CS336 作业 2（系统）：系统和并行性

版本 1.0.4

2025 年春季

1 作业结束 view

在本作业中，您将获得一些提高单 GPU 训练速度和将训练扩展到多个 GPU 的实践经验。

您将实施的内容。

1. 基准测试和分析线束
2. Flash Attention 2 Triton 内核
3. 分布式数据并行训练
4. 优化器状态分片

代码是什么样的。所有赋值代码以及这篇文章都可以在 GitHub 上获得：

github.com/stanford-cs336/assignment2-systems

请 git clone 存储库。如果有任何更新，我们会通知您，您可以 git pull 获取最新信息。

1. cs336-basics/：在此作业中，您将分析我们在作业 1 中构建的一些组件。此文件夹包含作业 1 的教职员工解决方案代码，因此您将找到一个 cs336-basics/pyproject.toml 和一个 cs336-basics/cs336_basics/* 模块。如果你想使用你自己的模型实现，你可以修改基目录中的 pyproject.toml 文件以指向你自己的包。

2. /：cs336-systems 基本目录。我们创建了一个名为 cs336_systems 的空模块。请注意，这里没有代码，所以你应该能够从头开始做任何你想做的事。

3. tests/*.py：这包含您必须通过的所有测试。这些测试调用 tests/adapters.py 中定义的钩子。你将实现适配器以将代码连接到测试。编写更多测试和/或修改测试代码有助于调试代码，但您的实现应通过原始提供的测试套件。

4. README.md：此文件包含有关预期目录结构的更多详细信息，以及有关设置环境的一些基本说明。

如何提交。您将向 Gradescope 提交以下文件：

- writeup.pdf：回答所有书面问题。请排版您的回复。
- code.zip：包含您编写的所有代码。

在 test_and_make_submission.sh 中运行脚本以创建 code.zip 文件。

在作业的第一部分中，我们将研究如何优化 Transformer 模型的性能，以最有效地利用 GPU。我们将分析我们的模型，以了解它在前向和后向传递期间花费的时间和内存的位置，然后使用自定义 GPU 内核优化自注意力作，使其比简单的 PyTorch 实现更快。在作业的后续部分，我们将利用多个 GPU。

1.1 分析和基准测试

在实施任何优化之前，首先分析我们的程序以了解它在哪里花费资源（例如时间和内存）会很有帮助。否则，我们可能会优化模型中不占用大量时间或内存的部分，因此看不到可衡量的端到端改进。

我们将实现三种性能评估路径：（a） 使用 Python 标准库进行简单的端到端基准测试，以计时我们的前向和后向传递，（b） 使用 NVIDIA Nsight Systems 工具进行分析计算，以了解该时间如何在 CPU 和 GPU 上的作中分配，以及（c） 分析内存使用情况。

1.1.1 设置 - 导入 Fundamentals Transformer 模型

首先，让我们确保您可以从上一个分配中加载模型。在上一个作业中，我们在 Python 包中设置了模型，以便以后可以轻松导入。我们在 ./cs336-basics 文件夹中添加了模型的 staff 实现，并在 pyproject.toml 文件中指向了它。通过像往常一样调用 uv run [command]，uv 会自动定位这个本地 cs336-basics 包。如果您想使用自己的模型实现，可以修改 pyproject.toml 文件以指向您自己的包。

您可以使用以下命令测试是否可以导入模型：

```
~$ uv 运行 python
使用 CPython 3.12.10
在以下位置创建虚拟环境：/path/to/uv/env/dir
        内置 cs336-systems @ file:///path/to/systems/dir
内置 cs336-basics @ file:///path/to/basics/dir
在 711 毫秒内安装了
85 个软件包
Python 3.12.10 （main， Apr 9， 2025， 04：03：51） [Clang 20.1.0 ] 在 linux 上
...
>>>导入 cs336_basics
>>>
```

作业 1 中的相关模块现在应该可用（例如，对于 model.py，您可以使用 import cs336_basics.model 导入它）。

1.1.2 模型大小调整

在整个任务中，我们将对模型进行基准分析和分析，以更好地了解它们的性能。为了了解事物如何大规模变化，我们将使用并参考以下模型配置。对于所有模型，我们将使用 10,000 的词汇表大小和 4 的批量大小，上下文长度不同。此作业（以及以后的作业）将需要大量结果以表格形式呈现。我们强烈建议您在代码中自动为写作构建表，因为在 LaTeX 或 Markdown 中格式化表可能非常乏味。参见熊猫。DataFrame.to_latex（） 和熊猫。DataFrame.to_markdown（）或编写您自己的函数以从您首选的表格表示形式生成它们。

尺码	d_model	d_ff	num_layers	num_heads
小号	768	3072	12	12
中号	1024	4096	24	16
大号	1280	5120	36	20
XL	1600	6400	48	25
2.7B	2560	10240	32	32

表 1：不同型号尺寸规格

1.1.3 端到端基准测试

现在，我们将实现一个简单的性能评估脚本。我们将测试模型的许多变体（更改精度、交换层等），因此让您的脚本通过命令行参数启用这些变体以使其易于以后运行将是值得的。我们还强烈建议使用 sbatch 或在 Slurm 上提交它对基准测试超参数（例如模型大小、上下文长度等）运行扫描，以便快速迭代。

首先，让我们通过计时前向和后向传递来对模型进行最简单的分析。由于我们只会测量速度和内存，因此我们将使用随机权重和数据。

衡量绩效是微妙的——一些常见的陷阱可能会导致我们无法衡量我们想要的东西。对于对 GPU 代码进行基准测试，需要注意的是 CUDA 调用是异步的。调用 CUDA 内核时，例如调用 torch.matmul 时，函数调用会将控制权返回给代码，而无需等待矩阵乘法完成。这样，CPU 可以继续运行，而 GPU 计算矩阵乘法。另一方面，这意味着天真地测量 torch.matmul 调用返回所需的时间并不能告诉我们 GPU 实际运行矩阵乘法需要多长时间在 PyTorch 中，我们可以调用 torch.cuda.synchronize（） 来等待所有 GPU 内核完成，从而使我们能够获得更准确的 CUDA 内核运行时测量。考虑到这一点，让我们编写基本的分析基础设施。

问题（benchmarking_script）：4 分

（a） 编写一个脚本来执行前向和后向传递的基本端到端基准测试你的模型。具体而言，脚本应支持以下内容：

- 给定超参数（例如，层数），初始化模型。
- 生成随机批次数据。
- 运行 w 个预热步骤（在开始测量时间之前），然后计时执行 n 个步骤（仅向前，或向前和向后传递，具体取决于参数）。对于计时，您可以使用 Python timeit 模块（例如，使用 timeit 函数或使用 timeit.default_timer（），它为您提供了系统的最高分辨率时钟，因此比 time.time（） 更好的基准测试默认值）。
- 在每个步骤之后调用 torch.cuda.synchronize（）。

可交付成果：一个脚本，它将使用给定的超参数初始化基本 Transformer 模型，创建随机批次数据，并对前进和后退传递进行时间。

（b） 对 §1.1.2 中描述的模型尺寸的前向和后向传递进行时间。使用 5 个预热步骤并计算 10 个测量步骤的时序的平均值和标准差。

向前传球需要多长时间？向后传球怎么样？您是否看到测量之间的差异很大，或者标准差很小？

可交付成果：1-2 句话的回复，说明您的时间安排。

（c） 基准测试的一个警告是不执行预热步骤。重复分析，无需预热步骤。这对您的结果有何影响？你认为为什么会这样？还要尝试使用 1 或 2 个预热步骤运行脚本。为什么结果可能仍然不同？

可交付成果：2-3 句话的回复。

1.1.4 Nsight 系统分析器

端到端基准测试不会告诉我们模型在前向和后向传递期间花费的时间和内存在哪里，因此不会暴露特定的优化机会。要了解我们的程序在每个组件（例如函数）上花费了多少时间，我们可以使用分析器。执行分析器通过在函数开始和完成运行时插入守卫来检测代码，因此可以在函数级别提供详细的执行统计信息（例如调用次数、平均花费的时间、在此函数上花费的累积时间等）。

标准 Python 分析器（例如 CProfile）无法分析 CUDA 内核，因为这些内核是在 GPU 上异步执行的。幸运的是，NVIDIA 提供了一个分析器，我们可以通过 CLI nsys 使用它，我们已经为您安装了它。在作业的这一部分中，您将使用 nsys 来分析 Transformer 模型的运行时。使用 nsys 很简单：我们可以简单地运行上一节中的 Python 脚本，并在前面添加 nsys 配置文件。例如，您可以使用以下命令 benchmark.py 分析脚本并将输出写入文件 result.nsys.rep：

```
~$ uv run nsys profile -o result python benchmark.py
```

然后，您可以使用 NVIDIA Nsight Systems 桌面应用程序在本地计算机上查看配置文件。在配置文件的 CUDA API 行中选择特定的 CUDA API 调用（在 CPU 上）将突出显示 CUDA 硬件行中所有相应的内核执行（在 GPU 上）。

我们鼓励您尝试 nsys profile 的各种命令行选项，以了解它可以做什么。值得注意的是，您可以使用 --python-backtrace=cuda 为每个 CUDA API 调用获取 Python 回溯，尽管这可能会带来开销。您还可以使用 NVTX 范围对代码进行注释，这些范围将显示为捕获所有 CUDA API 调用和相关内核执行的配置文件的 NVTX 行中的块。特别是，您应该使用 NVTX 范围来忽略基准测试脚本中的预热步骤（通过对配置文件中的 NVTX 行应用筛选器）。您还可以隔离哪些内核负责模型的前向和后向传递，甚至可以通过注释您的实现来隔离哪些内核负责自注意力层的不同部分，如下所示：

```
...
将 torch.cuda.nvtx 导入为 nvtx

@nvtx.range（“缩放点积注意”）
def annotated_scaled_dot_product_attention（ ...
# Q， K， V， 掩码
)
    ...
与 nvtx.range（“计算注意力分数”）:
        ... # 计算 Q 和 K 之间的注意力分数

    使用 nvtx.range（“计算 softmax”）
        ... # 计算注意力分数的 softmax

    使用 nvtx.range（“final matmul”）
        ... # 计算输出投影
```

```

        返回。。。
```

您可以通过以下方式将原始实现与基准测试脚本中的带注释的版本交换：

```
cs336_basics.model.scaled_dot_product_attention = annotated_scaled_dot_product_attention
```

最后，可以将 --pytorch 命令行选项与 nsys 一起使用，以使用 NVTX 范围自动注释对 PyTorch C++ API 的调用。

问题（nsys_profile）：5 分

使用 nsys 分析您的前向传递、后向传递和优化器步骤，表 1 中描述的每个模型大小和上下文长度为 128、256、512 和 1024（对于较大的模型，其中一些上下文长度可能会耗尽内存，在这种情况下，只需在报告中注明即可）。

（a） 您的前进传球所花费的总时间是多少？它是否与我们之前使用 Python 标准库测量的内容相匹配？

可交付成果：1-2 句话的回复。

（b） 哪个 CUDA 内核在前向传递过程中花费最多的累积 GPU 时间？在模型的单次前向传递期间，该内核被调用了多少次？当你同时进行前向和后向传递时，它是否是同一个内核占用最多的运行时间？（提示：查看“统计系统视图”下的“CUDA GPU 内核摘要”，并使用 NVTX 范围进行过滤，以确定模型的哪些部分负责哪些内核。可交付成果：1-2 句话的回复。

（c） 尽管绝大多数 FLOP 都发生在矩阵乘法中，但您会注意到其他几个内核仍然占用了整个运行时间的相当大的一部分。除了矩阵乘法之外，您还认为哪些其他内核考虑了前向传递中重要的 CUDA 运行时？

可交付成果：1-2 句话的回复。

（d） 在实现 AdamW 时分析运行一个完整训练步骤（即前向传递、计算损失并运行向后传递，最后是优化器步骤，就像您在训练期间所做的那样）。与进行推理（仅前向传递）相比，矩阵乘法所花费的时间比例如何变化？其他内核怎么样？

可交付成果：1-2 句话的回复。

（e） 在前向传递期间，比较模型自注意力层内 softmax 运算与矩阵乘法运算的运行时间。运行时间的差异与 FLOP 的差异相比如何？

可交付成果：1-2 句话的回复。

1.1.5 混合精度

到目前为止，我们一直以 FP32 精度运行，所有模型参数和激活都具有 torch.float32 数据类型。然而，现代 NVIDIA GPU 包含专门的 GPU 内核（张量核心），用于以较低的精度加速矩阵乘法。例如，NVIDIA A100 规格表称其 FP32 的最大吞吐量为 19.5 TFLOP/秒，而 FP16（半精度浮点）或 BF16（大脑浮点）的最大吞吐量明显更高，为 312 TFLOP/秒。因此，使用较低精度的数据类型应该有助于我们加快训练和推理速度。

然而，天真地将我们的模型转换为较低精度的格式可能会降低模型准确性。例如，实践中的许多梯度值通常太小而无法在 FP16 中表示，因此在使用 FP16 精度进行天真训练时变为零。为了解决这个问题，在使用 FP16 进行训练时通常使用损失缩放——损失只是乘以缩放因子，增加梯度幅度，这样它们就不会齐平为零。此外，FP16 的动态范围低于 FP32，这可能导致溢出，表现为 NaN 损失。完整的 bfloat16 训练通常更稳定（因为 BF16 与 FP32 具有相同的动态范围），但与 FP32 相比，仍然会影响最终模型的性能。

若要利用低精度数据类型的加速，通常使用混合精度训练。在 PyTorch 中，这是使用 torch.autocast 上下文管理器实现的。在这种情况下，某些运算（例如，矩阵乘法）以较低精度的数据类型执行，而其他需要 FP32 全动态范围的运算（例如，累积和缩减）则保持原样。例如，以下代码将自动识别在前向传递期间要以较低精度执行的作，并将这些作转换为指定的数据类型：

1 个模型： torch.nn.Module = ... # 例如您的 Transformer 模型

2 dtype： torch.dtype = ... # 例如，torch.float16 3 x：
torch.Tensor = ... # 输入数据

4

5 with torch.autocast（device=“cuda”，dtype=dtype）： 6 y = model（x）如上所述，即使正在累积的张量本身已被下调，通常最好保持更高的精度累积。以下练习将帮助您直觉了解为什么会出现这种情况。

问题 （mixed_precision_accumulation）：1 分

运行以下代码并赞扬结果的（准确性）。

```
s = torch.tensor（0，dtype=torch.float32） 对于
range（1000） 中的 i：
s += torch.tensor（0.01，dtype=torch.float32） 打印


s = torch.tensor（0，dtype=torch.float16） 对于
range（1000） 中的 i：
s += torch.tensor（0.01，dtype=torch.float16） 打印


s = torch.tensor（0，dtype=torch.float32） 对于
range（1000） 中的 i：
s += torch.tensor（0.01，dtype=torch.float16） 打印


s = torch.tensor（0，dtype=torch.float32） 对于
range（1000） 中的 i：
x = torch.tensor（0.01，dtype=torch.float16） s +=
x.type（torch.float32） 打印
```

可交付成果：2-3 句话的回复。

现在，我们将首先将混合精度应用于玩具模型以进行直觉，然后应用于基准测试脚本。

问题（benchmarking_mixed_precision）：2 分

（a） 考虑以下模式：

```
类 ToyModel（nn.模块）：
def __init__（self， in_features： int， out_features： int）：
        超级（）.__init__（）
self.fc1 = nn。线性（in_features， 10， bias=False） self.ln = nn。层规
范（10）
self.fc2 = nn。线性（10， out_features， bias=False） self.relu = nn。
ReLU（）

def forward（self， x）： x =
self.relu（self.fc1（x））
x = self.ln（x）
x = self.fc2（x）
        返回 x
```

假设我们在 GPU 上训练模型，并且模型参数最初位于 FP32 中。我们希望在 FP16 中使用自动转换混合精度。有哪些数据类型：

- 自动转换上下文中的模型参数，
- 第一个前馈层 （ToyModel.fc1） 的输出，
- 层范数 （ToyModel.ln） 的输出，
- 模型的预测 logits，
- 损失，
- 模型的梯度？

可交付成果：上面列出的每个组件的数据类型。

（b） 您应该已经看到，FP16 混合精度自动铸造对层归一化层的处理方式与前馈层不同。层归一化的哪些部分对混合精度敏感？如果我们用 BF16 代替 FP16，我们还需要区别对待层归一化吗？

为什么或为什么不？

可交付成果：2-3 句话的回复。

（c） 修改您的基准测试脚本，以选择使用混合精度与 BF16 运行模型。
对于 §1.1.2 中描述的每种语言模型大小，使用和不具有混合精度的前向和后向传递时间。比较使用全精度与混合精度的结果，并评论模型大小变化时的任何趋势。您可能会发现 nullcontext no-op 上下文管理器很有用。

可交付成果：2-3 句话的回复，其中包含您的时间安排和评论。

1.1.6 分析内存

到目前为止，我们一直在研究计算性能。现在，我们将注意力转移到记忆上，这是语言模型训练和推理的另一个主要资源。PyTorch 还附带了一个强大的内存分析器，它可以跟踪一段时间内的分配情况。

要使用内存分析器，您可以修改基准测试脚本，如下所示：

```

... # 基准测试脚本中的预热阶段

# 开始记录内存历史。
torch.cuda.memory._record_memory_history（max_entries=1000000）

... # 你想在基准测试脚本中分析什么

# 保存一个要由 PyTorch 的在线工具加载的 pickle 文件。
torch.cuda.memory._dump_snapshot（“memory_snapshot.pickle”）

# 停止记录历史记录。
torch.cuda.memory._record_memory_history（enabled=None）
```

这将输出一个文件 memory_snapshot.pickle, 您可以将其加载到以下在线工具中：https://pytorch.org/memory_viz 。此工具将允许你查看总体内存使用时间线以及所做的每个单独分配，以及其大小和指向其来源代码的堆栈跟踪。要使用此工具，您应该在 Web 浏览器中打开上面的链接，然后将 Pickle 文件拖放到页面上。

现在，您将使用 PyTorch 分析器来分析模型的内存使用情况。

问题（memory_profiling）：4 分

分析表 1 中 2.7B 模型的前向传递、后向传递和优化器步骤，上下文长度为 128、256 和 512。

（a） 在分析脚本中添加一个选项，以通过内存分析器运行模型。重用一些以前的基础设施（例如，激活混合精度、加载特定模型大小等）可能会有所帮助。然后，运行脚本以获取 2.7B 模型的内存配置文件，在仅执行推理（仅前向传递）或完整训练步骤时。你的记忆时间线是什么样的？你能根据你看到的峰值来判断哪个阶段正在运行吗？

可交付成果：来自 memory_viz 工具的 2.7B 模型的“活动内存时间线”的两张图像：一张用于前向传递，另一张用于运行完整的训练步骤（前向和向后传递，然后是优化器步骤），以及一个 2-3 句子的响应。

（b） 进行前向传递时，每个上下文长度的峰值内存使用量是多少？当进行完整的训练步骤时呢？

可交付成果：每个上下文长度包含两个数字的表。

（c） 在使用混合精度时，找到 2.7B 模型的峰值内存使用量，用于前向传递和全优化器步骤。混合精度是否显着影响内存使用？

可交付成果：2-3 句话的回复。

（d） 考虑 2.7B 模式。在我们的参考超参数中，Transformer 残差流中激活张量的大小是多少，以单精度表示？以 MB 为单位给出此大小（即，将字节数除以 1024）。

可交付成果：包含您的推导的 1-2 句话回复。

（e） 现在仔细查看 2.7B 模型进行前向传递的内存快照 pytorch.org/memory_viz 的“活动内存时间线”。当您降低“详细信息”级别时，该工具会隐藏相应级别的最小分配（例如，将“详细信息”设置为 10% 仅显示

10% 的最大分配)。显示的最大分配的规模是多少？查看堆栈跟踪，您能说出这些分配来自哪里吗？可交付成果：1-2 句话的回复。

1.2 使用 FlashAttention-2 优化注意力

1.2.1 基准测试 PyTorch 关注

您的分析可能表明，在注意力层中，在内存和计算方面都有机会进行优化。在高级别上，注意力运算包括矩阵乘法，然后是 softmax，然后是另一个矩阵乘法：

((QK √))

注意（Q、K、V）=软 max 面具 - $\overline{}$ V （1）

d

朴素注意力实现需要为每个批次/头元素保存形状 seq_len × seq_len 的注意力分数矩阵，这些矩阵可能会随着序列长度的延长而变得非常大，从而导致任何具有长输入或输出的任务出现内存不足错误。我们将在 FlashAttention2 论文之后实现一个注意力内核，它按图块计算注意力，并避免显式具体化 seq_len × seq_len 注意力分数矩阵，从而能够扩展到更长的序列长度。

问题（pytorch_attention)：2 分

（a） 在不同尺度上对注意力实施进行基准测试。编写一个脚本，该脚本将：

（a） 将批量大小固定为 8，不要使用多头注意力（即删除头尺寸)。
（b）迭代头嵌入维度 d 的笛卡尔积[16,32,64,128]，序列长度的[256,1024,4096,8192,16384]。

（c） 创建适当大小的随机输入 Q、K、V。
（d） 时间 100 向前使用输入通过注意力。
（e） 测量在向后传递开始之前使用了多少内存，并计时 100 次向后传递。
（f） 确保预热，并在每次向前/向后传递后调用 torch.cuda.synchronize（)。

报告您为这些配置获得的计时（或内存不足错误)。在多大时会出现内存不足错误？在您发现内存不足的最小配置之一中计算注意力的内存使用情况（您可以使用赋值 1 中的 Transformer 内存使用方程)。为向后保存的内存如何随着序列长度而变化？你会做些什么来消除这种内存成本？
可交付成果：包含您的时间表、内存使用情况和 1-2 段回复。

1.3 基准测试 JIT 编译注意力

从 2.0 版开始，PyTorch 还附带了一个强大的即时编译器，它会自动尝试对 PyTorch 函数应用许多优化：有关介绍，请参阅 https://pytorch.org/tutorials/intermediate/ torch_compile_tutorial.html。特别是，它将尝试通过动态分析您的计算图来自动生成融合的 Triton 内核。使用 PyTorch 编译器的界面非常简单。例如，如果我们想将其应用于模型的单个层，我们可以使用：

```
层 = SomePyTorchModule（...）
compiled_layer = torch.compile（层）
```

现在，compiled_layer 在功能上就像层一样（例如，它的前向和后向传递）。我们还可以使用 torch.compile（model） 编译整个 PyTorch 模型，甚至是调用 PyTorch 作的 Python 函数。

问题（torch_compile）：2 分

（a） 扩展您的注意力基准测试脚本以包含注意力的 PyTorch 实现的编译版本，并将其性能与与上述 pytorch_attention 问题具有相同配置的未编译版本进行比较。

可交付成果：一个表格，将已编译注意力模块的前向和后向传递时序与上述 pytorch_attention 问题中的未编译版本进行比较。

（b） 现在，在端到端基准测试脚本中编译整个 Transformer 模型。前传的性能如何变化？组合的前向和后退传递以及优化器步骤呢？

可交付成果：比较普通和编译的 Transformer 模型的表格。

鉴于我们在序列长度方面看到的缩放行为，我们需要进行重大改进来处理大型序列。即使使用 torch.compile，当前的实现在长序列长度下也存在非常差的内存访问模式。为此，我们将编写 FlashAttention-2 的 Triton 实现，在其中，我们将更好地控制内存的访问方式以及何时计算什么。

1.3.1 示例 - 加权和

为了介绍您需要了解的有关 Triton 的信息以及它如何与 PyTorch 互作，我们将通过一个示例内核来进行“加权和”作。有关快速使用 Triton 的更多资源，请参阅 Triton 的教程。我们注意到，这些教程没有使用新的、方便的块指针抽象，我们将在下面介绍它。

给定输入矩阵 X，我们将其条目乘以逐列权重向量 w，然后对每一行求和，得到 X 和 w 的矩阵向量乘积。我们将首先完成此作的前向传递，然后为后向传递编写 Triton 内核。

前向传递 我们内核的前向传递只是下面广播的内积。

```
def weighted_sum（x， weight）：
    # 这里，假设 x 具有 n-dim 形状 [...， D]，重量具有 1D 形状 [D]
    return （weight * x）.sum（axis=-1）
```

在编写 Triton 内核时，我们将让每个程序实例（可能并行运行）计算 x 行图块的加权和，并将相应的标量输出写入输出张量。在 Triton 中，程序实例是运行同一程序的线程块，这些线程块可以在 GPU 上并行运行。我们不将张量作为参数，而是将指向它们的第一个元素的指针，以及告诉我们如何沿轴移动的每个张量的步幅。

我们可以使用步幅加载与我们在运行实例中求和的 x 行图块相对应的张量，使用程序 ID 来划分工作（即，实例 i 将处理 x 行的第 i 个图块）。在这个简单的情况下，Triton 和 PyTorch 中的前向传递之间的主要区别在于需要进行指针算术和显式加载/存储。我们将使用

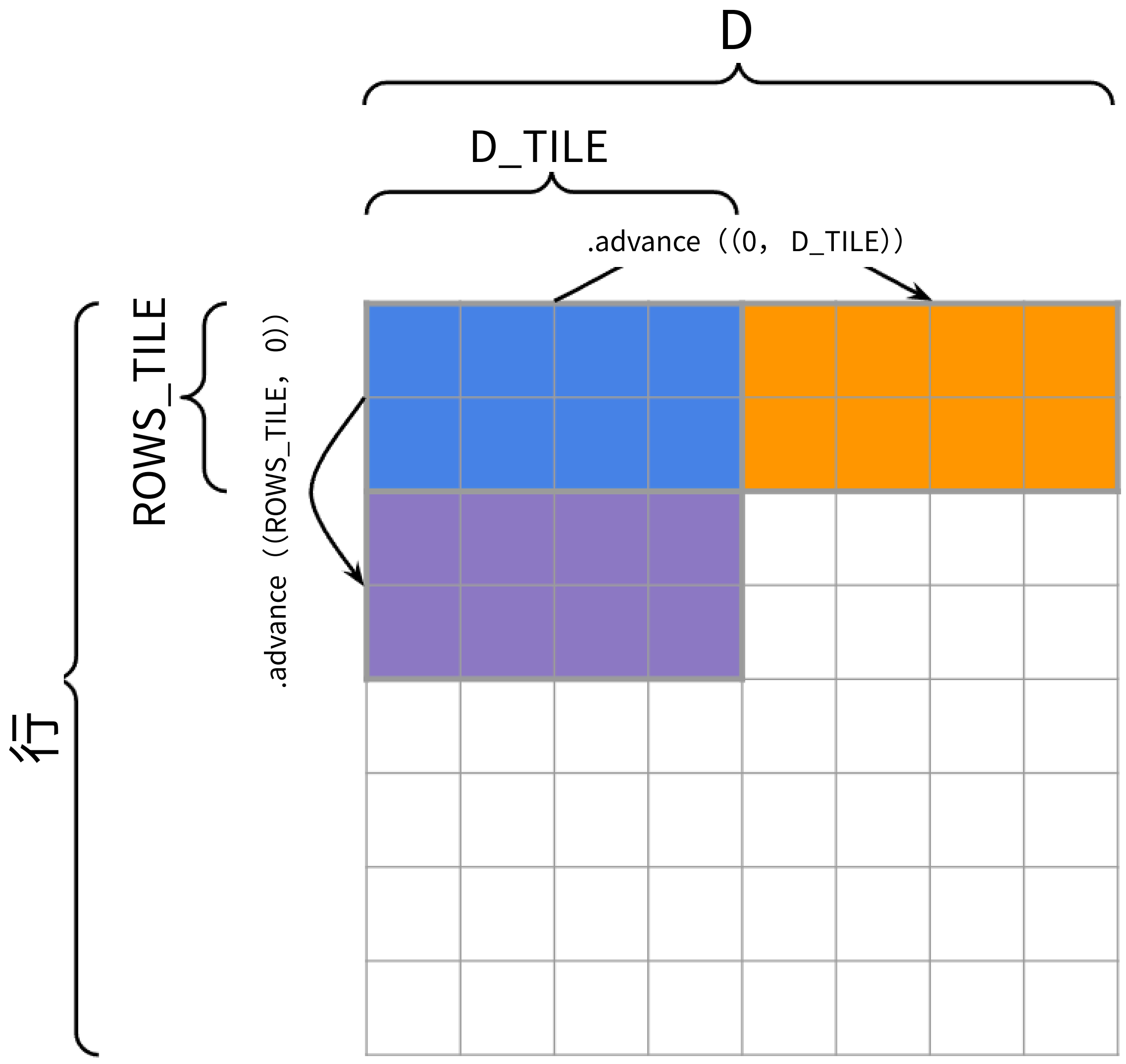

图 1：加权和内核示例中的平铺和前进块指针（第 1.3.1 节）。

tl.make_block_ptr 大大简化指针算术，尽管这意味着我们需要进行一些设置来准备块指针。请参阅图 1 了解平铺的示意图以及如何推进块指针。上面的加权和函数如下所示：

```
进口海卫一
将 triton.language 导入为 tl

@triton.jit
定义 weighted_sum_fwd（
x_ptr， weight_ptr， # 输入指针
output_ptr， # 输出指针
x_stride_row， x_stride_dim， # Strides 告诉我们如何在张量的每个轴上移动一个元素
weight_stride_dim， # 可能 1
output_stride_row，# 可能 1
行，D,
        ROWS_TILE_SIZE： tl.constexpr， D_TILE_SIZE： tl.constexpr， # 在编译时必须知道磁贴形状
    ):
        # 每个实例将计算 x 行图块的加权和。
        # 'tl.program_id' 为我们提供了一种检查我们在哪个线程块中运行的方法
        row_tile_idx = tl.program_id（0）

        # 块指针为我们提供了一种从内存的 ND 区域中进行选择的方法
        # 并移动我们的选择。
        # 块指针必须知道：
        # - 指向张量第一个元素的指针
        # - 用于处理越界访问的张量的整体形状
```

```
        # - 正确使用内存布局的每个维度的步幅
        # - 起始块的 ND 坐标，即“偏移量”
        # - 一次加载/存储使用的块形状
        # - 内存中维度从大到小的顺序
        # 用于优化的轴 （= np.argsort（strides）），在 H100 上特别有用

        x_block_ptr = tl.make_block_ptr（
x_ptr,
形状=（行，D, ），
步=（x_row_stride，x_stride_dim），
偏移量=（row_tile_idx * ROWS_TILE_SIZE， 0），
block_shape=（ROWS_TILE_SIZE， D_TILE_SIZE），
订单=（1， 0），
)

weight_block_ptr = tl.make_block_ptr（
weight_ptr,
形状=（D, ），
步=（weight_stride_dim, ），
个偏移量=（0, ），
block_shape=（D_TILE_SIZE, ），
订单=（0, ），
)

output_block_ptr = tl.make_block_ptr（
output_ptr,
形状=（行, ），
步=（output_stride_row, ），
偏移量=（row_tile_idx * ROWS_TILE_SIZE, ），
block_shape=（ROWS_TILE_SIZE, ），
订单=（0, ），
)

# 初始化要写入的缓冲区
        输出 = tl.zeros（(ROWS_TILE_SIZE, ）， dtype=tl.float32）

        对于 i in range（tl.cdiv（D， D_TILE_SIZE））：
            # 加载当前块指针
            # 由于 ROWS_TILE_SIZE 可能不会划分 ROWS，D_TILE_SIZE 可能不会划分 D,
            # 我们需要对两个维度进行边界检查
            row = tl.load（x_block_ptr， boundary_check=（0， 1）， padding_option=“zero”） # （ROWS_TILE_SIZE， D_TILE_SIZE）
            重量 = tl.load（weight_block_ptr， boundary_check=（0, ）， padding_option=“零”） # （D_TILE_SIZE, ）

            # 计算行的加权和。
            输出 += tl.sum（行 * 权重[无， :]， 轴=1）

            # 将指针移动到下一个图块。
            # 这些是（行、列）坐标增量
            x_block_ptr = x_block_ptr.advance（(0， D_TILE_SIZE)） # 在最后一个维度中按 D_TILE_SIZE 移动
            weight_block_ptr = weight_block_ptr.advance（(D_TILE_SIZE, )） # 移动 D_TILE_SIZE

# 将输出写入输出块指针（每行一个标量）。
# 由于 ROWS TILE  SIZE 可能不会划分 ROWS，因此我们需要边界检查
tl.store（output_block_ptr， output， boundary_check=（0, )）
```

现在让我们将这个内核包装在一个 PyTorch Autograd 函数中，该函数将与 PyTorch 互作（即，将张量作为输入，输出一个张量，然后在向后期间也与 autograd 引擎一起工作

通过）：

```
类 WeightedSumFunc（torch.autograd.Function）：
    @staticmethod
        def forward（ctx， x， weight）：
            # 缓存 x 和权重，用于后向传递，当我们
            # 只接收梯度 wrt。输出张量，以及
```

```
        # 需要计算梯度 wrt。x 和重量。
        D， output_dims = x.shape[-1]， x.shape[：-1]

        # 将输入张量重塑为 2D
        input_shape = x.shape
        x = rearrange（x， “...d -> （...） d “）

        ctx.save_for_backward（x，权重）

assert len（weight.shape） == 1 和 weight.shape[0] == D，“维度不匹配”
断言 x.is_cuda 和 weight.is_cuda，“预期的 CUDA 张量”
断言 x.is_contiguous（），“我们的指针算术将假设连续的 x”

ctx。D_TILE_SIZE = triton.next_power_of_2（D） // 16 # 大约 16 个通过嵌入维度的循环
ctx。ROWS_TILE_SIZE = 16 # 每个线程一次处理 16 个批处理元素
ctx.input_shape = input_shape

# 需要初始化空结果张量。请注意，这些元素不一定是 0！
y = torch.empty（output_dims， device=x.device）

# 在我们的 1D 网格中使用 n 个实例启动我们的内核。
n_rows = y.numel（）
weighted_sum_fwd[(cdiv（n_rows， ctx.ROWS_TILE_SIZE), )]（
x，重量，
岁，
x.步幅（0）， x.步幅（1），
weight.stride（0），
岁步幅（0），
行=n_rows，D=D，
ROWS_TILE_SIZE=ctx。ROWS_TILE_SIZE，D_TILE_SIZE=ctx。D_TILE_SIZE，
）

return y.view（input_shape[：-1]）
```

请注意，当我们使用 weighted_sum_fwd[(cdiv（n_rows， ctx.ROWS_TILE_SIZE),)]，我们通过传递元组（cdiv（n_rows， ctx.ROWS_TILE_SIZE),)。然后，我们可以在内核中使用 tl.program_id（0） 访问线程块索引。

向后传递由于我们正在定义自己的内核，因此我们还需要编写自己的向后函数。

在前向传递中，我们获得了层的输入，并且需要计算其输出。在向后传递中，回想一下，我们将获得目标相对于输出的梯度，并且需要计算相对于每个输入的梯度。在我们的例子中，我们的作有一个矩阵 x： R 和一个权重向量 w： R 作为输入。简而言之，我们将我们的作称为 f （x， w），其范围为 R。然后，假设给定 ∇L，即相对于层输出的损失 L 的梯度，我们可以应用多元链规则获得相对于 x 和 w 的梯度的以下表达式：

$$(\nabla L) = \sum_{k=1}^{n} \frac{\partial f\,(x,\ w)}{\partial x}(\nabla L) = w\cdot(\nabla L) \quad (2)$$

$$(\nabla L) = \sum_{\text{我}=1}^{n} \frac{\partial f\,(x,\ w)}{\partial w}(\nabla L) = \sum_{\text{我}=1}^{n} x\cdot(\nabla L) \quad (3)$$

这给出了一个计算向后传递的简单公式。为了获得相对于 x 的向后阶跃，我们应用方程 2 并取 w 和 ∇ 的外乘积。要计算相对于 w（即 （∇L））的向后步长，我们必须将输入梯度乘以相应的输出行。

我们用于反向传递的内核将首先定义所有块指针，然后计算 ∇L：

```
@triton.jit
def weighted_sum_backward（
    x_ptr， weight_ptr， # 输入
    grad_output_ptr， # 毕业生输入
    grad_x_ptr， partial_grad_weight_ptr， # Grad 输出
    stride_xr、stride_xd、
    stride_wd,
stride_gr,
stride_gxr、stride_gxd、
stride_gwb、stride_gwd、
NUM_ROWS，D,
ROWS_TILE_SIZE：tl.constexpr，D_TILE_SIZE：tl.constexpr,
):
row_tile_idx = tl.program_id（0）
n_row_tiles = tl.num_programs（0）

# 输入
grad_output_block_ptr = tl.make_block_ptr（
grad_output_ptr,
形状=（NUM_ROWS, ），步幅=（stride_gr, ），
偏移量=（row_tile_idx * ROWS_TILE_SIZE, ），
block_shape=（ROWS_TILE_SIZE, ），
订单=（0, ），
）

x_block_ptr = tl.make_block_ptr（
x_ptr,
形状=（NUM_ROWS，D, ），步幅=（stride_xr，stride_xd），
个偏移量=（row_tile_idx * ROWS_TILE_SIZE， 0），
block_shape=（ROWS_TILE_SIZE，D_TILE_SIZE），
订单=（1， 0），
）

weight_block_ptr = tl.make_block_ptr（
        weight_ptr,
        形状=（D, ），步幅=（stride_wd, ），
        偏移量=（0, ）， block_shape=（D_TILE_SIZE, ），
        order=（0, ），
    )

    grad_x_block_ptr = tl.make_block_ptr（
        grad_x_ptr,
        形状=（NUM_ROWS，D, ），步幅=（stride_gxr，stride_gxd），
        偏移量=（row_tile_idx * ROWS_TILE_SIZE， 0），
        block_shape=（ROWS_TILE_SIZE，D_TILE_SIZE），
        order=（1， 0），
    )

    partial_grad_weight_block_ptr = tl.make_block_ptr（
        partial_grad_weight_ptr,
形状=（n_row_tiles，D, ），步幅=（stride_gwb，stride_gwd），
个偏移量=（row_tile_idx， 0），
block_shape=（1， D_TILE_SIZE），
订单=（1， 0），
）

表示范围内的 i（tl.cdiv（D， D_TILE_SIZE））：
grad_output = tl.load（grad_output_block_ptr， boundary_check=（0, ）， padding_option=“零”） # （ROWS_TILE_SIZE, ）

# 外层产品用于 grad_x
重量 = tl.load（weight_block_ptr， boundary_check=（0, ）， padding_option=“零”） # （D_TILE_SIZE, ）
grad_x_row = grad_output[:， 无] * 权重[无， :]
tl.store（grad_x_block_ptr， grad_x_row， boundary_check=（0， 1））

# 为 grad_weight 结果尽可能多地减少行数
```

```
            row = tl.load（x_block_ptr， boundary_check=（0， 1）， padding_option=“zero”） #（ROWS_TILE_SIZE， D_TILE_SIZE）
            grad_weight_row = tl.sum（行 * grad_output[:， 无]， 轴=0， keep_dims=True）
            tl.store（partial_grad_weight_block_ptr， grad_weight_row， boundary_check=（1, )） # 暗淡 0 永远不会越界

            # 沿 D 将指针移动到下一个图块
            x_block_ptr = x_block_ptr.advance（(0， D_TILE_SIZE)）
            weight_block_ptr = weight_block_ptr.advance（(D_TILE_SIZE, )）
partial_grad_weight_block_ptr = partial_grad_weight_block_ptr.advance（(0， D_TILE_SIZE)）
grad_x_block_ptr = grad_x_block_ptr.advance（(0， D_TILE_SIZE)）
```

计算梯度∇很简单，我们将结果写入输出张量的相应图块中。然而，计算∇更具挑战性。每个内核实例负责 x 的一个行图块，但我们现在需要跨 x 行求和。我们不会直接在向后传递中执行此求和，而是假设 partial_grad_weight_ptr 包含一个 n_row_tiles × H 矩阵，其中第一个维度仅在 x 的行图块内减少。在写入此张量之前，我们在当前行图块内进行 reduce 。在内核之外，我们减少∇使用 torch.sum 来汇总每行图块的结果。决赛

Autograd 的一部分。然后函数相对简单:

```
类 WeightedSumFunc（torch.autograd.Function）:
@staticmethod
def forward（ctx，x，权重）:
# ...（前面定义）

    @staticmethod
    def backward（ctx， grad_out）:
        x，权重 = ctx.saved_tensors
        ROWS_TILE_SIZE，D_TILE_SIZE = ctx。ROWS_TILE_SIZE，ctx。D_TILE_SIZE # 这些不必相同
        n_rows，D = x.shape

        # 我们的策略是让每个线程块首先写入一个部分缓冲区,
        # 然后我们减少这个缓冲区以获得最终的梯度。
        partial_grad_weight = torch.empty（(cdiv（n_rows， ROWS_TILE_SIZE）， D）， device=x.device， dtype=x.dtype）
        grad_x = torch.empty_like（x）

        weighted_sum_backward[(cdiv（n_rows， ROWS_TILE_SIZE), )]（
            x，权重,
            grad_out,
            grad_x、partial_grad_weight、
x.步幅（0）， x.步幅（1）,
weight.stride（0）,
grad_out.stride（0）,
grad_x.stride（0）， grad_x.stride（1）,
partial_grad_weight.stride（0）， partial_grad_weight.stride（1）,
NUM_ROWS=n_rows，D=D,
ROWS_TILE_SIZE=ROWS_TILE_SIZE，D_TILE_SIZE=D_TILE_SIZE,
）
grad_weight = partial_grad_weight.sum（轴=0）
返回 grad_x，grad_weight
```

最后，我们现在可以获得一个与 torch.nn.functional 中实现的函数非常相似的函数:

```
f_weightedsum = WeightedSumFunc.apply
```

现在，在两个 PyTorch 张量 x 和 w 上调用 f_weightedsum 将给出如下所示的张量:

```
张量（[ 90.8563， -93.6815， -80.8884， ...， 103.4840， -21.4634， -24.0192],
        设备='cuda：0'， grad_fn=）
```

请注意附加到张量的 grad_fn——这表明当该张量出现在计算图中时，PyTorch 知道在向后传递中调用什么。这样就完成了我们对加权和运算的 Triton 实现。

---

[1] 或者，当然，我们可以为此编写自己的内核。

1.3.2 FlashAttention-2 前传

您将在 FlashAttention-2 之后用显着改进的 Triton 实现替换您的 PyTorch 注意力实现 [Dao， 2023]。
FlashAttention-2 采用了一些技巧来计算图块中的前向传递，这允许高效的内存访问模式，并避免在全局内存上实现完整的注意力矩阵。

在进入本节之前，我们强烈建议您至少阅读原始的 FlashAttention 论文 [Dao et al.， 2022]，这将使您直观地了解使用 FlashAttention 实现高效注意力的核心技术：以跨图块的在线方式计算 softmax（[Milakov 和 Gimelshein, 2018] 中提出的技术）。我们还建议查看 He [2022]，以了解更多关于 GPU 如何实际执行 PyTorch 代码的直觉。

了解普通注意力的低效率。回想一下，注意力的前向传递（暂时忽略掩码）可以写成：

$$S = QK/\sqrt{d} \quad (4)$$

P= 软最大值（S）（5）

O = 光伏 （6）

标准的后向传递是

$$dV = PdO \quad (7) \quad dP = dOV \quad (8)$$

dS= dsoftmax（dP） = $\big($诊断（P）− PP$\big)$ dP（9）

$$dQ = dSK/\sqrt{d} \quad (10)$$

$$dK = dSQ/\sqrt{d}, \quad (11)$$

正如我们所看到的，向后传递取决于前向传递的一些非常大的激活。例如，计算 （7） 中的 dV 需要 P，即形状（batch_size、n_heads、seq_len、seq_len）的注意力分数——这个激活矩阵的大小与序列长度成二次方有关，这解释了我们在上面对大序列长度的注意力进行基准测试时遇到的内存问题。在原版注意力的向前和向后传递期间，我们支付了大量的内存 IO 成本，以在片上 SRAM 和 GPU HBM 之间传输 P 和其他大型激活。在标准实现中进行了几种这样的传输：例如，标准向后传递实现将在 （7） 和 （9） 的计算中从 HBM 读取 P。

FlashAttention 的主要目标是避免向 HBM 读取和写入注意力矩阵，以降低 IO 和峰值内存成本。我们使用三种技术来实现这一点：平铺、重新计算和运算符融合。

铺瓷砖。为了避免在 HBM 之间读取和写入注意力矩阵，我们在不访问整个输入的情况下计算 softmax 减少。具体来说，我们重构注意力计算，将输入拆分为图块，并在输入图块上进行多次传递，从而逐步执行 softmax 减少。

重新计算。我们避免在 HBM 中存储形状（batch_size、n_头、seq_len、seq_len）的大型中间注意力矩阵。相反，我们将在 HBM 中保存某些“激活检查点”，然后在向后传递期间重新计算部分前向传递，以获得计算梯度所需的其他激活。FlashAttention-2 还存储注意力分数 L 的 logsumexp，这将是

这将用于简化后向传递计算。L 的表达式为：

$$L = \text{对数} \sum_{j} \text{经验值}(S) \quad (12)$$

在我们的最终内核中，我们将以在线方式计算这一点，但最终结果应该是相同的。通过平铺和重新计算，我们的内存 IO 和峰值使用不再依赖于 sequence_length，因此我们可能会使用更大的序列长度。

操作员融合。最后，我们通过在单个内核中执行所有作来避免注意力矩阵和其他中间激活的重复内存 IO，这称为运算符或内核融合。我们将为前向传递编写一个 Triton 内核，该内核执行 HBM 和 SRAM 之间有限的数据传输中涉及的所有作。运算符融合部分是通过重新计算实现的，因为我们可以避免通常的内存 IO，我们将每个中间激活存储到 HBM。

有关这些技术的更多直觉，请查看 FlashAttention 论文 [Dao 等人，2022 年，Dao，2023 年]。

通过重新计算进行后向传递。使用 L，我们可以进行适当的重新计算并有效地计算向后传递。在开始向后传递之前，我们将值 D = rowsum（O ∘ dO）（其中 ∘ 是元素乘法）预先计算到全局内存中，该值等于 rowsum（P ∘ dP），因为 PdP= P（dOV）= （PV）dO= OdO（和 rowsum（A ∘ B） = diag（AB） 对于任何矩阵 A 和 B）。使用 L 和 D 向量，可以在没有 softmax 的情况下计算向后传递。现在，后向传递的完整计算如下：

$$S = QK/\sqrt{d} \quad (13)$$

$$P = \exp(S - L) \quad (14)$$

$$dV = PdO \quad (15) \quad dP = dOV \quad (16) \quad dS = P \circ (dP - D) \quad (17) \quad dQ = dSK/\sqrt{d} \quad (18) \quad dK = dSQ/\sqrt{d}, \quad (19)$$

我们可以看到，作序列不需要我们在前向传递期间将注意力分数 P 存储在 HBM 中——我们从 （13） 和 （14） 中的激活 Q、K 和 L 重新计算它们。

闪光注意力前传的细节。现在我们对 FlashAttention-2 中使用的技术有了高层次的了解，我们将深入了解您将实现的 FA2 前向传递内核的详细信息。为了避免在 HBM 之间读取和写入注意力矩阵，我们希望使用平铺，即独立于其他图块计算输出的每个图块。这要求我们能够计算 P 的图块，最好在两个维度上平铺（用于查询和键）。

然而，当我们将 softmax 应用于 S 时，我们需要减少整行 S 来计算 softmax 分母，这意味着我们不能直接计算图块中的 P。FlashAttention-2 使用在线 softmax 解决了这个问题。在下面的文本中，我们将使用下标索引 i 来表示当前查询瓦片，并使用上标索引 （j） 来表示当前键瓦片。查询维度上的磁贴大小为 B，键维度为 B。我们不会沿着隐藏维度 d 平铺。

我们还保留一些按行运行的值，m∈ Rand li ∈ R。按行的 mvalue 是一个运行最大值，它被跟踪，以便我们可以以数值稳定的方式计算 softmax（回想一下我们在赋值 1 中的 softmax 实现中的这个技巧）。我们将更新 mS 的每个新行图块（当 j 增加时）。使用运行最大值，我们可以计算非规范化软最大值

我们可以将非规范化的运行代理 softmax(S)来分母使用非规范化的 P= exp softmax 值进行更新，即 l= exp（mi）。当我们最终写入输出时，我们需要使用 li 完成它的归一化，这是 l 在处理所有关键瓦片后的最终值。

算法 1 显示了应在 GPU 上实现的前向传递。

```
算法 1 FlashAttention-2 前向传递 需要：Q ∈ R、K、V ∈
R，图块大小 B、B 将 Q 拆分为 T= ⌈N/B⌉ 瓷砖 Q，. . . ，Q 尺寸 B× d
  将 K、V 拆分为 T= ⌈注 B⌉ 瓷砖 K，. . . . ，Kand V，. . . ，Vof 尺寸 B× d
  对于 i = 1，. . . ，Tdo
    加载 Q 从全局内存 初始化 O= 0 ∈ R，l= 0 ∈ R，m= −∞ ∈ R for
    j = 1，. . . ，Tdo 加载 K，V 从全局内存 计算前软最大注意力分数
    的图块 S= (j)) T √d ∈ R
      计算 mi = max ( mi，行最大 ( S )) ∈ R
      计算˜ P= exp ( Si − mi ) ∈ R
      计算 l= exp ( 米 - 米 ) li + 行和 ( 圆周率 ) ∈ R
      计算 O= diag ( 经验值 ( 米 - 米 )) Oi +˜ PV
    结束
    计算 O= diag ( 李 ) 大井
    计算 L= mi + log ( 李 )
  将 Oto 全局内存写入 O 的第 i 个图块。将 Lto 全局内存
  写入 L 的第 i 个图块。
```

在开始在 Triton 中实现前向传递之前，我们在这里收集了一些编写 Triton 内核的一般提示和技巧。

海卫一提示和技巧

- 您可以在 Triton 中使用 print 语句和 tl.device_print 进行调试：https://triton-lang。org/main/python-api/generated/triton.language.device_print.html。有一个设置 TRITON_INTERPRET=1 可以在 CPU 上运行 Triton 解释器，尽管我们发现它有错误。
- 定义块指针时，请确保它们具有正确的偏移量，并且块偏移量乘以适当的图块大小。
- 螺纹块的启动栅格设置为 kernel_fn[(launch_grid_d1， launch_grid_d2， ...)](...参数......

在 torch.autograd.Function 子类的方法中，正如我们在加权和示例中看到的那样。

- 使用 tl.dot 执行矩阵乘法。
- 要前进块指针，请使用 *_block_ptr = *_block_ptr.advance（...）

问题（flash_forward）：15 分

（a） 编写一个纯 PyTorch（无 Triton）autograd。实现 FlashAttention-2 前向传递的函数。这将比常规的 PyTorch 实现慢得多，但会帮助你调试 Triton 内核。

您的实现应采用输入 Q、K 和 V 以及标志 is_causal 并生成输出 O 和 logsumexp 值 L。您可以忽略此任务的 is_causal 标志。自动毕业。然后，函数 forward 应使用 save L、Q、K、V、O 进行向后传递并返回 O。请记住，autograd 的 forward 方法的实现。函数始终将上下文作为其第一个参数。任何自动毕业。函数类需要实现一个向后方法，但现在你可以让它只引发 NotImplementedError。如果您需要比较的东西，您可以在 PyTorch 中实现公式 4 到 6 和 12 并比较您的输出。然后接口为 def forward（ctx， Q， K， V， is_causal=False）。确定您自己的瓷砖尺寸，但要确保它们的尺寸至少为 16 × 16。我们将始终使用 2 和至少 16 的干净幂维度测试您的代码，因此您无需担心越界访问。可交付成果：一个 torch.autograd.Function 子类，用于在前向传递中实现 FlashAttention-2。若要测试代码，请实现 [adapters.get_flashattention_autograd_function_pytorch]。然后，使用 uv run pytest -k test_flash_forward_pass_pytorch 运行测试，并确保您的实现通过它。

（b） 按照算法 1 为 FlashAttention-2 的前向传递编写一个 Triton 内核。然后，编写另一个 torch.autograd.Function 的子类，在前向传递，而不是在 PyTorch 中计算结果。一些针对特定问题的提示：

- 要调试，我们建议将您执行的每个 Triton 作的结果与您在第 （a） 部分编写的平铺 PyTorch 实现进行比较。 · 您的启动网格应设置为 （T， batch_size），这意味着每个 Triton 程序实例将仅加载单个批处理索引中的元素，并且仅读取/写入 Q、O 和 L 的单个查询图块。

- 内核应该只有一个循环，它将迭代键瓦片 1 ⩽ j ⩽ T。
- 在循环末尾前进块指针。
- 使用下面的函数声明（使用我们为您提供的块指针，您应该能够推断出其余指针的设置）：

```
@triton.jit
def flash_fwd_kernel（
Q_ptr、K_ptr、V_ptr、
O_ptr、L_ptr、
stride_qb、stride_qq、stride_qd、
stride_kb、stride_kk、stride_kd、
stride_vb、stride_vk、stride_vd、
stride_ob、stride_oq、stride_od
stride_lb、stride_lq、
```

```
        N_QUERIES、N_KEYS、
        规模
        D：tl.constexpr,
        Q_TILE_SIZE：tl.constexpr,
        K_TILE_SIZE：tl.constexpr,
    ):
        # 程序索引
query_tile_index = tl.program_id（0）
batch_index = tl.program_id（1）

# 用对应的批处理索引偏移每个指针
# 乘以每个张量的批量步幅
Q_block_ptr = tl.make_block_ptr（
Q_ptr + batch_index * stride_qb,
形状=（N_QUERIES， D），
步=（stride_qq，stride_qd），
偏移量=（query_tile_index * Q_TILE_SIZE， 0），
block_shape=（Q_TILE_SIZE， D），
订单=（1， 0），
)

...其中 scale 是，Q_TILE_SIZE 和 K_TILE_SIZE 分别是 Band B。您可以稍后调整这些内容。
```

这些附加指南可以帮助您避免精度问题：

- 片上缓冲器（O、l、m）应具有 dtype tl.float32。如果要累积到输出缓冲区中，请使用 acc 参数（acc = tl.dot（...， acc=acc））。
- 将 P 转换为 V 的 dtype，然后再将它们相乘，并将 O 转换为适当的 dtype，然后再将其写入全局内存。铸造是用 tensor.to 完成的。您可以使用 tensor.dtype 获取张量的 dtype，使用 *_block_ptr.type.element_ty 获取块指针/指针的 dtype。

可交付成果：一个 torch.autograd.Function 子类，它使用 Triton 内核在前向传递中实现 FlashAttention-2。实现 [adapters.get_flash_autograd_function_triton]。然后，使用 uv run pytest -k test_flash_forward_pass_triton 运行测试，并确保您的实现通过它。

（c） 添加一个标志作为 autograd 的最后一个参数。因果掩蔽的函数实现。这应该是一个布尔标志，当设置为 True 时，将启用因果掩码的索引比较。你的 Triton 内核应该有一个相应的附加参数 is_causal：tl.constexpr（这是必需的类型注释）。在 Triton 中，为查询和键构造适当的索引向量，并将它们进行比较以形成一个大小为 B× B 的方形掩码。对于被屏蔽的元素，将常量值 -1e6 添加到注意力分数矩阵 S 的相应元素中。确保使用 ctx.is_causal = is_causal 保存向后保存掩码标志。

可交付成果：torch.autograd.Function 子类的附加标志，它使用 Triton 内核实现具有因果掩码的 FlashAttention-2 前向传递。确保该标志是可选的，默认为 False，以便以前的测试仍通过。

通过重新计算实现向后传递请注意，与公式 7 到 11 中的标准向后传递不同，我们可以使用重新计算来避免公式 13 到 19 所示的向后传递中的 softmax 运算。这意味着我们可以使用一个简单的内核来计算向后传递，并且不需要在线技巧。因此，对于这部分，您可以通过在常规

PyTorch 函数（不是 Triton）。

问题 （flash_backward）：5 分

为您的 FlashAttention-2 autograd 实现向后传递。使用 PyTorch（不是 Triton）和 torch.compile 进行函数。您的实现应将 Q、K、V、O、dO 和 L 张量作为输出，并返回 dQ、dK 和 dV。请记住计算和使用 D 向量。您可以按照公式 13 到 19 的计算进行作。

可交付成果：要测试您的实现，请运行 uv run pytest -k test_flash_backward。

现在让我们比较一下 FlashAttention-2 的（部分）Triton 实现与常规 Attention 的 PyTorch 实现的性能。

问题（flash_benchmarking）：5 分

（a） 使用 triton.testing.do_bench 编写一个基准测试脚本，将 FlashAttention-2 前向和后向传递的（部分）Triton 实现与常规 PyTorch 实现（即不使用 FlashAttention）的性能进行比较。

具体来说，您将报告一个表，其中包含 Triton 和 PyTorch 实现的前向、后向和端到端前向后传递的延迟。在开始基准测试之前随机生成任何必要的输入，并在单个 H100 上运行基准测试。始终使用批次大小 1 和因果掩码。扫描从 128 到 65536 的各种幂 2 的序列长度的笛卡尔积，嵌入从 16 到大小 128 的各种幂的 2 的维度大小，以及 torch.bfloat16 和 torch.float32 的精度。您可能需要根据输入大小调整磁贴大小。

可交付成果：使用上述设置比较 FlashAttention-2 实现与 PyTorch 实现的结果表，并报告向前、向后和端到端延迟。

1.3.3 FlashAttention-2 排行榜

作业 2 的排行榜将测试您实现 FlashAttention-2 的速度（包括向前和向后传递）。我们挑战您使用您能想到的任何技巧来进一步提高实施性能。限制是你不能改变函数的输入/输出，并且你必须使用 Triton（不幸的是没有 CUDA）。您的输入将在 BF16 上进行因果掩码测试，并且它必须通过与常规实现相同的测试。实现也必须是您自己的，并且不能使用预先存在的实现。您的计时应在 H100 上测量，样本的批量大小为 1，序列长度为 16,384，查询、键和值，并且 d= 1024 具有 16 个头。我们将验证前 5-10 名提交的正确性和性能。我们将运行的测试来计时您的实现如下：

```
def test_timing_flash_forward_backward（）：
n_heads = 16
    d_head = 64
sequence_length = 16384 q,　k,
v = torch.randn（
        3,　n_heads,　sequence_length,　d_head,　device='cuda',　dtype=torch.bfloat16,　requires_grad=True
)

    flash = torch.compile（FlashAttention2.apply）
```

```
def flash_forward_backward（）：应用）
o = 闪光（q，k，v，真）
损失 = o.sum（）
损失.backward（）

个结果 = triton.testing.do_bench（flash_forward_backward， rep=10000， warmup=1000）
打印（结果）
```

出于测试目的，您可以将重复和预热时间（以毫秒为单位）减少到更短的时间。

一些改进的想法：

- 调整内核的图块大小（为此使用 Triton 自动调整！
- 调整其他 Triton 配置参数
- 在 Triton 中实现反向传递，而不仅仅是 torch.compile（参见下面的第 1.3.4 节）
- 对输入进行两次反向传递，一次用于 dQ，另一次用于 dK 和 dV，以避免块之间的原子或同步。
- 在执行因果掩码时提前停止程序实例，跳过所有始终为零的图块
- 将未遮罩的瓦片与瓦片对角线分开，计算第一个不比较索引，第二个使用单次比较
- 在 H100 上使用 TMA（张量内存加速器）功能，遵循与本教程类似的模式。

将您的最佳成绩提交到排行榜，网址为

github.com/stanford-cs336/assignment2-systems-leaderboard

1.3.4 可选：Triton 后传

如果您有兴趣对 Triton 进行更多练习和/或快速提交排行榜，我们提供了平铺的 FlashAttention-2 向后传递，您可以在下面在 Triton 中实现。算法 2 显示了 FlashAttention-2 向后传递，因为它应该在 Triton 中实现。这里的一个关键技巧是计算两次 P，一次用于 dQ 的向后传递，另一次用于 dK 和 dV。这让我们可以跳过跨线程块的同步。

算法 2 Tiled FlashAttention-2 backward pass 需要：Q、O、dO ∈ R、K、V

```
∈ R、L ∈ R、图块大小 B、B 计算 D = rowsum（dO ∘ O） ∈ R 将 Q、O、
dO 拆分为 T= ⌈N/B⌉ 瓷砖 Q， . . . ， Q， O， . . . ， O， dO， . . . ， dO，每个尺寸为 B× d
  图块 K、V 拆分为 T= ⌈注 B⌉ Kand V， . . . ， V，每个大小为 B× d 将 L、D 拆分为 Ttiles L， . . . ， Land
  D， . . . ， D，每个大小为 B for j = 1， . . . ， Tdo 从全局内存加载 K， V 初始化 dK= dV= 0 ∈
  R
    对于 i = 1， . . . ， Tdo
      从全局内存加载 Q、O、dO、dQ 计算注意力分数
      的图块 S= (j) ) T · · √d ∈ R
      计算注意力概率 P= exp ( S− 升 ) ∈ R
      计算 dV+= （P）dO∈ R 计算 dP= dOV∈ R
      计算 dS= Pi ∘ ( dP− D ) / √d ∈ R
      从全局内存加载 dQ，然后更新 dQ+= dSi K∈R，并写回全局内存。必须是原子的，才能正确！计算 dK+=
      （dS）Q∈ R。
    结束
  将 dKand dVto 写入全局内存作为 dK 和 dV 的第 j 个图块。返回 dQ、dK、dV 结
  束。
```

2 分布式数据并行训练

在作业的下一部分中，我们将探索使用多个 GPU 来训练语言模型的方法，重点关注数据并行性。我们将从 PyTorch 中的分布式通信入门开始。然后，我们将研究分布式数据并行训练的朴素实现，然后实施和基准测试各种改进以提高通信效率。

2.1 PyTorch 中的单节点分布式通信

让我们首先看一下 PyTorch 中的一个简单的分布式应用程序，其目标是生成四个随机整数张量并计算它们的总和。

在下面的分布式案例中，我们将生成四个工作进程，每个进程生成一个随机整数张量。为了对工作进程中的这些张量求和，我们将调用全缩减集体通信作，该作将每个进程上的原始数据张量替换为全缩减结果（即总和）。

现在让我们看一些代码。

```
导入作系统
进口火炬
将 torch.distributed 导入为 dist
将 torch.multiprocessing 导入为 mp

def setup（rank， world_size）：
    os.environ[“MASTER_ADDR”] = “本地主机”
os.environ[“MASTER_PORT”] = “29500”
dist.init_process_group（“gloo”，rank=rank， world_size=world_size）

def distributed_demo（等级，world_size）：
设置（等级，world_size）
data = torch.randint（0， 10， （3,)）
print（f“rank {rank} data（before all-reduce）： {data}”）
    dist.all_reduce（data， async_op=False）
    print（f“rank {rank} data （after all-reduce）： {data}”）

如果 __name__ == “__main__”:
world_size = 4
    mp.spawn（fn=distributed_demo， args=（world_size， ）， nprocs=world_size， join=True）
```

运行上面的脚本后，我们得到下面的输出。正如预期的那样，每个工作进程最初都持有不同的数据张量。在对所有工作进程的张量求和的 all-reduce 作之后，每个工作进程上的数据都会就地修改，以保存全部缩减的结果。

```
$ uv run python distributed_hello_world.py
Rank 3 数据（全减前）：Tensor（[3， 7， 8]）
rank 0 数据（all-reduce 之前）：tensor（[4， 4， 7]）
排名 2 数据（全缩前）：张量（[6， 0， 7]）
排名 1 数据（全缩前）：张量（[9， 5， 3]）
排名 1 数据（全部减少后）：Tensor（[22， 16， 25]）
排名 0 数据（全部减少后）：Tensor（[22， 16， 25]）
```

---

[2] 如果多次运行此脚本，您会注意到打印输出的顺序不是确定性的。由于此应用程序在分布式设置中运行，因此我们无法控制命令的确切运行顺序——我们唯一的保证是，在全缩减作完成后，单独的进程将保存按位相同的结果张量。

```
排名 3 数据（全缩后）：Tensor（[22， 16， 25]）
排名 2 数据（全部减少后）：Tensor（[22， 16， 25]）
```

现在让我们更仔细地回顾一下上面的脚本。命令 mp.spawn 生成使用提供的参数运行 fn 的 nprocs 进程。此外，函数 fn 称为 fn（rank， *args），其中 rank 是工作进程的索引（介于 0 和 nprocs-1 之间的值）。因此，我们的 distributed_demo 函数必须接受这个整数秩作为其第一个位置参数。此外，我们还传入 world_size，它指的是工作进程的总数。

这些工作进程中的每一个都属于一个进程组，该进程组通过 dist.init_process_group 初始化。进程组表示多个工作进程，这些进程将通过共享主进程进行协调和通信。主站由其 IP 地址和端口定义，主站以 0 级运行进程。

集体通信作（如 all-reduce）对流程组中的每个流程进行作。

在这种情况下，我们使用“gloo”后端初始化了我们的进程组，但其他后端也可用。特别是，“nccl”后端将使用 NVIDIA NCCL 集体通信库，这通常对于 CUDA 张量的性能会更高。但是，NCCL 只能在配备 GPU 的机器上使用，而 Gloo 可以在仅 CPU 的机器上运行。一个有用的经验法则是使用 NCCL 进行分布式 GPU 训练，使用 Gloo 进行分布式 CPU 训练和/或本地开发。我们在此示例中使用了 Gloo，因为它支持在纯 CPU 机器上进行本地执行和开发。

运行多 GPU 作业时，请确保不同的等级使用不同的 GPU。一种方法是在设置函数中调用 torch.cuda.set_device（rank）），以便 tensor.to（“cuda”） 自动将其移动到指定的设备。或者，您可以显式创建每个等级的设备字符串（例如，device = f“cuda：{rank}”），然后将此设备字符串用作任何数据移动的目标设备（例如，tensor.to（f“cuda：{rank}”））。

术语。在作业的其余部分（以及您可能在网上看到的各种其他资源）中，您可能会在 PyTorch 分布式通信的上下文中遇到以下术语。尽管我们将在本作业中重点介绍单节点、多进程分布式训练，但该术语对于理解一般分布式训练很有用。有关视觉表示，请参见图 2。

node：网络上的计算机。

worker：参与分布式训练的程序实例。在此分配中，每个辅助角色将有一个流程，因此我们将互换使用辅助角色、流程和辅助角色流程。

然而，一个工作进程可能会使用多个进程（例如，加载数据进行训练），因此这些术语在实践中并不总是等效的。

world size：流程组中的工作线程总数。

全局排名：一个整数 ID（介于 0 和 world_size-1 之间），用于唯一标识流程中的辅助角色群。例如，对于世界大小为 2，一个进程将具有全局排名 0（主进程），另一个进程将具有等级 1。

local world size：跨不同节点运行应用时，local world size 是在给定节点上本地运行的 work。例如，如果我们有一个应用程序，每个应用程序在 2 个节点上生成 4 个工作程序，则世界大小将为 8，本地世界大小将为 4。请注意，在单个节点上运行时，worker 的本地世界大小等同于（全局）世界大小。

local rank：一个整数 ID（介于 0 和 local_world_size-1之间），用于唯一标识计算机上的本地工作人员。例如，如果我们有一个应用程序，每个应用程序在 2 个节点上生成 4 个进程，则每个节点将具有本地等级为 0、1、2 和 3 的工作线程。注意，在运行单节点多进程分布式应用时，进程的本地秩等同于其全局秩。

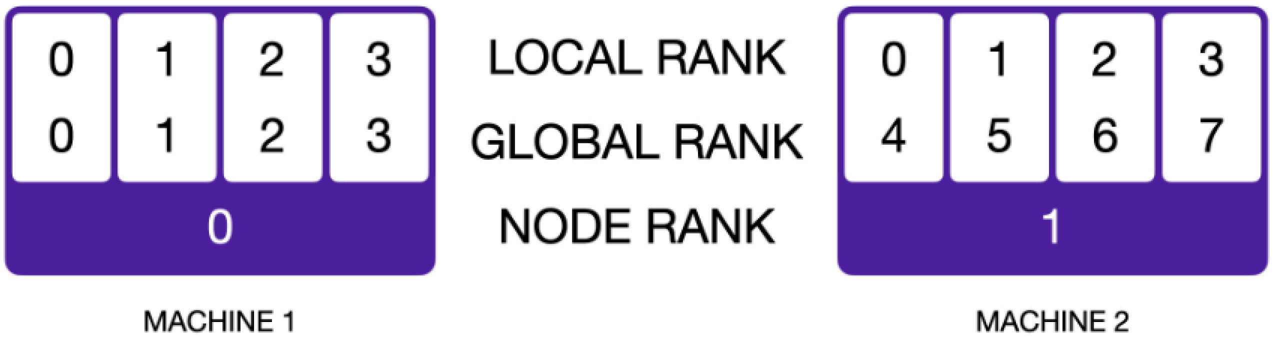

图 2：在 2 个节点上运行的分布式应用程序的示意图，世界大小为 8。每个工作进程都由全局排名（从 0 到 7）和本地排名（从 0 到 3）标识。图取自 lightning.ai/docs/fabric/stable/advanced/distributed_communication.html

2.1.1 对分布式应用程序进行基准测试的最佳实践

在作业的这一部分中，您将对分布式应用程序进行基准测试，以更好地了解通信的开销。以下是一些最佳实践：

- 只要有可能，就在同一台机器上运行基准测试，以促进受控比较。
- 在计时感兴趣的作之前执行几个预热步骤。这对于 NCCL 通信呼叫尤其重要。5 次迭代的预热一般就足够了。
- 在 GPU 上进行基准测试时，调用 torch.cuda.synchronize（） 等待 CUDA 作完成。请注意，即使使用 async_op=False 调用通信作，这也是必要的，当作在 GPU 上排队时返回（而不是当通信实际完成时）。
- 不同等级的时间可能略有不同，因此通常会汇总跨等级的测量值以改进估计值。您可能会发现 all-gather 集体（特别是 dist.all_gather_ 对象函数）对于收集所有等级的结果很有用。
- 通常，在 CPU 上使用 Gloo 进行本地调试，然后根据给定问题的需要，在 GPU 上使用 NCCL 进行基准测试。在后端之间切换仅涉及更改 init_process_group 调用和张量设备强制转换。

问题（distributed_communication_single_node）：5 分

编写一个脚本来对单节点多进程设置中全缩减作的运行时进行基准测试。上面的示例代码可以提供一个合理的起点。尝试改变以下设置：

后端 + 设备类型：Gloo + CPU、NCCL + GPU。

全部减少数据大小：float32 数据张量范围超过 1MB、10MB、100MB、1GB。

进程数：2、4 或 6 个进程。

资源要求：最多 6 个 GPU。每次基准测试运行应不到 5 分钟。

---

[3] 有关更多详细信息，请参阅 github.com/pytorch/pytorch/issues/68112#issuecomment-965932386。

可交付成果：比较各种设置的情节和/或表格，并附有 2-3 句关于您的结果的评论以及关于各种因素如何相互作用的想法。

2.2 分布式数据并行训练的朴素实现

现在我们已经了解了在 PyTorch 中编写分布式应用程序的基础知识，让我们构建分布式数据并行 （DDP） 训练的最小实现。

数据并行性将批次拆分为多个设备（例如 GPU），从而能够对不适合单个设备的大批量进行训练。例如，给定四个设备，每个设备可以处理最大批量大小 32，数据并行训练将使有效批量大小为 128。

以下是天真地进行分布式数据并行训练的步骤。最初，每个设备构建一个（随机初始化的）模型。我们使用广播集体通信作将模型参数从秩 0 发送到所有其他秩。在训练开始时，每个设备都保存模型参数和优化器状态的相同副本（例如，Adam 中的累积梯度统计量）。

1. 给定一个有 n 个示例的批次，该批次被分片，每个设备接收 n/d 个不相交的示例（其中 d 是用于数据并行训练的设备数量）。n 应除以 d，因为训练时间受到最慢过程的瓶颈。
2. 每个设备都使用其模型参数的本地副本在其 n/d 示例上运行前向传递，并运行后向传递来计算梯度。请注意，此时，每个设备都保存从它收到的 n/d 示例计算的梯度。
3. 然后，我们使用全缩减集体通信作对不同设备的梯度进行平均，因此每个设备都保存所有 n 个示例的平均梯度。
4. 接下来，每个设备都运行一个优化器步骤来更新其参数副本——从优化器的角度来看，它只是在优化本地模型。参数和优化器状态将在所有不同的设备上保持同步，因为它们都从相同的初始模型和优化器状态开始，并且每次迭代都使用相同的平均梯度。此时，我们已经完成了一次训练迭代，可以重复该过程。

问题（naive_ddp）：5 分

可交付成果：编写一个脚本，通过在向后传递后全减单个参数梯度来天真地执行分布式数据并行训练。要验证 DDP 实现的正确性，请使用它在随机生成的数据上训练小型玩具模型，并验证其权重是否与单进程训练的结果相匹配。

---

[a] 如果您在编写此测试时遇到问题，查看测试/test_ddp_individual_parameters.py 可能会有所帮助

问题（naive_ddp_benchmarking）：3 分

在这个朴素的 DDP 实现中，参数在每次向后传递后都会在各个等级之间单独全部减少。为了更好地了解数据并行训练的开销，请创建一个脚本，以便在使用这种朴素的 DDP 实现进行训练时对以前实现的语言模型进行基准测试。测量每个训练步骤的总时间以及通信梯度所花费的时间比例。在单节点设置（1 节点 x 2 GPU）中收集 §1.1.2 中描述的 XL 模型大小的测量值。

可交付成果：基准测试设置的描述，以及每次训练的测量时间

迭代和为每个设置传达梯度所花费的时间。

2.3 改进最小 DDP 实现

我们在 §2.2 中看到的最小 DDP 实现有几个关键限制：

1. 它对每个参数张量进行单独的全缩减作。每个通信调用都会产生开销，因此批量通信调用以最大程度地减少此开销可能是有利的。
2. 它等待向后传递完成，然后再通信梯度。但是，向后传递是增量计算的。因此，当参数梯度准备就绪时，可以立即对其进行通信，而无需等待其他参数的梯度。这使我们能够将梯度的通信与反向传递的计算重叠，从而减少分布式数据并行训练的开销。

在作业的这一部分中，我们将依次解决这些限制中的每一个，并衡量对训练速度的影响。

2.3.1 减少通信呼叫次数

与其为每个参数张量发出通信调用，不如看看我们是否可以通过批处理 all-reduce 来提高性能。具体来说，我们将采用我们想要全部减少的梯度，将它们连接成一个张量，然后对所有等级的组合梯度进行全部减少。使用 torch._utils._flatten_dense_tensors 和 torch._utils._unflatten_dense_tensors 可能会有所帮助。

问题（minimal_ddp_flat_benchmarking）：2 分

修改您的最小 DDP 实现，以使用来自所有参数的扁平梯度来通信张量。将其性能与最小 DDP 实现进行比较，该实现在以前使用的条件下为每个参数张量发出 allreduce（1 个节点 x 2 个 GPU，XL 模型大小，如 §1.1.2 中所述）。

可交付成果：每次训练迭代的测量时间以及在分布式数据并行训练下使用单个批处理全减少调用通信梯度所花费的时间。1-2 句话比较批处理与单独交流梯度时的结果。

2.3.2 与单个参数梯度通信的重叠计算

虽然批处理通信调用可能有助于降低与发出大量小型全缩减作相关的开销，但所有通信时间仍会直接增加开销。为了解决这个问题，我们可以利用反向传递以增量方式计算每一层的梯度（从损失开始并向输入移动）的观察结果，因此，我们可以在参数梯度准备就绪后立即全部减少参数梯度，通过将向后传递的计算与梯度通信重叠来减少数据并行训练的开销。

我们将首先实现一个分布式数据并行包装器并对其进行基准测试，该包装器在向后传递期间准备就绪时异步全缩减单个参数张量。以下指针可能有用： 向后钩子 要在参数的梯度在向后传递中累积后自动调用函数，您可以使用 register_post_accumulate_grad_hook 函数。

---

4 有关更多信息和使用示例，请参阅 pytorch.org/docs/stable/generated/torch.Tensor.register_post_accumulate_grad_hook.html。

异步通信：所有 PyTorch 集体通信作都支持同步 （async_op=False） 和异步执行 （async_op=True）。同步调用将阻塞，直到集体作在 GPU 上排队。这并不意味着 CUDA 作已完成，因为 CUDA 作是异步的。话虽如此，以后使用输出的函数调用将按预期运行。相比之下，异步调用将返回分布式请求句柄，因此，当函数返回时，不能保证集体通信作已在 GPU 上排队，更不用说完成了。要等待作在 GPU 上排队（从而使输出在以后的作中可用），您可以在返回的通信句柄上调用 handle.wait（）。

例如，以下两个示例使用同步或异步调用对张量列表中的每个张量进行全部缩减：

```
张量 = [torch.rand（5） for _ in range（10）]

    # 同步，阻塞直到作在 GPU 上排队。
    对于张量中的张量：
            dist.all_reduce（张量，async_op=False）

    # 异步，每次调用后立即返回，并且
    # 最后等待结果。
个手柄 = []
张量中的张量为：句柄 = dist.all_reduce（张量，
async_op=True）
            handles.append（句柄）


# ...
# 可能会执行其他不依赖于 all_reduce 结果的命令 # ...


# 确保所有 reduce 调用都已排队，# 因此其他作取决于

# all-reduce 输出可以排队。用于手柄中的
手柄：
            handle.wait（）
    handles.clear（）
```

问题（ddp_overlap_individual_parameters）：5 分

实现一个 Python 类来处理分布式数据并行训练。该类应该包装一个任意的 PyTorch nn。模块化并负责在训练前广播权重（因此所有等级具有相同的初始参数）并发出梯度平均的通信调用。我们推荐以下公共接口：

def __init__（self， module： torch.nn.Module）： 给定一个实例化的 PyTorch nn.要并行化的模块，构建一个 DDP 容器，该容器将处理跨等级的梯度同步。

def forward（self， *inputs， **kwargs）：使用提供的位置和关键字参数调用包装模块的 forward（） 方法。

def finish_gradient_synchronization（self）：调用时，等待异步通信

---

[5] 在高级情况下，如果您使用多个 CUDA 流，您可能需要跨流显式同步，以确保输出为以后的作做好准备。参见 pytorch.org/docs/stable/notes/cuda.html#cuda-streams。

调用在 GPU 上排队。

要使用此类进行分布式训练，我们将向其传递一个要包装的模块，然后添加一个
在运行 optimizer.step（）之前调用 finish_gradient_synchronization（）以确保
优化器步骤，一个依赖于梯度的作，可以排队：

```
模型 = ToyModel（）.to（设备）
ddp_model = DDP（模型）

对于 _ in range（train_steps）：
    x， y = get_batch（）logits = ddp_model（x）loss =
    loss_fn（logits， y）loss.backward（）
    ddp_model.finish_gradient_synchronization（）
    optimizer.step（）
```

可交付成果：实现容器类来处理分布式数据并行训练。此类应与梯度通信和反向传递的计算重叠。要测试您的 DDP 类，请首先实现适配器 [adapters.get_ddp_individual_parameters] 和 [adapters.ddp_individual_parameters_on_after_backward]（后者是可选的，具体取决于您的实现，您可能不需要它）。

然后，要执行测试，请运行 uv run pytest tests/test_ddp_individual_parameters.py。
我们建议多次运行测试（例如 5 次），以确保它可靠地通过。

问题 （ddp_overlap_individual_parameters_benchmarking）：1 分

（a） 在将反向传递计算与单个参数梯度的通信重叠时，对 DDP 实现的性能进行基准测试。将其性能与我们之前研究的设置（最小 DDP 实现，对每个参数张量发出 all-reduce，或对所有参数张量的串联发出单个 all-reduce）进行比较，具有相同的设置：1 个节点、2 个 GPU 和 §1.1.2 中描述的 XL 模型大小。

可交付成果：当反向传递与各个参数梯度的通信重叠时，每次训练迭代的测量时间，用 1-2 个句子比较结果。

（b） 使用 Nsight 分析器检测您的基准测试代码（使用 1 节点、2 个 GPU、XL 模型大小设置），比较初始 DDP 实现与此 DDP 实现（后向计算和通信重叠）。直观地比较这两个跟踪，并提供探查器屏幕截图，演示一个实现与通信重叠的计算，而另一个则没有。

可交付成果：2 个屏幕截图（一个来自初始 DDP 实现，另一个来自计算与通信重叠的此 DDP 实现），直观地显示通信与反向传递重叠或不重叠。

2.3.3 重叠计算与桶参数梯度通信

在上一节（§2.3.2）中，我们将反向支柱计算与单个参数梯度的通信重叠。然而，我们之前观察到，批处理通信调用可以提高性能，特别是当我们有许多参数张量时（这在深度 Transformer 模型中是典型的）。我们之前的批处理尝试一次发送了所有梯度，这需要等待向后

这需要等待向后传递完成。在本节中，我们将尝试通过将参数组织到桶（减少总通信调用次数）和所有减少桶（使我们能够将通信与计算重叠）中来获得两全其美的效果。

问题（ddp_overlap_bucketed）：8 分

实现一个 Python 类来处理分布式数据并行训练，使用梯度分桶来提高通信效率。该类应包装任意输入 PyTorch nn。模块化并负责在训练前广播权重（因此所有等级具有相同的初始参数）并发出用于梯度平均的分桶通信调用。我们推荐以下界面：

def __init__（self， module： torch.nn.Module， bucket_size_mb： float）： 给定一个实例化的 PyTorch nn。要并行化的模块，构建一个 DDP 容器，该容器将处理跨等级的梯度同步。梯度同步应分桶，每个桶最多包含 bucket_size_mb 参数。

def forward（self， *inputs， **kwargs）：使用提供的位置和关键字参数调用包装模块的 forward（） 方法。

def finish_gradient_synchronization（self）：调用时，等待异步通信调用在 GPU 上排队。

除了添加 bucket_size_mb 初始化参数之外，此公共接口还与我们之前单独通信每个参数的 DDP 实现的接口相匹配。我们建议使用 model.parameters（） 的相反顺序将参数分配给存储桶，因为在向后传递期间，梯度将大致按该顺序准备就绪。

可交付成果：实现容器类来处理分布式数据并行训练。此类应与梯度通信和反向传递的计算重叠。梯度通信应分桶，以减少通信调用的总数。要测试您的实现，请完成 [adapters.get_ddp_bucketed]、[adapters.ddp_bucketed_on_after_backward] 和 [adapters.ddp_bucketed_on_train_batch_start]（后两者是可选的，具体取决于您的实现，您可能不需要它们）。

然后，要执行测试，请运行 pytest tests/test_ddp.py。我们建议多次运行测试（例如 5 次），以确保它可靠地通过。

问题（ddp_bucketed_benchmarking）：3 分

（a） 使用与之前实验相同的配置（1 个节点、2 个 GPU、XL 模型大小）对分桶 DDP 实现进行基准测试，改变最大分桶大小（1、10、100、1000 MB）。

将您的结果与之前的实验进行比较，无需分桶 - 结果是否符合您的预期？如果它们不一致，为什么不呢？您可能需要根据需要使用 PyTorch 分析器，以更好地了解通信调用的排序和/或执行方式。您希望在实验设置中进行哪些更改才能产生符合您预期的结果？

可交付成果：针对各种存储桶大小，每次训练迭代的测量时间。关于结果、您的期望以及任何不匹配的潜在原因的 3-4 句话评论。

（b） 假设计算一个桶的梯度所需的时间与传达梯度桶所需的时间相同。编写一个方程，将 DDP 的通信开销（即向后传递后花费的额外时间）建模为函数

模型参数的总大小（字节）、全缩算法带宽（W，计算方式为每个排名的数据大小除以完成全缩减所需的时间）、与每个通信调用相关的开销（秒）（O）和桶数（N）。根据此等式，编写一个最佳存储桶大小的方程，以最大限度地减少 DDP 开销。

可交付成果：对 DDP 开销进行建模的方程，以及最佳存储桶大小的方程。

2.4 4D 并行度

向后传递后花费的额外时间）作为函数虽然实现要复杂得多，但我们可以沿着更多的轴并行化我们的训练过程。最常见的是，我们讨论 5 种并行方法：

- 数据并行性（DP）— 批量数据拆分到多个设备，每个设备为自己的批次计算梯度。这些梯度必须以某种方式在设备之间进行平均。
- 全分片数据并行性（FSDP）— 优化器状态、梯度和权重在设备之间拆分。如果我们只使用 DP 和 FSDP，则每个设备都需要从所有其他设备收集权重分片，然后我们才能执行前向或后向传递。
- 张量并行性（TP）— 激活在一个新的维度上分片，每个设备都会计算自己的分片的输出结果。使用 Tensor Parallel，我们可以沿着输入或输出分片我们正在分片的作进行分片。如果我们沿相应维度对权重和激活进行分片，则张量并行性可以与 FSDP 一起有效地使用。
- 流水线并行性（PP）— 模型按层拆分为多个阶段，其中每个阶段在不同的设备上运行。
- 专家并行性（EP）— 我们将专家（在专家混合模型中）分离到不同的设备上，每个设备为自己的专家计算输出结果。

通常，我们总是将 FSDP 和 TP 结合起来，因此我们可以将它们视为并行度的单个轴。这给我们留下了 4 个并行度轴：DP、FSDP/TP、PP 和 EP。我们还将关注密集模型（不是 MoE），因此不会进一步讨论 EP。

在推理分布式训练时，我们经常将集群描述为设备网格，其中网格的轴是我们定义并行度的轴。例如，如果我们有 16 个 GPU 和一个模型，其大小远大于单个设备上的容纳能力，我们可能倾向于将网格组织成 4 × 4 个 GPU 网格，其中第一个维度代表 DP，第二个维度代表组合的 FSDP 和 TP。

有关这些方法如何工作以及如何推导出其通信和内存成本的更多详细信息，请参阅 TPU Scaling Book（Austin et al. [2025]）第 5 部分中的概述（这对于解决以下问题特别有帮助）。有关更详细的管道并行讨论，请参阅超大规模剧本附录（Nouamane Tazi [2025]）。本书的其余部分还包含许多您可能会觉得有用的其他信息。

问题（communication_accounting）：10 分

考虑一个新的模型配置 XXL，其 d_model=16384、d_ff=53248 和 num_blocks=126。因为对于非常大的模型，绝大多数 FLOP 都在前馈网络中，所以我们做了一些简化的假设。首先，我们省略注意力、输入嵌入和输出线性层。然后，我们假设每个 FFN 只是两个线性层（忽略激活函数），其中第一个具有输入大小 d_model 和输出大小 d_ff，第二个具有输入大小 d_ff 和输出大小 d_model。您的模型由这两个线性层的 num_blocks 块组成。不要做任何激活检查点，并将激活和梯度通信保持在 BF16 中，而累积的梯度、主权重和优化器状态应该在 FP32 中。

（a） 在单个设备上存储 FP32 中的主模型权重、累积梯度和优化器状态需要多少内存？为向后节省了多少内存（这些将在 BF16 中）？ 这到底是多少 H100 80GB GPU 内存？

可交付成果：您的计算和一句话的回答。

（b） 现在假设你的主权重、优化器状态、梯度和一半的激活（实际上每隔一层）在 Ndevices 之间分片。编写一个表达式，说明每个设备将占用多少内存。总内存成本低于 1 v5p TPU（每台设备 95GB）需要多少值？可交付成果：您的计算和一句话的回答。

（c） 只考虑前传。使用 W=2·9 ·10，C = 4.6 ·10，适用于 TPU 扩展手册中给出的 TPU v5p。按照缩放手册的符号，使用 M= 2，M= 1（3D 网格），其中 X = 16 是您的 FSDP 维度，Y = 4 是您的 TP 维度。此模型的每个设备计算大小是多少？此设置中的总批量大小是多少？

可交付成果：您的计算和一句话的回答。

（d） 在实践中，我们希望整体批量大小尽可能小，并且我们也始终有效地使用我们的计算（换句话说，我们希望永远不会受到通信的束缚）。我们还可以使用哪些其他技巧来减小模型的批量大小，同时保持高吞吐量？

可交付成果：一段回复。用参考文献和/或方程来支持您的主张。

3 优化器状态分片

分布式数据并行训练在概念上很简单，通常非常有效，但要求每个等级都保存模型参数和优化器状态的不同副本。这种冗余可能会带来巨大的内存成本。例如，AdamW 优化器为每个参数维护两个浮点数，这意味着它消耗的内存是模型权重的两倍。Rajbhandari 等[2020]描述了几种减少数据并行训练中这种冗余的方法，方法是跨等级划分（1）优化器状态、（2）梯度和（3）参数，并根据需要在工作线程之间进行通信。

在赋值的这一部分中，我们将通过实现优化器状态分片的简化版本来减少每个等级的内存消耗。每个排名的优化器实例不会保留所有参数的优化器状态，而是只处理参数的子集（大约 1 / world_size）。当每个排名的优化器执行优化器步骤时，它只会更新其分片中的模型参数子集。然后，每个等级都会将其更新的参数广播到其他等级，以确保模型参数在每个优化器步骤后保持同步。

问题（optimizer_state_sharding）：15 分

实现一个 Python 类来处理优化器状态分片。该类应包装任意输入 PyTorch optim。优化器，并在每个优化器步骤后负责同步更新的参数。我们推荐以下公共接口：

def __init__（self， params， optimizer_cls： Type[Optimizer]， **kwargs： Any）： 初始化分片状态优化器。params 是要优化的参数的集合（或参数组，如果用户想要对模型的不同部分使用不同的超参数，例如学习率）;这些参数将在所有等级中分片。optimizer_cls 参数指定要包装的优化器类型（例如，optim.亚当 W）。最后，任何剩余的关键字参数都将转发给 optimizer_cls 的构造函数。确保在此方法中调用 torch.optim.Optimizer 超类构造函数。

def step（self， closure， **kwargs）：使用 pro- 调用包装优化器的 step（） 方法 vided 闭包和关键字参数。更新参数后，与其他等级同步。

def add_param_group（self， param_group： dict[str， Any]）：这个方法应该添加一个参数- ter 组添加到分片优化器。这在超类构造函数构建分片优化器期间调用，也可以在训练期间调用（例如，用于逐渐解冻模型中的层）。因此，此方法应处理在等级之间分配模型的参数。

可交付成果：实现一个容器类来处理优化器状态分片。若要测试分片优化器，请先实现适配器 [adapters.get_sharded_optimizer]。然后，要执行测试，请运行 uv run pytest tests/test_sharded_optimizer.py。我们建议多次运行测试（例如 5 次），以确保它可靠地通过。

现在我们已经实现了优化器状态分片，让我们分析一下它对训练期间峰值内存使用量的影响及其运行时开销。

问题（optimizer_state_sharding_accounting）：5 分

（a） 创建一个脚本来分析使用和不使用优化器状态分片训练语言模型时的峰值内存使用量。使用标准配置（1 个节点、2 个 GPU、XL 型号大小），

报告模型初始化后、优化器步骤之前和优化器步骤之后的峰值内存使用量。结果符合您的期望吗？细分每个设置中的内存使用情况（例如，参数的内存量、优化器状态的内存量等）。

可交付成果：2-3 句话的响应，其中包含峰值内存使用结果，以及内存在不同模型和优化器组件之间的分配方式的细分。

（b）我们对优化器状态分片的实现如何影响训练速度？测量标准配置（1 个节点、2 个 GPU、XL 模型大小）的每次迭代所花费的时间，无论是否使用优化器状态分片。

可交付成果：2-3 句话的回复和您的时间安排。

（c） 我们的优化器状态分片方法与 ZeRO 第 1 阶段（描述为 ZeRODP Pin Rajbhandari 等人，2020 年）有何不同？

可交付成果：2-3 句话总结任何差异，尤其是与记忆和交流量相关的差异。

4 结语

恭喜你完成作业！我们希望您觉得它有趣且有益，并且您了解了一些关于如何通过提高单 GPU 速度和/或利用多个 GPU 来加速语言模型训练的知识。

引用

特里道。Flashattention-2：更快的注意力，更好的并行性和工作分区，2023 年。网址 https://arxiv.org/abs/2307.08691。

Tri Dao、Daniel Y Fu、Stefano Ermon、Atri Rudra 和 Christopher Re. Flashattention：具有 IO 感知的快速且内存高效的精确注意力。在 Alice H. Oh、Alekh Agarwal、Danielle Belgrave 和

Kyunghyun Cho，编辑，神经信息处理系统的进展，2022 年。网址 https:// openreview.net/forum?id=H4DqfPSibmx。

马克西姆·米拉科夫和娜塔莉亚·吉梅尔辛。softmax 的在线归一化器计算，2018 年。网址 https：//arxiv.org/abs/1805.02867。

贺拉斯·赫。让深度学习从第一性原理开始。2022. 网址 https://horace.io/brrr_ intro.html。

雅各布·奥斯汀、肖尔托·道格拉斯、罗伊·弗罗斯蒂格、安塞姆·列夫斯卡娅、查理·陈、沙拉德·维克拉姆、费德里科·勒布朗、彼得·蔡、维奈·拉马塞什、阿尔伯特·韦伯森和莱纳·波普。如何缩放模型。2025.

取自 https://jax-ml.github.io/scaling-book/。

Haojun Zhao， Phuc Nguyen， Mohamed Mekkouri， Leandro Werra， Thomas Wolf， Nouamane Tazi， Ferdinand Mom.超大规模剧本：在 GPU 集群上训练 LLM，2025 年。

萨米亚姆·拉杰班达里、杰夫·拉斯利、奥拉通吉·鲁瓦斯和何宇雄。ZeRO：针对训练万亿参数模型的内存优化，2020 年。arXiv：1910.02054。