https://arxiv.org/pdf/2401.05391

摘要
基于 Transformer 的大型语言模型 (LLM) 已广泛应用于许多领域，并且 LLM 推理的效率成为实际应用中的热门话题。 然而，LLM 通常是模型结构设计复杂，运算量大，并以自回归模式进行推理，使得设计一个高效率的系统成为一项具有挑战性的任务。
在本文中，我们提出了一种具有低延迟和高吞吐量的高效 LLM 推理解决方案。首先，我们通过融合 data movement 和 element-wise 操作来简化 LLM decoder layer 降低内存访问频率并降低系统 latency。 我们还提出了 segment KV cache 将 token key/value 保持在 separate 物理内存中的策略，以确保高效内存管理，有助于扩大运行时 batch 并提高系统吞吐量。 定制的 Scaled-Dot-Product-Attention 内核旨在匹配我们基于 fusion 策略的 segment KV cache 解决方案。 我们在 Intel® GPU 上实现了 LLM 推理解决方案并发布公开。 与标准 HuggingFace 实现相比，所提出的解决方案实现了高达对于英特尔® GPU 上的一些流行的 LLM， token latency 降低了 7 倍，throughput 提高了 27 倍.

1. 简介
最近，基于 Transformers [1] 的大型语言模型 (LLM) [2-4] 得到了广泛的应用注意力。 凭借内容理解和生成的能力，LLMs 已被应用于许多领域下游应用[5-9]。 然而，LLMs 通常设计复杂，规模越来越大[10]更深入的[11]，使得实现高效的 LLM 推理系统成为一项具有挑战性的任务。LLM 推理系统通常用于不同的场景，例如延迟关键的在线和面向吞吐量的离线应用程序 [12, 13]。 对于在线服务，延迟表示时间生成一些 token 的消耗，体现在线系统流畅度，延迟越低越好用户体验。 对于离线应用来说，吞吐量指的是一次生成的 token 数量，反映系统资源利用率，吞吐量值越高，系统成本越低系统。
对于在线模式来说，降低延迟、提高请求响应效率至关重要。然而，深层解码器层与多种操作相结合，使得实现低解码器并不容易。单个令牌的延迟。 以 Llama2[14]为例，基本结构如图 1 所示，包含多个解码器层，每个解码器层有多个模块，例如线性、旋转位置嵌入(RoPE)、缩放点积注意力 (SDPA)、均方根层归一化 (RMSNorm) 和激活。 简化模型结构并减少 LLM 结构中的运算次数可以帮助降低延迟。

还应考虑吞吐量标准以降低成本。 增大批量大小值并提高计算资源的占用率有助于实现高吞吐量。 然而，自动回归原理使得 LLM 推理成为一个消耗内存的系统，限制了 batch size 值并且在设备内存受限的 HPC 平台上可以实现的吞吐量。 一般来说，期间 LLM 推理过程，输入 tokens（prompt）将首先在 prefill 阶段计算，然后输出 tokens（响应）将在解码阶段根据之前生成的 token 逐步生成直到达到终止 token 为止。 在解码阶段的每个时间步，一些候选 token 会一次生成，通常使用 beam search 方法[15]来选择一些标记高置信度分数。 通过这种自回归原理，每个解码器层的 key/value 在整个解码阶段，时间步都保留在设备内存中（称为 KV cache 策略 [13]），随着时间步长的增加，消耗越来越多的内存。有效管理设备至关重要内存来扩大批量大小并提高吞吐量。

1. 我们提出了一种高效的 LLM 推理解决方案，并在 Intel® GPU 上实现。 这可通过我们的 Intel®-Extension-for-Pytorch repo 在线实现。

2.为了降低延迟，我们简化了 LLM 解码器层结构以减少数据移动开销。此外，我们设计了一种深度融合策略来融合 GeMM 和 Element-wise 操作：尽可能多。 对于上面提到的一些参数大小从 6B ~ 176B 的流行 LLM，与标准相比，我们的推理解决方案将 token 延迟降低了 7 倍 HuggingFace 实现 (v4.31.0) [20]。

3.为了提高吞吐量，我们提出了 segment KV cache 策略以保持及时和响应 key/value 写入设备内存，使得 prompt key/value 被不同设备共享响应 token 以避免内存浪费。 在相同的设备配置下，吞吐量我们的推理解决方案（每秒令牌数）比标准高出 27 倍 HuggingFace 实现了流行的 LLM，参数大小为 6B ~ 176B。

4.基于我们的融合策略和分段 KV 缓存方法，我们设计了一个高效的内核将 SDPA 模块中的所有计算步骤与可能的 index selection 融合在一起，for beam search 使用过程。

2. 相关工作

基于 Transformer 的工作负载的效率通常受到内存访问的瓶颈 [21, 22]，其中读取和写入数据占据了运行时间的很大一部分。 减少内存限制 LLMs 的运营开销有助于提高效率。 Kernel fusion 是通用方法通过将多个连续操作组合在一起并在单个内核中计算它们来实现这样的开销以避免频繁访问速度较慢的 GPU 高带宽内存 (HBM)。 之前的一些作品[23-26]专注于 element-wise 操作融合，以减少内核启动和内存访问开销。除了按元素运算的一般融合之外，Fang 等人。 [26]提出 TurboTransformers 来融合两个 GeMM 之间的逐元素和归约运算。 阿米纳巴迪等人。 [12] 设计一些根据 Transformer 相关工作负载的特点定制的 GeMM 内核以及采用融合策略，将数据布局转换和缩减操作与定制 GeMM。 道等人。 [21]专注于提高 SDPA 模块效率并提出了一种 FlashAttention 算法结合所有计算步骤（Batch-GeMM、Softmax、可能的 Masking、Batch-GeMM）一起，在不同的情况下实现比标准注意力实现快 3 倍用户场景。 FasterTransformer [27]基于 FlashAttention [21]，进一步融合了 index selection,将 beam search 到 SDPA 内核以进一步删除数据移动操作。 考虑 LLMs 是权重有限的工作负载，压缩技术包括修剪 [28]、量化 [29, 30]、知识蒸馏[17]和低秩分解[31]通常用于压缩 LLM 权重，以减少内存访问大小并提高效率。 [29, 30] 关注 LLM 权重量化和应用低精度 Matmul 乘法进行 feed-forward 和 attention projection layers in transformer 中的层以减少推理内存的使用。 考虑不相关的信息使用长上下文提取，[32]采用稀疏变换器显式提取最相关的片段在注意模块中以减少计算量。

许多相关工作也致力于提高 LLM 推理系统的吞吐量。 批处理是一个提高硬件资源利用率和提高吞吐量的重要技术。 然而，LLMs 推理是消耗内存的，可以在硬件平台上设置限制批量大小值限制设备内存。 有效的内存管理对于帮助扩大批量大小和提高吞吐量。 由于自回归原理，LLM 中的 KV 缓存消耗较大的设备内存，大于 13B 参数 LLM 的 30% 设备内存 [33]。 有效管理 KV 缓存，Kwon 等人。 [33]提出了 PagedAttention 技术，将逻辑连续的键/值映射到分离物理内存，实现近乎零的内存浪费，灵活的 KV 缓存共享跨越不同的请求。 一般情况下，用户的提示请求都是不同的序列长度，静态的批处理（将不同长度的请求填充到批次中的最大长度）会浪费硬件资源因为在整个批次完成之前，之前完成的请求无法发送回客户端。 要解决这个问题问题，ORCA 系统[34]提出了一种迭代级调度机制来调用执行引擎仅在批次上运行模型的单个时间步，以便可以发送完成的令牌立即回来。

此外，一些 LLM 推理引擎（例如 TVM [35] 和 ONNXRumtime [24]）旨在在不同的硬件平台上提供高效的服务，提高系统效率根据不同的硬件特性优化内核。 除了单 GPU 卡上的推理外，DeepSpeed [12] 还考虑在多个 GPU 卡上扩展 LLM 推理，并提出张量/管道/专家并行技术。

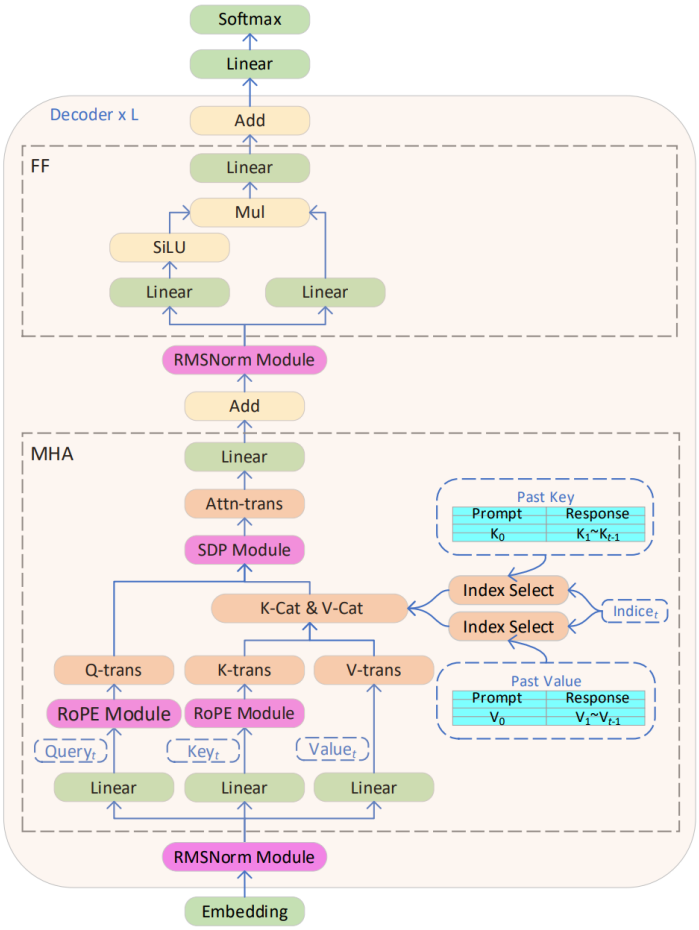
3. Proposed Method

在本文中，我们设计了一种高效的 LLM 推理解决方案，并在英特尔® GPU 上实现。 降低延迟，我们通过减少数据移动操作来简化 Transformer 解码层的结构并应用定制的内核融合策略。 为了更有效地管理设备内存和提高吞吐量，我们提出了一种分段 KV 缓存算法，使键/值能够及时共享不同响应令牌之间。 定制的 SDPA 内核旨在支持我们的融合策略和分段 KV 缓存算法。

3.1 简化的模型结构

正如我们上面提到的，LLM 通常设计复杂，具有多个解码器层由捕获上下文信息的大量操作组成。解码器层通常有两个基本的模块多头注意力（MHA）和前馈（FF），它们通过一些连接标准化操作。 以 Llama2 为例，基本模型结构如图 1 所示。时间步t，在 MHA 模块中，首先采用三个线性运算来生成$\mathbf{Query}t$, $\mathbf{Key}t$,$\mathbf{Value}t$，然后可能是用于位置嵌入的 RoPE。 然后应用 SDPA 模块注意上下文计算，最后使用另一个输出 Linear 进行特征投影。 在 SDPA 之前模块中，使用了一些数据

移动操作，包括 Transpose、Cat 和 Index Select。 这转置操作应用于**Query*t***, **Key*t***, **Value*t*** 用于数据布局转换[BS × BW，N*t*, H,D] 到 [BS × BW,H,N*t*,D]，其中 BS、BW 表示运行时 batch size、beam widthvalue，H，D 表示模型的注意力头数量和头维度，而 N*t*表示时间步 *t* 处的序列长度。 然后过去的键/值（由提示**Key0**/**Value0**和响应组成）**Key1~*t*−1** / **Value1~*t*−1**）通过基于**Indice*t***的索引选择操作提取生成张量来自 Beam Search 模块，它将与当前的 **Key*t*** 连接起来/**Value*t***创建 KV 缓存用于 SDPA 模块中的注意力上下文计算。 内存布局中生成的注意力上下文 [BS × BW,H,N*t*,D] 将转换回 [BS × BW,N*t*,H,D] 用于最终线性投影



Prefill phase & Decoding phase

Figure 1. The standard flowchart of LLM inference with Llama2.

MHA 中的数据移动操作，例如 Transpose、Cat、Index Select，在图 1 中以橙色标记将导致内存访问和内核启动开销。 在本文中，我们简化了 MHA 结构减少这些运算符，如图 2 所示。在预填充阶段，内存布局中的 **Query0**、**Key0**、**Value0**生成 [BS × BW,NO,H,D]（称为批量优先布局），然后直接生成自定义的设计了 SDPA 内核用于注意力上下文计算，无需任何转换操作。 这定制设计的 SDPA 内核以批量优先布局接受输入并生成输出张量([BS × BW,NO,H,D])以避免转换开销，如图 2(a)所示。 在解码阶段，图 1 中所示的 Cat 操作用于将不同时间步生成的键/值组合在一起。为了消除 Cat 开销，我们为具有序列长度的响应令牌预先分配 KV 缓存缓冲区[Nstep, BS × BW, H,D] 内存布局中的 Nstep（称为序列优先布局）。 在时间步 *t*解码阶段，**Key*t***和**Value*t***保存在预分配缓冲区的相应部分[N*t*−1:N*t*，BS×BW，H，D]。 序列优先布局确保 **Key*t*** 和 **Value*t*** 始终连续排列以便于计算，无需了解超参数 Nstep。 然后提示 **Key0**、**Value0** 和响应键/值（**Key1~*t***, **Value1~*t***在预先分配的 KV 缓存中缓冲区）以及 **Indice1~*t***张量（由 Beam Search 模块生成）将被我们使用定制 SDPA 内核来计算最终内存布局 [1, BS × BW,H, D] 中的注意力上下文无需任何变换的线性计算。 对齐 KV 缓存的序列优先布局策略在不引入额外数据移动开销的情况

下，我们转换输入张量（称为隐藏状态图 2）在第一个解码器层之前从批量优先布局到序列优先布局。 序列第一个布局将在所有解码器层的内部和外部传播，直到到达最后一个解码器层，其输出将转换回批量优先布局。 优化后的 MHA 结构可以如图 2 所示，说明图 1 中以橙色标记的所有数据移动操作都是已删除

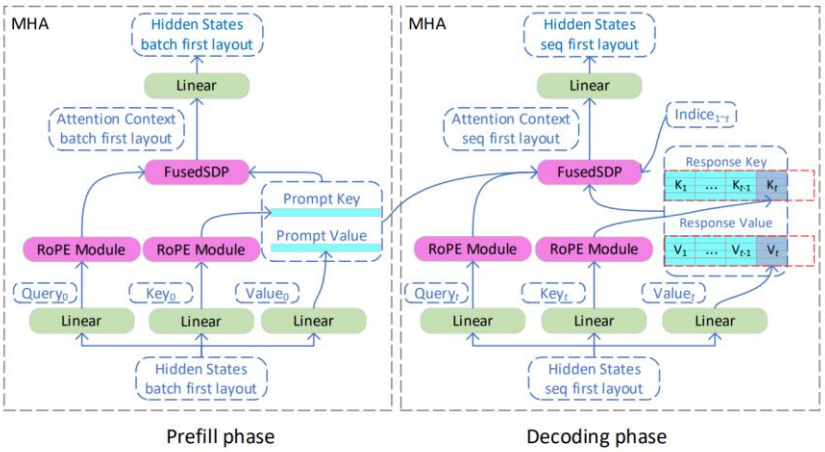


Figure 2. Optimized MHA module in Llama2 (a) at prefill phase, and (b) decoding phase, respectively.

此外，我们分别融合了 RMSNorm、RoPE 和 SDPA 模块中的多个操作（图 1 中以粉红色标记）为单粒。 三个线性运算（分别生成查询/键/值）也合并为一个，并且元素明智的操作标记为黄色图 1 进一步融合了之前的线性运算。 优化后的 Llama2 结构为如图 3 所示。与图 1 所示的原始结构相比，数据移动操作橙色标记和黄色标记的逐元素操作都被删除。 标记的模块粉红色的包含多个操作被融合到单个内核中。 即使我们额外介绍解码阶段的序列/批量优先布局转换操作，它们在解码阶段仅发生一次每个时间步的第一个和最后一个解码器层，与整个推理相比几乎没有开销。通过我们的优化，每个 Llama2 解码器层的操作数量已减少到九个，比原来的数百次操作要小得多。

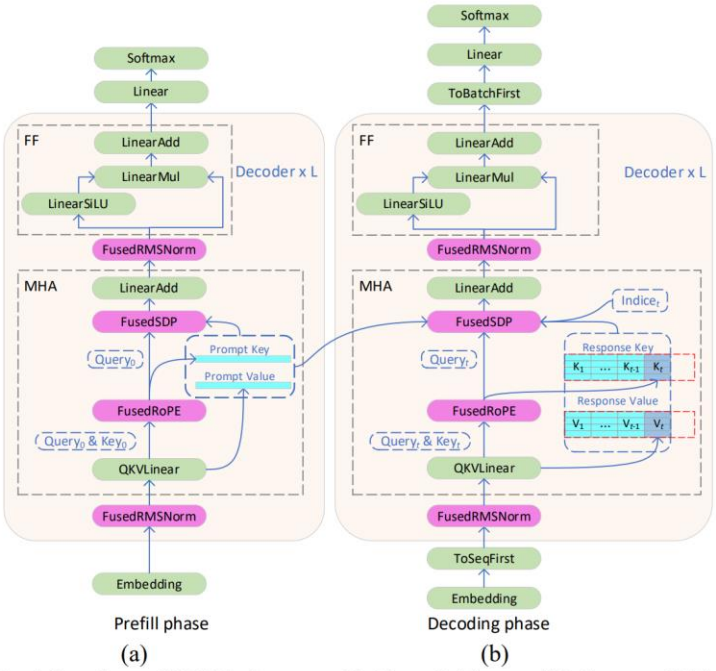


Figure 3. Optimized flowchart of LLM inference with Llama2 (a) at prefill phase, and (b) decoding phase, respectively.

3.2 segment kv cache

LLM 推理由于参数大小和 KV 缓存策略而导致内存消耗，限制了批量大小值，然后进一步影响系统吞吐量。 假设 LLM 有 L 名解码器层数、H 个注意力头、D 个头维度，则一个 token 的 KV 缓存大小可以为计算公式为（1），其中 2 表示键和值。 运行时批量大小、波束宽度分别为 BS、BW、总序列长度与提示序列长度 Nprompt 和响应序列相结合长度 N 响应。 在标准的 KV 缓存实现中，提示和响应键/值将是连接在一起以创建形状为 [BS × BW, Nprompt + Nresponse,H,D] 的连续 KV 缓存。那么最后一个时间步的 KV 缓存大小可以计算为 Eq（2）。 在标准实施中，将提示和响应键/值保存在连续的 KV 缓存缓冲区中会浪费设备内存，因为提示键/值应延长 BW 倍。 此外，由于 KV Cache 缓冲区变得更大在解码阶段的每个时间步，将分配新的更大的缓冲区，同时可能不会重用之前的缓冲区 KV Cache 缓冲区较小，导致内存碎片较多。 考虑到极端情况，所有每个时间步的 KV 缓存的物理设备内存都不能被复用，那么总内存 fragment 应该是每个时间步的 KV Cache 的总和，远大于该时刻的 KV Cache 本身最后一个时间步。

$$\text{cache}_{\text{token}} = 2 \times L \times H \times D \times \text{sizeof(datatype)} \quad (1)$$

$$\text{cache}_{\text{standard}} = BS \times BW \times (N_{\text{prompt}} + N_{\text{response}}) \times \text{cache}_{\text{token}} \quad (2)$$

为了避免保留重复的提示键/值并尽力减少内存碎片，我们建议分段 KV 缓存策略，将提示和响应键/值保存在不同的缓冲区中并定期保存在每个时间步清空缓存片段。 在预填充阶段，提示键/值的形状为每个解码器层的 [BS,Nprompt,H,D] 都会生成并保存在设备上，如图 3(a) 所示。然后，在解码阶段，我们以 [Nstep, BS × BW, H,D] 的形式预先分配响应 KV 缓存，每个解码器层。 每次响应的 key/value 都会保存在 KV 的相应部分缓存，如图 3(b)所示。 通过建议的段 KV 缓存策略，提示键/值可以与不同的响应令牌共享。

在之前的一些工作中，Nstep 通常被设置为模型的最大位置长度，如 4096Llama2，可以帮助减少内存碎片，同时如果运行时会导致内存浪费响应的句子长度较短。 为了解决这个问题，我们设置 Nstep 值动态增加 step = 16。以响应长度大于 16 为例，在第 1 个时间步（Nresponse = 1），我们预分配 KV 缓存，Nstep = 16。然后在第 17 个时间步（Nresponse = 17），Nstep 将设置为 32，会分配新的 KV 缓存，对应的部分会用原来的来完成 KV 缓存数据，另一部分将保留新的键/值数据。 同时，之前的 KVNstep=16 的 cache 会被手动清空 cache 以提高内存复用的概率。 在这样，响应 KV 缓存的长度会随着 step=16 而增加，减少一些数据移动每个时间步的操作，同时不会引入太多内存浪费或碎片。 记忆随着 Nstep 随着步长 = 16 的增加，段 KV 缓存策略的消耗可以计算为等式（3）。

$$\text{cache}_{\text{segment}} = BS \times (N_{\text{prompt}} + BW \times \text{Ceil}(\frac{N_{\text{response}}}{\text{step}}) \times \text{step}) \times \text{cache}_{\text{token}} \quad (3)$$

Table 1. Different LLM Configurations

Model	Decoder Layer (L)	Attention Heads (H)	Head Dim (D)
GPT-J-6B	32	32	128
Llama2-13B	40	40	128
OPT-30B	48	56	128
Bloom-176B	70	112	128

以表 1 所示不同参数大小从 6B 到 176B 的 LLM 为例，运行时 BW、Nprompt 和 Nresponse 分别设置为 4、1024、1024。 不考虑内存片段，上次不同 BS 值的 Float16 数据类型的 KV 缓存内存消耗步骤如图 4 所示，其中 standard 和 segment 代表标准的 KV Cache 策略和我们的分别提出了分段 KV 缓存策略。 以 GPT-J-6B 为例，BS=32，建议的分段 KV 缓存消耗 86GB 设备内存，仅为标准 KV 缓存

的 63%左右策略为 137GB，总共节省 51GB 设备内存。 因此，对于具有约束设备的 GPU 硬件内存，建议的 Segment KV 缓存策略消耗更少的内存，使得 batch size 值更大可以用来提高系统吞吐量。

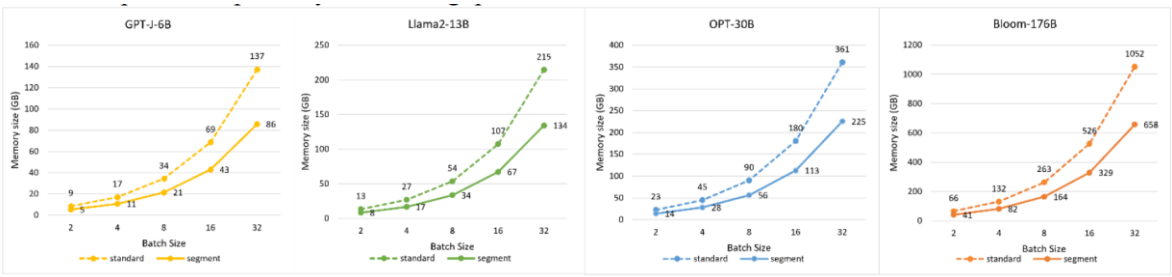


Figure 4. Memory consumption of standard KV cache and segment KV cache on different LLMs

受 FlashAttention [21] 和 FasterTransformer [27] 的启发，我们融合了 SDPA 中的所有计算步骤模块以及可能的索引选择操作基于我们的段 KV 缓存到单个内核中政策。 对于解码阶段定制的 SDPA 内核，查询、提示键/值的输入张量响应键/值的形状为 [1, BS × BW, H,D]、[BS,Nprompt,H,D] 和 [Nresponse, BS ×分别为 BW、H、D]。 注意上下文的输出张量的形状为 [1, BS × BW, H,D]。 在我们的实现中，BS 和 H 在不同的 GPU 工作组（块）中并行。 在单个 GPU 上工作组，查询**Q**的输入张量，提示键/值（**Kprompt**，**Vprompt**）和响应键/值（**Kresponse**，**Vresponse**）的形状分别为 [1, BW, D]，[Nprompt,D] 和 [Nresponse, BW, D]。 这注意力上下文 O 的输出张量的形状为 [1, BW, D]。 预填充时定制的 SDPA 内核阶段是解码阶段的特例，仅使用查询和提示键/值的输入张量用于注意力上下文计算。 我们在 SDPA 内核的单个工作组中进行计算以解码阶段为例来阐述我们如何实现内核。

Index select fusion

Beam index tensor **Indice1~*t***[BW,Nresponse] 的形状是在与 FasterTransformer [27] 中的方式相同。 基于**Indice1~*t***，每个上的实际响应键/值选择不同时间步长的 beam 位置。 采取 2 第一个时间步长的 nd beam 位置如图 5(a)所示，该位置的索引值为 3，则该位置的响应键值位置应该是 3 键缓存中第一个时间步的 rd 项。

内核实现。 在单个工作组中，**Q**首先会被加载到寄存器中。 然后我们 for 循环在 Nprompt 维度加载**Kprompt**和**Vprompt**的相应部分来计算注意力像 FlashAttention 这样的上下文。 下一步，我们在 KV 上应用上面提到的索引选择过程缓存和 for 循环在 BW 和 Nresponse 上加载 **response** 和 **Vresponse** 的相应部分注意上下文计算的维度。 最后分别计算出注意力上下文值根据提示和响应键/值将累加在一起得到结果并写入 HBM。 相应的流程如图 5(b)所示。

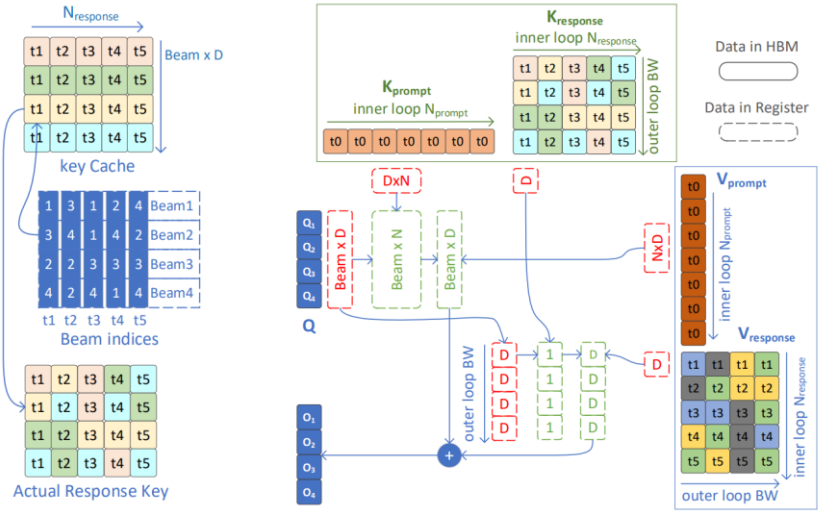


Figure 5. Implementation of customized SDPA kernel. (a) Index select; (b) Customized SDPA kernel based segment KV cache policy.

4. experiments

我们在英特尔® GPU 上实施我们的 LLM 推理解决方案，并在集群上执行实验 4 × 英特尔® Data Center Max 1550 GPU 卡，每卡 2 个 Tiles，64 Xe-每块核心数和 512 个 EU。 这每个 Tile 的设备内存为 64GB，有效内存带宽约为 1000GB/s。 这些 GPU 是托管在运行 Ubuntu 22.04.3 的 2 个 Intel® Xeon® 8480+ 系统上。 我们的优化方案代码是在 Intel®-Extension-for-Pytorch (v2.1.10) 中发布，我们的软件堆栈可在此处公开获取性能再现。

在我们的实验中，如果没有指定，我们将使用 Float16 数据类型的配置，输入提示长度 Nprompt = 1024 和输出响应长度 Nresponse = 128 用于性能评估。 首先，我们在定制的 SDPA 内核上进行实验以显示内核效率。 然后我们申请我们在表 1 中列出的一些流行的 LLM 上的推理解决方案。对于参数较小的模型 GPT-J-6B 和 Llama2-13B，我们在 Max 1550 GPU 单块上运行它们。 对于大参数模型尺寸为 OPT-30B 和 Bloom-176B，我们分别在 Max 1550 GPU 一张卡、两张图块和四张卡上运行它们卡八块。 对于大参数尺寸模型，我们使用自动张量并行（autoTP）[12]水平分割模型层以跨 GPU 设备运行它们的算法。

我们使用延迟和吞吐量标准来评估性能。 延迟包括两个比率：第一个令牌延迟指示批次中第一个令牌生成的延迟，以及下一个令牌延迟表示批次中倒数第二个令牌生成的平均延迟。 供推论吞吐量评估中，我们计算出可以设置的最大批量大小并收集相应的延迟。然后吞吐量可以按式（4）计算。

吞吐量 = (BSmax × Nresponse)/延迟 (4)

4.1 SDPA kernel 性能评估

我们使用提示键/值和响应的输入张量实现定制的 SDPA 内核形状分别为 [BS,Nprompt, H,D] 和 [Nresponse, BS × BW, H,D] 的键/值。 带着骆驼 2-以图 13B（H = 40 和 D = 128）为例，BW = 4，有效内存带宽值具有不同 BS 的定制 SDPA 内核如图 6 所示。对于 BS = 1，我们定制的内核（提示批量优先布局，响应序列优先布局）达到 727GB/s，仅约峰值内存带宽的 72%。 有 64 个

Xe- GPU 单块最多 1550 个核心，至少 64 个需要多个工作组来完成所有 Xe-核心，否则只有低硬件资源占用率高，效率低。 在我们的实现中，BS 和 H 是并行计算的在不同的工作组中。 当 BS = 1 时，并行工作组数为 H = 40，不满足总数 64 氙-核心。 BS = 1 的工作负载规模较小会浪费硬件资源并导致效率低下。当 BS 变大并等于 16 时，BS × H = 640 个工作组被调度在 64 Xe 上-核心和我们实现高硬件资源占用，内存带宽 942GB/s。 内核效率将随着 BS 继续变大而减少，因为对于响应键/值，我们 for 循环 Nresponse 以 stride BS × BW 加载 D 元素，stride 值越大，缓存命中的概率越低。

我们还首先进行实验来比较 SDPA 内核效率与批量提示键/值（建议的方法）和序列优先布局（与响应 KV 缓存布局对齐）。 如图 6、对于 BS = 1，两种布局都可以简化为 [Nprompt,H,D] 并实现相似的内存带宽值约 720GB/s。 对于较大的 BS，批量优先布局中的提示键/值[BS,Nprompt,H,D] 步长 H 比顺序优先布局具有更高的缓存命中概率[Nprompt, BS,H,D] 与 stride BS × H，这样我们定制的 SDPA 内核就可以用提示符实现 key/value 采用批量优先布局（与 KV 缓存布局不同）可以达到更好的性能效率。

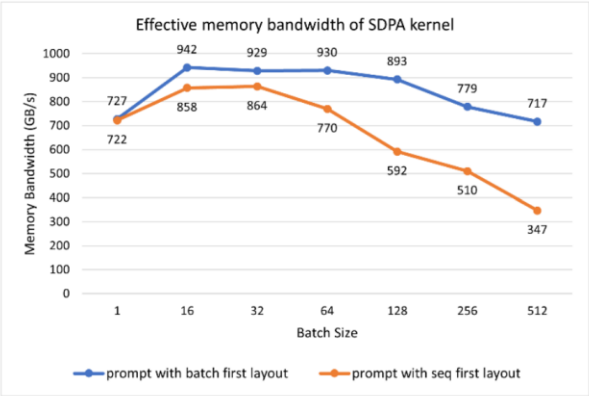
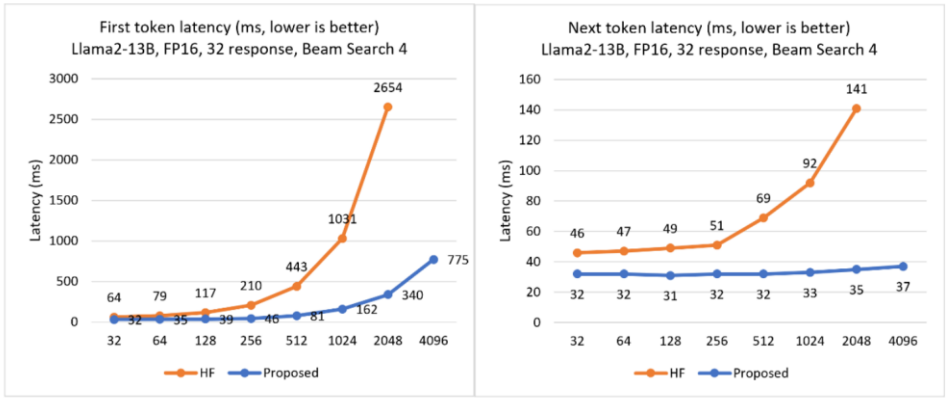


Figure 6. SDPA kernel performance with different batch sizes for different prompt key/value memory layout

4.2 System Latency Evaluation

4.2.1 不同提示长度下的延迟评估首先我们在 Llama2-13B 上进行实验，找出提示序列的影响带有响应令牌的延迟长度 Nprompt Nresponse = 32，BS = 1，BW = 4。我们收集第一个 Nprompt 从 32 到 4096 的不同情况下的令牌延迟和下一个令牌延迟。性能结果标准的 HuggingFace（命名为 HF）和我们提出的解决方案（Propose）如图 7 所示，其中，当 Nprompt = 4096 时，HF 实现会出现内存不足的情况。从图 7(a) 中我们可以看出可以看到，随着 Nprompt 变长，HF 实现的第一个令牌延迟变得更高。Nprompt = 128 是拐点，小于该拐点，低硬件下延迟增加缓慢占用率，而 Nprompt 大于 128 则增加 2 倍，几乎占满硬件资源占用。 与 HF 实现相比，所提出的方法仅访问 1/BW 量的数据在 HBM 中，相应的拐点发生在 Nprompt = 512，BW 倍长于 128。此外，所提出的方法比标准 HF 实现的首次令牌延迟要低得多对于任何 Nprompt 值，并且当 Nprompt 大于 512 并使用完整硬件时，超过 BW 倍占用。 图 7(b) 显示了相同的情况，所提出的方法接下来实现的效果要低得多任何不同提示序列长度的令牌延迟。



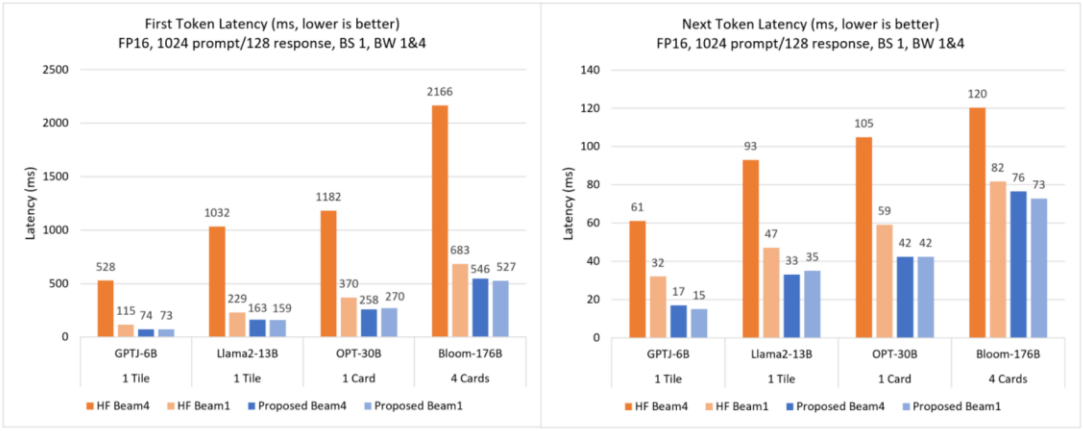
(a) (b)

Figure 7. Llama2-13B latency of (a) the first token, and (b) the next token with different prompt sequence length, respectively.

4.2.2 不同 LLM 的延迟评估

我们还进行了实验来评估不同 BW 值对性能的影响（BW =1 和 4），BS = 1。不同 LLM 上第一个和下一个令牌延迟的性能结果为如图 8 所示。标准 HF 实现的第一个和下一个令牌延迟均呈线性增长随着 BW 变大。 然而，BW 值对所提出的方法影响很小，几乎 BW 4 和 1 的第一个和下一个令牌延迟相同，如图 8 所示。

此外，我们还比较了所提出的方法和标准 HF 实现的令牌延迟。 为了 BW 1，该方法的第一个和下一个令牌延迟比之前的方法快约 1.1 倍~2 倍不同法学硕士的标准 HF 实施。 好处来自我们的深度融合政策高效的内核，删除数据移动操作并融合逐元素操作，这可以帮助减少 HBM 访问和内核启动开销。 对于 BW 4，第一个令牌延迟值我们提出的方法的性能比标准 HF 实现低 4 倍~7 倍。 除了好处之外从深度融合策略来看，分段 KV 缓存方式（无需扩展提示 BW=4 次）有助于减少内存访问和内核计算成本。 下一个代币的表现所提出方法的延迟也比 HF 实现低得多，特别是当 BW = 4 时如图 8(b)所示。



(a) (b)

Figure 8. (a) First and (b) next token latency performance of different LLMs with different parameters sizes

4.3 system throughput evaluation

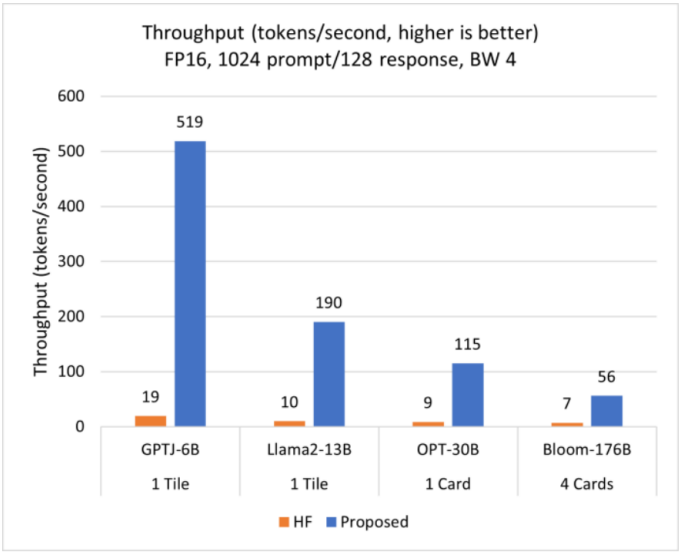


Figure 9. Throughput performance of different LLMs with different parameters sizes

我们进一步将我们提出的 LLM 推理解决方案与标准 HF 实施进行比较吞吐量标准由方程（4）计算。 性能结果如图 9 所示，说明所提出的方法实现了吞吐量 8 倍~27 倍的提高。 除了延迟优化之外，主要好处来自我们的段 KV 缓存，它可以帮助节省设备内存，然后扩大运行时批量大小值可提高硬件资源利用率。

5. 结论

在本文中，我们提出了一种具有低延迟和高吞吐量的高效 LLM 推理解决方案。由于 LLM 推理是一项内存密集型任务，因此降低内存访问频率有助于提高效率。 LLM 通常由多个解码器层组成，我们专注于简化解码器层结构通过融合数据移动和逐元素操作来减少内存访问。考虑到 LLM 推理工作在自回归模式并且消耗大量设备内存，我们提出分段 KV 缓存策略，使提示在不同的响应之间共享定期清空缓存中的碎片，从而节省设备内存并提高吞吐量。 我们也基于我们的融合策略和分段 KV 缓存方法设计一个高效的 SDPA 内核。 这所提出的解决方案已在参数大小从 6B~176B 的几个流行的 LLM 上得到验证英特尔® GPU。 实验结果表明，我们的延迟降低了 7 倍，延迟提高了 27 倍吞吐量高于标准 HuggingFace 实现。