# SARATHI 之Chunked Prefill

## 背景介绍

在基于 Transformer 的大模型推理中，整个生成过程可分为两大阶段：

1. **预填充（Prefill）阶段**：一次性并行处理完整 Prompt，prefill阶段计算量较大，较小的batch-size就能充分利用GPU计算资源（矩阵-矩阵乘为主）。
2. **解码（Decode）阶段**：自回归逐 Token 生成，batch-size小时GPU利用率低，且需要多轮迭代。

为了提升Decode阶段的效率，主流做法是采用**张量并行（TP）**或**流水线并行（PP）+micro-batch**来放大有效batch-size。

但TP 通信开销高，只在高带宽互联（如 NVLink）场景适用；PP+micro-batch 则会引入流水线气泡（流水线气泡指多 GPU 流水线中某些 stage 因等待上/下游或负载不匹配而出现的空闲时间片，降低整体吞吐；后文将展示其形成原因）

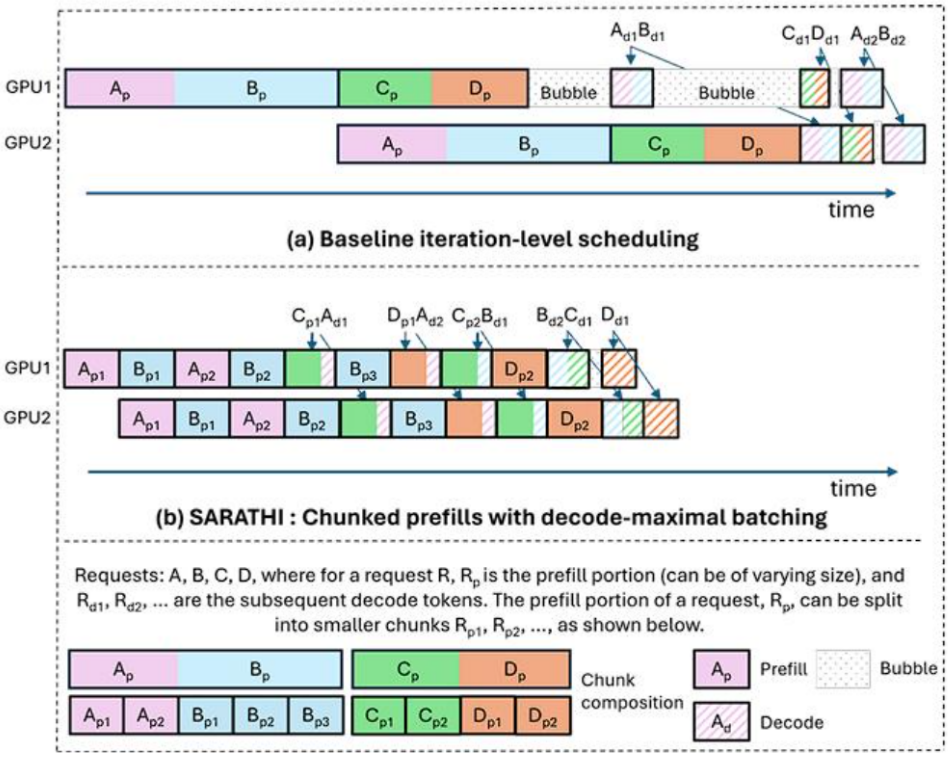
因此提出 **SARATHI** 算法以同时提升解码效率并缓解流水线气泡：

- **Chunked prefill**：将一个用户问题的prefill阶段拆成多个计算量相同的chunk，每个chunk都能单独饱和 GPU 运算单元。并且各块独立计算，有利于构建足够的混合batch，提升与Decode请求捆绑执行的机会。
- **Decode-Maximal Batching**：将一个Chunked-Prefill计算块与尽可能多的Decode请求打包到同一batch中。
  - 由于Chunked-Prefill块本身已经极大地饱和了GPU，额外加入多个Decode请求仅需极少额外开销（相较于纯Decode的batch），并且可以通过复用相同Chunked-Prefill块的KV cache，从而提升Decode效率。
  - 统一的计算量则显著减少了流水线气泡。

  下图展示了2路-流水线的情形。图中A、B、C、D表示4个独立的用户query；micro-batch size = 2，则 (A,B) 形成第一 micro-batch，(C,D) 形成第二 micro-batch。

  图（a）中，**第一个气泡**来自用户query长度差异，GPU1在跑完 $D_p$ 后，需要等待GPU2prefill完 $A_p$、$B_p$ 的后半截模型，并传回对应KV cache才能开始第一次解码 $A_{d1}B_{d1}$ 。**第二个气泡**来自prefill和decode计算时间不匹配，GPU1解码完 $A_{d1}B_{d1}$ 仍需等待GPU2完成 $C_pD_p$ 的prefill阶段的后半截模型，才能开始解码 $C_{d1}D_{d1}$ 。

  图（b）中，先将长prefill切分为等算力chunk（chunked-prefill），再进行多GPU之间的交错调度。具体来说，GPU1上 $C_{p1}$ 执行完成后，chunk内还有额外的计算时间，可以搭载已经prefill完的请求A，执行第一次decode $A_{d1}$ ，**由于Chunked-Prefill块已经极大地饱和了计算资源，额外加入Decode 几乎不额外占用时间**。Chunked-Prefill 因此产生更多且均匀的高算力时间片，让 Decode 步骤填入原有 Prefill/Decode 切换间隙，显著减少图（a）中的两类气泡。

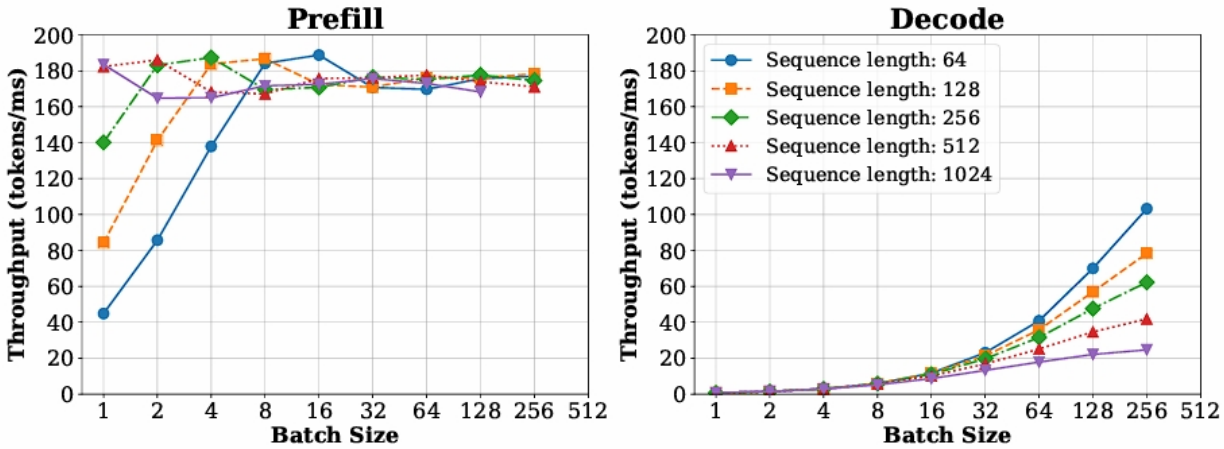



## LLM推理低效源头

LLM推理低效的两个原因：

- **Decode 主要受内存带宽限制颈（memory-bound）**。
- 流水线并行调度导致的**流水线气泡**。

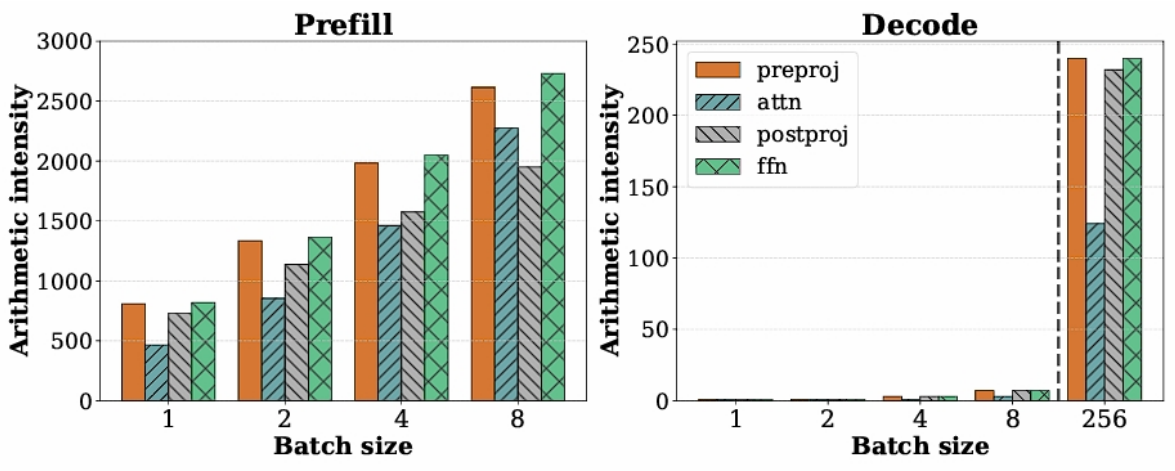
## Decode阶段memory-bound




(a) Throughput of a single layer of LLaMA-13B on A6000 GPU.

- **Prefill**阶段在batch-size(B), sequence length(L) 满足 $B \times L \geq 512$ 时，就能达到巅峰吞吐量180 tokens/ms，增大batch-size对吞吐量没有影响。Prefill阶段的输入是 $[batch\_size, sequence\_length, hidden\_size]$ ,执行的是矩阵-矩阵乘法。
- **Decode**阶段，只有当batch-size极大时（L=1024时，B>256）才开始接近饱和，但这么大的batch-size会导致KV cache占极大显存，无法实现。Decode阶段的输入是 $[batch\_size, 1, hidden\_size]$ ,执行的是向量-矩阵乘法。

定义**算术强度**为：**每次内存读写可使用的计算量**。如下图所示，Prefill阶段即使batch-size为1时算术强度也很高，能充分利用GPU算力（**compute-bound**）。而在Decode阶段，只有batch-size很大时，算术强度才能比较高，但大batch-size会增大KV cache的存储和读取压力，算术强度比较低（**memory-bound**）。




(b) Arithmetic intensity with 1K sequence length (per-request).

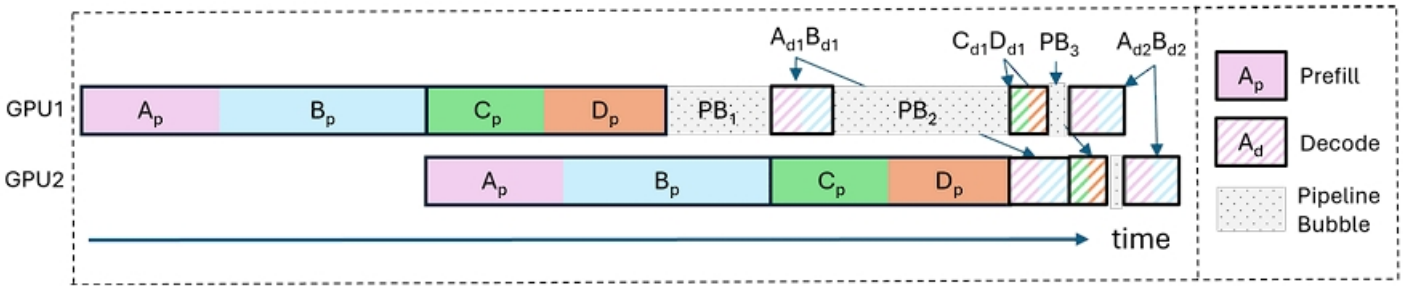
流水线气泡

现有调度主要有两种范式：

- **请求级调度 (request-level)：**一次性打包一批请求，先对整批做完整 Prefill，再顺序或分阶段 Decode。此方式简单，但短序列被 padding 到最长序列长度，造成大量无效计算，并且早完成的请求被迫等待，延迟高。
- **迭代级调度 (iteration-level / continuous batching)（Orca / vLLM / TGI 等）：**在每个生成迭代（生成下一个 token 之前）动态维护一个“活跃批”，允许新请求加入、已完成请求退出，仅保证当前迭代的批大小相对稳定，从而显著减少 padding 浪费并改善尾延迟。

然而，**迭代级调度仍无法消除流水线（多 GPU 分段）中的空闲时间片（流水线气泡）**，其主要根源有三类（对应图中的三个气泡）：

- **PB1：Prefill 长度差异导致等待。**相邻micro-batch中需要预填充的token数不同，导致prompt短的需要等待prompt长的。
- **PB2：Prefill和Decode 时长不匹配。**同一micro-batch中不同GPU分别进行prefill和decode，计算快的decode要等待计算慢的prefill。
- **PB3：Decode KV cache深度差异。**不同请求的 KV cache 深度不同，使得不同请求的Decode时间不一。




## SARATHI：Prefill多趟车+Decode搭便车

SARATHI的核心思想为：将大 Prefill 切分为等算力的小块（**chunked-prefills**），每块都能饱和GPU算力的同时，与尽可能多的 Decode 请求混合成同一微批次（**decode-maximal batching**）。这样既能保持高GPU利用率，又能将Decode的高KV cache压力摊薄到多块中，同时消除各micro-batch之间的时长差异，从根本上减少流水线气泡，提高整体吞吐。

## Chunked-Prefill

**核心思想：将一次完整的 Prefill切分成多个等算力的chunk，使得每个chunk都能单独饱和GPU计算。**

之所以可以将prefill进行分块，主要基于两个关键洞见：

1. **Prefill 吞吐饱和点**：不同模型/显卡组合下，Prefill 对 token 数的吞吐量随着输入序列长度增长，在某个阈值便趋于饱和；选用比饱和点略小的 chunk（如 256）只损失约 12.5% 的吞吐，却能带来更细的调度粒度。
2. **实际 Prompt 足够长**：生产环境中常见的prompt在 1K–4K 之间（这篇论文写于2023年，近期随着推理模型的火热，大多数模型支持的上下文长度也在变得越来越长），有足够空间拆分出多个 chunk。

Chunked-Prefill的注意力mask 矩阵设置如下图所示，当前chunk中的每个token可以访问之前的所有token，不能访问之后的token，保证分块计算与一次性prefill在数值上完全等价。

| | k0 | k1 | k2 | k3 |
|---|---|---|---|---|
| q0 | 1 | - | - | - |
| q1 | 1 | 1 | - | - |
| q2 | 1 | 1 | 1 | - |
| q3 | 1 | 1 | 1 | 1 |

attention mask during first chunk prefill

| | k0 | k1 | k2 | k3 | k4 | k5 | k6 | k7 |
|---|---|---|---|---|---|---|---|---|
| q4 | 1 | 1 | 1 | 1 | 1 | - | - | - |
| q5 | 1 | 1 | 1 | 1 | 1 | 1 | - | - |
| q6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | - |
| q7 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

attention mask during second chunk prefill

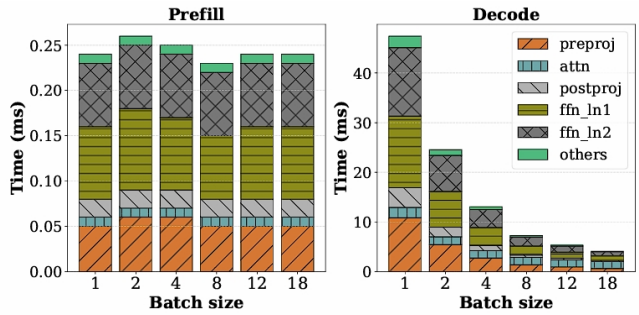
| | k0 | k1 | k2 | k3 | k4 | k5 | k6 | k7 | k8 | k9 | k10 | k11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| q8 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | - | - | - |
| q9 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | - | - |
| q10 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | - |
| q11 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

attention mask during third chunk prefill

**引入Chunked-Prefills 的开销：每个 chunk 都要重新读取前面所有已生成 token 的 KV 缓存。**

但是，作者通过实验验证了以下观点：

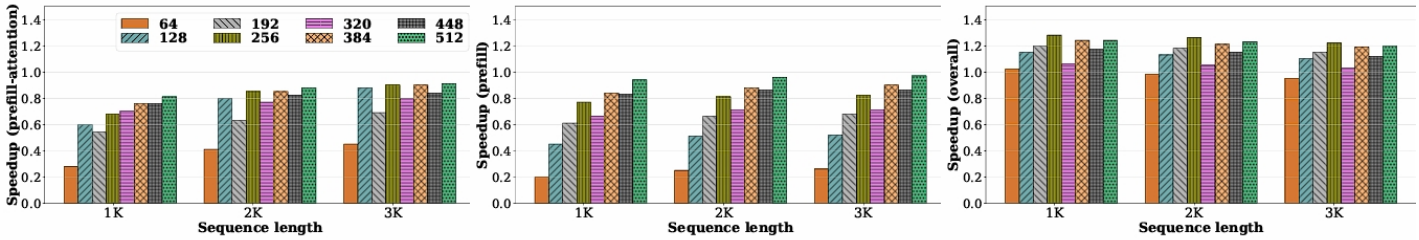
- **注意力计算在一次前向传播中耗时占比小**。KV cache只在注意力计算时使用，而注意力计算在前向传播中占比很小，主要时间都消耗在FFN层。并且考虑到prefill阶段是**compute-bound**任务，主要时间耗费在计算上，增加通信时长影响不大。




| Batching Scheme | Operation(s) | | Total Time | Per-token Time | |
|---|---|---|---|---|---|
| | Linear | Attn | | Prefill | Decode |
| Prefill-only | 224.8 | 10 | 234.8 | 0.229 | - |
| Decode-only | 44.28 | 5.68 | 49.96 | - | 12.49 |
| Decode-maximal | 223.2 | 15.2 | 238.4 | 0.229 | 1.2 |

Table 2: **Per-token prefill and decode time (in ms)** For LLaMA-13B on A6000 GPU, the rows show operation times for 1) prefill-only requests of prompt size 1024 of batch size 4, 2) decode-only batch size of 4 with sequence length 1024, and c) a mixed batch of a single 1021 prefills and 3 decodes. *Decode-maximal batching* reduces the decode time per token by an order of magnitude.

- 单看Chunked-Prefill这一个阶段，当chunk比较小时（64），对注意力计算和整个prefill阶段都会造成很大的速度损失，但当chunk提升到256/512时，损失就只有20%和10%了。而如果结合上decode-maximal batching来看，Chunked-Prefill在chunk大于128时都能提升吞吐量，因为可以携带更多得Decode任务。




(a) Self-attention (prefill-only). (b) chunked-prefills vs. full prefill. (c) End-to-end speedup for the entire batch.

Figure 13: **Ablation study:** Effect of varying the chunk size on different components of the system for LLaMa 13B on A6000 GPU.

## Decode-Maximal Batching

**核心思想：每个batch选取一个Prefilled-chunk，再用尽可能多的 Decode 任务填满剩余的slots。**这样既保证了batch的算术强度（Prefilled-chunk），保障每个batch的计算量相同，又能将多个 Decode 请求“搭载”进来，摊薄它们的内存带宽开销。

首先要确定能搭便车的最大Decode任务数。给定可用 GPU 显存 $M_G$ 、模型参数占显存大小 $M_S$ ，最大序列长度 $L$ ，每个token占用的KV cache $m_{kv}$ ，定义 $B = \lfloor (M_G - M_S)/(L \times m_{kv}) \rfloor$ ，最大Decode任务数为： $B - 1$ ，(减去一个prefilled-chunk）。

对于注意力运算，Decode和Prefilled-Chunk分别计算（因为输入形状不统一），线性计算对输入形状没要求，为了提升解码效率就把所有Decode和Prefilled-Chunk拼一起高效计算。线性计算中，Decode和Prefill阶段使用相同的权重张量，将Prefill-chunk和Decode的线性计算合并成一次矩阵-矩阵乘运算，只需要加载一次模型权重。如表2所示，纯 Decode 阶段每token平均耗时约12.49 ms；而在Decode-Maximal Batching 中，每token耗时仅1.2 ms，约 10× 加速。

**如何确定Prefilled-chunk大小**

定义**P : D比为某个混合批次中 Prefill token 数 P 与 Decode token 数 D 的比值**。

举例：batch size = 4（1 个 Prefill + 3 个 Decode），若 chunk size = 128，则一个 Prefill 长度为 P 的请求可分出 P/128 个 chunk，每个 chunk 最多搭载 3 个 Decode，共能覆盖约 P/42 个 Decode；要完全覆盖，则需 P:D＞42。

- Chunk size小会导致P:D小，虽然能携带更多的Decode，但会降低prefill效率（因为chunk size小可能无法充分利用GPU算力，并且增加了Prefilled-chunk访问KV cache的次数）
- 反之，Chunk size大，会减少能携带的Decode。
- 理想的chunk size取决于目标场景的平均P:D，以及Prefill 与 Decode 在该场景下的时间分布。(如果Decode占用了90%的时间，Prefill占了10%的时间，那为了Decode能加速2倍，prefill效率降低5倍也是值得的)。
- 此外，由于GPU执行矩阵乘法时，会将矩阵划分为若干tile并行计算。如果矩阵维度不是tile大小的整数倍，那就会浪费算力。因此，prefilled-chunk和携带的Decode token之和应该是tile大小的整数倍。

参考文章：《SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills》