严格嵌套。

2. **开始/结束式范围 (Start/End Ranges):** - **工作机制：** ‘Start‘ 操作会返回一个唯一的**句柄（Handle）**，这个句柄就像一个凭证，唯一对应。你必须在未来的某个时刻调用 ‘End‘，并将这个凭证传递给它，才能正确地关闭对应的范围。- **核心特点：** 它们可以**任意重叠**，并且一个范围的开始和结束可以发生在**不同的线程**上。- **适用场景：** 用于标记复杂的、异步的或并行的操作，这些操作的生命周期不是简单的嵌套关系。

```
主程序线程:
        ...
        // 步骤 1: 加载数据
        // 步骤 2: 并行处理数据
        // 为工作线程 1 的任务创建一个范围，并拿到凭证 handle1
        handle1 = nvtx.start_range("并行任务 1")
        // 启动工作线程 1，把 handle1 交给它

        // 为工作线程 2 的任务创建一个范围，并拿到凭证 handle2
        handle2 = nvtx.start_range("并行任务 2")
        // 启动工作线程 2，把 handle2 交给它

        // 主线程可以继续做其他事，或者等待线程结束...

        - ---------- 时间流逝 -----------

        工作线程 1 (在另一个 CPU 核心上):
        ...执行任务 1 的代码...
        // 任务完成，用凭证 handle1 关闭对应的范围
        nvtx.end_range(handle1)
```

```
工作线程 2（在另一个 CPU 核心上）：
...执行任务 2 的代码...
// 任务完成，用凭证 handle2 关闭对应的范围
nvtx.end_range(handle2)
```

在性能分析工具的时间轴上，你会看到：

- “并行任务 1”和“并行任务 2”这两个范围是在主线程上**几乎同时开始**的。- 它们的执行过程在时间上是**重叠**的。- 它们的结束点发生在各自的工作线程上，并且结束时间**可能不同**。

3. **Resources**

资源命名：

对于新的线程，在 Nsight 里面分析的结果默认显示 python (TID: 54321) 这种线程名，资源命名就是给这个线程起一个名字方便观测处理。

资源追踪：

是命名的扩展，对于一些标准的比如加锁、解锁的过程，NVTX 只需命名，其他工具可以自动识别并给予、释放资源；

对于一些不标准的，比如自己写的自旋锁，NVTX 除了命名之外，还需要明确指出哪里加锁了，哪里解锁了。

我的结果示例

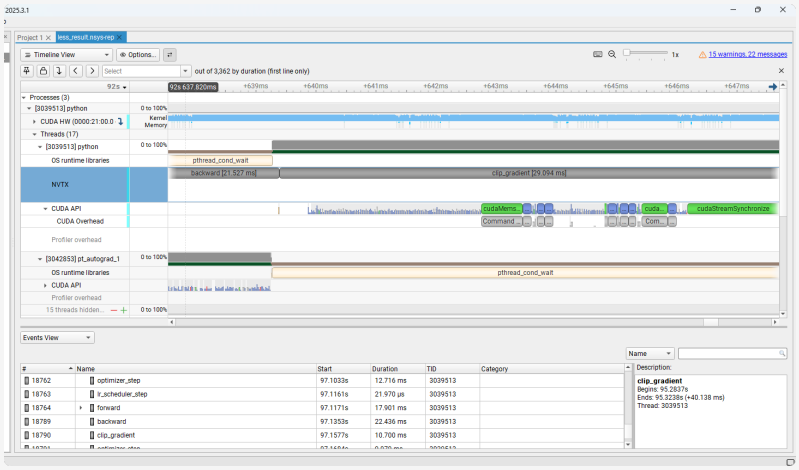

图 8.1: nvtx 结果示例

从结果上来看，没想到裁剪梯度这一步居然这么耗时。。。

8.3. 混合精度训练

到目前为止，在这个任务中，我们一直在使用 FP32 精度——所有模型参数和激活值都具有 torch.float32 数据类型。然而，现代 NVIDIA GPU 包含专门的 GPU 核心（张量核心），用于以较低精度加速矩阵乘法。例如，NVIDIA A100 规格表显示，其在 FP32 下的最大吞吐量为 19.5 TFLOP /秒，而使用 FP16（半精度浮点）或

BF16（脑浮点）时，最大吞吐量显著提高至 312 TFLOP /秒。因此，使用较低精度的数据类型有助于加快训练和推理速度。

然而，若简单地将我们的模型转换为低精度格式，可能会导致模型准确度降低。在此情况下，混合精度训练（Mixed Precision Training）提供了一种解决方案。它允许我们使用较低精度（如 FP16 或 BF16）来计算模型参数和激活值，同时保持 FP32 精度来计算梯度。这不仅提高了训练速度，还避免了精度丢失问题。

总结混合精度训练的特点:

朴素的训练默认使用 32 位浮点数（FP32），而混合精度训练引入了 16 位浮点数（FP16 或 BF16），主要带来两大好处：

1. **减少显存占用**：模型参数、梯度和优化器状态占用的显存几乎减半。这意味着可以用同样的显存训练更大的模型，或者使用更大的批量（batch size）。
2. **加快训练速度**：现代 NVIDIA GPU 内置了 Tensor Cores，专门用于加速 FP16/BF16 的矩阵运算，其理论吞吐量远高于 FP32。使用低精度计算能充分利用这部分硬件性能，带来显著的训练加速（通常能提升 1.5 倍至 3 倍）。

但是，混合精度训练也存在两大挑战：

1. **数据溢出**：直接把所有东西都变成 FP16 是行不通的，因为它的数值范围（约 $6e-5$ 到 65504）太小，非常容易出现**上溢（Overflow）**变成 'inf'和**下溢（Underflow）**变成 '0'，导致训练崩溃。
2. **下溢 (Underflow)**：指一个数因为**太小**（过于接近零），超出了当前数据类型能表示的最小精度范围，结果被强制舍入为**零**。这就像用一把最小刻度是“毫米”的尺子去测量一粒灰尘，由于灰尘太小，你只能记录下它的长度是“0 毫米”。
3. **上溢 (Overflow)**：指一个数因为**太大**，超出了当前数据类型能表示的最大范围，结果变成了一个特殊值——**无穷大 ('inf')**。这就像用一把 30 厘米的尺子去测量一张 1 米长的桌子，你只能说它的长度“超出了尺子的范围”。
4. **舍入误差**：当网络模型的反向梯度很小时，一些 FP32 能够表示的数值可能不能满足 FP16 精度下的表示范围，导致被强行舍入，带来误差。比如 0.66⋯.6（32 位）被舍入成 0.66..7（16 位）。

Keynote 8.3

溢出是范围问题，它是指数值超出了 *FP16* 表示的范围；而舍入误差是精度问题，它发生在数值可以被 *FP16* 表示范围呢，但精度不能够被 *FP16* 表示。

8.3.1. 浮点数知识回顾

定点数表示小数点位置是固定的，而浮点数则是用科学计数法来表示数值。

IEEE 浮点表示

$$V = (-1)^s \times M \times 2^E$$

- 符号 s：s=1 代表负数，s=0 代表正数
- 尾数 M：尾数是浮点数的有效精度部分；
  e.g. $6.75 = 1.6875 \times 2^2$, 1.6875 就是尾数 M。**尾数隐含以 1 开头，M=1+f, 实际存的只有小数点后面的数**
- 阶码 E：决定数值的范围；
  值得注意的是在规格化下：指数位会存在一个偏移，偏移的好处是为了方便数值大小的比较和避免 0 的不唯一表示。偏移量 $bias = 2^{-1} - 1$；比如 8 位指数的偏移量 $bias = 127$ 。
- 最后存储的指数 $E = e() + bias$

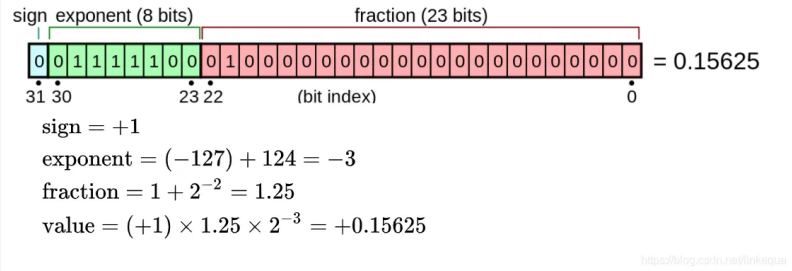

图 8.2: IEEE 浮点表示

半精度浮点数、单精度浮点数和双精度浮点数

- 半精度浮点数 (FP16) 包括 1 位符号位，5 位指数，10 位尾数。
- 单精度浮点数 (FP32) 包括 1 位符号位，8 位指数，23 位尾数。
- 双精度浮点数 (FP64) 包括 1 位符号位，11 位指数，52 位尾数。

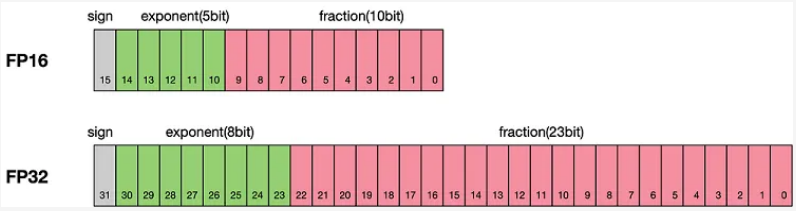

图 8.3: 半精度浮点数、单精度浮点数和双精度浮点数

规格化、非规格化和特殊值

规格化：指数 E 既不是全 0，也不是全 1；$E = e - bias$, 对于单精度全 0→-126, 全 1→127。$bias = 2^{k-1} - 1$；尾数部分隐含 1 开头的情况；大部分都是这种情况。

非规格化：指数 E 全 0；阶码 $E = 1 - bias$; 尾数不全 0，没有隐含的 1,0.$xxxxx \times 2^{-126}$ 为范围 (单精度)。

特殊值：尾数和指数全为 0 时，代表 ±0

指数全为 1，尾数全为 0 时，为 ±∞

当指数全为 1，尾数非全 0，为 NaN

8.3.2. 混合精度训练的实现

梯度缩放 (Loss Scaling): 解决数据溢出问题

举个例子，在模型前向传播、计算损失使用 FP16 精度计算完了之后，某个权重计算得到的真实梯度是 ‘1e-6‘，如果不使用缩放还使用 FP16 去存储的话，那么因为 ‘1e-6‘已经不能被 FP16 半精度所表示，它就自动被视为 0，这样就成了梯度消失，学习无效。

而缩放就是在计算得到梯度之后给它乘一个缩放因子，比如 65536，乘了之后的结果就是 FP16 可以表示的了。

Keynote 8.4

在深度学习训练中，尤其是在模型后期趋于收敛时，计算出的梯度值经常会变得非常非常小，比如 *‘1e-6‘, ‘1e-7‘* 等，所以一般缩放因子都比较大，只针对下溢而非上溢出。

权重备份（Weight Backup）：解决舍入误差

假设你有一个 FP16 格式的权重，值是 ‘0.9824‘。现在计算出一个梯度更新量，在 FP32 下是 ‘0.0010001‘。但在 FP16 下，这个更新量可能被舍入成 ‘0.001000‘。所以 ‘0.9824 + 0.001000‘ 在 FP16 下可能是 ‘0.9834‘。而最后一位的这个小更新就**永久丢失**了。一次两次没问题，成千上万次迭代后，大量的小更新都丢失了，模型就学偏了或不收敛了。

权重备份的思路就是**把权重备份一个 FP32 精度的**，每次计算梯度更新权重的时候就用 FP32 的来计算，得到新结果再转换成 FP16，方便下一次计算梯度、激活等操作。

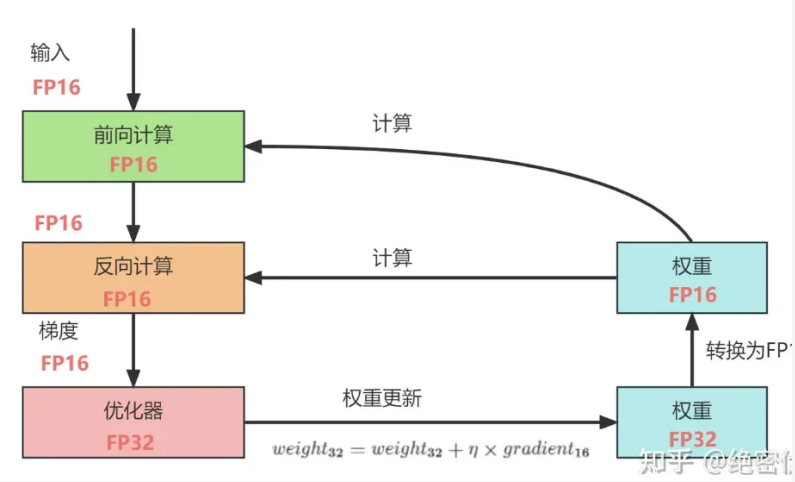

图 8.4: 权重备份

精度累加（precision accumulated）：解决舍入误差

与权重备份类似，精度累加也是利用了 FP32 精度作为中介来解决舍入误差的问题，区别在于精度累加的过程是**在计算矩阵乘法运算**的时候，计算的结果用 FP32 保存减少舍入误差，然后再转成 FP16 格式进行下一步计算。

图 8.5: 精度累加

Keynote 8.5

事实上，在解决舍入误差的问题上，更新后的 *FP32* 主权重转换回 *FP16* 用于下一次计算时，会引入一次新的舍入误差。但这个误差是一次性的、非累积的。真正的关键在于，所有微小的梯度更新信息已经被精确地、永久地累积到了 *FP32* 的主权重上。我们避免了最致命的累积误差问题。

8.4. 显存分析

参考资料 8.3

https://hugging-face.cn/blog/train_memory

此前我们主要探讨了计算的时间性能，接下来将聚焦于**显存**这一语言模型训练与推理中的关键资源。PyTorch 自带强大的内存分析工具 Memory Profiler，可实时追踪内存分配动态。

对于训练和推理大语言模型（LLM）来说，显存（GPU Memory）和计算的时间性能一样，都是极其宝贵的资源。显存瓶颈常常会导致 “Out of Memory” (OOM) 错误，限制了我们能使用的模型大小和批量大小（batch size）。因此，精确地分析显存都用在了哪里，是性能优化的关键一步。

Memory Profiler 的核心流程是：

1. 通过代码启动内存历史记录，运行需要分析的目标代码，

```
torch.cuda.memory._record_memory_history(max_entries=1000000)
```

2. 然后将记录到的内存分配快照（snapshot）保存成一个 ‘pickle‘ 文件。

```
torch.cuda.memory._dump_snapshot("memory_snapshot.pickle")
```

3. 停止记录。

```
torch.cuda.memory._record_memory_history(enabled=None)
```

4. 可视化工具分析。
   - 使用官方工具：访问图片中提到的在线工具 ‘https://pytorch.org/memory_viz‘。
   - 加载快照文件：将你本地生成的 ‘memory_snapshot.pickle‘ 文件拖拽到这个网页上。
   - 解读分析结果：
     - 内存时间线 (Timeline)：你会看到一个图表，显示了总的预留显存（Reserved Memory）和已使用显存（Allocated Memory）随时间的变化。这可以帮助你快速定位显存峰值出现在哪个阶段。
     - 内存分配详情 (Allocations)：工具会列出每一次具体的内存分配事件。最关键的是，它会提供每一次分配的**大小（Size）**和**代码堆栈追踪（Stack Trace）**。通过堆栈追踪，你可以精确地知道是你的代码中的哪一行（例如，‘model.forward()‘ 里的某个特定操作）导致了这次内存分配，从而实现精确优化。

示例代码：

```
import torch
import torch.nn as nn

# 确认你的环境支持 CUDA

if not torch.cuda.is_available():
print("CUDA is not available. This script requires a GPU.")
else:
device = torch.device("cuda")
# 1. 定义一个简单的模型
class SimpleModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = nn.Linear(in_features=1024, out_features=2048)
        self.relu = nn.ReLU()
        self.layer2 = nn.Linear(in_features=2048, out_features=512)

    def forward(self, x):
        return self.layer2(self.relu(self.layer1(x)))

# 2. 实例化模型和输入数据，并移动到 GPU
model = SimpleModel().to(device)
# 创建一个需要梯度的输入张量
input_tensor = torch.randn(256, 1024, device=device, requires_grad=True)

# 3. 开始记录内存历史
print("======== 开始内存性能分析 ========")
```

```
torch.cuda.memory._record_memory_history(max_entries=1000000)

# --- 我们想要分析的核心代码段 ---
# 执行一次前向传播
output = model(input_tensor)
# 计算一个标量损失
loss = output.sum()
# 执行一次反向传播
loss.backward()
# ---------------------------------

# 4. 保存内存快照到文件
snapshot_filename = "simple_model_snapshot.pickle"
torch.cuda.memory._dump_snapshot(snapshot_filename)

# 5. 停止记录
torch.cuda.memory._record_memory_history(enabled=None)

print(f"======== 性能分析结束，快照已保存至 '{snapshot_filename}' ========")
print("请访问 <https://pytorch.org/memory_viz> 并上传该文件进行分析。")
```

薇言大义 8.1

注意这里分析的是 GPU 的显存而非主机的内存。
分析计算性能 (对应时间) 和显存使用 (对应空间) 是非常重要的优化手段。
它让我们从**完成**跨越到**完美**。

9 FlashAttention-2 优化

在上一节利用 Nsight Systems 等工具对模型进行计算分析, 可以看到限制模型内存和计算资源的主要瓶颈主要还是在 Attention 注意力计算上。因此，我们优化的目标也会集中在此。仔细分析过后可以发现，Attention 计算的瓶颈主要在于 IO 感知上，FlashAttention 的优化方法应运而生。

9.1. GPU 内存架构

参考资料 9.1

- *https://www.cnblogs.com/ArsenalfanInECNU/p/18021724*

9.1.1. 物理内存架构：片上内存与片下内存

从硬件位置来看,GPU 内存主要分为两类:**片上 (on-chip) 内存**和**片下 (off-chip) 内存** 。

- **片上内存 (On-chip Memory)**: 位于 GPU 芯片内部, 主要用作高速缓存 (Cache)、共享内存 (Shared Memory) 和寄存器 (Register)。
  - **特点**: 速度极快，但存储空间非常小 。
  - **硬件类型**: 通常是**SRAM**(静态随机存取存储器)，其优点是访问速度快，无需刷新即可保存数据。
- **片下内存 (Off-chip Memory)**: 位于 GPU 芯片外部, 主要用作全局内存 (Global Memory), 也就是我们常说的 “显存”。
  - **特点**: 容量大，但速度相对较慢 。
  - **硬件类型**: 通常是**HBM**(高带宽内存), 它通过将多个 DDR 芯片堆叠并与 GPU 封装在一起, 以实现大容量和高位宽。HBM 属于**DRAM**(动态随机存取存储器), 其成本较低、密度高, 但访问速度慢于 SRAM。

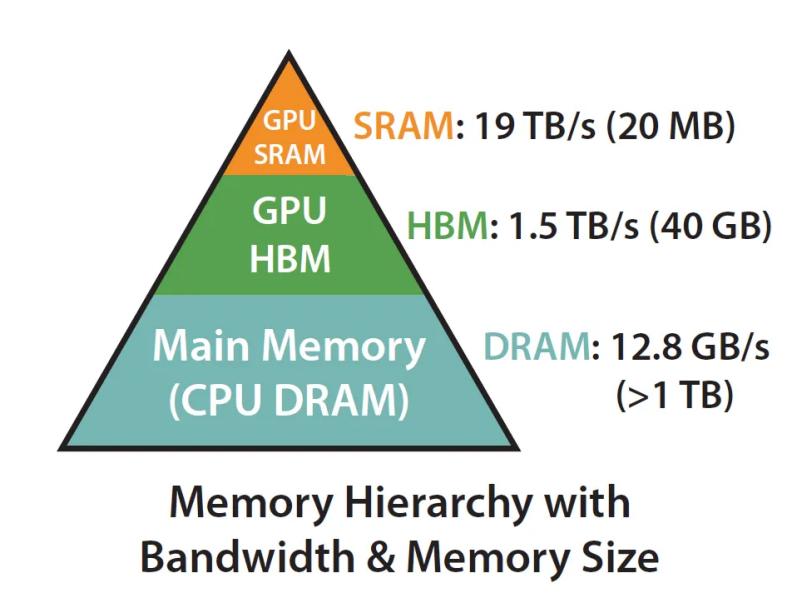

图 9.1: GPU 内存架构

9.1.2. 逻辑内存层次与功能划分

在 CUDA 编程模型中,GPU 内存根据其作用域、生命周期和访问特性被划分为不同的逻辑类型。其速度和容量关系通常是:寄存器 > 共享内存/L1 缓存 > L2 缓存 > 全局内存。

以下是各类内存的详细介绍:

1. **寄存器 (Register)**
   - **位置与速度**: 位于片上, 是 GPU 中速度最快的内存空间。
   - **作用域**: 每个线程独享的私有资源, 用于存储线程内频繁使用的临时变量。
   - **生命周期**: 与核函数 (Kernel) 的执行周期一致, 核函数运行结束即被释放。
   - **特点**: 容量非常有限。如果一个线程需要的变量过多, 寄存器会发生“溢出”, 多出的变量将被存放到速度慢得多的本地内存中, 严重影响性能。
2. **本地内存 (Local Memory)**
   - **位置与速度**: 物理上它与全局内存位于同一区域, 即片下的 HBM/DRAM 中, 因此访问延迟高、速度慢。
   - **作用域**: 和寄存器一样，是每个线程的私有内存。
   - **用途**: 主要用于存放两种数据: □ 编译器无法确定索引的本地数组; □ 因体积过大或数量过多而无法放入寄存器的变量（即寄存器溢出）。
3. **共享内存 (Shared Memory)**

- **位置与速度**: 位于片上 (on-chip), 是可编程的内存, 访问速度非常快, 几乎和寄存器一样。
- **作用域**: 被一个线程块 (Block) 内的所有线程共享, 可用于块内线程间的高效通信。不同线程块之间无法通过共享内存通信。
- **生命周期**: 与线程块的生命周期一致, 线程块执行开始时分配, 执行结束时释放。

4. **全局内存 (Global Memory)**

- **位置与速度**: 位于片下 (off-chip) 的 DRAM/HBM 中, 是 GPU 上容量最大但访问速度最慢的内存空间。通过 ‘cudaMalloc‘等函数分配的内存就在这里。
- **作用域**: 所有线程都可以访问, 是实现不同线程块之间数据通信的唯一途径。
- **生命周期**: 与整个应用程序的生命周期相同, 除非被显式释放。
- **缓存**: 对全局内存的访问会经过 L1 和 L2 缓存来提速。

5. **常量内存 (Constant Memory)**

- **位置与速度**: 物理上驻留在片下的设备内存中, 但每个 SM 都有一个专用的只读常量缓存 (Constant Cache), 因此读取速度很快。
- **作用域**: 对所有线程可见, 但它是只读的。
- **用途**: 主要用于存储在核函数执行期间不会改变的数据。当一个 Warp(一组线程) 中的多个线程需要访问同一个常量数据时, 常量缓存可以实现广播 (broadcast), 一次性满足所有请求, 避免了串行访问, 从而提高效率。

6. **L1/L2 缓存 (Cache)**

- **位置**:L1 缓存位于每个 SM 内部, 被该 SM 内的 CUDA 核心共享。L2 缓存则被 GPU 上所有的 SM 共享。
- **硬件类型**: 它们都是片上 SRAM。
- **作用**: 由系统自动控制, 对程序员不完全透明。它们主要用于缓存对慢速内存 (如全局内存和本地内存) 的访问, 以减少访问延迟。FlashAttention 这类算法的核心思想就是尽可能利用 L1/L2 缓存来减少对 HBM 的读写。

9.2. FlashAttention

参考资料 9.2

- ***参考:FlashAttention–猛猿–知乎***
- ***falshAttention 动画演示-bilibili***

开宗明义：FlashAttention 的背景是在当前原始的 Attention 在 GPU 中，存写速度是逊色于计算速度的，也就是说存写相比于计算时拖了后腿的。为了达到最佳效率，我们需要减少多次存写，哪怕以稍微降低计算速度的代价。

9.2.1. GPU 的 FLOPs 计算能力与内存吞吐能力

多年来，GPU 的计算能力（FLOPS）的增长速度比增加内存吞吐量（TB/s）更快。

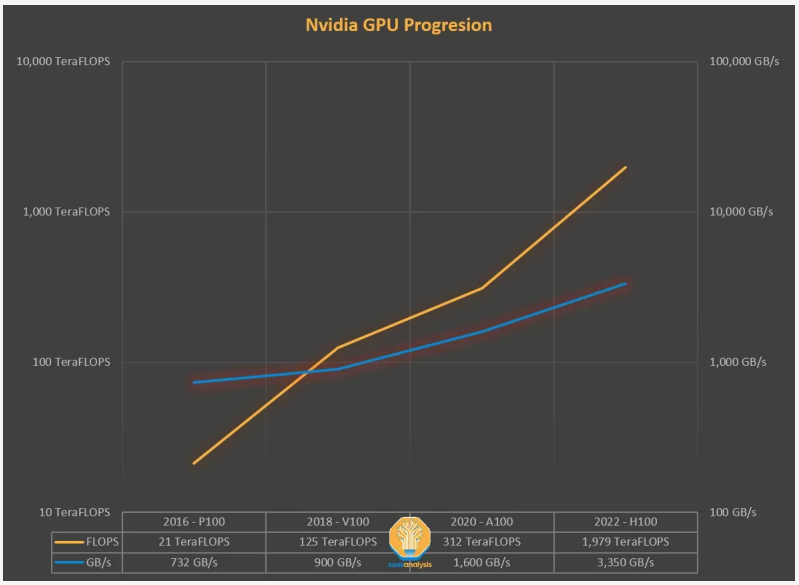

图 9.2: GPU 的 FLOPs 计算能力与内存吞吐能力

如果没有数据需要处理，那么额外的 FLOPS 的计算能力是没有意义的。总而言之，只有**存算**合理配合，才能达到最佳效率。

9.2.2. I/O 感知

IO 感知（IO-awareness）是一种在设计算法时充分考虑**硬件输入/输出（I/O）开销**的优化思想，特别是在 GPU 这类**计算速度远超内存访问速度**的设备上尤为重要。

在现代 GPU 中，性能瓶颈往往不是浮点运算（FLOPs）的速度，而是从**显存（HBM）读写数据的速度**。因此，IO 感知的算法旨在通过优化内存访问模式，特别是**减少对速度较慢、但容量较大的高带宽内存（HBM）的读写次数**，来减少整体的运行时钟时间（wall-clock time）。

简单来说，IO 感知的核心就是：**尽可能减少与慢速内存的数据交换，让计算尽可能在快速内存中完成**。

9.2.3. FlashAttention 如何充分考虑 GPU 的内存层级结构？

标准注意力算法的缺点

标准注意力算法没有感知到 IO 的成本，它会频繁地读写 HBM。例如，它需要计算并存储一个巨大的 $N \times N$ 的中间注意力矩阵（S 和 P）到 HBM 中，然后再从 HBM 中读出进行下一步计算。这些冗余的 HBM 读写操作（IO）占据了大量的计算时间，成为了性能瓶颈。

Transformer 的计算瓶颈不在运算能力，而在读写速度上。

Keynote 9.1

- ***FLOPS***：等同于 *FLOP/s*，表示***Floating Point Operations Per Second***，即每秒执行的浮点数操作次数，用于衡量硬件计算性能。
- ***FLOPs***：表示***Floating Point Operations***，表示某个算法的总计算量（即总浮点运算次数），用于衡量一个算法的复杂度。

FlashAttention 的性能提升方式

1. **利用 SRAM 进行分块计算（Tiling）**：为了避免对 HBM 的反复读写，FlashAttention 的核心思路是将多个操作融合成一个单一的 GPU 核（Kernel），并利用速度快 10 倍左右的 SRAM 进行计算。

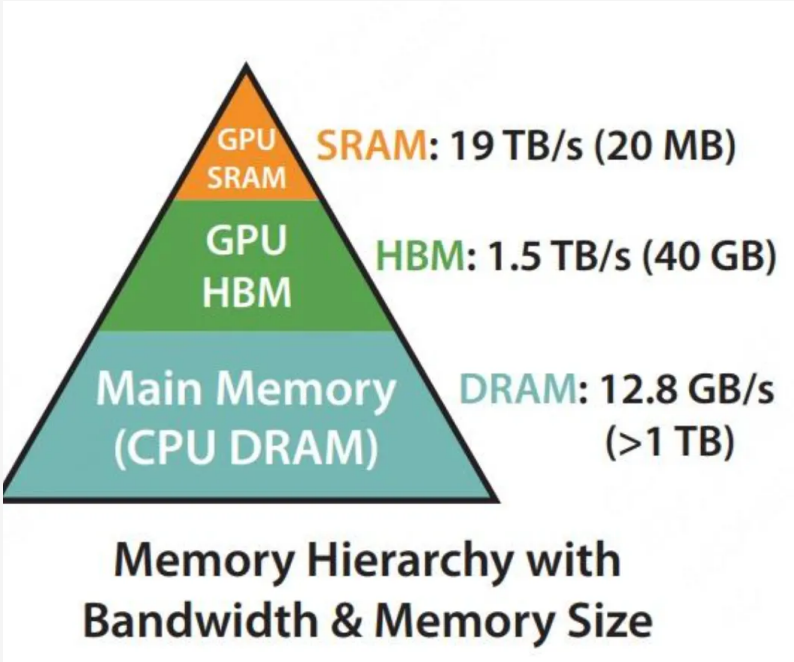

图 9.3: GPU 体系架构

- 它将输入的查询（Q）、键（K）、值（V）矩阵从 HBM 中切分成块（Blocks/Tiles）。
- 然后，它逐块将这些数据加载到高速的 SRAM 中 。所有的中间计算步骤，包括矩阵乘法和 Softmax，都在 SRAM 内部完成，完全不产生将巨大的 ‘N×N‘中间矩阵写回 HBM 的操作。
- 最后，只将计算完成的最终输出结果从 SRAM 一次性写回 HBM 。

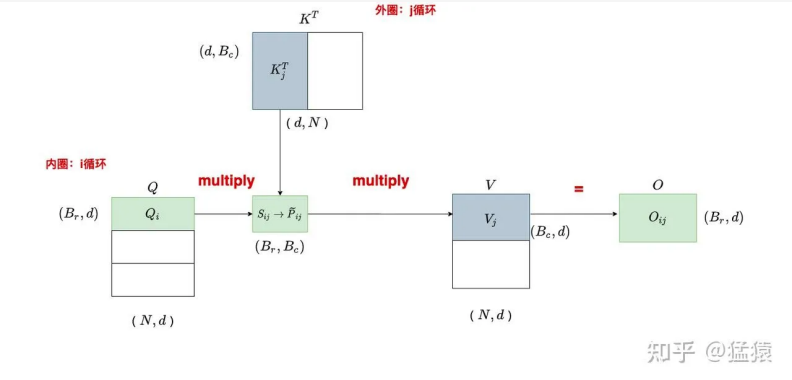

图 9.4: FlashAttention v1 内外循环

2. **通过重计算（Recomputation）减少内存占用**：为了进一步优化内存，FlashAttention 在前向传播时不会保存用于反向传播的中间注意力矩阵（S 和 P）。在反向传播时，它会利用保存在 SRAM 上的输入块以及少量统计数据，快速地重新计算出这些中间值。这种“以时间换空间”的策略极大地节省了 HBM 的占用。

9.2.4. 实现细节

原始 Attention 计算

$$Attention(Q,K,V) = softmax((Q * K^T)/d_k) * V$$

缩放部分 $\sqrt{d_k}$ 的计算不占额外存储，可忽略掉。主要考量 $Q*K^T softmax()*V$ 这三个运算

FlashAttention 的分块计算流程

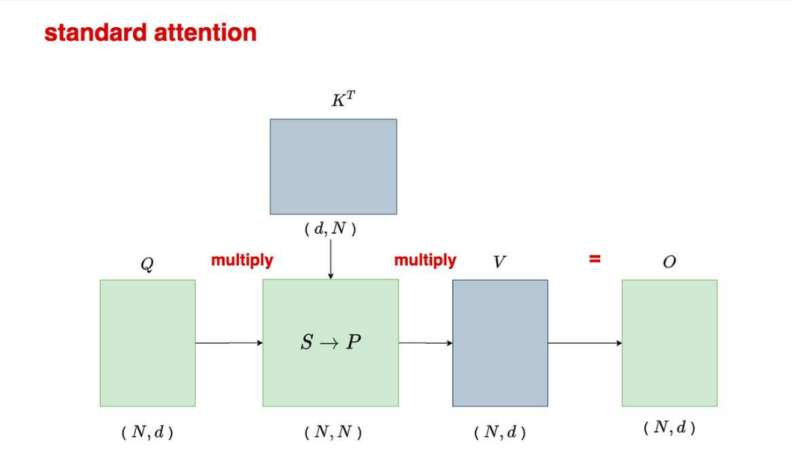

图 9.5: 分块计算

1. 首先，将 Q 矩阵 (形状为 (N,d)) 按照 **“行”** 切为 T_r 块（block），每块的长度为 B_r 。用 Q_i 来表示切完后的某块矩阵，则 Q_i 的维度为 (B_r,d) 。不难

理解，Q_i 中存储着某 B_r 个 token 的 query 信息。

2. 然后，将 K^T 矩阵 (形状为 (d, N)) 按照 **"列"** 切为 T_c 块，每块的长度为 B_c 。用 K_j^T 表示切完后的某块矩阵，则 K_j^T 的维度为 (d, B_c) 。易知 K_j^T 中存储着某 B_c 个 token 的 key 信息。
3. 同样，将 V 矩阵也按照 **"行"** 切为 T_c 块，每块长度为 B_c 。用 V_j 表示切完后的某块矩阵，则 V_j 的维度为 (B_c, d) 。易知 V_j 中存储着某 B_c 个 token 的 value 信息。

计算初始 attention 分数：

$S_{ij} = Q_i * K_j^T = (B_r, d) * (d, B_c) = (B_r, B_c)$

图中的 S_{ij} 表示前 B_r 个 token 和前 B_c 个 token 间的原始相关性分数。

具体循环中的操作示意：

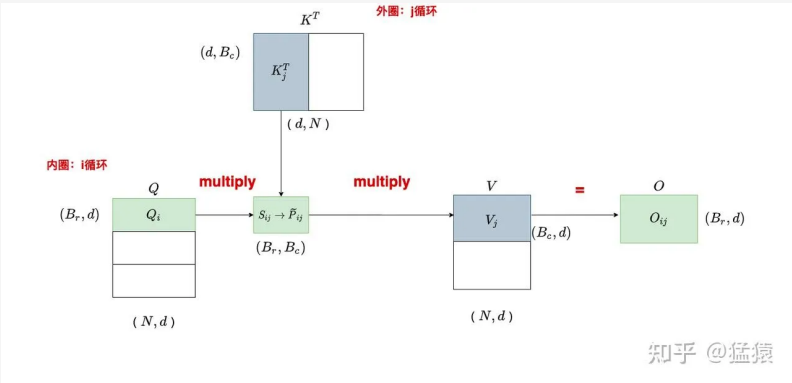

图 9.6: FlashAttention v1 内外循环

softmax 计算操作

原始 softmax 计算：$softmax(x_i) = exp(x_i - m)/_j exp(x_j - m)$ 其中 m 是某一行列的最大值，减去它是为了数值稳定性。

定义:

- $m(x)$：标准场景下，该行的全局最大值
- $m(x^{(1)})$：分块 1 的全局最大值
- $m(x^{(2)})$：分块 2 的全局最大值

那么易知: $m(x) = m([x^{(1)}, x^{(2)}]) = \max(m(x^{(1)}), m(x^{(2)}))$

- $f(x)$：标准场景下，$e^{x-m(x)}$ 的结果
- $f(x^{(1)})$：分块场景下，$e^{x^{(1)}-m(x^{(1)})}$ 的结果
- $f(x^{(2)})$：分块场景下，$e^{x^{(2)}-m(x^{(2)})}$ 的结果

那么易知: $f(x) = [e^{m(x^{(1)})-m(x)} f(x^{(1)}), e^{m(x^{(2)})-m(x)} f(x^{(2)})]$

- $l(x)$：标准场景下，$\text{rowsum}[f(x)]$ 的结果

- $l(x^{(1)})$：分块场景下，rowsum$[f(x^{(1)})]$ 的结果
- $l(x^{(2)})$：分块场景下，rowsum$[f(x^{(2)})]$ 的结果

那么易知: $l(x) = [l(x^{(1)}), l(x^{(2)})]$。

$$\text{softmax}(x) = \frac{f(x)}{l(x)} = \frac{[e^{m(x^{(1)})-m(x)}f(x^{(1)}), e^{m(x^{(2)})-m(x)}f(x^{(2)})]}{e^{m(x^{(1)})-m(x)}l(x^{(1)}) + e^{m(x^{(2)})-m(x)}l(x^{(2)})}$$

分块计算操作：

$$m(x) = \max_i (x_i)$$

$$f(x) = \left[e^{x_1 - m(x)}, \ldots, e^{x_n - m(x)}\right]$$

$$l(x) = \sum_i f(x)_i$$

$$\text{softmax}(x) = \frac{f(x)}{l(x)}$$

FlashAttention 的在线 Softmax 算法

FlashAttention 的核心思想是：我们可以**迭代地**更新 Softmax 的计算结果。每当一个新的数据块到来时，我们用它来**修正**之前基于旧数据块计算出的（不完整的）结果。

让我们把注意力计算中的一行（例如，Q_i 和所有 K_j 的点积）看作是我们要计算 Softmax 的输入向量 x。这个向量 x 被分成了多个块 $x_1, x_2, ..., x_{T_c}$。

算法为这一行维护三个核心变量：

- O: 当前的输出（对 V 的加权和）。
- m: 到目前为止见过的所有 x 元素中的最大值（**running maximum**）。
- l: 到目前为止 Softmax 分母的累加和（**running denominator**）。

现在，我们模拟算法的迭代过程：

假设我们已经处理了第 1 到 j-1 个块，得到了当前的统计量 m_{old} 和 l_{old}，以及当前的输出 O_{old}。

现在，我们处理第 j 个块，x_j。

1. **计算当前块的局部统计量**

   我们只看当前块 x_j，可以计算出它的：

   - 局部最大值: $m_j = max(x_j)$
   - 局部 Softmax 分子: $P_tilde_j = e^{x_j - m_j}$ (注意，减去的是局部最大值)
   - 局部 Softmax 分母: $l_j = sum(P_tilde_j)$

2. **合并统计量，更新全局最大**

   新的全局最大值，一定是旧的全局最大值和当前块最大值中的较大者：$m_{new} = max(m_{old}, m_j)$

3. **重新缩放（Re-scaling）并更新分母 l**

   这是最关键的数学技巧。我们之前的 l_{old} 是基于 m_{old} 计算的，而 l_j 是基于 m_j 计算的。现在我们有了新的全局最大值 m_{new}，我们需要把这两部分都统一到新的基准上再相加。

$l_{new} = l_{old} * e^{m_{old}-m_{new}} + l_j * e^{m_j-m_{new}}$

理解这个公式:

- l_{old} 最初是 $e^{x_{old}-m_{old}}$。乘以 $e^{m_{old}-m_{new}}$ 后，它就变成了 $e^{x_{old}-m_{new}}$。
- l_j 最初是 $e^{x_j-m_j}$。乘以 $e^{m_j-m_{new}}$ 后，它就变成了 $e^{x_j-m_{new}}$。
- 两者相加，就得到了处理完 'j' 块数据后，基于新的全局最大值 m_{new} 的正确的分母总和!

4. **重新缩放并更新输出 O**

输出 O 是对值矩阵 V 的加权和。它的更新逻辑和分母 l 完全一样，也需要重新缩放。

$O_{new} = (O_{old} * l_{old} * e^(m_{old} - m_{new}) + (P_tilde_j * V_j) * e^(m_j - m_{new}))/l_{new}$

理解这个公式:

- 分子部分 (...) 是计算了未归一化的加权和，并将其统一到了 m_{new} 基准上。
- $P_tilde_j * V_j$ 是当前块 x_j 对 V_j 的加权和。
- 最后，除以新的、正确的总分母 l_{new}，就得到了更新后的、正确的输出 O。

在论文的算法图中，这个公式被写作：$O_i diag(l_i^{new})^-1(diag(l_i) * e^{(m_i-m_i^{new})} * O_i + e^{(m_i j-m_i^{new})} * P - tilde_{ij} * V_j)$

这只是上述逻辑的矩阵化写法。$diag(l)$ 乘以 O 是为了还原出未归一化的加权和。

在反向传播中，因为中间计算的结果没有存储，所以要重新计算一下。

9.2.5. Flash Attention 的计算量分析

主要设计两部分矩阵计算:

对于 $S_{ij} = Q_i K_j^T$，其中 $Q_i \in \mathbb{R}^{B_r \times d}, K_j^T \in \mathbb{R}^{d \times B_c}$。根据前置知识，求 S_{ij} 的计算量为 $O(B_r B_c d)$。

对于 $\tilde{P}_{ij} V_j$，其中 $\tilde{P}_{ij} \in \mathbb{R}^{B_r \times B_c}, V_j \in \mathbb{R}^{B_c \times d}$。则这里的计算量同样为 $O(B_r B_c d)$。

接下来我们看一共计算了多少次（1）和（2），也就是执行了多少次内循环: $T_c T_r = \frac{N}{B_c} \frac{N}{B_r}$

综合以上三点，flash attention 的 forward 计算量为: $O\left(\frac{N^2}{B_c B_r} B_r B_c d\right) = O(N^2 d)$

注意，因为计算量是用大 O 阶表示的，所以这里我们把常数项都省略了。

IO 复杂度分析

操作 (Operation)	标准注意力 (Standard Attention)	FlashAttention	备注 (Remarks)
HBM -> SRAM (读操作)			
读 Q, K, V	$O(Nd)$	$O(Nd)$	这是无法避免的初始加载
读中间矩阵 S 或 P $(N \times N)$	$O(N^2)$	$\emptyset$ (不读取)	核心区别: 标准注意力需要 S/P 写回 HBM 再读出, FlashAttention 完全避免了这一步。
SRAM -> HBM (写操作)			
写中间矩阵 S 或 P $(N \times N)$	$O(N^2)$	$\emptyset$ (不写入)	核心区别: FlashAttention 的中间计算结果保留在 SRAM 中, 用完即弃。
写最终输出 O	$O(Nd)$	$O(Nd)$	最终结果必须写回。
总 I/O 复杂度	$O(N^2 + Nd)$	$O(Nd)$	当 $N \gg d$ 时, N^2 项成为不可逾越的瓶颈。

9.3. FlashAttention2 与 FlashAttention1 的区别

v2 主要提升有两点：

1. **引入多线程提升了并行的能力**
2. **继续优化了细节，比如延迟归一化使得内部的除法运算进一步减少。**

v2 最主要的改进：原 v1 在进行 QK^TV 的分块计算时，是把 K^T 放到最外层循环的，QV 放到内层循环。v2 将 Q 放到最外层，K^TV 放到了内层。

从图里面可以看到，对于 v1，K^T 是外层循环，这时候每次和 Q 相乘的内循环得到的中间结果是红色圈中的 **“列”**，这个列是没办法直接和 V 相乘得到完整的结果的，它需要等到下一次 K^T 的循环得到下一个列与 V 相乘得到的结果求和才是最终的 O。

而对于 v2,Q 是外层循环，Q 的每一行是每次循环的变量，这时候这一行可以和所有的内循环的 K^T 列相乘得到中间结果的行，即图中画蓝、红、紫的圈，这几个行可以单独与 V 也进行相乘，得到最终结果 O_{00}、O_{10} 等单个元素，不需要等待下一次循环。这样做好处就是可以把 Q 按照行进行各自独立的分割，每一行之间互不影响，就可以放到多个线程里面并行优化。

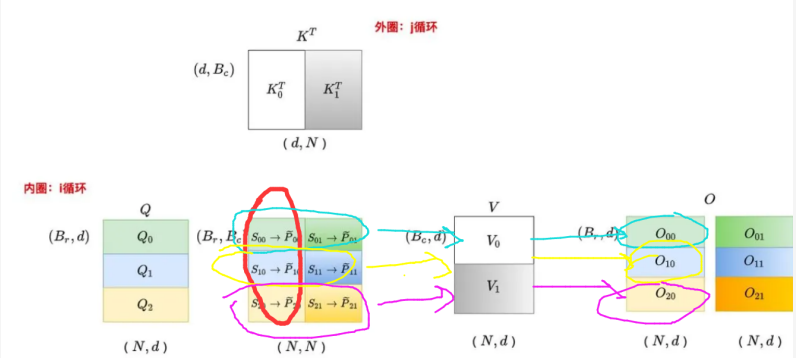

图 9.7: FlashAttention v2 内外循环

并行策略的进一步增强

- **FlashAttention-1**: 其并行化主要在 **批次（batch size）和头（head）** 这两个维度上进行。每个注意力头分配一个线程块。当序列很长而批次和头数较少时，可并行的线程块数量可能远少于 GPU 上的流多处理器（SM）数量，导致 GPU 利用率低下。
- **FlashAttention-2**: 在 V1 的基础上，增加了在**序列长度（sequence length）维度上的并行化** 。
  - **前向传播**：将 Q 矩阵按行切分，不同的**行块**（row blocks）分配给不同的线程块并行计算 。
  - **反向传播**：由于梯度计算的依赖关系不同（dK 和 dV 需要按行累加），为了优化，反向传播是按**列块**（column blocks）分配给不同的线程块并行计算的 。

Warp 间工作分区的优化：减少内部通信

即使在单个线程块（Thread Block）内部，V2 也优化了工作分配方式。一个线程块由多个 Warp（通常每个包含 32 个线程）组成。

- **FlashAttention-1**: 采用了一种称为 **“Split-K”** 的策略。它将 K 和 V 矩阵沿着序列维度切分给不同的 Warp，而 Q 块对所有 Warp 可见。这种方式的缺点是，每个 Warp 计算出的是部分结果，为了得到最终的行输出，必须将这些中间结果写入共享内存（SRAM），进行 Warp 间的同步和聚合（相加），这引入了不必要的通信和共享内存读写开销，成为性能瓶颈。
- **FlashAttention-2**: 转而采用 **“Split-Q”** 策略。它将 Q 矩阵切分给不同的 Warp，而 K 和 V 块对所有 Warp 可见。因为不同 Q 的计算是独立的，每个 Warp 在计算完自己的 $(Q_slice * K^T) * V$ 后，直接得到最终输出 O 的一个分片，**无需与其他 Warp 进行通信或聚合** 。这极大地减少了共享内存的读写和同步开销，是 V2 速度提升的关键之一。

算法层面的优化

FlashAttention-2 在算法层面进行了一些精简，以减少开销较高的非矩阵乘法浮点运算（non-matmul FLOPs）。

- **延迟归一化 (Rescaling)**: 在 V1 中，每一次内循环迭代，都需要用当前的局部 softmax 分母 l 来重新缩放（rescale）累积的输出 O，这涉及到除法运算。FlashAttention-2 优化了这一点，在内循环中，它只更新 O 的“未归一化”版本，将最终的除法操作**延迟到外循环结束时才执行一次**。这个调整减少了中间步骤的除法运算次数，更充分地利用了为矩阵乘法优化的 Tensor Cores 。
- **反向传播存储优化**: V1 为了反向传播时的重计算，需要从 HBM 中读取并保存两个统计量 m (最大值) 和 l (exp 分数总和) 。V2 则将它们**合并，只存储一个 $L = m + log(l)$（即 log-sum-exp）**，在反向传播时由此恢复所需信息。这进一步减少了一次 HBM 的 I/O 操作，节省了内存带宽。

9.3.1. FlashAttentionv3

相比于 FlashAttentionv1、v2，v3 没有太多算法上的创新，主要有两个性能提升的点：

1. **矩阵计算 (GEMM) 和 softmax 计算流水线化**，交错并行，不严格顺序执行，而是会有时间的重叠。
2. **硬件加速的 FP8 低精度计算与精度保持**。