LLM 的推理优化技术总结

作者：Calvin
QQ: 179209347
Mail: 179209347@qq.com

介绍

笔记简介：

- 面向对象：深度学习初学者
- 依赖课程：**线性代数，统计概率**，优化理论，图论，离散数学，微积分，信息论

知乎专栏：

https://zhuanlan.zhihu.com/p/693738275

Github & Gitee 地址：

https://github.com/mymagicpower/AIAS/tree/main/deep_learning

https://gitee.com/mymagicpower/AIAS/tree/main/deep_learning

*** 版权声明：**

- 仅限用于个人学习
- 禁止用于任何商业用途

背景

LLM之推理成本高主要原因在于：

- 模型自身复杂性：模型参数规模大，对计算和内存的需求增加
- Auto-Regressive Decoding：逐token进行，速度太慢

推理加速一般可以从两个层面体现：Latency和Throughput

- **Latency：**延迟，主要从用户的视角来看，从提交一个prompt，至返回response的响应时间，衡量指标为生成单个token的速度，如16 ms/token；若batch_size=1，只给一个用户进行服务，Latency最低。
- **Throughput：**吞吐，主要从系统的角度来看，单位时间内能处理的token数，如16 tokens/s；增加Throughput一般通过增加batch_size，即，将多个用户的请求由串行改为并行

LLM 服务的重要指标

四个关键指标：

- **首次令牌时间 (TTFT)**：
用户输入查询后开始看到模型输出的速度。较短的响应等待时间对于实时交互至关重要，但对于离线工作负载则不太重要。该指标由处理提示然后生成第一个输出令牌所需的时间驱动。

- **每个输出令牌的时间 (TPOT)**：
为查询我们系统的每个用户生成输出令牌的时间。该指标与每个用户如何感知模型的“速度”相对应。例如，100 毫秒/token 的 TPOT 意味着每个用户每秒 10 个 token，或每分钟约 450 个单词，这比一般人的阅读速度要快。

- **延迟**：
模型为用户生成完整响应所需的总时间。总体响应延迟可以使用前两个指标来计算：
延迟 = (TTFT) + (TPOT) *（要生成的令牌数量）。

- **吞吐量**：
推理服务器每秒可以针对所有用户和请求生成的输出令牌数。

推理加速

延迟(Latency)：
- 关注服务体验，返回结果越快，用户体验越好；
- 针对Latency的优化，主要还是底层的OP算子、矩阵优化、并行、更高效的C++推理等。针对Latency的优化可以提升Throughput，但没有直接增加batch_size来的更显著。

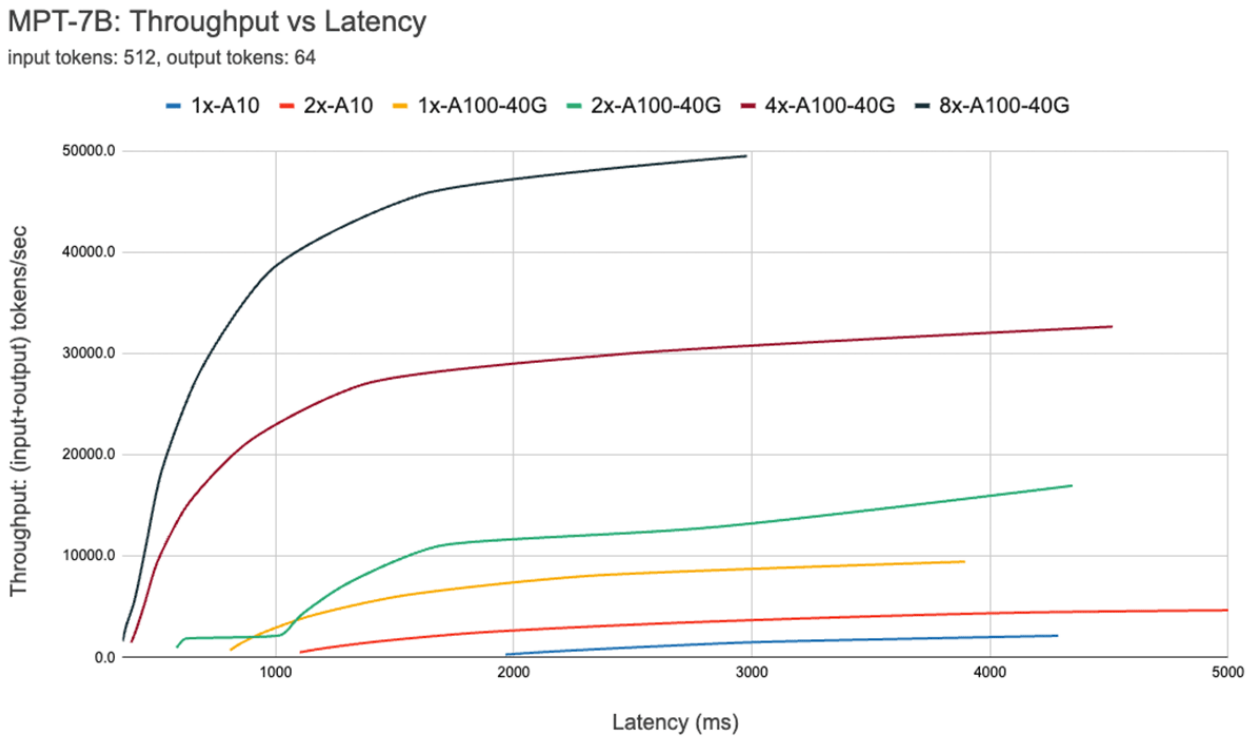
吞吐量(Throughput)：
- 关注系统成本，系统单位时间处理的量越大，系统利用率越高。
- 针对Throughput优化，主要是KV Cache存取优化，本质是降低显存开销，从而可以增加batch_size。

权衡折中(trade-off)：
- 高并发时，将用户请求组batch后能提升Throughput，但一定程度上会损害每个用户的Latency，因为之前只计算一个请求，现在合并计算多个请求，每个用户等待的时间就长了。通常，Throughput随batch_size增大而增大，Latency也随之提升，当然Latency在可接受范围内就是ok的，因此二者需要trade-off。
- 对于一些离线的场景，我们对Latency并不敏感，这时会更多的关注Throughput，但对于一些实时交互类的应用，我们就需要同时考虑Latency和Throughput。

推理加速

权衡折中(trade-off)：

- 高并发时，将用户请求组batch后能提升Throughput，但一定程度上会损害每个用户的Latency，因为之前只计算一个请求，现在合并计算多个请求，每个用户等待的时间就长了。在达到一定的批量大小之后，即当我们进入计算限制状态时，批量大小的每次加倍只会增加延迟，而不会增加吞吐量。通常，Throughput随batch_size增大而增大，Latency也随之提升，当然Latency在可接受范围内就是ok的，因此二者需要trade-off。
- 对于一些离线的场景，我们对Latency并不敏感，这时会更多的关注Throughput，但对于一些实时交互类的应用，我们就需要同时考虑Latency和Throughput。



1. 运算符融合

图融合技术即通过将多个 OP（算子）合并成一个 OP（算子），来减少Kernel的调用。因为每一个基本 OP 都会对应一次 GPU kernel 的调用，和多次显存读写，这些都会增加大量额外的开销。

- FasterTransformer by NVIDIA
- DeepSpeed Inference by Microsoft
- MLC LLM by TVM
- TensorRT by NVIDIA
- TurboTransformers by Tencent 等。

2. 模型压缩（Model Compression）

模型压缩的基本动机在于当前的模型是冗余的，可以在精度损失很小的情况下实现模型小型化，主要包括3类方法：

- 稀疏(Sparsity)、
- 量化(Quantization)、
- 蒸馏(Distillation)

2.1 稀疏(Sparsity)

实现稀疏(Sparsity)的一个重要方法是剪枝(Pruning)。
剪枝是在保留模型容量的情况下，通过修剪不重要的模型权重或连接来减小模型大小。 它可能需要也可能不需要重新培训。 修剪可以是非结构化的或结构化的。

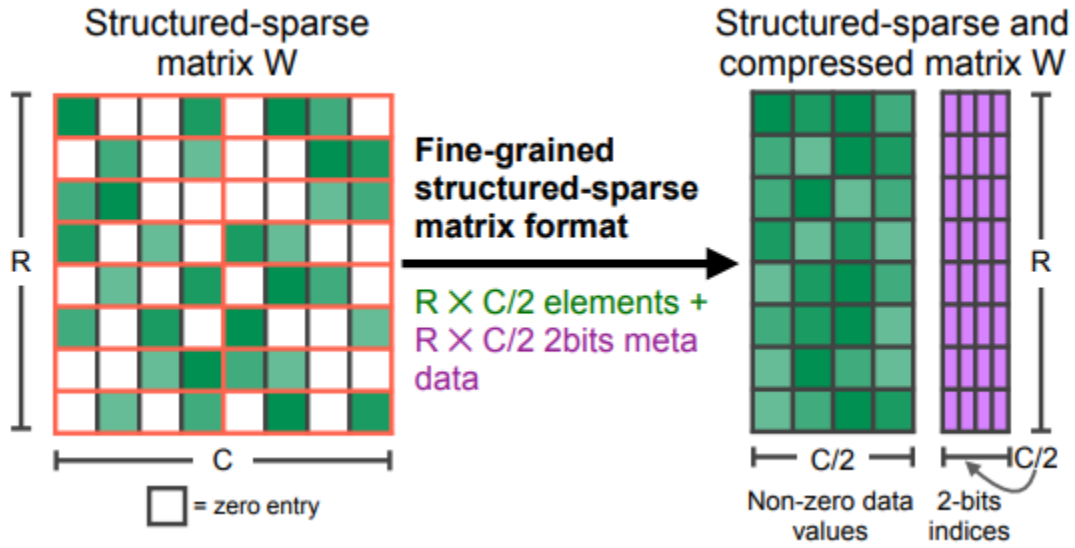
- 非结构化剪枝允许删除任何权重或连接，因此它不保留原始网络架构。 非结构化剪枝通常不适用于现代硬件，并且不会带来实际的推理加速。
- 结构化剪枝旨在维持某些元素为零的密集矩阵乘法形式。 他们可能需要遵循某些模式限制才能使用硬件内核支持的内容。 当前的主流方法关注结构化剪枝，以实现 Transformer 模型的高稀疏性。

其他的稀疏化方法，包括但不限于：

- SparseGPT：该方法的工作原理是将剪枝问题简化为大规模的稀疏回归实例。它基于新的近似稀疏回归求解器，用于解决分层压缩问题，其效率足以在几个小时内使用单个 GPU 在最大的 GPT 模型（175B 参数）上执行。同时，SparseGPT 准确率足够高，不需要任何微调，剪枝后所损耗的准确率也可以忽略不计。
- LLM-Pruner：遵循经典的“重要性估计-剪枝-微调”的策略，能够在有限资源下完成大语言模型的压缩，结果表明即使剪枝 20% 的参数，压缩后的模型保留了 93.6% 的性能。
- Wanda: 该方法由两个简单但必不可少的组件构成——剪枝度量和剪枝粒度。剪枝度量用来评估权重的重要性，然后按照剪枝粒度进行裁剪。该方法在 65B 的模型上只需要 5.6 秒就可以完成剪枝，同时达到SparseGPT相近的效果。

2.1 稀疏(Sparsity)

对于稀疏后的模型如何加速呢？NVIDIA Ampere 架构对与结构化稀疏做了专门的稀疏加速单元，下图展示了结构化稀疏的物理表示：



2.2 量化(Quantization)

随着以Transformer为基础语言模型规模的快速增大，对GPU显存和算力的需求越来越大，如何减少模型的存储大小，提高计算效率就显得愈发重要。模型的大小由其参数量及其精度决定，精度通常为 float32、float16 或 bfloat16.

在训练时，为保证精度，主权重始终为 FP32。而在推理时，FP16 权重通常能提供与 FP32 相似的精度，这意味着在推理时使用 FP16 权重，仅需一半 GPU 显存就能获得相同的结果。那么是否还能进一步减少显存消耗呢？答案就是使用量化技术，最常见的就是 INT8 量化。

常见量化有两种常见方法：

- 量化感知训练（Quantization-Aware Training，QAT）：在预训练或进一步微调期间应用量化。 QAT 能够获得更好的性能，但需要额外的计算资源和对代表性训练数据的访问。
- 训练后量化（Post-Training Quantization，PTQ）：模型首先经过训练以达到收敛，然后我们将其权重转换为较低的精度，而无需进行更多训练。 与训练相比，实施起来通常相当便宜。

2.2 量化方法

在低批量大小下使用 LLM 生成令牌是一个 GPU 显存带宽限制问题，即生成速度取决于模型参数从 GPU 显存移动到片上缓存的速度。将模型权重从 FP16（2 字节）转换为 INT8（1 字节）或 INT4（0.5 字节）需要移动更少的数据，从而加快令牌生成速度。然而，量化可能会对模型生成质量产生负面影响。

LLM.int8 量化

LLM.int8() 方法的主要目的是在不降低性能的情况下降低大模型的应用门槛，使用了 LLM.int8() 的 BLOOM-176B 比 FP16 版本慢了大约 15% 到 23%。bitsandbytes 是基于 CUDA 的主要用于支持 LLM.int8() 的库。

SmoothQuant 量化方案

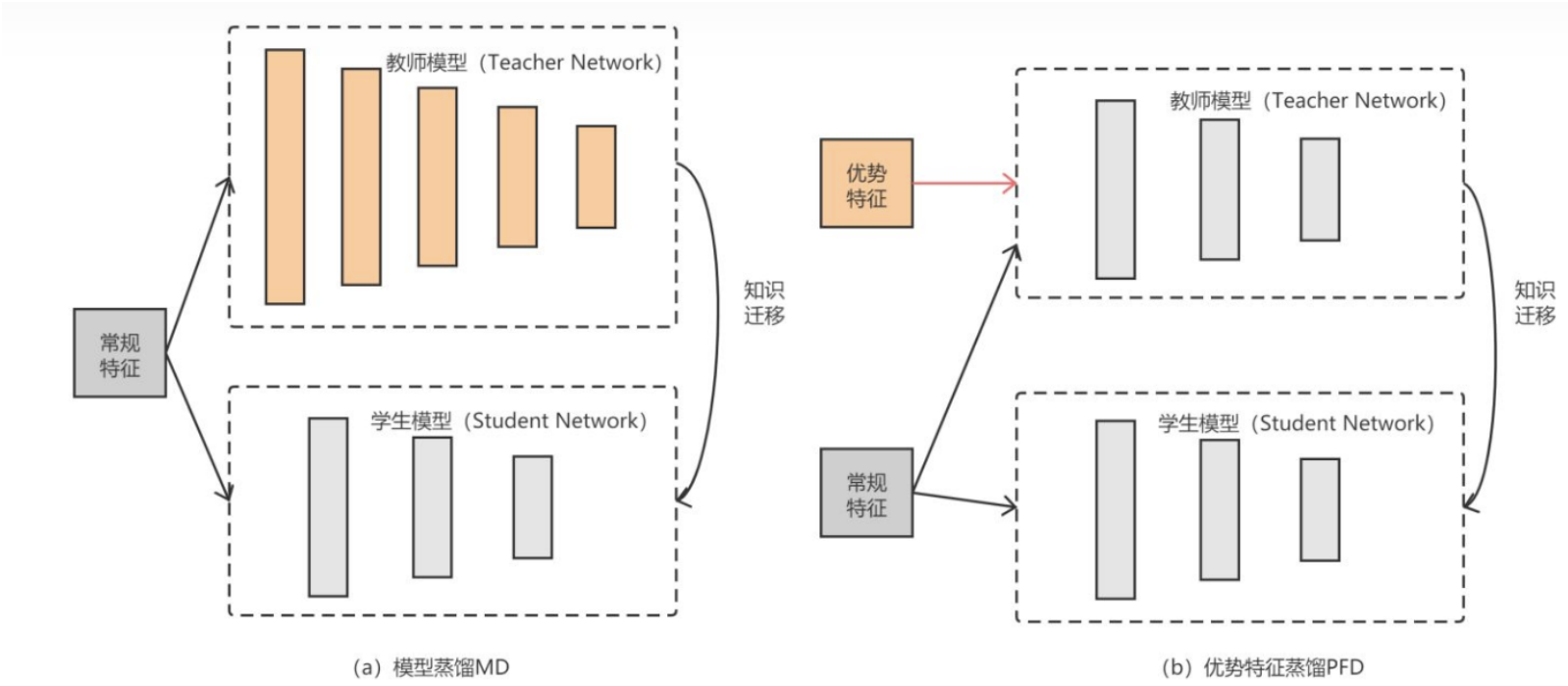
当模型规模更大时，单个token的值变化范围较大，难以量化，相比之下 weight 的变化范围较小，即 weight 较易量化，而 activation 较难量化。基于此，SmoothQuant 核心思想是引入一个超参，减小激活值的变化范围，增大权重的变化范围，从而均衡两者的量化难度。

GPTQ 量化训练方案

GPTQ 的核心思想是逐一量化模型的各个层.即对于每个需要被量化的层(对应参数W)，希望量化前后该层输出变化尽量小。

2.3 蒸馏(Distillation)

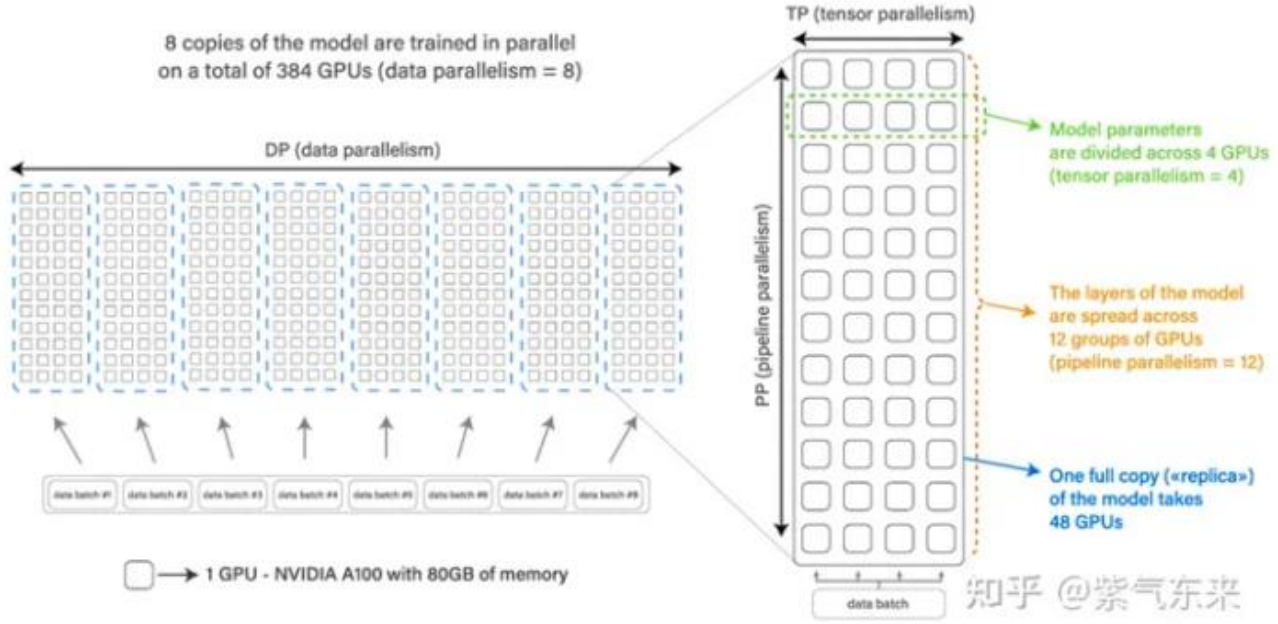
知识蒸馏是一种构建更小、更便宜的模型（“student 模型”）的直接方法，通过从预先训练的昂贵模型中转移技能来加速推理（“ teacher 模型”）融入 student。 除了与 teacher 匹配的输出空间以构建适当的学习目标之外，对于如何构建 student 架构没有太多限制。
在 Transformer 中一个典型案例是DistilBERT，模型参数减少 40%，速度提升71%。在大模型时代，蒸馏可以与量化、剪枝或稀疏化技术相结合，其中 teacher 模型是原始的全精度密集模型，而 student 模型则经过量化、剪枝或修剪以具有更高的稀疏级别，以实现模型的小型化。



3. 并行化（Parallelism）

当前的推理的并行化技术主要体现在3个维度上，即 3D Parallelism:

- Data Parallelism(DP)在推理中，DP 主要是增加设备数来增加系统整体 Throughput，其中最经典的即DeepSpeed的Zero系列
- Tensor Parallelism(TP)在推理中，TP 主要是横向增加设备数通过并行计算来减少 latency
- Pipeline Parallelism(PP)在推理中，PP 主要是纵向增加设备数通过并行计算来支持更大模型，同时提高设备利用率。



3D Parallelism 的3个维度

4. Transformer 结构优化

该类方法主要通过优化 Transformer 的结构以实现推理性能的提升。

在自回归 decoder 中，所有输入到 LLM 的 token 会产生注意力 key 和 value 的张量，这些张量保存在 GPU 显存中以生成下一个 token。这些缓存 key 和 value 的张量通常被称为 KV cache，其具有以下特点：

- 显存占用大：在 LLaMA-13B 中，缓存单个序列最多需要 1.7GB 显存；
- 动态变化：KV 缓存的大小取决于序列长度，这是高度可变和不可预测的。因此，这对有效管理 KV cache 挑战较大。该研究发现，由于碎片化和过度保留，现有系统浪费了 60% - 80% 的显存。

4.1 FlashAttention

- 在不访问整个输入的情况下计算 softmax
- 不为反向传播存储大的中间 attention 矩阵

4.2 PagedAttention

PagedAttention 允许在非连续的内存空间中存储连续的 key 和 value 。
具体来说，PagedAttention 将每个序列的 KV cache 划分为块，每个块包含固定数量 token 的键和值。在注意力计算期间，PagedAttention 内核可以有效地识别和获取这些块。

4.3 FLAT Attention

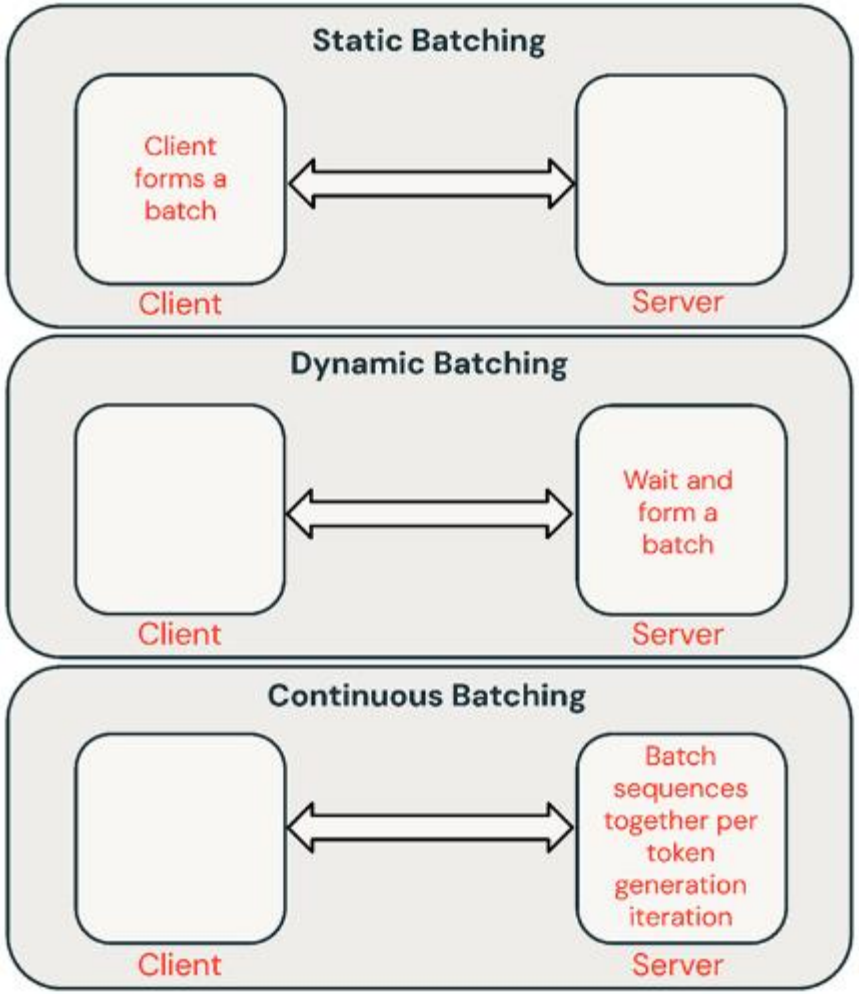
FLAT-Attention 与 FlashAttention 采取不同的路线来解决同一问题。 提出的解决方案有所不同，但关键思想是相同的（tiling 和 scheudling）。

5. 吞吐量

我们可以通过将请求一起批处理来权衡每个token的吞吐量和时间。与按顺序处理查询相比，在 GPU 评估期间对查询进行批量推理可提高吞吐量，但每个查询将需要更长的时间才能完成（忽略排队效应）。

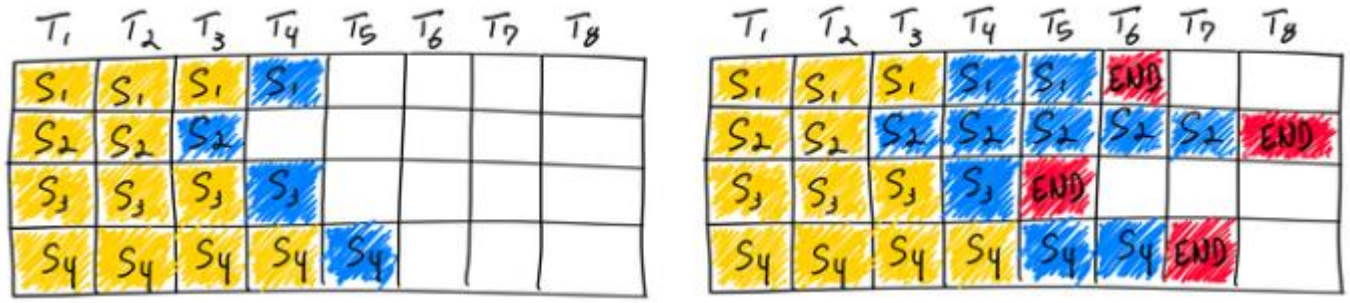
有一些用于批量推理请求的常用技术：
静态批处理：客户端将多个提示打包到请求中，并在批处理中的所有序列完成后返回响应。我们的推理服务器支持这一点，但不要求它。
动态批处理：在服务器内动态批处理在一起。通常，此方法的性能比静态批处理差，但如果响应较短或长度统一，则可以接近最优。当请求具有不同参数时效果不佳。

连续批处理：目前是 SOTA 方法。它不是等待批次中的所有序列完成，而是在迭代级别将序列分组在一起。它可以实现比动态批处理高 10 倍到 20 倍的吞吐量。

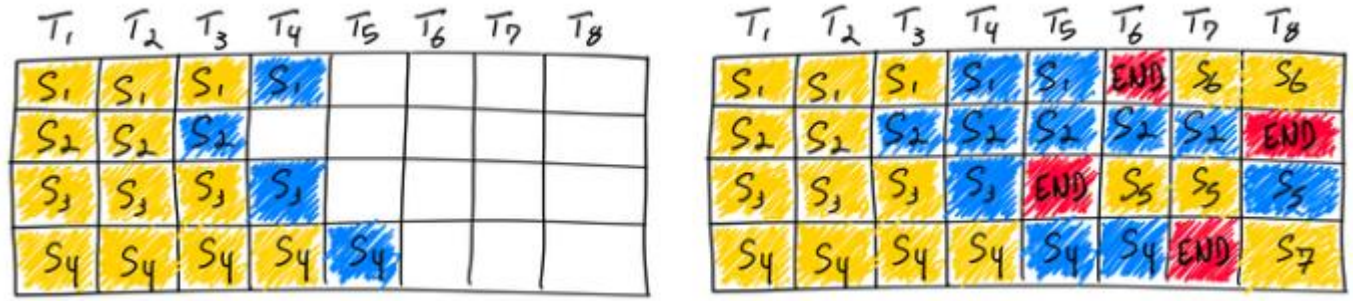


6. 连续批处理（Continuous batch）

该类方法主要是针对多 Batch 的场景，通过对 Batch 的时序优化，以达到去除 padding、提高吞吐和设备利用率。传统的 Batch 处理方法是静态的，因为Batch size 的大小在推理完成之前保持不变。



Continuous Batch不是等到 Batch 中的所有序列完成生成，而是实现 iteration 级调度，其中Batch size 由每次迭代确定。 结果是，一旦 Batch 中的序列完成生成，就可以在其位置插入新序列，从而比静态 Batch 产生更高的 GPU 利用率。



KV 缓存

Llama2 模型大小，架构：

- d，可以表示为d_head，是单个注意力头的维度。
  - 对于Llama2 7B d = 128，.
- n_heads是注意力头的数量。
  - 对于Llama2 7B n_heads = 32，.
- n_layers是注意力块出现的次数。
  - 对于Llama2 7B n_layers = 32，.
- d_model是模型的维度。d_model = d_head * n_heads。
  - 对于Llama2 7B d_model = 4096，.

params	dimension	n heads	n layers
6.7B	4096	32	32
13.0B	5120	40	40
32.5B	6656	52	60
65.2B	8192	64	80

https://arxiv.org/pdf/2302.13971.pdf

KV 缓存

Llama2 模型使用一种称为分组查询注意（GQA）的注意力变体。

GQA 通过共享键/值来帮助缩小 KV 缓存大小。KV缓存大小的计算公式为：
batch_size * seqlen * (d_model/n_heads) * n_layers * 2 (K and V) * 2 (bytes per Float16) * n_kv_heads

批量大小	GQA KV 高速缓存 (FP16)	GQA KV 缓存 (Int8)
1	0.312 GiB	0.156 GiB
16	5GiB	2.5GiB
32	10GiB	5GiB
64	20GiB	10GiB

序列长度为 1024 时 Llama-2-70B 的 KV 缓存大小

LLM七种推理服务框架总结

Framework	Tokens Per Second	Query per second	Latency	Adapters	Quantisation	Variable precision	Batching	Distributed Inference	Custom models	Token streaming	Prometheus metrics
vLLM	115	0.94	4.8 s	✗	✗	✗	✓	✓	✓	✓	✗
Text generation inference	50	0.26	4.8 s	✗	✓	✓	✓	✓	✓	✓	✓
CTranslate2	93	0.55	4.5 s	✗	✓	✓	✓	✓	✓	✓	✗
DeepSpeed-MII	80	0.47	2.5 s	✗	✗	✓	✓	✓	✗	✗	✗
OpenLLM	30	0.15	6.6 s	✓	✓	✓	✗	✗	✓	✗	✓
Ray Serve	28	0.15	7.1 s	✗	✓	✓	✓	✓	✓	✗	✓
MLC LLM	25	0.13	7.2 s	✗	✓	✗	✗	✗	✓	✓	✗

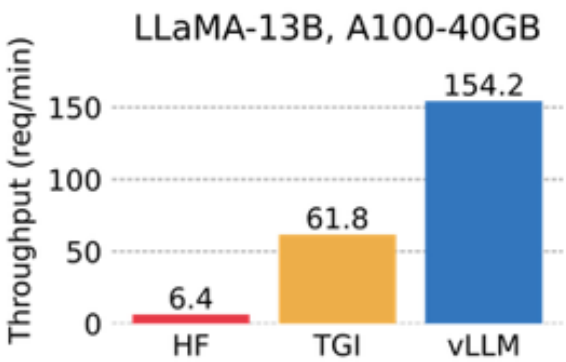
LLM七种推理服务框架总结

框架	速度	易用性	功能	文档
vLLM	10	10	5	3
TGI	4	10	7	7
CTranslate2	8	9	5	8
DeepSpeed-MII	7	1	5	1
OpenLLM	3	9	9	6
Ray Serve	2	3	9	9
MLC LLM	1	3	4	6

1. vLLM

https://vllm.ai/

吞吐性能



功能:
- 支持动态 batch。
- 使用 **PagedAttention** 优化 KV cache 显存管理。
- 支持linux。

优点:
- 最快的推理速度
- 最好的服务吞吐性能
- 与 HuggingFace 模型无缝集成（目前支持GPT2, GPTNeo, LLaMA, OPT 系列）
- 高吞吐量服务与各种 decoder 算法，包括并行采样、beam search 等
- 张量并行(TP)以支持分布式推理
- 流输出
- 兼容 OpenAI 的 API 服务

缺点:
- 添加自定义模型：虽然可以合并自己的模型，但如果模型没有使用与vLLM中现有模型类似的架构，则过程会变得更加复杂。例如，增加Falcon的支持，这似乎很有挑战性；
- 缺乏对适配器（LoRA、QLoRA等）的支持：当针对特定任务进行微调时，开源LLM具有重要价值。然而，在当前的实现中，没有单独使用模型和适配器权重的选项，这限制了有效利用此类模型的灵活性。
- 缺少权重量化：这对于减少GPU显存消耗至关重要。

1. vLLM支持的模型

- Aquila & Aquila2 (BAAI/AquilaChat2-7B, BAAI/AquilaChat2-34B, BAAI/Aquila-7B, BAAI/AquilaChat-7B, etc.)
- Baichuan (baichuan-inc/Baichuan-7B, baichuan-inc/Baichuan-13B-Chat, etc.)
- BLOOM (bigscience/bloom, bigscience/bloomz, etc.)
- ChatGLM (THUDM/chatglm2-6b, THUDM/chatglm3-6b, etc.)
- Falcon (tiiuae/falcon-7b, tiiuae/falcon-40b, tiiuae/falcon-rw-7b, etc.)
- GPT-2 (gpt2, gpt2-xl, etc.)
- GPT BigCode (bigcode/starcoder, bigcode/gpt_bigcode-santacoder, etc.)
- GPT-J (EleutherAI/gpt-j-6b, nomic-ai/gpt4all-j, etc.)
- GPT-NeoX (EleutherAI/gpt-neox-20b, databricks/dolly-v2-12b, stabilityai/stablelm-tuned-alpha-7b, etc.)
- InternLM (internlm/internlm-7b, internlm/internlm-chat-7b, etc.)
- LLaMA & LLaMA-2 (meta-llama/Llama-2-70b-hf, lmsys/vicuna-13b-v1.3, young-geng/koala, openlm-research/open_llama_13b, etc.)
- Mistral (mistralai/Mistral-7B-v0.1, mistralai/Mistral-7B-Instruct-v0.1, etc.)
- MPT (mosaicml/mpt-7b, mosaicml/mpt-30b, etc.)
- OPT (facebook/opt-66b, facebook/opt-iml-max-30b, etc.)
- Phi-1.5 (microsoft/phi-1_5, etc.)
- Qwen (Qwen/Qwen-7B, Qwen/Qwen-7B-Chat, etc.)
- Yi (01-ai/Yi-6B, 01-ai/Yi-34B, etc.)

https://github.com/vllm-project/vllm

2. Text generation inference

Text generation inference是用于文本生成推断的Rust、Python和gRPC服务器，在HuggingFace中已有LLM 推理API使用。

https://github.com/huggingface/text-generation-inference

功能：

- **内置服务评估：**可以监控服务器负载并深入了解其性能；
- **使用flash attention（和v2）和Paged attention优化transformer推理代码：**并非所有模型都内置了对这些优化的支持，该技术可以对未使用该技术的模型可以进行优化；

优点：

- **所有的依赖项都安装在Docker中：**会得到一个现成的环境；
- **支持HuggingFace模型：**轻松运行自己的模型或使用任何HuggingFace模型中心；
- **对模型推理的控制**：该框架提供了一系列管理模型推理的选项，包括精度调整、量化、张量并行性、重复惩罚等；

缺点：

- **缺乏对适配器的支持：**需要注意的是，尽管可以使用适配器部署LLM（可以参考https://www.youtube.com/watch?v=HI3cYN0c9ZU），但目前还没有官方支持或文档；
- **从源代码（Rust+CUDA内核）编译：**对于不熟悉Rust的人，将客户化代码纳入库中变得很有挑战性；
- **文档不完整**：所有信息都可以在项目的自述文件中找到。尽管它涵盖了基础知识，但必须在问题或源代码中搜索更多细节；

3. CTranslate2

CTranslate2是一个C++和Python库，用于使用Transformer模型进行高效推理。

https://github.com/OpenNMT/CTranslate2

功能：

- **在CPU和GPU上快速高效地执行：**得益于内置的一系列优化：层融合、填充去除、批量重新排序、原位操作、缓存机制等。推理LLM更快，所需内存更少；
- **动态内存使用率：**由于CPU和GPU上都有缓存分配器，内存使用率根据请求大小动态变化，同时仍能满足性能要求；
- **支持多种CPU体系结构：**该项目支持x86–64和AArch64/ARM64处理器，并集成了针对这些平台优化的多个后端：英特尔MKL、oneDNN、OpenBLAS、Ruy和Apple Accelerate；

优点：

- **并行和异步执行**--可以使用多个GPU或CPU核心并行和异步处理多个批处理；
- **Prompt缓存**——在静态提示下运行一次模型，缓存模型状态，并在将来使用相同的静态提示进行调用时重用；
- **磁盘上的轻量级**--量化可以使模型在磁盘上缩小4倍，而精度损失最小；

缺点：

- **没有内置的REST服务器**——尽管仍然可以运行REST服务器，但没有具有日志记录和监控功能的现成服务
- **缺乏对适配器（LoRA、QLoRA等）的支持**

4. DeepSpeed-MII

在DeepSpeed支持下，DeepSpeed-MII可以进行低延迟和高通量推理。

https://github.com/microsoft/DeepSpeed-MII

功能：

- **多个副本上的负载平衡：**这是一个非常有用的工具，可用于处理大量用户。负载均衡器在各种副本之间高效地分配传入请求，从而缩短了应用程序的响应时间。
- **非持久部署：**目标环境的部署不是永久的，需要经常更新的，这在资源效率、安全性、一致性和易管理性至关重要的情况下，这是非常重要的。

优点：

- **支持不同的模型库：**支持多个开源模型库，如Hugging Face、FairSeq、EluetherAI等；
- **量化延迟和降低成本：**可以显著降低非常昂贵的语言模型的推理成本；
- **Native和Azure集成：**微软开发的MII框架提供了与云系统的出色集成；

缺点：

- **支持模型的数量有限：**不支持Falcon、LLaMA2和其他语言模型；
- **缺乏对适配器（LoRA、QLoRA等）的支持；**

5. OpenLLM

OpenLLM是一个用于在生产中操作大型语言模型（LLM）的开放平台。

https://github.com/bentoml/OpenLLM

功能：

- **适配器支持：**可以将要部署的LLM连接多个适配器，这样可以只使用一个模型来执行几个特定的任务；
- **支持不同的运行框架：**比如Pytorch（pt）、Tensorflow（tf）或Flax（亚麻）；
- **HuggingFace Agents：**连接HuggingFace上不同的模型，并使用LLM和自然语言进行管理；

优点：

- **良好的社区支持：**不断开发和添加新功能；
- **集成新模型：**可以添加用户自定义模型；
- **量化：**OpenLLM支持使用bitsandbytes和GPTQ进行量化；
- **LangChain集成：**可以使用LangChian与远程OpenLLM服务器进行交互；

缺点：

- **缺乏批处理支持：**对于大量查询，这很可能会成为应用程序性能的瓶颈；
- **缺乏内置的分布式推理**——如果你想在多个GPU设备上运行大型模型，你需要额外安装OpenLLM的服务组件Yatai[14]；

6. Ray Serve

Ray Serve是一个可扩展的模型服务库，用于构建在线推理API。Serve与框架无关，因此可以使用一个工具包来为深度学习模型的所有内容提供服务。

https://www.ray.io/

功能：

- **监控仪表板和Prometheus度量：**可以使用Ray仪表板来获得Ray集群和Ray Serve应用程序状态；
- **跨多个副本自动缩放：**Ray通过观察队列大小并做出添加或删除副本的缩放决策来调整流量峰值；
- **动态请求批处理：**当模型使用成本很高，为最大限度地利用硬件，可以采用该策略；

优点：

- **文档支持：**开发人员几乎为每个用例撰写了许多示例；
- **支持生产环境部署：**这是本列表中所有框架中最成熟的；
- **本地LangChain集成：**您可以使用LangChain与远程Ray Server进行交互；

缺点：

- **缺乏内置的模型优化：**Ray Serve不专注于LLM，它是一个用于部署任何ML模型的更广泛的框架，**必须自己进行优化**；
- **入门门槛高：**该库功能多，提高了初学者进入的门槛；

7. MLC LLM

LLM的机器学习编译（MLC LLM）是一种通用的部署解决方案，它使LLM能够利用本机硬件加速在消费者设备上高效运行。

https://github.com/mlc-ai/mlc-llm

功能：

- **平台本机运行时：**可以部署在用户设备的本机环境上，这些设备可能没有现成的Python或其他必要的依赖项。应用程序开发人员只需要将MLC编译的LLM集成到他们的项目中即可；
- **内存优化：**可以使用不同的技术编译、压缩和优化模型，从而可以部署在不同的设备上；

优点：

- **所有设置均可在JSON配置中完成：**在单个配置文件中定义每个编译模型的运行时配置；
- **预置应用程序：**可以为不同的平台编译模型，比如C++用于命令行，JavaScript用于web，Swift用于iOS，Java/Kotlin用于Android；

缺点：

- **使用LLM模型的功能有限**：不支持适配器，无法更改精度等，该库主要用于编译不同设备的模型；
- **只支持分组量化[15]：**这种方法表现良好，但是在社区中更受欢迎的其他量化方法（bitsandbytes和GPTQ）不支持；
- **复杂的安装：**安装需要花几个小时，不太适合初学者开发人员；

如果需要在iOS或Android设备上部署应用程序，这个库正是你所需要的。它将允许您快速地以本机方式编译模型并将其部署到设备上。但是，如果需要一个高负载的服务器，不建议选择这个框架。

其它推理框架总结

1. Fastllm
2. LMDeploy
3. llama.cpp
4. LightLLM
5. TensorRT-LLM　TensorRT-LLM 发布之后FT(**FasterTransformer**)不再维护
6. DJL serving
7. FastServe

1. fastllm

纯c++的全平台llm加速库，支持python调用，chatglm-6B级模型单卡可达10000+token / s，支持glm, llama, moss基座，手机端流畅运行.

https://github.com/ztxz16/fastllm

功能概述:

- 纯c++实现，便于跨平台移植，可以在安卓上直接编译
- ARM平台支持NEON指令集加速，X86平台支持AVX指令集加速，NVIDIA平台支持CUDA加速，各个平台速度都很快就是了
- 支持浮点模型（FP32), 半精度模型(FP16), 量化模型(INT8, INT4) 加速
- 支持多卡部署，支持GPU + CPU混合部署
- 支持Batch速度优化
- 支持并发计算时动态拼Batch
- 支持流式输出，很方便实现打字机效果
- 支持python调用
- 前后端分离设计，便于支持新的计算设备
- 目前支持ChatGLM模型，各种LLAMA模型(ALPACA, VICUNA等)，BAICHUAN模型，MOSS模型

2. LMDeploy

LMDeploy 由 MMDeploy 和 MMRazor 团队联合开发，是涵盖了 LLM 任务的全套轻量化、部署和服务解决方案。

http://github.com/InternLM/lmdeploy

功能概述:

- **高效推理引擎 TurboMind**：基于 FasterTransformer，我们实现了高效推理引擎 TurboMind，支持 InternLM、LLaMA、vicuna等模型在 NVIDIA GPU 上的推理。
- **交互推理方式**：通过缓存多轮对话过程中 attention 的 k/v，记住对话历史，从而避免重复处理历史会话。
- **多 GPU 部署和量化**：我们提供了全面的模型部署和量化支持，已在不同规模上完成验证。
- **persistent batch 推理**：进一步优化模型执行效率。
- 不支持GGUF。

模型	模型并行	FP16	KV INT8	W4A16	W8A8
Llama	Yes	Yes	Yes	Yes	No
Llama2	Yes	Yes	Yes	Yes	No
SOLAR	Yes	Yes	Yes	Yes	No
InternLM-7B	Yes	Yes	Yes	Yes	No
InternLM-20B	Yes	Yes	Yes	Yes	No
QWen-7B	Yes	Yes	Yes	Yes	No
QWen-14B	Yes	Yes	Yes	Yes	No
Baichuan-7B	Yes	Yes	Yes	Yes	No
Baichuan2-7B	Yes	Yes	Yes	Yes	No
Code Llama	Yes	Yes	No	No	No

3. llama.cpp

llama.cpp 满足了开源大模型 + 本地化部署的需求，吸引了大量没有昂贵 GPU 的玩家，成为了在消费级硬件上玩大模型的首选。
由于目前不支持动态batch，所以不适合服务器端部署，并发访问的场景。

https://github.com/ggerganov/llama.cpp

功能概述:

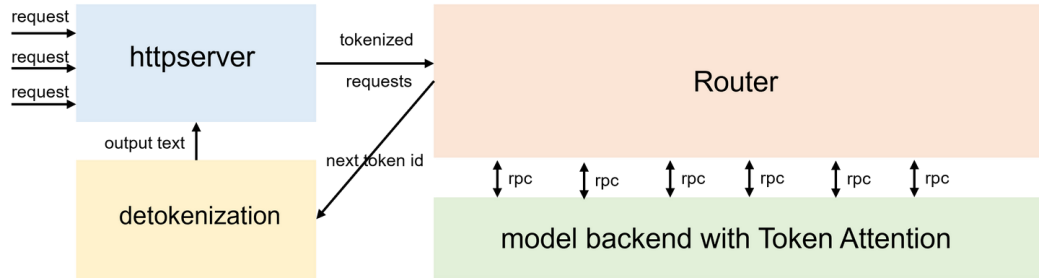
- 无需任何额外依赖，相比 Python 代码对 PyTorch 等库的要求，C/C++ 直接编译出可执行文件，跳过不同硬件的繁杂准备；
- 支持 Apple Silicon 芯片的 ARM NEON 加速，x86 平台则以 AVX2 替代；
- 具有 F16 和 F32 的混合精度；
- 支持 4-bit 量化；
- 支持CPU，GPU；

4. LightLLM

LightLLM是一个基于Python的LLM（大型语言模型）推理和服务框架，以其轻量级设计、易于扩展和高速性能而闻名。
LightLLM利用了许多备受好评的开源实现优势，包括FasterTransformer、TGI、vLLM和FlashAttention等。

LIGHT LLM

A Light and Fast Inference Service for LLM



https://github.com/ModelTC/lightllm

LightLLM包括以下特点：
1.三进程异步协作：token化、模型推理和去token化是异步执行的，从而大大提高了GPU的利用率。
2.Nopad（Unpad）：支持跨多个模型的无填零注意操作，有效处理长度差异较大的请求。
3.动态批处理：请求的动态批处理调度。
4.FlashAttention：结合FlashAttention（“**FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness**“，2022），提高速度并减少推理过程中GPU内存占用。
5.张量并行性：利用多个GPU的张量并行性进行更快的推理。
6.TokenAttention：实现逐Token的KV缓存内存管理机制，允许在推理过程中零内存浪费。
7.高性能路由器：与TokenAttention合作，精心管理每个token的GPU内存，从而优化系统吞吐量。
8.Int8KV缓存：此功能将使token的容量几乎增加一倍。只有LLAMA支持。

支持的模型列表：

- BLOOM
- LLaMA
- LLaMA V2
- StarCoder
- Qwen-7b
- ChatGLM2-6b
- Baichuan-7b
- Baichuan2-7b
- Baichuan2-13b
- Baichuan-13b
- InternLM-7b
- Yi-34b

5. TensorRT-LLM

TensorRT-LLM提供了一个易于使用的Python API，用于定义大型语言模型（LLM）并构建TensorRT引擎，该引擎包含了在NVIDIA GPU上高效执行推理的最新优化技术。TensorRT-LLM还包含用于创建执行这些TensorRT引擎的Python和C++运行时的组件。它还包括与NVIDIA Triton推理服务器集成的后端，这是一个用于提供LLMs的生产级系统。使用TensorRT-LLM构建的模型可以在各种配置上执行，从单个GPU到具有多个GPU的多个节点（使用张量并行和/或流水线并行）。

https://github.com/NVIDIA/TensorRT-LLM

TensorRT-LLM 的特性：

1、支持模型架构定义、预训练权重编译、推理加速；
2、提供 Python API 编译 TensorRT 引擎；
3、提供组件创建 Python/C++ Runtimes，用于执行 TensorRT 引擎；
4、GPU 上的高效推理，做了 SOTA 级别的优化；
5、包含了一个可与 Triton Inference Server 集成的 backend；
6、自带主流的预定义热门大语言模型，包括 baichuan、LlaMA、ChatGLM、BLOOM、GPT等。

支持的模型列表：

- Baichuan
- BART
- Bert
- Blip2
- BLOOM
- ChatGLM
- FairSeq NMT
- Falcon
- Flan-T5
- GPT
- GPT-J
- GPT-Nemo
- GPT-NeoX
- InternLM
- LLaMA
- LLaMA-v2
- mBART
- Mistral
- MPT
- mT5
- OPT
- Qwen
- Replit Code
- SantaCoder
- StarCoder
- T5
- Whisper

6. DJL LMI

DJLServing具有托管大型语言模型和基础模型的能力，这些模型无法适应单个GPU。维护了一系列专门用于使用大型模型进行推理的深度学习容器（DLC）。DJL Serving是由DJL驱动的高性能通用独立模型服务解决方案。它接受深度学习模型、多个模型或工作流，并通过HTTP端点使它们可用。

https://docs.djl.ai/docs/serving/serving/docs/large_model_inference.html

可以直接支持以下模型类型：

- PyTorch TorchScript model
- TensorFlow SavedModel bundle
- Apache MXNet model
- ONNX model (CPU)
- TensorRT model
- Python script model

可以安装额外的扩展来启用以下模型：

- PaddlePaddle model
- TFLite model
- XGBoost model
- LightGBM model
- Sentencepiece model
- fastText/BlazingText model

关键特性：

- **性能** - 在单个JVM中运行多线程推理。吞吐量比市场上大多数C++模型服务器更高。
- **易于使用** - 可以直接为大多数模型提供服务。
- **易于扩展** - 插件使添加自定义扩展变得容易。
- **自动扩展** - 根据负载自动扩展/缩减工作线程。
- **动态批处理** - 支持动态批处理以提高吞吐量。
- **模型版本控制** - 允许用户在单个端点上加载不同版本的模型。
- **多引擎支持** - 允许用户同时为不同引擎的模型提供服务。

6. DJL LMI - DeepSpeed 调优参考

模型	数据类型	大小 (GB)	实例类型	并行度	最大批次大小 (rolling)
Llama2 7B	fp16	14	g5.2xlarge	1	8
Llama2 7B smoothquant	int8	7	g5.2xlarge	1	16
Llama2 7B	fp16	14	g5.12xlarge	2	32
Llama2 13B	fp16	26	g5.12xlarge	4	32
Llama2 13B smoothquant	int8	13	g5.12xlarge	4	64

- g5.2xlarge 8vCPUs 2.8GHz, 32G 内存, 1 GPU- A10 24G
- g5.12xlarge 48vCPUs 2.8GHz, 192G 内存, 4 GPU- A10 96G

6. DJL LMI - TensorRT 调优参考

模型	大小 (GB)	实例类型	并行度	模型大小/GPU显存	最大批次大小 (rolling)
Llama2 7B	14	g5.2xlarge	1	0.58333	16
Llama2 13B	26	g5.12xlarge	4	0.27083	32
Llama2 70B	134	g5.48xlarge	8	0.69792	8
Llama2 70B	134	p4d.24xlarge	8	0.41875	128
Falcon 7B	16	g5.2xlarge	1	0.66667	16
Falcon 40B	84	g5.48xlarge	8	0.4375	32
Falcon 40B	84	p4d.24xlarge	8	0.2625	128
Code Llama 34B	63	p4d.24xlarge	8	0.19688	128

- g5.2xlarge 8vCPUs 2.8GHz, 32G 内存, 1 GPU- A10 24G
- g5.12xlarge 48vCPUs 2.8GHz, 192G 内存, 4 GPU- A10 96G
- g5.48xlarge 192vCPUs 2.8GHz, 2048G 内存, 8 GPU- A10 192G
- p4d.24xlarge 96vCPUs 3GHz, 1152G 内存, 8 GPU- A100 320G

6. DJL LMI - TransformersNeuronX 调优参考

模型	大小 (GB)	实例类型	并行度	最大批次大小 (rolling)	n_positions
Llama2 7b	14	inf2.xlarge	2	4	2048
Llama2 7b	14	inf2.24xlarge	8	4	2048
Llama2 13b (int8)	13	inf2.xlarge	2	4	512
Llama2 13b	26	inf2.24xlarge	8	4	2048

- inf2.xlarge 4vCPUs 2.95GHz, 16G 内存, 1 GPU- aws inferentia2 32G
- inf2.24xlarge 96vCPUs 2.95GHz, 384G 内存, 6 GPU- aws inferentia2 192G

6. DJL LMI - LMI Dist 调优参考

模型	大小 (GB)	实例类型	并行度	最大批次大小 (rolling)	max_rolling_batch _prefill_tokens
GPT-NeoX-20B	39	g5.12xlarge	4	64	19200
Flan-T5 XXL	42	g5.12xlarge	4	64	
Flan-UL2	37	g5.12xlarge	4	64	
Llama2 7B	13	g5.12xlarge	1	32	11000
Llama2-13B	25	g5.12xlarge	2	32	14500
Llama2 70B	129	g5.48xlarge	8	8	
Llama2 70B	129	p4d.24xlarge	8	64	
Falcon 7b	14	g5.2xlarge	1	64	
Falcon 40b	78	g5.48xlarge	8	64	18000

- g5.2xlarge 8vCPUs 2.8GHz, 32G 内存, 1 GPU- A10 24G
- g5.12xlarge 48vCPUs 2.8GHz, 192G 内存, 4 GPU- A10 96G
- g5.48xlarge 192vCPUs 2.8GHz, 2048G 内存, 8 GPU- A10 192G
- p4d.24xlarge 96vCPUs 3GHz, 1152G 内存, 8 GPU- A100 320G

7. FastServe 框架

FastServe系统是由北京大学研究人员开发的，针对大语言模型的分布式推理服务进行了设计和优化。整体系统设计目标包含以下三个方面。

- 低作业完成时间：专注于交互式大语言模型应用，用户希望作业能够快速完成，系统应该在处理推理作业时实现低作业完成时间。
- 高效的GPU 显存管理：大语言模型的参数和键值缓存占用了大量的GPU 显存，系统应该有效地管理GPU 显存，以存储模型和中间状态。
- 可扩展的分布式系统：大语言模型需要多块GPU 以分布式方式进行推理，系统需要可扩展的分布式系统，以处理大语言模型的推理作业。

FastServe 的整体框架如图所示:

- 用户将作业提交到作业池（Job Pool）中，跳跃连接多级反馈队列（Skip-join MLFQ）调度器使用作业分析器（Job Profiler）根据作业启动阶段的执行时间决定新到达作业的初始优先级。
- FastServe 作业调度采用迭代级抢占策略，并使用最小者（Leastattained）优先策略，以解决头部阻塞问题。一旦选择执行某个作业，调度器会将其发送到分布式执行引擎（Distributed Execution Engine），该引擎调度GPU 集群为大语言模型提供服务，并与分布式键值缓存（Distributed Key-Value Cache）进行交互，在整个运行阶段检索和更新相应作业的键值张量。
- 为了解决GPU 显存容量有限的问题，键值缓存管理器（Key-Value Cache Management）会主动将优先级较低的作业的键值张量转移到主机内存，并根据工作负载的突发性动态调整其转移策略。

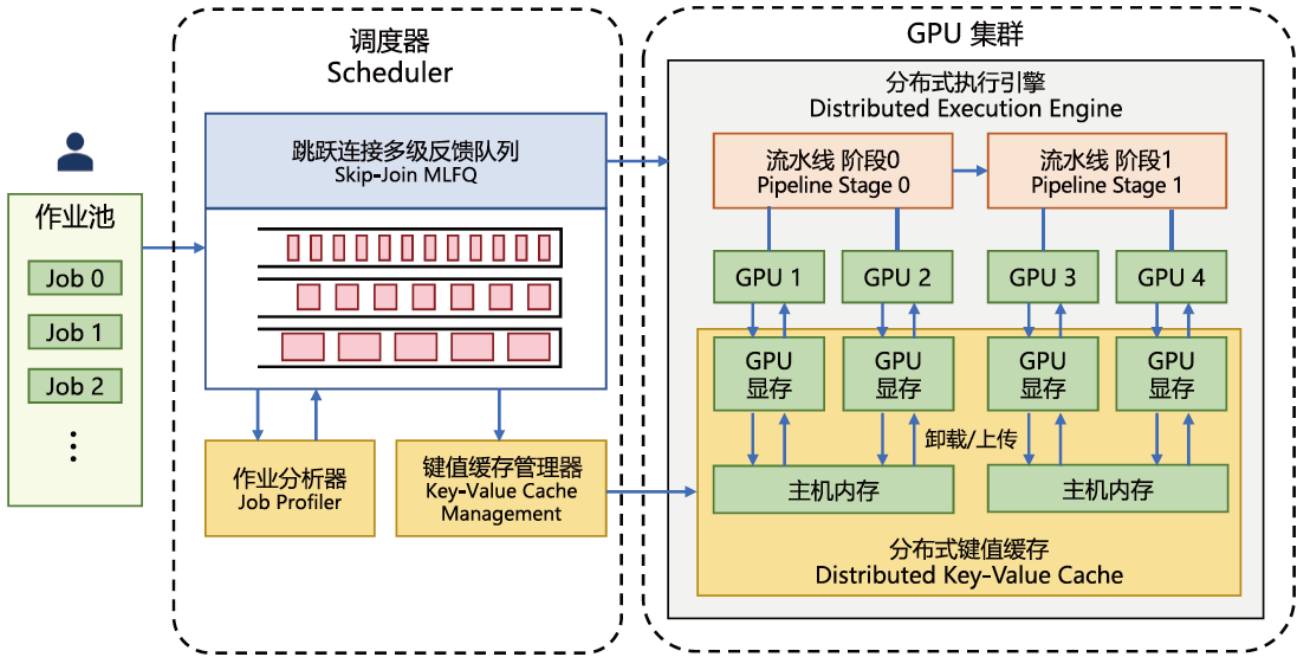


图 7.17　FastServe 的整体框架[193]

参考资料

1. 小记：主流推理框架在Llama 2 的上性能比较
https://zhuanlan.zhihu.com/p/646772063

2. [fastllm]多线程下动态组batch实现解析
https://www.cnblogs.com/wildkid1024/p/17658849.html

3. 大模型推理加速工具：vLLM
https://zhuanlan.zhihu.com/p/642802585

4. NLP（十七）：从 FlashAttention 到 PagedAttention, 如何进一步优化 Attention 性能
https://zhuanlan.zhihu.com/p/638468472

5. NLP（十八）：LLM 的推理优化技术纵览
https://zhuanlan.zhihu.com/p/642412124

6. LLM 量化技术小结
https://zhuanlan.zhihu.com/p/651874446

7. 7 Frameworks for Serving LLMs
https://betterprogramming.pub/frameworks-for-serving-llms-60b7f7b23407


Thank
You