Continuous batching

看vllm的时候注意到了continuous batching这个技术点，按图索骥找到了最初提出它的论文《**Orca: A Distributed Serving System for Transformer-Based Generative Models**》，在这篇论文里它最初被称为 **Iteration Batching**。得益于它对吞吐能力的数倍提升，现在已成为 LLM 推理框架的标配能力了。一起来回顾一下它是如何被提出，又有哪些技术细节吧！

先解释一下后面会用到的图示的意思：大模型推理过程分为两部分，如下图，其中绿色为**prompt**，蓝色为**生成的每个token**，生成过程持续直到遇到stop token或达到最长长度。

t1	t2	t3	t4	t5	t6	t7	t8
S1	S1	A1	A1	A1	A1	EOS	

传统 static batching 存在什么问题?

Static batching指的是**按 batch 处理请求**，每个 batch 同时生成下一个 token，直到所有请求完成。但这会带来两个主要问题：

1. **Early-finished（提前完成）与Late-joining（迟到加入）**

指的是某些请求可以在batch中很早就完成，但并未释放资源，以让新的请求加入到batch中。这意味着GPU未被充分利用，比如下图中，请求s1生成了4个token，s3生成了3个，s2生成了1个，s4生成了1个，seq1、2、4结束标记后的**白色方块就是GPU在空闲**，什么都没有做，此时GPU利用率非常低，新请求也无法插队进来。同时这也意味着已经生成的短文本，要等待同一batch内其他文本生成完成才能返回给用户。

t1	t2	t3	t4	t5	t6	t7	t8
S1	S1	A1	A1	A1	A1	EOS	
S2	S2	S2	A2	EOS			
S3	S3	S3	S3	A3	A3	A3	EOS
S4	S4	S4	S4	S4	A4	EOS	

2. **Batching an arbitrary set of requests（Batch 内部异构请求难以高效处理）**

另一个问题在于，如何将多个请求组合成一个 batch 进行处理，以充分利用 GPU 的并行计算能力。然而，对同一 batch 内的请求进行处理时，要求在此次迭代中执行完全相同的操作，即每个请求待处理

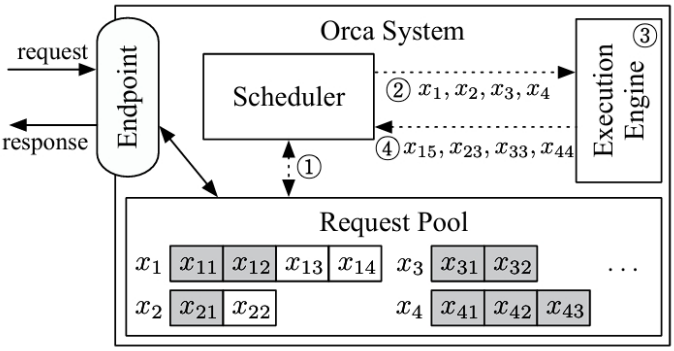
的张量形状需保持一致。尽管可以通过 padding（填充）和掩码矩阵的方式解决这一问题，但这种方式会严重浪费算力，尤其是在 Batch 内部存在异构请求的情况下，高效处理会变得更为困难。

如右图所示：

x3和x4：都在 `initiation` 阶段，但输入长度不同。可以通过补齐（padding）合批处理。

x1和x2：都在 `increment` 阶段，但生成进度不同，KV 缓存形状不一致，难以直接合批。

x1和x3：分别处于不同阶段（一个在 `initiation` 阶段，另一个在 `increment` 阶段），不能合批处理。`initiation` 阶段的所有token可以并行计算，`increment` 阶段不能并行。



> 因为是22年的文章，这里的 `initiation` 指的是“请求刚进入系统、prompt 输入的预填充阶段”；而 `increment` 表示正式进入 **decode 阶段**，也就是每次生成下一个 token 的过程。

综上所述，只有处在相同阶段、且输入或张量形状完全相同的请求才能同一个batch处理。

Continuous Batching 如何解决以上两个问题？

对于第一个问题的解决方法就是一旦一个batch中的某个请求完成生成，发射了一个end-of-seq token，就可以在其位置插入新的请求继续生成token，从而达到比static batching更高的GPU利用率。而之所以能这样实现，是因为一次推理只生成一个token，利用两次推理之间的空隙，就可以进行请求增减和显存分配。如下图所示，请求S1、S2、S4被处理完毕后，马上开始处理新的请求。

t1	t2	t3	t4	t5	t6	t7	t8
S1	S1	A1	A1	A1	A1	EOS	S6
S2	S2	S2	A2	EOS	S5	S5	S5
S3	S3	S3	S3	A3	A3	A3	EOS
S4	S4	S4	S4	S4	A4	EOS	S7

接下来根据论文中的伪代码详细讲解原理：

1-2行：`n_scheduled`：当前正进行推理但尚未返回结果的microbatch数（与流水线并行有关）

`n_rsrv`：当前尚未开始的请求在 K/V cache中预留的最大 token slots数。

4-5行：基于到达时序，从请求池中选出最多 `max_bs` 个请求，并交给推理引擎，生成每个请求的下一个token。

7行：每个加入当前batch的请求都被标记为 `RUNNING`，对应19行。

9-10行：当正在执行的microbatch数等于 `n_workers` 时，则不再调度

11-15行：每个返回结果的请求状态更新为 `INCREMENT`，如果某个请求预测是 `《EOS》`，则释放对应的KV cache slots。

18-20行：进程池中过滤掉 `RUNNING` 的请求，按照FCFS的方式排序

22行：如果达到了MicroBatch的大小，结束循环。

23-26行：针对新到的、未分配显存资源的请求，为其预留 `req.max_tokens` 个 token Slot。若分配Slot数超过最大阈值，则退出循环，否则更新已预留Slot数，并将该请求加入到batch中。

Algorithm 1: ORCA scheduling algorithm

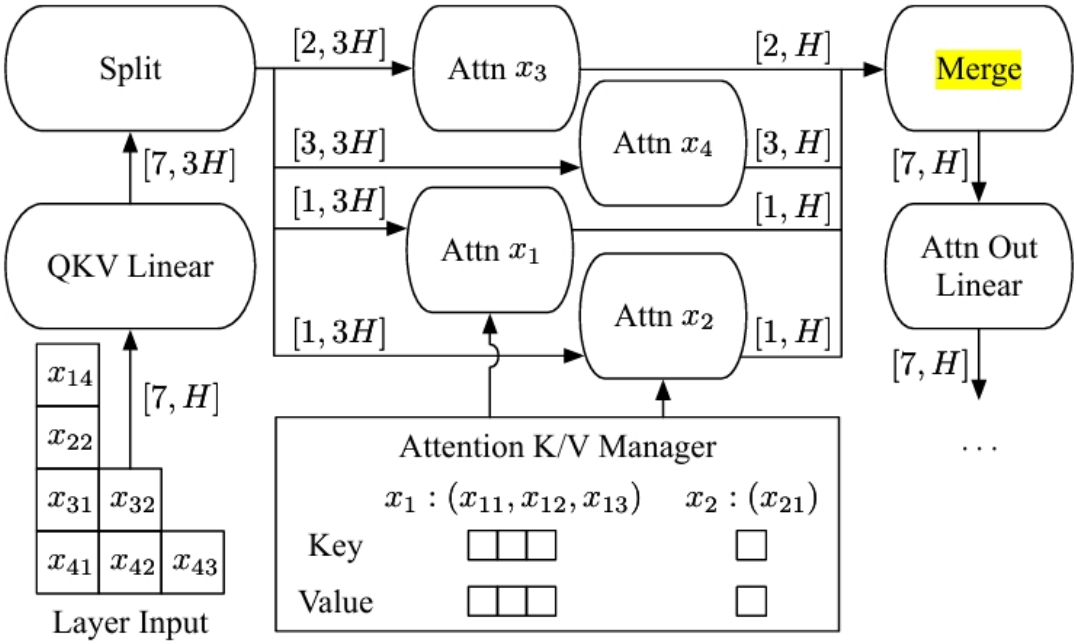
Params: *n_workers*: number of workers, *max_bs*: max batch size, *n_slots*: number of K/V slots

```
1  n_scheduled ← 0
2  n_rsrv ← 0
3  while true do
4      batch, n_rsrv ← Select(request_pool, n_rsrv)
5      schedule engine to run one iteration of
       the model for the batch
6      foreach req in batch do
7          req.state ← RUNNING
8      n_scheduled ← n_scheduled + 1
9      if n_scheduled = n_workers then
10         wait for return of a scheduled batch
11         foreach req in the returned batch do
12             req.state ← INCREMENT
13             if finished(req) then
14                 n_rsrv ← n_rsrv − req.max_tokens
15         n_scheduled ← n_scheduled − 1
16
17 def Select(pool, n_rsrv):
18     batch ← {}
19     pool ← {req ∈ pool | req.state ≠ RUNNING}
20     SortByArrivalTime(pool)
21     foreach req in pool do
22         if batch.size() = max_bs then break
23         if req.state = INITIATION then
24             new_n_rsrv ← n_rsrv + req.max_tokens
25             if new_n_rsrv > n_slots then break
26             n_rsrv ← new_n_rsrv
27         batch ← batch ∪ {req}
28     return batch, n_rsrv
```

对于第二个问题continuous batching采用**Selective batching**方法。

哪些运算对输入形状没要求（比如 Linear 层），就把所有请求拼一起高效计算；哪些运算必须对每个请求单独处理（比如 Attention），就分别算，保证计算正确。

- **非 Attention 运算：**诸如线性变换、LayerNorm、加法和 GeLU 等运算，不需要进行Token之间的交互，也就不需要区分出不同请求，因此可以将多个请求的张量先进行拼接，比如下图中x3和x4可以拼接为成一个形状为 $[\sum L, H] = [5, H]$ 的二维张量，再送入GPU进行计算 。图中QKV Linear层的输入和输出分别是 $[7, H]$、$[7, 3H]$，因为用一个Linear层计算出了QKV3个矩阵，故为3H。
- **Attention 运算：**Attention 要求只能在同一个请求内计算token之间的关系，因此在进入 Attention 前插入一个 **Split** 操作，分别计算每个请求的Attention；随后再通过**Merge** 操作将各子结果重新合并，以便后续运算继续使用批处理 。


Split
[2, 3H]
Attn x_3
[2, H]
Merge
[3, 3H]
Attn x_4
[3, H]
[7, H]
[7, 3H]
[1, 3H]
[1, H]
Attn Out Linear
QKV Linear
Attn x_1
Attn x_2
[1, 3H]
[1, H]
[7, H]
x_{14}
x_{22}
x_{31} x_{32}
x_{41} x_{42} x_{43}
Layer Input
Attention K/V Manager
$x_1 : (x_{11}, x_{12}, x_{13})$
$x_2 : (x_{21})$
Key
Value
...