Week 9 Reading Summaries: PagedAttention and FlashAttention

Amogh Patankar
University of California, San Diego
3869 Miramar Street Box #3037, La Jolla, CA- 92092
apatankar@ucsd.edu

Abstract

In this set of readings, we go over PagedAttention and FlashAttention, which are two methods of attention algorithms that use various strategies to reduce the number of memory read/writes that occur between high bandwidth memory (HBM) and GPU on-chip SRAM. Paged Attention uses classic virtual memory and paging techniques for operating systems, and the authors build vLLM, which is a serving system for minimal KV cache waste and sharing of KV cache. Flash Attention is extended to block-sparse attention, and they train transformers in a more efficient manner

1 PagedAttention

1.1 Introduction

In recent times, with LLMs on the rise, especially GPT, PaLM, and many more, there have been a wider variety of applications. Namely, chatbots and assistants are rising, with many cloud companies providing these services as features and hosted services. But, these applications require a lot of compute, i.e. GPUs. Given these costs, LLM serving systems come to the forefront of our attention, in order to improve throughput.
Improving throughput can be achieved by multibatching, but memory needs to be managed properly. The authors note that existing systems fall short because the KV cache memory isn't managed well. Moreover, these systems can't use memory sharing properly. The authors, in this paper, build vLLM, which is a LLM servicing engine using PagedAttention, achieving close to zero waste in the KV cache memory

1.2 Background

1.2.1 Transformer-Based LLMs

For transformer based LLMs, the whole idea is modeling probabilities as a list of tokens; this is done using a technique called autoregressive decomposition. A self-attention layer within a transformer uses linear transformations, attaining query, key, and value vectors, finally computing the attention score.

1.2.2 LLM Service and Autoregressive Generation

LLMs are essentially used as a generational API service; the KV cache stores key and value vectors in the generation process. The prompt phase takes the prompt and computes probabilities of new tokens, while the autoregressive generation phase generates the new tokens sequentially.

1.2.3 Batching Techniques for LLMs

Batching multiple requests in serving LLMs can help to decrease compute utilization; however, it's non-trivial. As requests arrive at different times, and the requests' input and output lengths may be different, standard batching doesn't work. Hence, batching techniques like cellular or iteration-level scheduling are proposed.

1.3 Memory Challenges in LLM Serving

Surpassing memory requirements has a few challenges, namely, large KV Cache, complex decoding algos, and scheduling for unknown input/output lengths.

1.3.1 Memory Management in Existing Systems

LLM serving systems require a dynamic allocation of memory for newer systems, storing KV caches efficiently. Typical static allocation of memory ends with huge pure memory waste.

1.4 Method

The authors develop PagedAttention to solve this exact problem.

1.4.1 PagedAttention

PagedAttention allows for the storage of keys and values in non-contiguous memory space, by creating KV blocks. Each block stores an attention score, at the ij^{th} index. In doing so, the KV blocks are stored in non-contiguous memory, meaning vLLM can handle memory more efficiently.

1.4.2 KV Cache Manager

vLLM's memory manager operates similarly to virtual memory in OS'. vLLM organizes KV cache as fixed-size KV blocks (like pages), with a request's KV cache represented as a series of KV blocks. The block manager maintains block tables.

1.4.3 Decoding with PagedAttention and vLLM

When doing decoding, vLLM doesn't reserve memory; it reserves KV blocks. In the first autoregressive decoding step, vLLM generates new tokens with PagedAttention on various blocks, and stores the newest KV cache in a new block for he second decoding step. Finally, vLLM dynamically assigns new physical blocks to logical blocks as tokens get generated.

1.4.4 Application to Other Decoding Scenarios

- Parallel Sampling: For parallel sampling scenarios, only one copy of the prompt's state will be reserved, and we keep a reference count. When generating, we use the copy-on-write mechanism.
- Beam Search: For beam search, the beam width determines the top candidates for each step. We have beam candidates vLLM allocates blocks, and vLLM reduces overhead by sharing physical blocks.
- Shared prefix: vLLM reserves a set of physical blocks for shared prefixes.
- Mixed Decoding Methods: vLLM allows for different decoding preferences.

1.4.5 Scheduling and Preemption

vLLM adopts a first-come-first-serve scheduler, ensuring fairness and preventing starvation.

- Swapping: vLLM selects a set of sequence to remove, then swaps their KV cache to the CPU.
- Recomputation: vLLM recomputes the KV cache for each preempted sequences that are rescheduled.

1.5 Distributed Execution

vLLM efficiently manages memory for large language models using Megatron-LM's tensor model parallelism and an SPMD execution strategy. Linear layers are partitioned for block-wise matrix multiplication, with GPUs synchronizing intermediate results via all-reduce. A centralized KV cache manager handles memory mapping, which all GPU workers share. Each worker stores only part of the KV cache for its assigned attention heads. The scheduler coordinates memory management upfront, allowing GPU workers to execute independently and sync only intermediate results.

1.6 Implementation

vLLM is an end-to-end serving system with a FastAPI frontend and a GPU-based inference engine, featuring Python and CUDA code for control and custom kernels, supporting popular LLMs like GPT, OPT, and LLaMA with NCCL for distributed communication.

1.6.1 Kernel-level Optimization

To optimize PagedAttention's memory access patterns, vLLM introduces custom GPU kernels that fuse key operations: (1) reshaping, block writing, and saving KV cache to minimize kernel launch overheads; (2) block reading and attention to improve efficiency with coalesced memory access; and (3) a fused block copy kernel to reduce overhead from fragmented data movement.

1.7 Supporting Various Decoding Algorithms

vLLM supports decoding algorithms using three key methods: 'fork' (to create new sequences), 'append' (to add tokens), and 'free' (to delete sequences), enabling techniques like parallel sampling, beam search, and prefix sharing.

1.8 Evaluation

The authors evaluate vLLM under a bunch of different workloads; they find that vLLM susains much higher request rates than existing solutions.

1.9 Ablation Studies

The authors evaluate design choices through ablation experiments.vLLM's PagedAttention introduces some overhead (20–26% higher latency) in the attention kernel due to dynamic block mapping, but this impact is limited to the attention operator. Optimal performance is achieved with a block size of 16, balancing GPU utilization and minimizing fragmentation. For recovery, recomputation outperforms swapping with small block sizes, while both methods perform similarly for medium block sizes (16–64).

1.10 Discussion

The authors discuss the idea of applying virtual memory and paging to other GPU workloads.

2 FlashAttention

2.1 Introduction

The authors develop FlashAttention, a novel attention algorithm used to improve the efficiency of transformers by reducing memory access overhead.They use tiling and recomputation methods to restructure attention computation, avoiding repeatedly read/writes of the attention matrix to GPU memory. The authors' approach enables FlashAttention to run close to 8x faster, and consumes less memory than vanilla attention. FlashAttention is able to accelerate model training, getting much faster results on BERT-large, GPT-2, and long-range tasks. FlashAttention also improves model quality; block-sparse FlashAttention enhances speed and scalability even further, achieving faster performance than all known approximate attention methods.

2.2 Background

The authors provide some background about performance of DL operations on GPUs.

2.2.1 Hardware Performance

The authors focus on GPU performance, specifically its memory hierarchy and execution model. GPUs have larger, slower HBM, with small, fast on-chip SRAM, making it crucial to use SRAM efficiently. For kernel execution, data is loaded from HBM to SRAM, computed, and written back to HBM. Depending on the balance of computation and memory access, operations can be compute-bound or memory-bound. To optimize memory-bound operations, the authors used kernel fusion, reducing redundant HBM access by combining multiple ops. However, in model training, saving intermediate values for backpropagation limits the full benefits of naive kernel fusion.

2.2.2 Standard Attention Implementation

In regular attention, $S = QK^T$, $P = softmax(S) \in R^{NxN}$, and $O = PV \in R^{Nxd}$, with row-wise softmax. THe problem with this, is the memory-bound operations, slowing down the entire computation.

2.3 FlashAttention: Algorithm, Analysis, and Extensions

2.3.1 An Efficient Attention Algorithm With Tiling and Recomputation

The authors compute the attention and write it to HBM, reducing the r/w to HBM. Then, they apply tiling and recomputation to overcome the exact computations for attention. With tiling, the attention is computed by blocks, and compute softmax block by block, then combine. For recomputation, the authors recompute S and P easily, and often, with gradient checkpoint.

2.3.2 Analysis: IO Complexity of FlashAttention

FlashAttention significantly reduces high-bandwidth memory (HBM) accesses compared to standard attention, having faster execution and lower memory usage, especially for typical head dimensions. The authors show that exact attention algorithms can't improve HBM accesses, and show that HBM accesses influence runtime. Increasing block sizes in FlashAttention reduces HBM accesses and runtime, though performance flattens due to arithmetic ops.

2.3.3 Extension: Block-Sparse FlashAttention

Block-sparse FlashAttention is an extension of FlashAttention, where the authors compute only the non-zero blocks of the attention matrix, i.e. dense blocks. This optimization has a $O(N\sqrt{N})$ complexity, decreasing from FlashAttention by a factor proportional to the sparsity.

2.4 Experiments

In their experiments, the authors show that FlashAttention outperforms standard attention, on models such as GPT-2 small, medium, and BERT by up to 3.5x. Moreover, FlashAttention and Block-Sparse FlashAttention outperform every model in long range, long sequences, and long-context.

2.5 Limitations and Future Directions

The authors note that future work can be compiling to CUDA, IO-Aware Deep Learning methods, and Multi-GPU IO-Aware methods, for data transfer between GPUs.