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

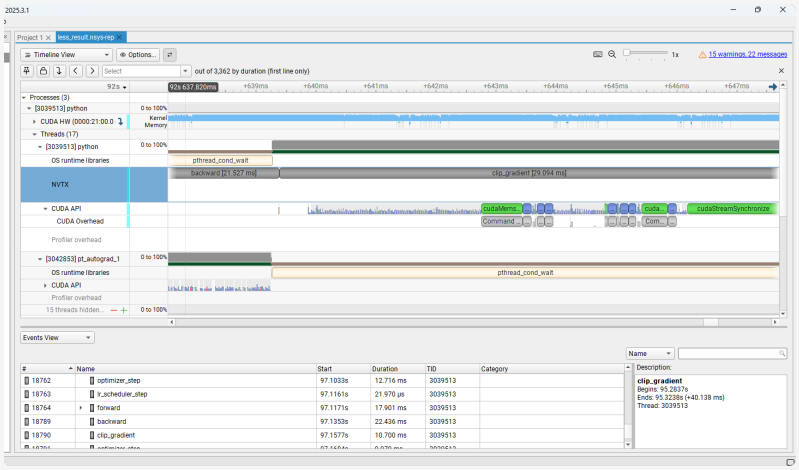

图 8.1: nvtx 结果示例

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。

7.4.2.1 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 $E = e() + bias$

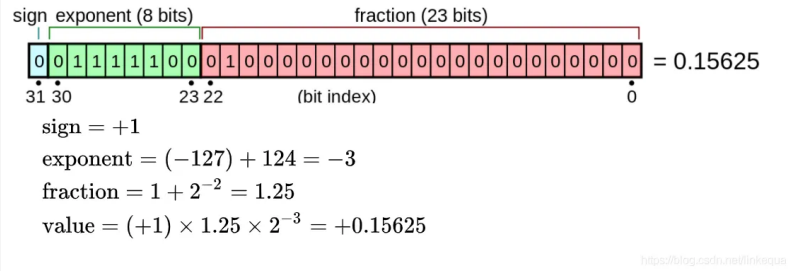

图 8.2: IEEE 浮点表示

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。

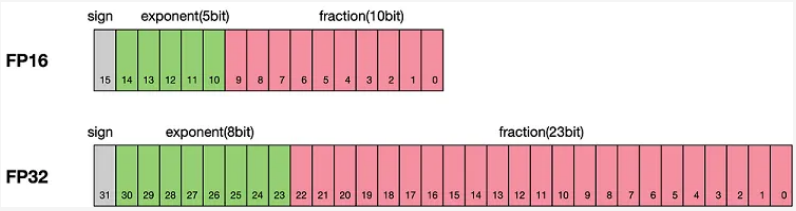

图 8.3: 半精度浮点数、单精度浮点数和双精度浮点数

7.4.2.2 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

7.4.3.1 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。

7.4.3.2 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。

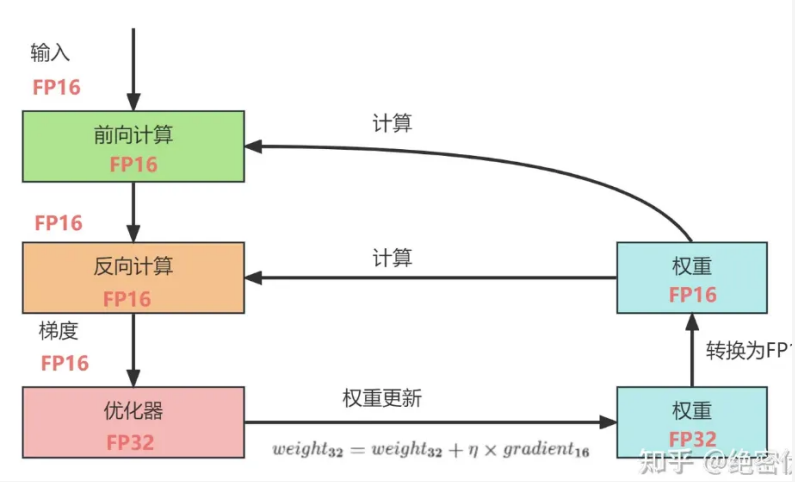

图 8.4: 权重备份

7.4.3.3 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**。