Prompt Cache: Modular Attention Reuse for Low-Latency Inference

SL LV

2024 年 8 月 16 日

1 摘要的中文解读

我们提出了一种名为*Prompt Cache*的技术，旨在通过重用不同大型语言模型（LLM）提示间的注意力状态来加速推理过程。许多输入提示存在文本段的重叠，例如系统消息、提示模板以及用于提供上下文的文档。我们的关键见解是：通过在推理服务器上预先计算并存储这些经常出现的文本段的注意力状态，可以在这些文本段在用户提示中再次出现时，有效地进行重用。Prompt Cache 采用了一种模式（*schema*）来显式定义这些可重用的文本段，我们称之为提示模块。这种模式确保了在重用注意力状态时的位置精确性，并为用户提供了在其提示中访问缓存状态的接口。通过在多种LLM上使用原型实现进行评估，我们展示了Prompt Cache在处理基于文档的问答和推荐等长提示时，显著减少了至首标记时间（time-to-first-token，TTFT）的延迟，改善幅度从GPU推理的8倍提升至CPU推理的60倍，同时保持输出的准确性

2 引言

大量的大型语言模型（LLM）提示经常被重复使用。例如，提示通常从相同的“系统消息”开始，这些消息提供了功能的初始指导。文档也可能在多个提示中重叠。在广泛的领域中，例如法律分析、医疗应用和教育，提示可能包含一组或多个文档，这些文档来自于一个资源池。此外，提示通常由可重用的模板格式化，这是提示工程的结果。这些情况在用于机器人学和工具学习的LLM中很常见，从而导致在处理这些提示时需要高度的计算资源。

为了减少生成性LLM推理中的计算开销，我们引入了一种名为*Prompt Cache*的新技术。Prompt Cache的动机在于观察到LLM的输入提示往往具有可重用的结构。核心思想是预先计算经常重用的文本段的注意力状态。我们提出了Prompt Markup Language（PML），使提示的结构在PML中显式可见，从而使文本段可重用。除了解决第二个问题外，PML还为每个提示模块分配了唯一的位置ID。我们的第二个思路是基于经验发现LLM可以在具有非连续位置ID的注意力状态上运行。这意味着我们可以提取不同的注意力状态片段并将它们连接起来，以形成不同的含义，使用户可以根据需要选择提示模块，甚至在运行时更新某些提示模块。

总结而言，LLM用户可以基于提示模块重用注意力状态来撰写他们的提示。重要的是，他们必须从一个PML中书写的模式派生提示。当Prompt Cache接收到一个提示时，它首先处理其模式并计算其提示模块的注意力状态。然后，它将这些状态用于提示中的提示模块以及从相同模式派生的其他提示。

3 Prompt Cache的构建和优化

构建在KV缓存之上，Prompt Cache通过使注意力状态重用变得模块化，扩展了注意力状态的重用范围，涵盖了多个提示。在我们的方法中，频繁重用的文本段被单独预计算并存储在内存中。当这些“缓存”的片段出现在输入提示中时，系统使用内存中的预计算键值注意力状态，而不是重新计算它们。因此，只有未缓存的文本段需要进行注意力计算。我们注意到，随着缓存片段大小的增长，性能优势变得更加明显，因为注意力状态的计算开销与输入序列大小呈二次方关系，而Prompt Cache的空间和计算复杂性与大小呈线性关系。

我们使用原型实现对Prompt Cache进行了广泛的基准测试，以评估其在多个长文本数据集上的准确性，包括个性化、代码生成和参数化提示等生成任务。我们展示了Prompt Cache的提示模式和性能改进的表现力，并发现即使在质量几乎不降低的情况下也有明显的性能改善。

未来工作中，我们预见Prompt Cache将被用作未来LLM服务系统的基础组件，这些系统可能包括增强的提示模块管理和GPU缓存替换策略，优化主机DRAM和GPU HBM的使用。我们的源代码和用于评估的数据可在GitHub上找到。

4 背景和相关工作

Prompt Cache构建在键值缓存（KV Cache）的思想之上，即在大型语言模型（LLM）的自回归解码过程中重用注意力状态。本节回顾了自回归标记生成和其他低延迟LLM推理的方法。

4.1 自回归标记生成

LLM通过自回归方式生成输出标记，这意味着它从初始输入开始，逐步添加新标记，直至达到停止条件，如预定的标记数量或生成特定的序列结束标记。在此过程中，模型在每一步都使用输入和迄今为止生成的所有标记作为输入。这一自回归标记生成过程导致在每一步中对整个输入应用自注意力机制，从而在每次生成新标记时产生了大量的计算。

4.2 键值缓存

键值缓存（KV Cache）是一种优化技术，它在整个输入期间只计算每个标记的键和值嵌入一次，并在生成过程中重用这些键值状态。例如，假设用户提示是一个由n个标记组成的序列$s_1, \ldots, s_n$，并且接下来生成k个标记$s_{n+1}, \ldots, s_{n+k}$。在原始的自回归标记生成中，每一步都需要重新计算所有标记的注意力状态。而使用KV缓存，模型只在第一步计算输入$s_0 = \{(k_i, v_i) | i \leq n\}$的注意力状态，并将其缓存。对于每一个后续步骤$j \leq k$，模型重用缓存的值$S_j = \{(k_i, v_i) | i < n + j\}$来计算新标记$s_{n+j}$的注意力状态$(k_{n+j}, v_{n+j})$。这显著减少了自注意力的计算需求，特别是在标记数量很大的情况下。

4.3 其他低延迟LLM推理方法

除了KV缓存，还有其他几种加速LLM推理的方法。一些系统专门针对多GPU推理进行了优化，而其他系统则优化了软件栈以提高计算性能。尽管这些方法在当前的硬件架构上可以实现低延迟推理，但Prompt Cache通过优化注意力状态的重用进一步提高了效率，并减少了计算需求。此外，Prompt Cache的内存管理优化也有助于提高处理速度，尤其是在使用分页注意力（Paged Attention）等技术时，可以共享不同提示的相同模块，而不是重复缓存注意力状态。

5 Prompt Cache的概述

5.1 概述

文本段的注意力状态只能在LLM输入中的相同位置被重用。这是因为变换器架构将唯一的位置嵌入集成到了注意力状态中，这不会影响使用KV缓存为单个提示服务的情况，因为相同的提示文本位于所有步骤的输入开始位置。然而，共享的文本段可以出现在不同的提示中的不同位置。为了重用这些注意力状态，缓存系统必须解决两个问题：首先，系统必须能够有效地识别出可能已缓存其注意力状态的文本段；其次，当系统接收到新提示时，必须能够在不同的提示中重用这些状态。

为了解决这两个问题，我们结合了两个想法。首先是使提示的结构通过Prompt Markup Language（PML）显式化，从而使文本段可重用。这不仅解决了第二个问题，而且打开了解决第一个问题的大门，因为每个提示模块可以被分配唯一的位置ID。我们的第二个想法是基于经验发现LLM可以在具有不连续位置ID的注意力状态上运行。这意味着我们可以提取不同的注意力状态片段并将它们连接起来，从而形成不同的含义。

5.2 Prompt Markup Language（PML）

接下来，我们描述了PML的关键特性，这些特性用于定义提示模式和由模式派生的提示。PML是一个定义提示模块和模式的文档，每个模块和模式都有唯一的标识符，并通过¡module¿标签指定。提示模块如果不被¡module¿标签或未指定标识符包围，则被视为匿名提示模块，总是被包括在从模式构建的提示中。

5.3 编码模式

第一次使用提示模块的注意力状态时，需要在设备内存中计算并存储这些状态，我们将此过程称为提示模块编码。Prompt Cache首次提取模式中的提示模块，然后为其分配位置ID，这些位置ID是根据提示模块在模式中

的绝对位置确定的。从提示模块的标记序列和对应的位置ID中，这些标记序列和位置ID被传递到LLM以计算注意力状态。我们注意到，分配的位置ID不是从零开始的，这在语义上是可以接受的，因为空白不改变预先计算文本的含义。

5.4 缓存推理

当提示提供给Prompt Cache时，Prompt Cache会解析它以确保与声称的模式一致。如图2所示，Prompt Cache从缓存中检索导入的提示模块的注意力状态，为新文本段计算这些状态，并将这些注意力状态连接起来，形成整个提示的注意力状态，替换了预填充操作。详细来说，Prompt Cache通过连接每个导入提示模块的KV状态张量来开始，然后计算未缓存的提示部分的注意力状态，并将这些状态与整个提示的KV缓存一起传递到LLM，以计算整个提示的注意力状态。

这种方法不仅优化了注意力计算，而且提高了系统在处理批量提示时的内存优化，因为来自同一模式的不同提示可能包括相同的提示模块，这允许通过减少KV缓存冗余来增加批量大小，从而提高系统吞吐量。

6 实施

我们使用HuggingFace的transformers库构建了一个Prompt Cache原型，在PyTorch框架中整合了3K行的Python代码。我们的目标是无缝接入现有的LLM代码库并复用其权重，我们实现了Prompt Cache以便在CPU和GPU内存中存储提示模块，并在这两种平台上进行评估。

6.1 存储提示模块在内存中

我们以两种类型的内存存储编码过的提示模块：CPU内存（DRAM）和GPU内存（HBM）。为了优化通过CPU和GPU进行访问的速度，我们雇用了PyTorch的内存分配器来管理跨这两种内存类型的张量。除了将CPU用于CPU内存中的提示模块和GPU用于GPU内存外，我们还使GPU能够访问存储在CPU内存中的提示模块，这通过需要时将提示模块从主机复制到设备来完成。尽管这一过程引入了主机到设备的内存复制开销，但它允许GPU利用丰富的CPU内存，这可以扩展到数TB级别。我们将在第5节展示，Prompt Cache带来的计算节省超过了由内存复制操作引起的延迟。

6.2 适配变换器架构

实现Prompt Cache需要支持不连续的位置ID，尽管transformers库当前并不提供这些功能，但可以通过轻微的修改进行整合。例如，我们大约为每个LLM需要20行的额外代码来完成所需的调整。具体调整如下：

- **嵌入表**：早期模型如BERT和GPT-2使用查找表来将位置ID映射到学习到的嵌入或固定偏差，不需要任何修改。
- **RoPE**：如Llama2和Falcon采用RoPE系统，它在注意力计算中使用旋转矩阵进行位置编码。我们为每个旋转矩阵创建一个查找表，根据位置ID检索。
- **ALiBi**：如MPT和Bloom采用ALiBi系统，在softmax分数计算中加入静态偏差。类似于RoPE，我们设计了一个查找表来调整根据提供的位置ID的偏差矩阵。

我们还重写了PyTorch的连接操作符，以更有效地分配内存。PyTorch仅支持连续张量，因此两个张量的连接总是导致新的内存分配。Prompt Cache需要连接提示模块的注意力状态，而默认行为会导致冗余的内存分配。我们实现了一个缓冲连接操作符，该操作符在连接张量时重用内存。这种优化改善了Prompt Cache的内存足迹并减少了内存分配的开销。

7 评估

我们的评估重点在于回答以下三个研究问题：Prompt Cache对首标记时间（TTFT）的影响是什么？存储内存开销是多少？哪些应用和KV Cache相比最合适？我们使用TTFT延迟作为测量时间，这是从开始解码到首标记生成的时间。我们在两个CPU配置上对Prompt Cache进行了评估。

7.1 评估环境

我们在两种CPU配置上评估Prompt Cache：一种是搭载128 GB DDR5 RAM和Intel i9-13900K CPU，另一种是搭载128 GB DDR4 RAM和AMD Ryzen 7 7950X

CPU。对于GPU评估，我们部署了三种NVIDIA GPU：RTX 4090、A40和A100，这些GPU安装在NCSA提供的16核AMD EPIC 7763服务器上，每台服务器都配备224 GB RAM。我们使用多种开源LLM进行测试，包括Llama2、CodeLlama、MPT和Falcon。

7.2 基准数据集上的延迟改进

我们观察到在所有数据集上，无论使用CPU还是GPU内存，TTFT延迟都有显著减少。由于篇幅限制，我们只展示了8个基准测试结果，完整的21个数据集的基准测试可以在附录中找到。

7.2.1 GPU推断延迟

我们的结果总结在图3中，评估了三种NVIDIA GPU。黄色条表示从CPU内存中加载提示模块的情况，蓝色条表示使用GPU内存的情况。由于LongBench数据集中的样本长度相似，平均约有5K个标记，我们观察到在所有数据集和GPU中，TTFT延迟减少了1.5倍到3倍。

7.2.2 CPU推断延迟

图4显示Prompt Cache在Intel和AMD CPU上能实现高达70倍的延迟改善。这种差异可能受到系统设置中内存带宽差异的影响，Intel CPU的DDR5 RAM与AMD CPU的DDR4 RAM相比，前者的性能更优。

7.3 与Prompt Cache的准确性

为了验证Prompt Cache对LLM响应质量的影响，我们没有使用支架对LLM进行评分。我们使用LongBench数据集和不同的transformer架构进行了准确性基准测试。结果显示，Prompt Cache保持了生成的内容质量，所有数据集的输出与基线相当。

7.4 延迟改进的理解

理论上，Prompt Cache应该在常规KV Cache上提供二次方级的TTFT改进。这是因为，随着序列长度的增加，Prompt Cache的内存复制成本线性增长，而其节省的自注意力计算开销呈二次方增长。为了验证这一点，我们在具有不同序列长度的合成数据集上测试了Prompt Cache，假设所有提示都已缓存。

7.5 内存开销

Prompt Cache的内存开销与缓存的标记总数成正比。我们使用了不同的LLM来说明在较小的模型中如Falcon 1B，每个提示模块的内存占用约为180 MB，而在大型模型如Llama 70B中，每个提示模块需要约2.5 GB的内存。

7.6 Prompt Cache的应用

我们通过示例用例展示了PML的表现力，这些用例需要比LongBench基准测试更复杂的提示结构和更高级的功能。我们还展示了代码生成、个性化和参数化提示的用例，这些用例展示了Prompt Cache如何在保持LLM响应质量的同时降低TTFT延迟。

8 结论和未来工作

我们引入了Prompt Cache，一种加速技术，基于注意力状态可以在不同LLM提示中重用的洞察。Prompt Cache利用模式模式化这些可重用的文本段，形成所谓的“提示模块”，使LLM用户能够无缝地将这些模块整合到他们的提示中，从而利用其模块性结构来增强上下文内容，并具有可忽略的延迟影响。我们的数据集评估表明，Prompt Cache在GPU上可以实现最高8倍、在CPU上可以实现最高60倍的TTFT延迟减少。

对于未来的工作，我们计划使用Prompt Cache作为构建未来LLM服务系统的基础，这样的系统可以配备改进的缓存替换策略，以实现更低的边界延迟。不同策略的实施，如优化KV缓存中的不同缓存技术，利用分组查询注意力或共享注意力状态的探索，都有助于减少并发请求的TTFT延迟。我们也讨论了通过将更多请求打包成批次来减少网络通信开销的可能性。除此之外，Prompt Cache还可以直接作为信息检索增强生成系统（如实时生成问答系统和对话系统）的实时响应系统，其中Prompt Cache可以作为提示模块数据库，实现实时答案和对话生成。

9 Prompt Cache的重用机制

图2展示了Prompt Cache中的重用机制。首先，PML（Prompt Markup Language）使得提示模块在Schema和Prompt中显式化。提示模块可以具有参数，例如`trip-plan`。在Prompt中导入模块时，用户提

供参数值（例如`duration="3 days"`）。Prompt可以包括排除某些模块和参数末尾的新文本段。

在实际操作中，Prompt Cache首先为Schema中的所有模块预计算并缓存它们的注意力状态（步骤1），之后当Prompt被调用时，Prompt Cache使用缓存的推断（步骤3.4）：它检索为导入的提示模块缓存的注意力状态，为参数和新文本段计算它们（步骤4），最后将它们连接起来生成整个Prompt的注意力状态（步骤5）。

此图是对图1c中步骤1的详细说明，揭示了如何通过PML在Prompt Cache中有效地利用预先计算的注意力状态来减少计算需求并提高处理速度。此外，这种模块化和参数化的方法允许高度灵活的Prompt构建，能够适应多变的用户需求和场景，同时保持高效的执行性能。

10 论文创新点及其实现

本论文提出的Prompt Cache技术，为大型语言模型（LLM）的推理加速提供了一种新颖的方法。其主要创新点及实现方式如下：

10.1 模块化提示重用

创新点：通过Prompt Markup Language (PML) 显式定义可重用的提示模块，实现了提示内容的模块化管理和重用。

实现方式：PML允许开发者定义包含多个模块的Schema，每个模块都可以被参数化并在不同的提示中重用。这些模块预计算并缓存其注意力状态，当它们被再次调用时，可以直接使用这些缓存的状态，而不需要重新计算。

相关知识点：- *注意力机制*：在Transformer模型中，注意力机制是通过计算键值对（Key-Value pairs）来动态选择输入的相关部分，这是现代NLP模型中处理和生成文本的核心。- *缓存机制*：在计算机科学中，缓存是一种保存数据副本的技术，旨在加速未来对这些数据的访问。在Prompt Cache中，注意力状态的缓存减少了重复计算的需求，提高了处理效率。

10.2 缓存推断

创新点：采用缓存的注意力状态进行推断，减少了生成每个新标记所需的计算量。

实现方式：Prompt Cache在LLM推理过程中，通过串联缓存的注意力状态和新计算的状态来生成响应。这种方式利用了已缓存的数据来加速推理过程，特别是在处理长序列或复杂模板时更显著。

相关知识点：- *自回归生成*：自回归模型在生成每个新标记时依赖于之前的所有标记，这通常需要大量的重复计算。Prompt Cache通过重用先前计算的注意力状态来减轻这一负担。- *Transformer架构的适应性*：为了支持不连续的位置标识符和模块化的输入，修改了基础的Transformer模型，允许在不牺牲模型性能的情况下进行灵活的输入处理。

10.3 性能与应用

创新点：通过减少计算需求，显著提高了在多种硬件配置上的推理性能。

实现方式：通过详细的基准测试，证明了Prompt Cache在CPU和GPU上实现了显著的性能提升，特别是在延迟敏感的应用场景中。

相关知识点：- *硬件优化*：利用不同类型的内存和处理器的特性，优化数据存储和处理策略，以适应不同的运行环境。- *延迟与吞吐量*：在系统设计中，延迟和吞吐量是衡量性能的关键指标。通过减少延迟，可以提高用户体验，而提高吞吐量则可以处理更多的数据或请求。

以上各点展示了Prompt Cache不仅提出了一种理论上的模型优化方案，还具体实现了这些优化，使其能够在实际应用中提供显著的性能提升。