大模型推理加速综述

1. LLM 基本概念 和 推理过程中的瓶颈

- 大模型(Transformer Block)的流程

$$Q_i = XW^{Q_i}, K_i = XW^{K_i}, V_i = XW^{V_i},$$

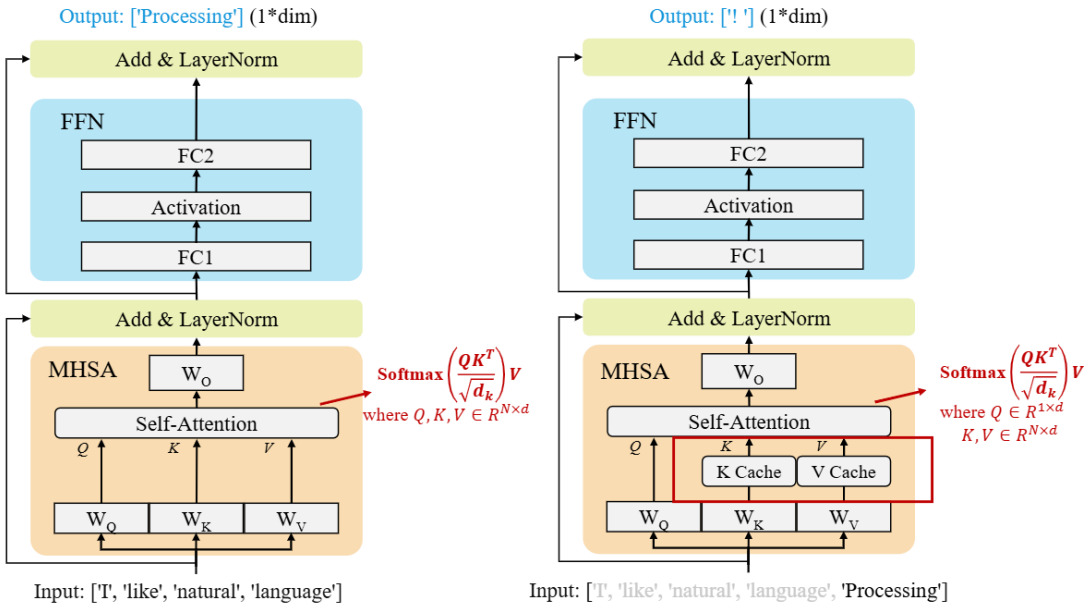
$$Z_i = \text{Attention}(Q_i, K_i, V_i) = \text{Softmax}(\frac{Q_i K_i^T}{\sqrt{d_k}})V_i,$$

$$Z = \text{Concat}(Z_1, Z_2, ..., Z_h)W^O$$

$$\text{FFN}(X) = W_2\sigma(W_1 X)$$

10.6. The Encoder–Decoder Architecture — Dive into Deep Learning 1.0.3 documentation (d2l.ai)
Transformer | 鲁老师 (lulaoshi.info)

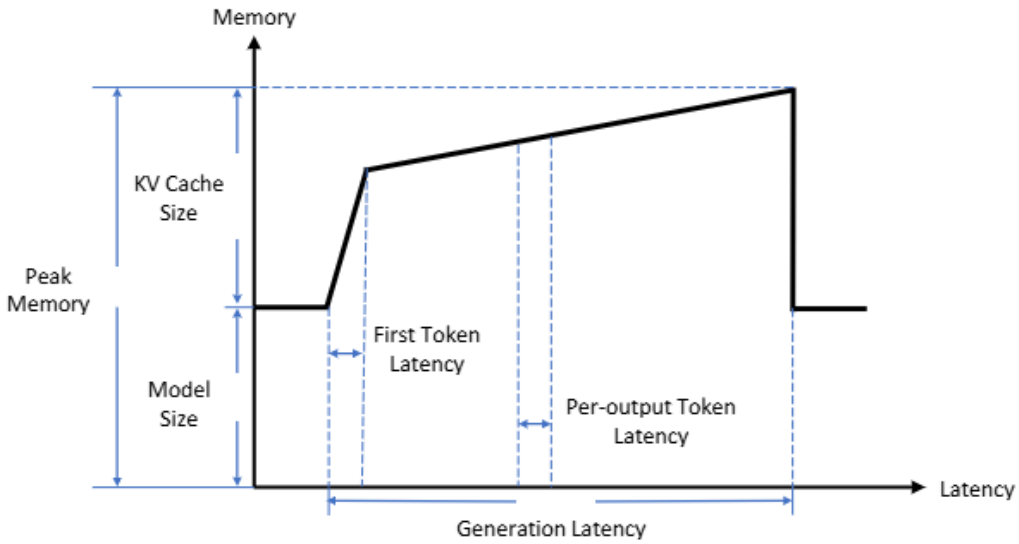
- 大模型推理过程
  主要分为两个阶段：prefill 和 decoding



(a) The prefilling stage **(b)** The decoding stage

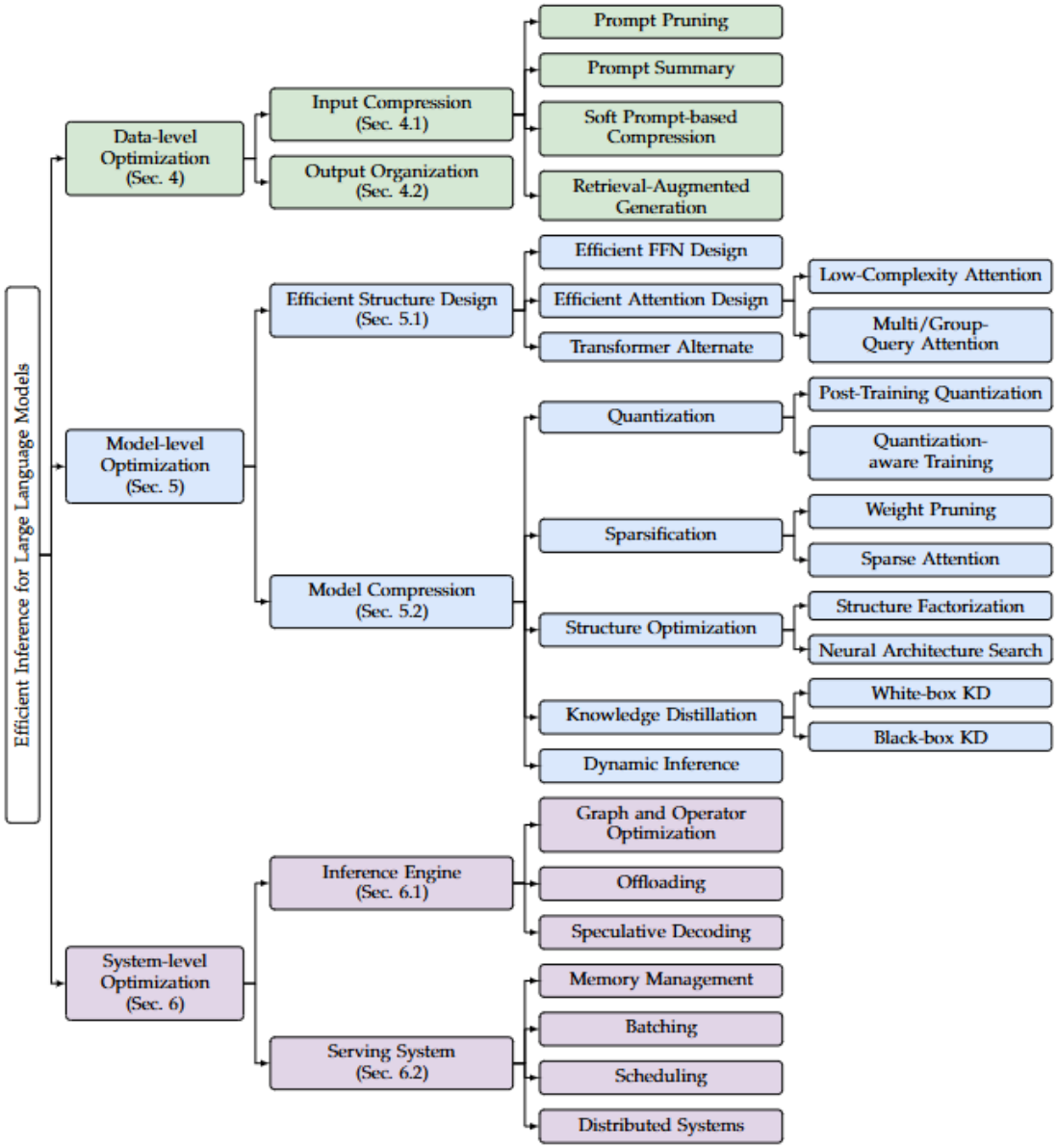
prefill： LLM计算并存储初始输入token的KV缓存，并生成第一个输出token
decoding： LLM通过KV缓存一一生成输出令牌，然后用新生成的令牌的键（K）和值（V)对进行更新

- 推理过程性能分析



峰值内存表示生成过程中的最大内存使用量，大约等于模型权重和KV缓存的内存总和。此外，大模型推理重点关注三个成本： **计算成本 、 内存使用成本 、 内存访问成本** 主要有三个原因： 模型尺寸 、 注意力操作 、 解码步骤(自回归解码方法——生成标记。在每个解码步骤中，所有模型权重都从片外 HBM 加载到 GPU 芯片，导致很大的内存访问成本。此外，KV缓存的大小随着输入长度的增长而增加，可能导致内存碎片和不规则的内存访问模式)

2. 大模型推理加速方法的分类

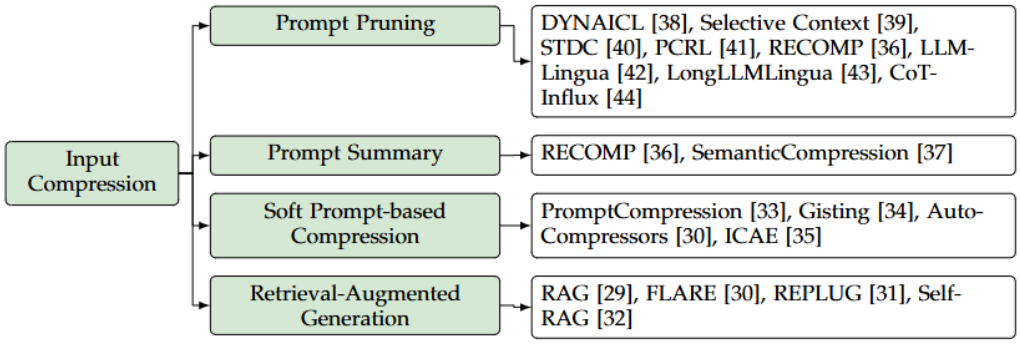


3. 数据级优化

数据层面，先前的研究可以分为两类，即**输入压缩和输出组织。**输入压缩技术直接缩短模型输入以降低推理成本。而输出组织技术通过组织输出内容的结构来实现批量（并行）推理，这可以提高硬件利用率并减少生成延迟。

情境学习（ICL）建议在提示中包含多个相关示例。这种方法鼓励LLLM通过类比学习。**思想链（CoT）**建议在上下文示例中合并一系列中间推理步骤，这有助于法学硕士进行复杂的推理。然而，这些提示技术**不可避免地会导致更长的提示**，这带来了挑战。在该领域内，相关研究分为四个方向：**提示剪枝、提示摘要、基于软提示的压缩和检索增强生成**

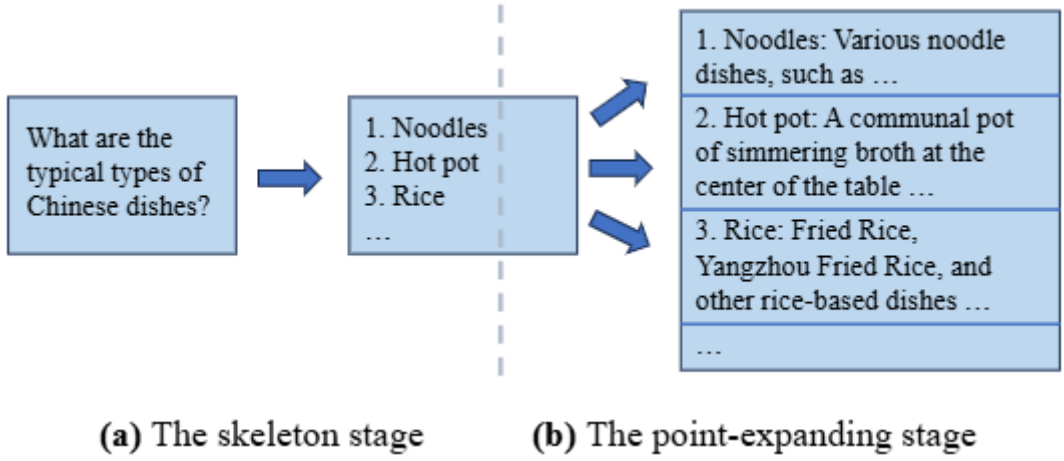
输入压缩：



- Prompt Pruning
  提示修剪背后的核心思想是根据预定义或可学习的重要性指标，从每个输入提示中在线删除不重要的标记、句子或文档。
- Prompt Summary
  提示摘要的核心思想是将原始提示压缩为较短的摘要，同时保留相似的语义信息。
- Soft Prompt-based Compression
  核心思想是设计一个比原始提示短得多的软提示，用作法学硕士的输入。
- Retrieval-Augmented Generation
  检索增强生成（RAG）[29]旨在通过整合外部知识源来提高法学硕士回答的质量。 RAG也可以被视为一种在处理大量数据时提高推理效率的技术。 RAG 不会将所有信息合并到过长的提示中，而是仅将检索到的相关信息添加到原始提示中，确保模型接收必要的信息，同时显着缩短提示长度。

输出组织：

输出组织技术旨在通过组织输出内容的结构来（部分）并行化生成。**思想骨架（SoT）** 是这个方向的先驱。 SoT 背后的核心思想是利用LLM的新兴能力来规划输出内容的结构。具体来说，SoT 由两个主要阶段组成。在第一阶段（即骨架阶段），SoT 指示 LLM使用预定义的“骨架提示”生成简明的答案骨架。例如，给出一个“中国菜的典型类型是什么？”这样的问题，此阶段的输出将是一个菜品列表（例如面条、火锅、米饭），没有详细的描述。然后，在第二阶段（即点扩展阶段），SoT 指示 LLM 使用“点扩展提示”同时扩展骨架中的每个点，然后将这些扩展连接起来形成最终答案。当应用于开源模型时，可以通过批量推理来执行点扩展，从而优化硬件利用率并使用相同的计算资源减少总体生成延迟。



(a) The skeleton stage **(b) The point-expanding stage**

SGD通过将子问题点组织成有向无环图（DAG）并在一轮中并行回答逻辑无关的子问题，进一步扩展了 SoT 的思想。

APAR采用了与SoT类似的想法，利用LLM输出特殊的控制令牌（即[fork]）来自动动态地触发并行解码。为了有效利用输出内容中固有的可并行结构并准确生成控制令牌，APAR 对在特定树结构中形成的精心设计的数据上的 LLM 进行微调。此外，APAR 将其解码方法与推测解码技术（即 Medusa）和服务系统（即 vLLM）相结合，以分别进一步提高推理延迟和系统吞吐量。 **SGLang** 在 Python 中引入了一种特定于领域的语言（DSL），其特征在于灵活地促进 LLM 编程。 **SGLang背后的核心思想**是自动分析各个生成调用之间的依赖关系，并基于此分析进行批量推理和KV缓存共享。使用这种语言，用户可以轻松实现各种提示策略，并受益于 SGLang 的自动效率优化（例如 SoT 、ToT ）。此外，SGLang 引入并结合了多种系统级编译技术，例如代码移动和预取注释。

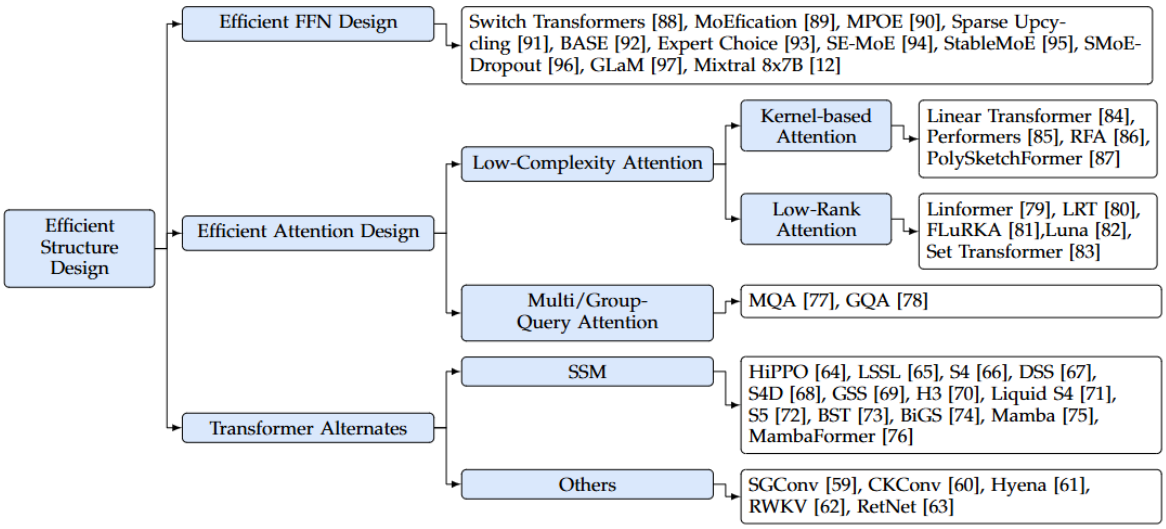
总结： 输入压缩方法主要目标是通过减少注意力操作产生的计算和内存成本来增强预填充阶段。相比之下，输出组织方法专注于通过减轻与自回归解码方法相关的大量内存访问成本来优化解码阶段。

4. 模型级优化

模型级优化主要集中在优化模型结构或数据表示。模型结构优化涉及直接设计有效的模型结构、修改原始模型以及调整推理时间架构。

4.1 高效结构设计：

FFN 在基于 Transformer 的 LLM 中贡献了很大一部分模型参数，导致显着的内存访问成本和内存使用，特别是在解码阶段。例如，FFN模块在LLaMA-7B模型中占参数的63.01%，在LLaMA-70B模型中占71.69%。
注意力操作表现出输入长度的二次复杂性，导致大量的计算成本和内存使用，特别是在处理较长的输入上下文时



- Efficient FFN Design
  在这一领域，许多研究集中于将专家混合（MoE）技术。**MoE 的核心思想**是将不同的计算预算动态分配给不同的输入令牌。在基于 MoE 的 Transformer 中，多个并行前馈网络 (FFN)（即专家）与可训练路由模块一起使用。在推理过程中，模型有选择地为路由模块控制的每个令牌激活特定的专家。 一些研究集中在FFN专家的构建上，主要集中在**优化专家权重的获取过程或使这些专家更加轻量级**以提高效率。
  另一条研究重点是改进 MoE 模型中路由模块（或策略）的设计。在之前的 MoE 模型中，路由模块经常会导致负载不平衡问题，即一些专家被分配大量令牌，而其他专家则只有几个。这种不平衡不仅浪费了未充分利用的专家的能力，从而降低了模型性能，而且还降低了推理效率。还有一些研究专注于改进基于 MoE 的模型的训练方法。

- Efficient Attention Design
  其与输入长度相关的二次复杂度导致大量的计算成本、内存访问成本和内存使用，特别是在处理长上下文时。为了解决这个问题，研究人员正在探索更有效的方法来近似原始注意力操作的功能。这些研究可以大致分为两个主要分支：多查询注意力和低复杂性注意力。

  多查询注意力
  多查询注意力（MQA）通过在不同的注意力头之间共享键（K）和值（V）缓存来优化注意力操作。该策略有效降低了推理过程中的内存访问成本和内存使用量，有助于提高 Transformer 模型的效率。Transformer 式 LLM 通常采用多头注意力（MHA）操作。此操作需要在解码阶段存储和检索每个注意力头的 K 和 V 对，导致内存访问成本和内存使用量大幅增加。 MQA 通过在不同头使用相同的 K 和 V 对来应对这一挑战，同时保持不同的查询 (Q) 值。MQA 显着降低了内存需求，对模型性能的影响极小，使其成为提高推理效率的关键策略。**分组查询注意力（GQA）**进一步扩展了 MQA 的概念，它可以被视为 MHA 和 MQA 的混合。具体来说，GQA 将注意力头分成组，为每个组存储一组 K 和 V 值。该方法不仅保留了 MQA 在减少内存开销方面的优势，而且还增强了推理速度和输出质量之间的平衡。
  低复杂性注意力
  低复杂度注意力方法旨在设计新的机制来降低每个注意力头的计算复杂度。提出了**基于内核的注意力和低秩注意力**方法，将复杂度降低到 O(n)

  - Kernel-based Attention

    $$\text{Softmax}(QK^T)V \approx \phi(Q)(\phi(K)^T V).$$

    输入 Q 和 K 矩阵首先使用核函数 φ 映射到核空间，同时保持其原始维度。利用矩阵乘法的关联特性，可以在与 Q 交互之前将 K 和 V 相乘。策略有效地将计算复杂度降低到 O(nd^2)，使其相对于输入长度呈线性。 Linear Transformer 是第一个提出基于核的注意力的工作。它采用 φ(x) = elu(x) + 1 作为核函数，其中 elu(·) 表示指数线性单元激活函数。

  - Low-Rank Attention
    低秩注意力技术在执行注意力计算之前，将 K 和 V 矩阵的token维度（即 n）压缩为更小的固定长度（即 k）。该方法基于这样的见解：**n×n 注意力矩阵通常表现出低秩属性**，使得在令牌维度上压缩它成为可能。这方面研究的主要焦点是设计有效的压缩方法，其中X可以是上下文矩阵或K和V矩阵：

    $$X \in \mathbb{R}^{n\times d} \rightarrow X' \in \mathbb{R}^{k\times d}.$$

- Transformer Alternates
  一些代表性非 Transformer 模型

Model	Training Form	Training Computational Complexity	Training Memory Complexity	Inference Form	Inference Computational Complexity	
					Prefilling	Decoding (per token)
Transformer [26]	Transformer-like	$\mathcal{O}(n^2d)$	$\mathcal{O}(n^2+nd)$	Transformer-like	$\mathcal{O}(n^2d)$	$\mathcal{O}(nd)$
S4 [66]	Convolution	$\mathcal{O}(nd^2\log n)$	$\mathcal{O}(nd)$	Recurrence	$\mathcal{O}(nd^2)$	$\mathcal{O}(d^2)$
Mamba [75]	Recurrence	$\mathcal{O}(nd^2\log n)$	$\mathcal{O}(nd)$	Recurrence	$\mathcal{O}(nd^2)$	$\mathcal{O}(d^2)$
Hyena [61]	Convolution	$\mathcal{O}(nd\log n)$	$\mathcal{O}(nd)$	Convolution	$\mathcal{O}(nd\log n)$	$\mathcal{O}(nd\log n)$
RetNet [63]	Transformer-like	$\mathcal{O}(n^2d)$	$\mathcal{O}(n^2+nd)$	Recurrence	$\mathcal{O}(nd^2)$	$\mathcal{O}(d^2)$
RWKV [62]	Recurrence	$\mathcal{O}(nd^2)$	$\mathcal{O}(nd)$	Recurrence	$\mathcal{O}(nd^2)$	$\mathcal{O}(d^2)$

在这个研究领域中，两个突出的研究方向引起了极大的关注。其中一项研究集中在**状态空间模型**（SSM)上，该模型将序列建模表示为基于 HiPPO 理论的递归变换[64]。此外，其他研究主要集中在**使用长卷积或设计类似注意力的公式**来建模序列。

State Space Model
与基于注意力的 Transformer 相比，SSM 表现出与输入序列长度相关的线性计算和存储复杂性，这提高了其处理长上下文序列的效率。在本次调查中，SSM是指满足以下两个属性的一系列模型架构：1. 基于 HiPPO和 LSSL提出的以下公式对序列进行建模：

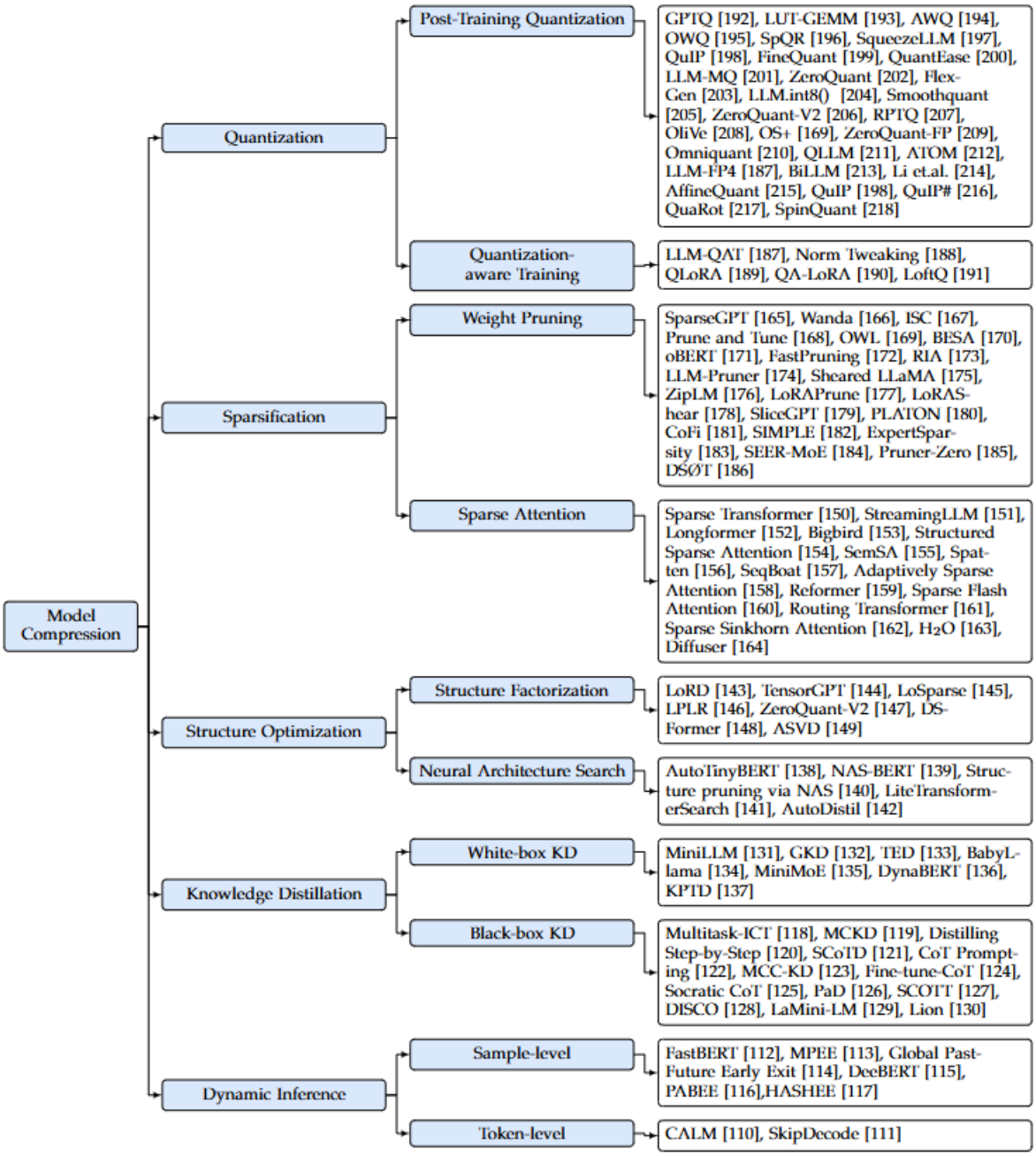
$$x_k = \overline{A}x_{k-1} + \overline{B}u_k,$$
$$y_k = \overline{C}x_k,$$

其中 A、B 和 C 表示转移矩阵，x表示中间状态，u 表示输入序列。 2. 他们基于HiPPO理论设计了转移矩阵A。具体来说，HiPPO 提出通过将输入序列投影到一组多项式基上，将其压缩为系数序列（即状态）

Other Alternates.

除了 SSM 之外，其他几种高效的替代方案也引起了人们的广泛关注，包括长卷积和类似注意力的递归操作。

4.2 模型压缩



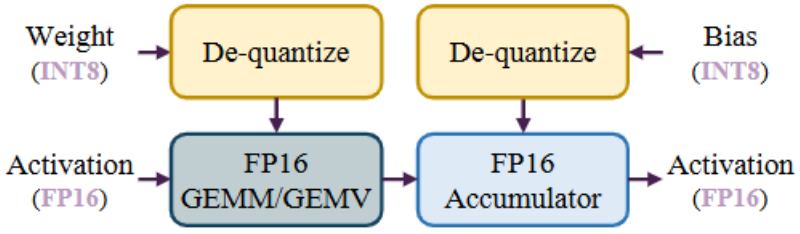
- 量化

$$\mathbf{X}_{\mathrm{INT}} = \left\lfloor \frac{\mathbf{X}_{\mathrm{FP16}} - Z}{S} \right\rceil,$$

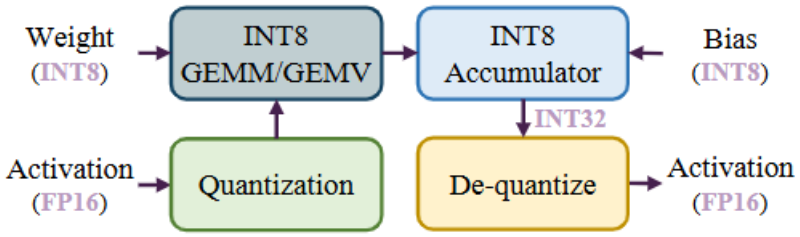
$$S = \frac{\max(\mathbf{X}_{\mathrm{FP16}}) - \min(\mathbf{X}_{\mathrm{FP16}})}{2^{N-1} - 1},$$

N表示位数，S和Z表示缩放因子和零点

LLM的推理过程涉及两个阶段：预填充阶段和解码阶段。**在预填充阶段，LLM 通常处理长令牌序列，主要操作是通用矩阵乘法（GEMM）。**预填充阶段的延迟主要受到高精度 CUDA 核心执行的计算的限制。为了应对这一挑战，现有方法对权重和激活进行量化，以使用低精度张量核心加速计算。如下图b所示，在每个GEMM操作之前在线执行激活量化，允许低精度张量核计算。这种量化方法也叫量化方法被称为**权重激活量化。**



(a) Weight-only Quantization



(b) Weight-Activation Quantization

相比之下，在解码阶段，LLM 在每个生成步骤仅处理一个标记，使用通用矩阵向量乘法 (GEMV) 作为核心运算。**解码阶段的延迟主要受到大权重张量加载的影响。**为了应对这一挑战，现有方法专注于仅量化权重以加速内存访问。这种方法称为“仅权重量化”，涉及权重的离线量化，然后将低精度权重反量化为 FP16 格式进行计算，如上图 a所示。

值得注意的是，在 LLM 中，KV 缓存代表了影响内存和内存访问的独特组件。因此，一些研究提出了 KV 缓存量化。关于数据格式，大多数算法都采用统一的格式，以便于硬件实现。在确定量化参数（例如比例、零点）时，大多数研究依赖于从权重或激活值导出的统计数据。尽管如此，一些研究工作[194]、[210]主张**根据重建损失搜索最佳参数。**此外，某些研究[192]、[194]、[205]建议在量化过程之前或期间更新未量化权重（称为量化值更新）以提高性能。

Model	Quantized Tensor Type			Data Format	Quantization Parameter Determination Scheme	Quantized Value Update
	Weight	Activation	KV Cache			
GPTQ [192]	✓			Uniform	Statistic-based	✓
LUT-GEMM [193]	✓			Non-uniform	Statistic-based	
AWQ [194]	✓			Uniform	Search-based	✓
SqueezeLLM [197]	✓			Non-uniform	Statistic-based	
LLM.int8() [204]	✓	✓		Uniform	Statistic-based	
SmoothQuant [205]	✓	✓		Uniform	Statistic-based	✓
RPTQ [207]	✓	✓		Uniform	Statistic-based	
OmniQuant [210]	✓	✓		Uniform	Search-based	
FlexGen [203]	✓		✓	Uniform	Statistic-based	
Atom [212]	✓	✓	✓	Uniform	Statistic-based	
KVQuant [219]			✓	Non-uniform	Statistic-based	
KIVI [220]			✓	Uniform	Statistic-based	

AWQ 观察到权重通道对性能的重要性各不相同，特别强调那些与激活中表现出异常值的输入通道对齐的权重通道。为了增强关键权重通道的保存，AWQ 采用了重新参数化方法。该技术通过网格搜索选择重新参数化系数，以有效地最小化重建误差。 OWQ观察到量化与激活异常值相关的权重的困难。【weight-only】

对于权重激活量化，ZeroQuant [202]对权重和激活采用更细粒度的量化，利用内核融合来最小化量化期间的内存访问成本，并进行逐层知识蒸馏以恢复性能。 [203]将权重和KV缓存直接量化到INT4中，以减少大批量推理期间的内存占用。LLM.int8()[204]识别出激活中的异常值集中在一小部分通道内。根据这一见解 LLM.int8() 根据输入通道内的异常值分布将激活和权重分为两个不同的部分，以最大限度地减少激活中的量化误差。激活和权重中包含异常值数据的通道以 FP16 格式存储，而其他通道则以 INT8 格式存储。Atom [212] 采用了一种涉及混合精度和动态量化激活的策略。值得注意的是，它扩展了这种方法，将 KV 缓存量化为 INT4，以提高吞吐量性能。 LLM-FP4 [187] 致力于将整个模型量化为 FP4 格式，并引入预移位指数偏差技术。这种方法将激活值的缩放因子与权重相结合，以解决异常值带来的量化挑战。【weight-activation】上述都是PTQ

在QAT方面，当前的研究重点是减少训练数据需求或减轻与 QAT 实施相关的计算负担的策略。为了减少数据需求，LLM-QAT [187]引入了一种无数据方法，使用原始的 FP16 LLM 生成训练数据。具体来说，LLM-QAT 使用标记化词汇表中的每个标记作为生成句子的起始标记。基于生成的训练数据，LLM-QAT 应用基于蒸馏的工作流程来训练量化的 LLM，以匹配原始 FP16 LLM 的输出分布。QLoRA [189] 将 LLM 的权重量化为 4 位，随后在 BF16 中对每个 4 位权重矩阵采用 **LoRA【了解一下】** 来微调量化模型。 QLoRA 允许在一个仅具有 30GB 内存的 GPU 上对 65B 参数 LLM 进行高效微调。

一个实验：

		TensorRT-LLM				
	B	128	256	512	1024	2048
LLaMA-2-7B	1	1.06/2.40/2.37	0.90/2.38/2.34	0.92/2.30/2.28	0.88/2.19/2.17	0.91/2.00/1.98
	2	0.88/2.10/2.05	0.91/2.07/2.04	0.89/2.01/1.98	0.91/1.92/1.89	0.88/1.78/1.76
	4	0.92/1.72/1.67	0.89/1.67/1.64	0.90/1.61/1.58	0.87/1.53/1.51	0.84/1.42/1.40
	8	0.91/1.43/1.36	0.88/1.38/1.33	0.83/1.33/1.28	0.77/1.25/1.21	0.78/1.16/1.14
	16	0.91/1.43/1.36	0.88/1.38/1.33	0.83/1.33/1.28	0.77/1.25/1.21	0.78/1.16/1.14
	B	128	256	512	1024	2048
LLaMA-2-13B	1	1.24/2.51/2.50	0.89/2.45/2.47	0.94/2.34/2.42	0.90/2.18/2.32	0.83/1.94/2.16
	2	0.90/2.51/2.50	0.95/2.45/2.47	0.90/2.34/2.42	0.83/2.18/2.32	0.80/1.94/2.16
	4	0.96/1.80/1.76	0.91/1.78/1.74	0.83/1.73/1.69	0.80/1.65/1.62	0.83/1.54/1.52
	8	0.91/1.86/1.77	0.83/1.81/1.73	0.80/1.73/1.66	0.82/1.62/1.56	0.75/1.46/1.41
	16	0.84/1.84/1.69	0.81/1.77/1.63	0.82/1.63/1.53	0.78/1.46/1.39	OOM
		LMDeploy				
	B	128	256	512	1024	2048
LLaMA-2-7B	1	1.30/2.11/2.09	0.94/2.07/2.05	0.90/2.03/2.02	0.88/1.97/1.96	0.94/1.92/1.91
	2	1.03/2.24/2.20	0.90/2.19/2.15	0.88/2.11/2.08	0.93/1.97/1.95	0.85/1.78/1.76
	4	0.90/2.18/2.10	0.87/2.12/2.05	0.93/2.01/1.96	0.92/1.86/1.83	0.92/1.64/1.62
	8	0.92/1.92/1.77	0.91/1.82/1.71	0.92/1.65/1.57	0.93/1.45/1.41	0.94/1.28/1.26
	16	0.92/1.92/1.77	0.91/1.82/1.71	0.92/1.65/1.57	0.93/1.45/1.41	0.94/1.28/1.26
	B	128	256	512	1024	2048
LLaMA-2-13B	1	1.32/2.34/2.32	0.94/2.31/2.28	0.92/2.22/2.20	0.94/2.15/2.13	0.94/2.01/1.99
	2	1.06/2.42/2.36	0.92/2.37/2.32	0.94/2.29/2.25	0.94/2.15/2.12	0.95/1.95/1.93
	4	0.93/2.36/2.26	0.94/2.29/2.21	0.94/2.18/2.12	0.95/2.01/1.97	0.96/1.78/1.75
	8	0.92/2.24/2.10	0.93/1.93/2.02	0.94/1.81/1.89	0.94/1.65/1.71	0.95/1.45/1.49
	16	0.93/2.02/1.85	0.94/1.90/1.76	0.94/1.73/1.63	0.95/1.50/1.45	OOM

Comparison of speed-ups in different scenarios (e.g., model size, batch size, input context length, inference framework) with W4A16 quantization based on TensorRT-LLM [222] and LMDeploy [223] framework, respectively. We test the speed-ups of prefilling/decoding/end-to-end latency on a single NVIDIA A100 GPU. OOM denotes "Out Of Memory".

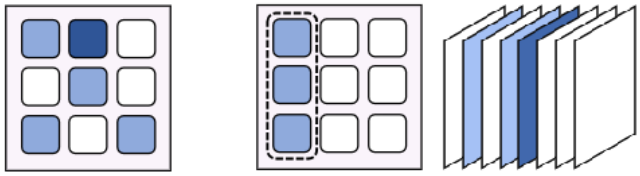
通过上述实验可以得到：(1) **仅权重量化可以大大加速解码阶段**，从而改善端到端延迟。这种增强主要源于从高带宽存储器（HBM）更快地加载具有低精度权重张量的量化模型的能力。因此，这种方法显着减少了内存访问开销。 (2) **对于预填充阶段，仅权重量化实际上可能会增加延迟。**这是因为预填充阶段的瓶颈是计算成本而不是内存访问成本。因此，仅量化权重而不量化激活对延迟的影响最小。此外，仅权重量化需要对 FP16 的低精度权重进行反量化，从而导致额外的计算开销，从而减慢预填充阶段的速度。**【那么一个问题：量化的目的是为了压缩模型 会不会考虑计算？】** (3) 随着批量大小和输入长度的增加，仅权量化实现的加速程度逐渐减小。这主要是因为，随着批量大小和输入长度的增加，计算成本构成了更大比例的延迟。**虽然仅权重量化主要降低了内存访问成本**，但随着批量大小和输入长度的增大，计算需求变得更加突出，它对延迟的影响变得不那么重

要。 (4) 由于与较大模型大小相关的显着内存访问开销，仅权重量化为较大模型提供了更大的好处。随着模型复杂性和大小的增加，存储和访问权重所需的内存量也会成比例增加。通过量化模型权重，仅权重量化有效地减少了内存占用和内存访问开销。

- 稀疏
  该方法旨在通过在计算过程中有效地忽略零元素来降低计算复杂度和内存使用量。在LLM的背景下，稀疏化通常应用于权重参数和注意力激活。它促使了**权重剪枝策略和稀疏注意力机制**的发展。

  权重剪枝

  权重修剪系统地从模型中删除不太关键的权重和结构，旨在减少预填充阶段和解码阶段的计算和内存成本，而不会显着影响性能。这种稀疏化方法分为两种主要类型：非结构化修剪和结构化修剪。



  一种剪枝标准是最小化模型的重构损失。

  SparseGPT [165]是该领域的代表性方法。它遵循OBS[225]的思想，考虑去除每个权重对网络重建损失的影响。 OBS迭代地决定剪枝掩模来剪枝权重并重建未剪枝的权重以补偿剪枝损失。SparseGPT通过最优部分更新技术克服了OBS的效率瓶颈，并设计了一种基于OBS重建误差的**自适应mask选择**技术。 Prune and Tune [168] 通过在修剪期间用最少的训练步骤微调 LLM，改进了SparseGPT。 ISC[167]通过结合OBS[225]和OBD[226]中的显着性标准设计了一种新颖的剪枝标准。它进一步根据 **Hessian 信息**为每一层分配**非均匀剪枝比率**。 oBERT [171]和FastPruning [172]利用损失函数的二阶信息来决定剪枝权重。 BESA [170] 通过重建损失的梯度下降来学习可微分的二元掩模。每层的剪枝比率是通过最小化重建误差来顺序决定的。另一种流行的修剪标准是基于幅度的。 **Wanda** [166]建议使用权重大小和输入激活范数之间的元素乘积作为剪枝标准。 RIA [173]通过使用相对重要性和激活的度量来共同考虑权重和激活，该度量根据所有相关的权重来评估每个权重元素的重要性。此外，**RIA将非结构化稀疏模式转换为结构化N:M稀疏模式，可以在NVIDIA GPU上享受实际的加速。** 【以上是 非结构化剪枝】

  LLM-Pruner [174]提出了一种与**任务无关的**结构化剪枝算法。具体来说，它首先根据神经元之间的连接依赖关系识别LLM中的耦合结构。然后，它根据精心设计的分组修剪指标来决定删除哪些结构组。剪枝后，它进一步提出通过参数高效的训练技术（即 LoRA ）来恢复模型性能。LoRAPrune [177] 为带有 LoRA 模块的预训练 LLM 提出了一个结构化剪枝框架，以实现基于 LoRA 的模型的快速推理。它设计了一个使用LoRA的权重和梯度的LoRA引导的剪枝准则，以及基于该准则去除不重要权重的迭代剪枝方案。 LoRAShear [178] 还为基于 LoRA 的 LLM 设计了一种剪枝方法。

  稀疏注意力

  Transformer 模型的多头自注意力（MHSA）组件中的稀疏注意力技术策略性地省略了某些注意力计算，以提高主要在预填充阶段的注意力操作的计算效率。

  静态稀疏注意力独立于特定输入而删除激活值。这些方法预先确定稀疏注意掩码，并在推理过程中将其强制执行到注意矩阵上。先前的研究结合了不同的稀疏模式来保留每个注意力矩阵中最基本的元素。如图 下图(a) 所示，最常见的稀疏注意力模式是**局部和全局注意力模式。局部注意模式通过围绕每个标记的固定大小的窗口注意来捕获每个标记的局部上下文。全局注意力模式通过计算和关注序列中的所有标记来捕获特定标记与所有其他标记的相关性。**请注意，利用**全局模式可以消除未使用的令牌存储键值 (KV) 对的需要**，从而减少解码阶段的内存访问成本和内存使用。 局部、全局和随机模式的组合被证明可以封装所有连续的序列到序列函数，证实了其图灵完备性。如图（b）所示，Longformer [152]另外引入了扩张的滑动窗口模式。它类似于扩张的 CNN，使滑动窗口“扩张”以增加感受野。

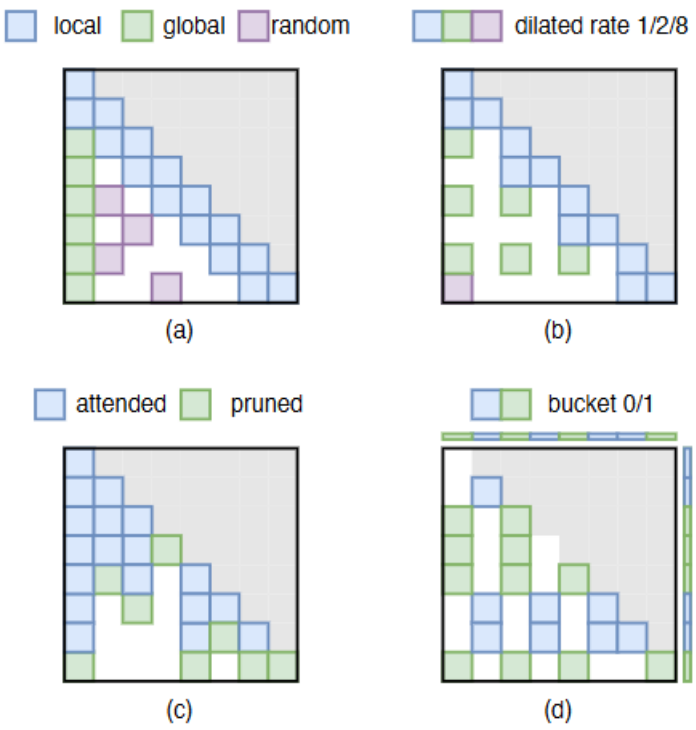


Fig. 11. Examples of different sparse attention masks. (a) Static mask with local, global, and random attention pattern. (b) Static mask with dilated attention pattern of different dilated rate. (c) Dynamic token pruning. (d) Dynamic attention pruning.

相比之下，**动态稀疏注意力**根据不同的输入自适应地消除激活值，利用神经元激活值的实时监控来绕过对影响可忽略不计的神经元的计算，从而实现剪枝。大多数动态稀疏注意力方法都采用动态标记修剪方法，如上图(c) 所示。

除了动态标记剪枝之外，还采用了动态注意力剪枝策略[159]，[160]，[161]，[162]，[163]。如图（d）所示，这些方法不是修剪某些标记的所有注意力值，而是根据输入动态修剪注意力的选择性部分。该领域中的一个突出方法是将输入令牌动态分割为组（称为存储桶），并策略性地忽略驻留在单独存储桶中的令牌的注意力计算。

除了**注意力级别和令牌级别的稀疏性**之外，注意力修剪的范围还扩展到各种粒度。 Spatten [156]还将修剪从令牌粒度扩展到了**注意头粒度**，消除了对不重要的注意头的计算，以进一步减少计算和内存需求。

- 结构优化

  结构优化的目标是细化模型架构或结构，以增强模型效率和性能之间的平衡。在这一研究领域中，有两种突出的技术脱颖而出：神经架构搜索（NAS）和低阶分解（LRF）。

Neural Architecture Search

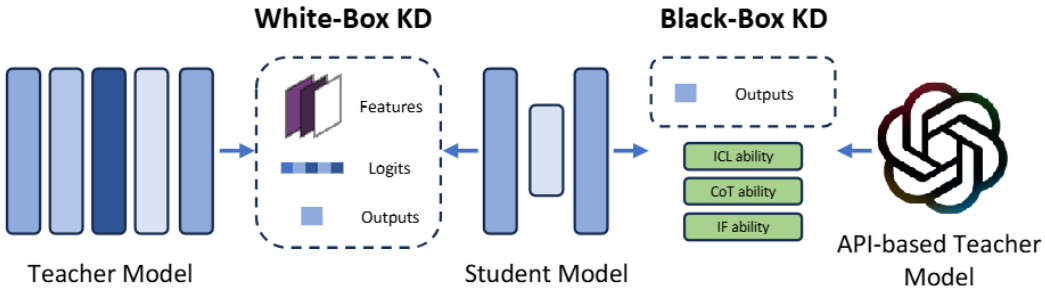
神经架构搜索（NAS）[228]旨在自动搜索在效率和性能之间取得优化平衡的最佳神经架构。

Low Rank Factorization

低秩分解 (LRF) 旨在用两个低秩矩阵 B 和 C近似矩阵 A： $A^{m\times n} \approx B^{m\times r} \times C^{r\times n}$ 其中r远小于m和n

LRF 可以减少内存使用并提高计算效率。此外，在LLM推理的解码阶段，内存访问成本成为解码速度的瓶颈。因此，**LRF可以减少需要加载的参数数量，从而加快解码速度。** LoRD [143] 显示了通过 LRF 压缩 LLM 而不大幅降低性能的潜力。具体来说，它采用**奇异值分解（SVD）来分解权重矩阵**，并成功地将具有16B参数的LLM压缩到12.3B，同时性能下降最小。 TensorGPT [144]引入了一种使用张量训练分解来压缩嵌入层的方法。每个令牌嵌入都被视为矩阵乘积状态（MPS），并以分布式方式有效计算。

- 知识蒸馏
  在LLM的背景下，KD 涉及使用原始LLM作为教师模型来提炼较小的LLM。许多研究都致力于有效地将LLM的各种能力转移到更小的模型上。在这个领域，方法可以分为两种主要类型：白盒KD和黑盒KD（如下图所示）



白盒蒸馏

白盒 KD 是指利用对教师模型的结构和参数的访问的蒸馏方法。这种方法使 KD 能够有效地利用教师模型的中间特征和输出逻辑来增强学生模型的性能。

黑盒蒸馏

黑盒KD是指教师模型的结构和参数不可用的知识蒸馏方法。通常，黑盒 KD 仅使用教师模型获得的最终结果来提取学生模型。在LLM领域，黑盒KD主要引导学生模型学习LLM的泛化能力和涌现能力，包括情境学习（ICL： In-Context - Learning）能力、思想链（CoT）推理能力和指令跟随（IF： Instruction Following）能力。

- 动态推理
  动态推理涉及在推理过程中**根据输入数据自适应选择模型子结构。**本节重点介绍早期现有的技术，这些技术使法学硕士能够**根据特定的样本或标记在不同的模型层停止推理。**值得注意的是，虽然 MoE 技术也在推理过程中调整模型结构，但它们通常涉及昂贵的预训练成本。相比之下，早期的技术只需要训练一个小模块来确定何时得出推理。本文将LLM提前退出技术的研究分为两种主要类型：**样本级别提前退出和令牌级别提前退出**（如下图所示）。

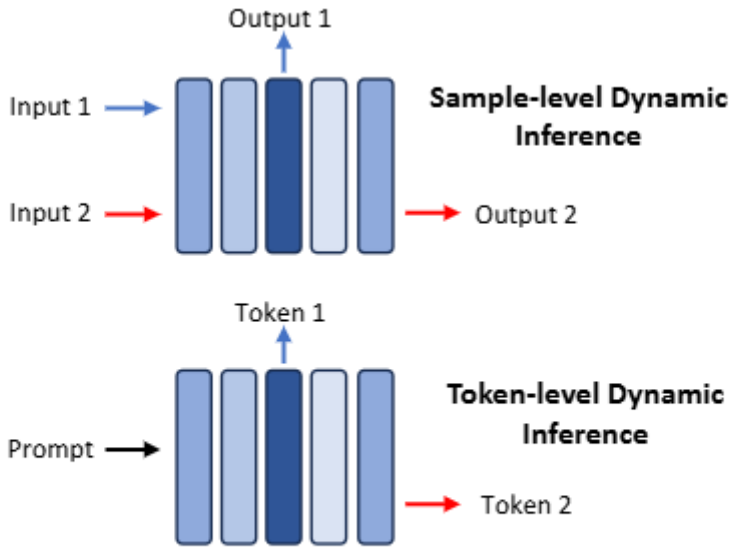


Fig. 13. Illustration of Token-level (up) and Sample-level (down) dynamic inference.

Sample- Level

样本级别提前退出技术侧重于确定单个输入样本的语言模型 (LLM) 的最佳大小和结构。一种常见的方法是在每一层之后使用额外的模块来增强LLM，利用这些模块来决定是否在某一层终止推理。FastBERT [112]、DeeBERT [115]、MP [233] 和 MPEE [113] 直接训练这些模块以根据当前层的特征做出决策（例如，输出 0 继续或 1 停止）。

Token-Level

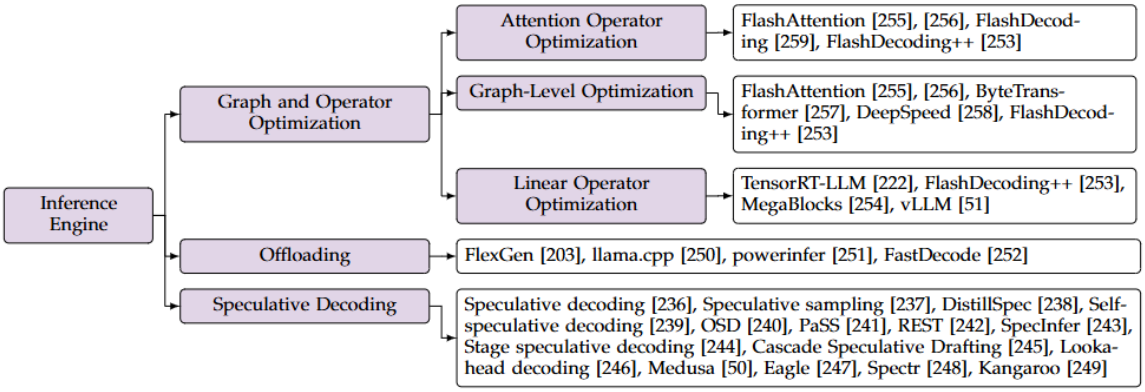
在LLM推理的解码阶段，token是顺序生成的，令牌级提前退出技术旨在优化每个输出令牌的LLM的大小和结构。 CALM [110] 在每个 Transformer 层之后引入早期退出分类器，训练它们输出置信度分数，以确定是否在特定层停止推理。值得注意的是，在 self-attention 块中，计算每层当前 token 的特征依赖于同一层中所有先前 token 的特征（即 KV 缓存）。**为了解决由于先前令牌提前退出而导致 KV 缓存缺失的问题，CALM 提出直接将特征从退出层复制到后续层，实验结果显示仅出现轻微的性能下降。** SkipDecode [111] 解决了先前早期现有方法的局限性，这些局限性阻碍了它们对批量推理和 KV 缓存的适用性，从而限制了实际的加速增益。对于批量推理，SkipDecode 为批量内的所有令牌提出了统一的退出点。关于KV缓存，SkipDecode确保退出点单调减少，以防止KV缓存重新计算，有利于推理过程中的效率提升。

5. 系统级优化

LLM推理的系统级优化主要涉及增强模型前向传递。考虑到 LLM 的计算图，存在多个运算符，其中**注意力运算符和线性运算符主导了大部分运行时间。**系统级优化主要考虑注意力算子的独特特征和LLM中的解码方法。特别是，为了解决与LLM解码方法相关的具体问题，**线性算子需要特殊的平铺设计【了解一下】**，并且提出了推测解码方法来提高利用率。 LLM 的大量内存需求导致参数或 KV 缓存卸载到 CPU。此外，在在线服务的背景下，请求来自多个用户。因此，除了前面讨论的优化之外，在线服务还面临着异步请求带来的内存、批处理和调度方面的挑战。

5.1 推理引擎

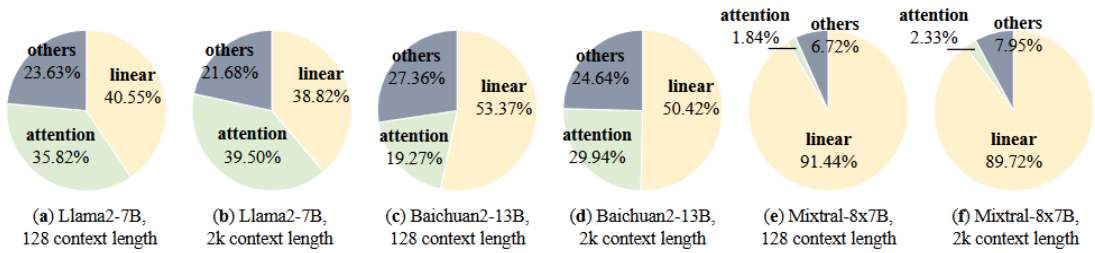
推理引擎的优化致力于加速模型前进过程。 LLM推理中的主要算子和计算图都经过高度优化。此外，提出了**推测解码**技术来加速推理速度而不降低性能，并引入**卸载技术**来减轻内存压力



- 图和算子优化

  Runtime Profile

  借助于HuggingFace，我们使用不同的模型和上下文长度来分析推理运行时。



(a) Llama2-7B, 128 context length (b) Llama2-7B, 2k context length (c) Baichuan2-13B, 128 context length (d) Baichuan2-13B, 2k context length (e) Mixtral-8x7B, 128 context length (f) Mixtral-8x7B, 2k context length

上图中的分析结果表明，**注意力算子和线性算子共同主导了运行时间**，它们的总持续时间通常**超过推理持续时间的 75%。**因此，算子级别的优化工作的很大一部分致力于提高两个算子的性能。此外，其他算子占用的运行时间比例较小，导致算子执行时间线碎片化，**增加了CPU侧内核启动的成本【原因如下】**。为了解决这个问题，在计算图级别，当前优化的推理引擎实现了高度融合的算子。

在深度学习模型的推理过程中，尽管大多数计算密集型任务（如矩阵乘法和卷积运算）通常在GPU或其他加速器上执行，但CPU仍然在整个过程中扮演着关键角色，特别是在调度和管理方面。这里是为什么CPU会负责算子执行时间碎片化引起的内核启动和调度的几个原因：

1. **任务调度和管理**：CPU 通常负责管理和调度计算任务，包括启动、调度和终止各种算子。即使主要的计算在 GPU 上进行，CPU 仍然需要控制任务的协调和调度。例如，当多个算子需要依次运行时，CPU 负责确定任务的顺序和调度时间。
2. **控制和同步**：在深度学习推理中，CPU 负责协调 GPU 或其他加速器的工作。对于每个算子，CPU 需要发出指令以启动 GPU 上的计算，监控其进度，并在任务完成后进行同步。碎片化的执行时间意味着 CPU 频繁地启动和停止这些任务，从而导致更高的开销。
3. **数据传输和内存管理**：CPU 通常还负责处理数据传输的准备工作，将数据从内存中提取并准备发送到 GPU 或其他加速器。如果算子执行时间碎片化，意味着 CPU 需要频繁地进行数据准备和传输管理，这也增加了开销。
4. **轻量级算子的执行**：一些轻量级的算子可能直接在 CPU 上执行，特别是当这些算子的计算量不足以让 GPU 高效利用时。因此，这些小算子执行时间的碎片化直接影响 CPU 的负载和调度。

注意力算子优化

如上图所示，**注意力算子的时间比例随着上下文长度的增加而增加。**这意味着对内存大小和计算能力的高要求，尤其是在处理长序列时。为了解决 GPU 上标准注意力计算的计算和内存开销**，定制注意力算子**至关重要。 **FlashAttention将整个注意力操作融合到一个单一的、内存高效的运算符中，以减轻内存访问开销。输入矩阵（Q，K，V）和注意力矩阵被平铺为多个块，这消除了完整数据加载的需要。**FlashDecoding [259] 建立在 FlashAttention 的基础上，旨在最大化解码的计算并行性。由于解码方法的应用，Q矩阵在解码过程中退化为一批向量，如果并行性仅限于批量大小维度，则这使得填充计算单元变得具有挑战性。 FlashDecoding 通过沿序列维度引入并行计算来解决这个问题。虽然这给 softmax 计算带来了一些同步开销，但它导致了并行性的显着改进，特别是对于小批量和长序列。随后的工作 FlashDecoding++ [253] 观察到，在之前的工作 [255]、[256]、[259] 中，softmax 内的最大值仅充当缩放因子以防止数据溢出。**FlashDecoding++提出提前根据统计确定缩放因子。这消除了 softmax 计算中的同步开销，从而能够与 softmax 计算一起并行执行后续操作。**

线性算子优化

线性算子在 LLM 推理中发挥着关键作用，在特征投影和前馈神经网络 (FFN) 中发挥作用。在传统的神经网络中，线性算子可以抽象为通用矩阵-矩阵乘法**（General Matrix-Matrix Multiplication GEMM）**运算。然而，**就 LLM 而言，解码方法的应用导致维度显着减小，与传统的 GEMM 工作负载不同。**传统GEMM的底层实现已经高度优化，主流LLM框架（例如DeepSpeed[258]、vLLM[51]、OpenPPL[263]等）主要调用cuBLAS[264]提供的GEMM API来实现线性算子。**如果没有针对降维 GEMM 进行明确定制的实现，解码过程中的线性算子就会效率低下。**在最新版本的 TensorRT-LLM中观察到了解决该问题的显着趋势。它引入了专用的**通用矩阵向量乘法（General Matrix-Vector Multiplication GEMV）实现**，有可能提高解码步骤的效率。最近的研究 FlashDecoding++ [253] 更进一步，解决了 cuBLAS 和 CUTLASS 库在解码步骤中处理小批量时效率低下的问题。作者首先引入了 FlatGEMM 操作的概念，以高度缩减的维度（FlashDecoding++ 中的维度大小 < 8）表示 GEMM 的工作负载。由于 FlatGEMM 提出了新的计算特性，传统 GEMM 的平铺策略需要进行修改。作者观察到，随着工作负载的变化，存在两个挑战：低并行性和内存访问瓶颈。为了应对这些挑战，FlashDecoding++采用细粒度的平铺策略来提高并行性，并利用双缓冲技术来隐藏内存访问延迟。此外，认识到典型 LLM（例如 Llama2 [261]、ChatGLM [262]）中的线性运算通常具有固定的形状，FlashDecoding++ 建立了一种启发式选择机制。该机制根据输入大小动态选择不同的线性运算符。这些选项包括 cuBLAS [264]、[265] 库提供的 FastGEMV [266]、FlatGEMM 和 GEMM。这种方法确保为给定的线性工作负载选择最有效的运算符，从而可能带来更好的端到端性能

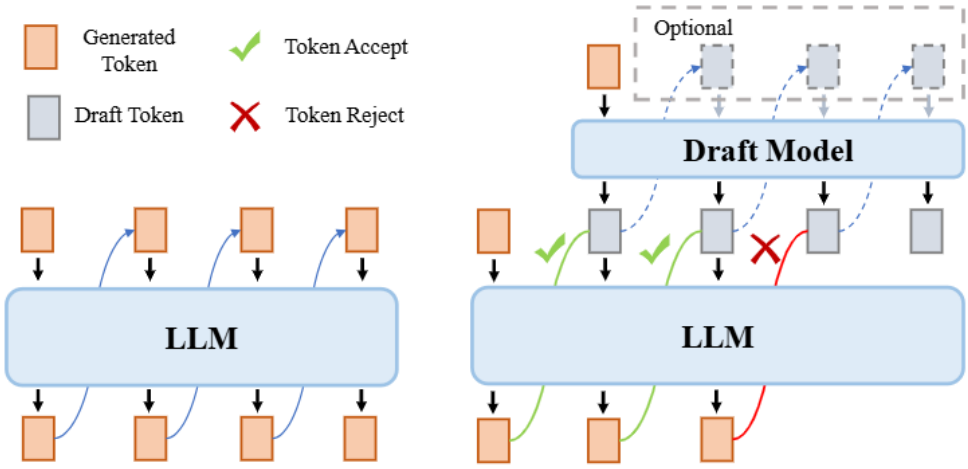
最近，应用MoE FFN来增强模型能力已成为法学硕士的趋势[12]。这种模型结构也对算子优化提出了新的要求。如上图 所示，在具有 MoE FFN 的 Mixtral 模型中，由于 HuggingFace 实现中的非优化 FFN 计算，线性算子在运行时占据主导地位。此外，Mixtral采用GQA注意力结构，降低了注意力算子的运行时间比例，**这进一步指出了优化FFN层的迫切需要。 MegaBlocks [254] 是第一个优化 MoE FFN 层计算的。**这项工作将 MoE FFN 计算公式化为块稀疏操作，并提出了定制的 GPU 内核以进行加速。然而，MegaBlocks 专注于 MoE 模型的有效训练，因此忽略了推理的特征（例如解码方法）。现有框架正在努力优化 MoE FFN 推理阶段的计算。 vLLM [51] 的官方存储库在 Triton [267] 中集成了 MoE FFN 的融合内核，无缝地消除了索引开销。

图级别优化

内核融合因其能够减少运行时间而成为流行的图级优化。 **【为什么 kernel fusion能被看作 Graph-Level Optimization?** 以下是一个可能的解释：内核融合可以被视为图级别优化的后续步骤或实现方式。图级别优化可能识别出可以融合的算子，然后在硬件实现（如 GPU）上通过内核融合来实际执行这些优化。在一些高性能的深度学习编译器或框架（如 TVM、TensorRT、XLA）中，内核融合确实被作为一种图级别优化策略来处理。**编译器在处理计算图时，可以自动识别和重写计算图以使用内核融合技术，将多个算子合并为一个内核执行。**在一些高性能的深度学习编译器或框架（如 TVM、TensorRT、XLA）中，内核融合确实被作为一种图级别优化策略来处理。编译器在处理计算图时，可以自动识别和重写计算图以使用内核融合技术，将多个算子合并为一个内核执行。】

应用kernel fusion有三个主要优点：（1）**减少内存访问。**融合内核本质上消除了中间结果的内存访问，减轻了运算符的内存瓶颈。 (2) **减轻内核启动开销。**对于一些轻量级算子（例如残差添加），内核启动时间占据了大部分延迟，而内核融合减少了单个内核的启动。 (3)**增强并行性。**对于那些没有数据依赖性的算子，当一对一的内核执行无法填满硬件容量时，通过融合并行内核是有利的。kernel fusion技术在 LLM 推理中被证明是有效的，具有上述所有优点。 FlashAttention [255]将注意力算子表述为一个内核，消除了访问注意力结果的开销。基于注意力算子受内存限制的事实，内存访问的减少有效地转化为运行时加速。 ByteTransformer [257]和DeepSpeed [258]建议将包括残差加法、层范数和激活函数在内的轻量级算子融合到以前的线性算子中，以减少内核启动开销。结果，这些轻量级运算符在时间线中消失，几乎没有额外的延迟。此外，还采用核融合来增强LLM推理的利用率。查询、键和值矩阵的投影最初是三个单独的线性运算，并融合为一个线性运算符以部署在现代 GPU 上。目前，kernel fusion技术已在LLM推理实践中得到应用，高度优化的推理引擎在运行时仅使用少量融合核。例如，在 FlashDecoding++ [253] 实现中，transformer block仅集成了七个融合内核。利用上述运算符和内核融合优化，FlashDecoding++ 比 HuggingFace 实现实现高达 4.86 倍的加速

- 推测解码
  推测解码是一种用于自回归LLM的创新解码技术，旨在提高解码效率而不影响输出的保真度。这种方法的核心思想涉及采用一个较小的模型（称为草稿模型）来有效地预测几个后续标记，然后使用目标 LLM 并行验证这些预测。该方法旨在使法学硕士能够在单个推理通常所需的时间范围内生成多个令牌。分两个步骤 ——Draft construction 和 Draft verification



(a) Auto-regressive Decoding **(b)** Speculative Decoding

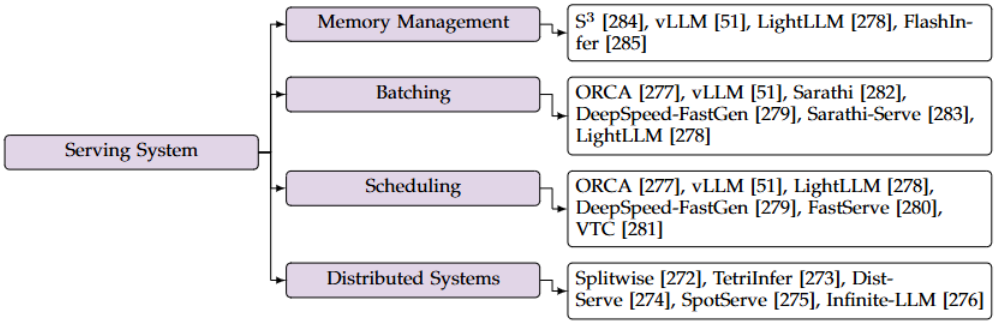
接受率代表每个推理步骤接受的草稿令牌的平均数量，是评估推测解码算法性能的关键指标
传统的解码技术通常采用两种初级采样：**贪婪采样和核采样(greedy sampling and nucleus sampling)。**贪婪采样涉及在每个解码步骤中选择概率最高的令牌来生成特定的输出序列。推测性解码的最初尝试称为“块式并行解码”[270]，旨在确保草稿token与通过贪婪采样采样的token精确匹配，从而保持输出token等价性。相比之下，核心采样涉及从概率分布中采样token，从而在每次运行中产生不同的token序列。这种多样性使得核采样很受欢迎。为了在推测性解码框架内适应核采样，已经提出了推测性采样技术[236]、[237]。推测采样保持输出分布的等价性，与核心采样的概率性质相一致，以生成不同的token序列。

token tree verifier[243]已成为这种情况下广泛采用的验证策略。该方法利用草稿令牌集的树结构表示，并采用树注意机制来有效地执行验证过程。在推测性解码方法中，草案代币的接受率受到草案模型的输出分布与原始法学硕士的输出分布一致程度的显着影响。因此，大量的研究工作集中在改进草案模型的设计上。DistillSpec [238] 直接从目标 LLM 中提取出较小的草稿模型。 SSD [239] 涉及从目标 LLM 自动识别子模型（模型层的子集）作为草稿模型，消除了对草稿模型进行单独训练的需要。
Spectr [248]提倡生成**多个草案token序列**，并采用 k 顺序草案选择技术来**同时验证 k 个序列。**该方法利用推测抽样，确保输出分布的等效性。类似地，**SpecInfer** [243] 采用了类似的方法。然而，与 Spectr 不同的是，SpecInfer 将草稿令牌序列合并到“令牌树”中，并引入了用于验证的树注意机制。这种策略被称为“令牌树验证器”。由于其功效，令牌树验证器已广泛应用于众多推测性解码算法中[50]、[242]、[244]、[247]。除了这些努力之外，阶段推测解码[244]和级联推测起草（CS Drafting）[245]建议通过将推测解码直接集成到令牌生成过程中来加速草稿构建。

- Offloading
  卸载的本质是在空闲时将部分存储从GPU卸载到CPU。直观上，此类研究的重点在于隐藏 GPU 和 CPU 之间昂贵的数据移动延迟。 **FlexGen [203] 能够卸载权重、激活和 KV 缓存，并进一步制定用于卸载的图遍历问题以最大化吞吐量。**下一批的数据加载和上一批的数据存储可以与当前批次的计算重叠。另一项工作 **llama.cpp [250] 也将计算任务分配给 CPU，以低功耗 CPU 进行计算为代价来减轻数据传输开销。 Powerinfer** [251] 在 LLM 中使用 ReLU [271] 利用激活的稀疏性，并将激活分为代表计算频率的冷和热神经元的子集。冷神经元被卸载到 CPU，用于 Powerinfer 中的存储和计算。利用自适应预测器和稀疏算子，Powerinfer 通过卸载显着提高了计算效率。 **FastDecode [252]建议将整个注意力算子的存储和计算卸载到CPU。由于注意力操作是在CPU上计算的，KV缓存的数据移动被减少到仅仅一些激活。**选择 CPU 数量以匹配 GPU 上的工作负载延迟，从而减少异构管道中的气泡。

5.2 Serving System

服务系统的优化致力于**提高处理异步请求的效率**。优化**内存管理**以容纳更多请求，并集成高效的**批处理和调度策略**以提高系统吞吐量。此外，还提出了针对**分布式系统**的优化来利用分布式计算资源。



- 内存管理

  KV 缓存的存储在 LLM 服务中主导着内存使用，特别是当上下文长度很长时。由于生成长度不确定，提前分配KV缓存存储空间具有挑战性。早期的实现通常根据每个请求的预设最大长度提前分配存储空间。然而，在请求生成提前终止的情况下，这种方法会导致存储资源的严重浪费。为了解决这个问题，S3[284]建议预测每个请求的生成长度的上限，以减少预分配空间的浪费。然而，当不存在这么大的连续空间时，静态的KV缓存内存分配方式仍然会失败。为了处理碎片存储，**vLLM [51]建议按照操作系统以分页方式存储 KV 缓存(Paged attention)。** vLLM首先分配尽可能大的内存空间，并将其均分为多个物理块。当请求到来时，vLLM 以不连续的方式动态地将生成的 KV 缓存映射到预先分配的物理块。通过这种方式，vLLM显着减少了存储碎片，并在LLM服务中实现了更高的吞吐量。在vLLM的基础上，LightLLM[278]使用更细粒度的KV缓存存储来减少不规则边界发生的浪费。 LightLLM 将令牌的 KV 缓存视为一个单元，而不是块，因此生成的 KV 缓存始终使预分配的空间饱和。

  当前优化的服务系统通常采用这种分页方法来管理KV缓存存储，从而减少冗余KV缓存的浪费。然而，**分页存储导致注意力算子中的内存访问不规则。对于使用分页KV缓存的注意力算子，这需要考虑KV缓存的虚拟地址空间与其对应的物理地址空间之间的映射关系。为了提高注意力算子的效率，必须调整 KV 缓存的加载模式以促进连续内存访问。**

- 连续批处理

  批次中的请求长度可能不同，当较短的请求完成而较长的请求仍在运行时，会导致利用率较低。由于服务场景中请求的异步性质，存在可以缓解这种低利用率时期的机会。提出了连续批处理技术，以便在一些旧请求完成后通过批处理新请求来利用这一机会。 ORCA [277] 是第一个在 LLM 服务中使用连续批处理技术的人。每个请求的计算包含多次迭代，每次迭代代表预填充步骤或解码步骤。作者建议可以在迭代级别对不同的请求进行批处理。这项工作在线性运算符中实现了迭代级批处理，在序列维度中将不同的请求连接在一起。因此，与已完成的请求相对应的空闲存储和计算资源被及时释放。继 ORCA 之后，vLLM [51]将该技术扩展到注意力计算，使得具有不同 KV 缓存长度的请求能够批量处理在一起。

- 调度策略

  在LLM服务中，每个请求的作业长度表现出可变性，因此执行请求的顺序会显着影响服务系统的吞吐量。当长请求被赋予优先级时，会发生队头阻塞[280]。具体来说，内存使用量会因响应长请求而快速增长，从而在系统耗尽内存容量时阻碍后续请求。开创性的工作 ORCA [277] 和开源系统，包括 vLLM [51] 和 LightLLM [278]，采用简单的先到先服务（FCFS）原则来安排请求。 DeepSpeed-FastGen [279]优先考虑解码请求以增强性能。 FastServe [280]提出了一种抢占式调度策略来优化队头阻塞问题，在LLM服务中实现较低的作业完成时间（JCT）。 FastServe 采用多级反馈队列 (MLFQ) 对剩余时间最短的请求进行优先级排序。由于自回归解码方法会导致未知的请求长度，因此 FastServe 首先预测长度，并利用跳跃连接方式为每个请求找到适当的优先级。与之前的工作不同，VTC [281]讨论了 LLM 服务的公平性。 VTC引入了基于代币数量的成本函数来衡量客户端之间的公平性，并进一步提出了公平调度器来保证公平性。

- 分布式系统
  为了实现高吞吐量，LLM服务通常部署在分布式平台上。最近的工作还侧重于通过利用分布式特性来优化此类推理服务的性能。**值得注意的是，观察到预填充和解码的计算相互干扰另外，splitwise [272]、TetriInfer [273] 和 DistServe [274] 展示了分解请求的预填充和解码步骤的效率。**通过这种方式，两个不同的步骤根据它们的特性被独立处理。 ExeGPT [287]也采用了这种分解架构，并提出了具有可控变量的不同策略，以在一定延迟约束下最大化系统吞吐量。 Llumnix [288] 在运行时重新安排请求以实现不同的服务目标，包括负载平衡、碎片整理和优先级排序。 Spot-Serve [275] 旨在通过可抢占的 GPU 实例在云上提供 LLM 服务。 SpotServe 能够有效应对动态并行控制和实例迁移等挑战，并利用 LLM 的自回归特性来实现令牌级状态恢复。此外，Infinite-LLM [276]在整个数据中心的注意力算子中并行序列的不同部分，以解决服务极长上下文时的挑战。LoongServe [289]提出弹性序列并行性来管理弹性资源需求迭代级别，通过精心设计的调度减少KV缓存的数据移动。

5.3 硬件加速器设计(FACT ALLO FlightLLM)

5.4 LLM 框架对比

TABLE 6
Comparison of multiple open-source inference engines and serving systems. "-" denotes no serving support. Note that the scheduling method of TensorRT-LLM is not open-sourced.

Model	Inference Optimization				Inference (token/s)	Serving Optimization			Serving (req/s)
	Attention	Linear	Graph	Speculative Decoding		Memory	Batching	Scheduling	
HuggingFace [260]				✓	38.963	-	-	-	-
DeepSpeed [258]	✓		✓		80.947	blocked	split-and-fuse	decode prioritized	6.78
vLLM [51]	✓			✓	90.052	paged	continuous batching	prefill prioritized	7.11
OpenPPL [263]	✓		✓		81.169	-	-	-	-
FlashDecoding++ [253]	✓	✓	✓		106.636	-	-	-	-
LightLLM [278]	✓				73.599	token-wise	split-and-fuse	prefill prioritized	10.29
TensorRT-LLM [222]	✓	✓	✓	✓	92.512	paged	continuous batching	-	5.87

上表比较了多个 LLM 框架的性能。推理吞吐量是用 Llama2-7B 测量的（批量大小=1，输入长度=1k，输出长度=128)。服务性能是在 ShareGPT [297] 数据集上测量的最大吞吐量。**两者【两个moe？】**均源自单个 NVIDIA A100 80GB GPU。在提到的框架中，DeepSpeed [258]、vLLM [51]、LightLLM [278] 和 TensorRT-LLM [222] 集成了服务功能来服务来自多个用户的异步请求。我们还在表中列出了每个框架的优化。除了 HuggingFace 之外的所有框架都实现了算子级或图级优化来增强性能，其中一些框架还支持推测解码技术。请注意，当我们测量所有框架的推理性能时，**推测解码技术已关闭**。推理吞吐量的结果表明，FlashDecoding++ 和 TensorRT-LLM 在涵盖主要运算符和计算图的优化方面优于其他算法。从服务的角度来看，**所有框架都使用细粒度和不连续的存储来进行KV缓存，并应用连续批处理技术来提高系统利用率。**与 vLLM 和 LightLLM 不同，**DeepSpeed 在调度时优先考虑解码请求**，这意味着如果批次中有足够的现有解码请求，则不会合并新请求

最近，算子优化已与实际服务场景紧密结合，例如，专为**前缀缓存设计的RadixAttention [52]【了解一下】**

6. 关键应用场景

6.1 Agent & Multi-Model Framework

越来越多的研究趋势[298]专注于将人工智能代理扩展到多模态领域，通常利用大型多模态模型（LMM）作为这些代理系统的核心。为了提高这些新兴的基于 LMM 的智能体的效率，为 LMM 设计优化技术是一个有前途的研究方向。

6.2 Long-Context LLMs

值得注意的是，具有次二次或线性复杂度的非 Transformer 架构最近引起了研究人员的极大兴趣。尽管它们效率很高，但与 Transformer 架构相比，这些新颖架构在各种能力（例如上下文学习能力和long-range建模能力）方面的竞争力仍然受到审查[76]，[299]。因此，从多个角度探索这些新架构的功能并解决其局限性仍然是一个有价值的追求。此外，确定各种场景和任务所需的上下文长度以及确定未来将作为LLM基础骨干的下一代架构至关重要。

6.3 Edge Scenario Deployment

MiniCPM [309] 进行沙箱实验来确定最佳的预训练超参数。 PanGu-π-Pro [302] 建议使用模型剪枝的指标和技术从预训练的 LLM 初始化模型权重。 MobileLLM[310]采用"深而薄"的架构进行小模型设计，并提出跨不同层的权重共享以增加层数，而无需额外的内存成本。然而，小型模型和大型模型之间仍然存在性能差距，需要未来的研究来缩小这一差距。未来，迫切需要研究识别边缘场景下的模型规模限制，并探索各种优化方法在设计较小模型时的边界。除了设计更小的模型之外，系统级优化还为 LLM 部署提供了一个有希望的方向。最近一个著名的项目 **MLC-LLM [311] 成功在手机上部署了 LLaMA-7B 模型。** MLC-LLM 主要采用融合、内存规划和循环优化等编译技术来增强推理过程中的延迟并降低内存成本。此外，采用云边协作技术，或者设计更复杂的硬件加速器也可以帮助将LLM部署到边缘设备上

6.4 Security-Efficiency Synergy

一个有前途的方向将涉及开发新的优化方法或改进现有的方法，以在LLM的效率和安全性之间实现更好的权衡。

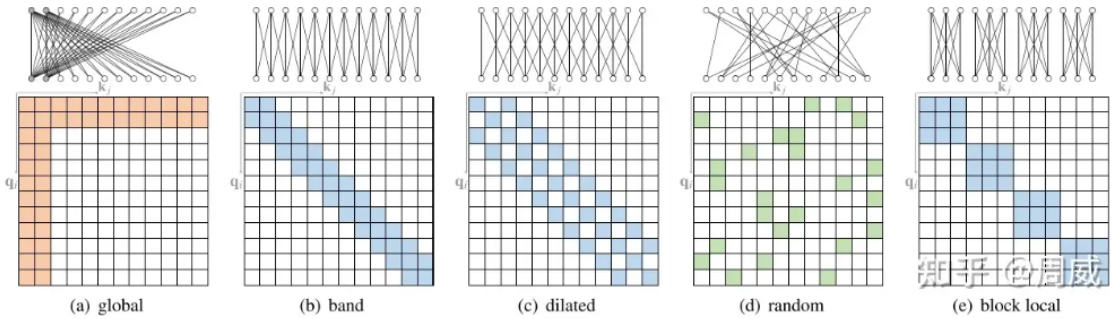
补充：

Sparse Attention

基于位置的稀疏注意力（position-based sparse attention）：这种方法是根据位置信息来确定哪些位置之间的注意力权重是稀疏的。在基于位置的稀疏注意力中，可能会使用一些规则或启发式方法来确定哪些位置之间的交互是重要的，而其他位置之间的交互可以被忽略或减少。这种方法通常依赖于位置之间的相对距离或其他位置特征来确定稀疏连接。

基于内容的稀疏注意力（content-based sparse attention）：这种方法是根据输入内容的特征来确定哪些位置之间的注意力权重是稀疏的。在基于内容的稀疏注意力中，可能会根据输入数据的特征向量来动态地确定哪些位置之间的交互是重要的，从而实现稀疏连接。这种方法通常会根据输入数据的内容特征来自适应地调整注意力权重，以实现更高效的计算和更好的性能。

基于位置的稀疏注意力：



Global： 通过添加一些全局节点来工作，这些节点可以被看作是信息传播的枢纽。这些全局节点有能力关注序列中的每一个节点，这意味着它们可以捕获和累积整个序列的信息。同时，序列中的每个节点也都会关注这些全局节点。这样，即使是序列中相距很远的节点，它们之间的信息也可以通过这些全局节点进行有效传播。通过这种方式，**全局节点充当了一个信息集散地**，帮助远距离的节点间接地相互"通信"。

Band： 是一种在处理数据时**考虑数据的局部性质**的注意力机制。

Dilated Attention： 是一种灵感来源于**扩张卷积神经网络**（Dilated CNNs）的注意力机制，旨在在不增加计算复杂度的情况下增加带状注意力的接受域（即模型能够感知到的输入范围）。这对于处理需要理解长范围上下文信息的任务特别有价值，如某些自然语言处理和时间序列分析任务。

Random： 旨在增强模型处理非局部（或长距离）交互的能力，这是通过为每个查询随机抽样一些边来实现的。

Block： 一个查询块中的所有查询仅关注对应记忆块中的键（keys），而不是整个序列。这样做的目的是减少计算复杂度，并允许模型更专注于局部上下文信息，从而提高处理长序列数据时的性能。

现有一些方法主要是**通过组合上面提到的几种基本Attention**

基于内容的稀疏注意力：
在标准的 Transformer 模型中，自注意力机制需要计算查询（query）与所有键（key）之间的相似度，这在处理长序列时会导致巨大的计算负担。为了解决这个问题，可以构建一个基于内容的稀疏图，从而**只关注与给定查询最相关的一小部分键，而不是所有键。**

稀疏连接：在稀疏图中，每个查询不是与所有键计算相似度，而是只与一部分键（即稀疏连接）进行交互。这些连接是基于输入内容的，意味着它们是根据查询和键之间的相关性条件动态确定的。
选择关键键：构建稀疏图的一个直接方法是选择那些与给定查询有较大相似度得分的键。这意味着，对于每个查询，我们只关注那些最可能与其高度相似的键，从而减少计算量。

最大内积搜索（MIPS）

- **MIPS问题**：为了高效地构建这种基于内容的稀疏图，可以利用最大内积搜索（Maximum Inner Product Search，MIPS）问题的解决方案。MIPS的目标是找到与查询具有最大点积（dot product）的键，而不需要计算查询与所有键之间的点积。
- **效率**：通过解决MIPS问题，可以**快速**识别出与给定查询最相关的键，从而构建稀疏图。这种方法避免了计算查询与所有键之间的相似度，显著提高了计算效率。