Reading Summary Week 9

Aakash Agrawal
HDSI
University of California San Diego
San Diego, CA, 92092
aaa015@ucsd.edu

1 FlashAttention: IO-Aware Fast and Memory-Efficient Exact Attention

Transformers require a significant amount of memory when processing longer sequences because of both the time and memory complexity of the self-attention **scale quadratically** with sequence length. Many approximation methods, such as low-rank and sparse approximations, have attempted to reduce the compute time of attention to linear or near-linear complexity. However, these approaches often result in poor overall (wall-clock) speed compared to standard attention mechanisms. This is primarily because they focus heavily on FLOP reduction while overlooking IO and latency overheads introduced by frequent data access.

The authors argue that any effort to make attention more efficient should also account for **IO overhead**, i.e., the cost of reads and writes between different levels of GPU memory. Reducing FLOPs alone is insufficient if memory access remains a bottleneck, as frequent data movement between HBM (High Bandwidth Memory), cache, and registers can significantly impact wall-clock speed.

The authors propose **FlashAttention**, an **IO-aware exact** attention algorithm designed to reduce both memory usage and computational cost, particularly for long sequences. This approach avoids reading and writing the attention matrix to slow HBM by:

1. Computing the softmax reduction without needing access to the entire input (**online** fashion).
2. Eliminating the need to store the large intermediate attention matrix for the backward pass.

1.1 Background

GPU Memory Hierarchy: GPUs have a hierarchical memory system where smaller memory is faster but more limited in size. For example, the A100 GPU has:

- **HBM**: 40-80GB, 1.5-2.0TB/s bandwidth
- On-chip **SRAM**: 192KB per 108 streaming multiprocessors, 19TB/s bandwidth

Since SRAM is much faster but significantly smaller than HBM, memory access has become a major bottleneck as compute speed outpaces memory speed. Efficiently using SRAM is crucial for improving performance.

Performance Characteristics: Operations are classified as compute-bound or memory-bound based on their **arithmetic intensity** (AI = operations per byte of memory access):

- **Compute-bound operations**, such as matrix multiplication with large inner dimensions or convolutions with many channels, are limited by the number of computations rather than memory access.
- **Memory-bound operations** like elementwise functions (e.g., activation, dropout) and reduction tasks (e.g., softmax, batch norm, layer norm) are constrained by memory access speed rather than computation.

Kernel Fusion: means that if there are multiple operations applied to the same input, the input can be loaded once from HBM instead of multiple times for each operation.

Standard Attention Implementation: The standard attention implementation is highly inefficient in terms of memory usage due to its large IO overhead and the high number of **elementwise operations**. Frequent reads and writes between different levels of memory, particularly HBM, significantly slow down performance, making memory access a major bottleneck.

Algorithm 0 Standard Attention Implementation

```
Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.
1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.
```

1.2 Flash Attention

The key idea of Flash Attention is to compute attention by blocks to reduce global memory access. It uses two main techniques under the hood:

- **Tiling**: This involves restructuring the attention computation to load the query/key/values block by block from global to shared memory and incrementally performing the softmax reduction in an online fashion.
- **Recomputation**: This involves storing and tracking the softmax statistics for the block, like the normalization factor from the forward pass, and quickly recomputing attention on-chip (SRAM) in the backward pass. Hence, we avoid materializing the entire matrix while still obtaining precise softmax results. This technique called **selective gradient checkpointing**, trades extra compute (FLOPs) for memory. However, even with this extra forward-pass recomputation, we still speed up backward computation due to reduced HBM access.

The FlashAttention algorithm is both memory-efficient (**linear** in sequence length) and faster compared to other approaches. The key steps in the algorithm involve splitting the inputs (Q, K, V) into blocks, loading them from slow HBM to fast SRAM, and computing the attention output for each block. By scaling each block's output with the correct normalization factor before summing them, the algorithm ensures an accurate final result. The authors claim that no exact attention algorithm can asymptotically improve on the number of HBM accesses over all SRAM sizes.

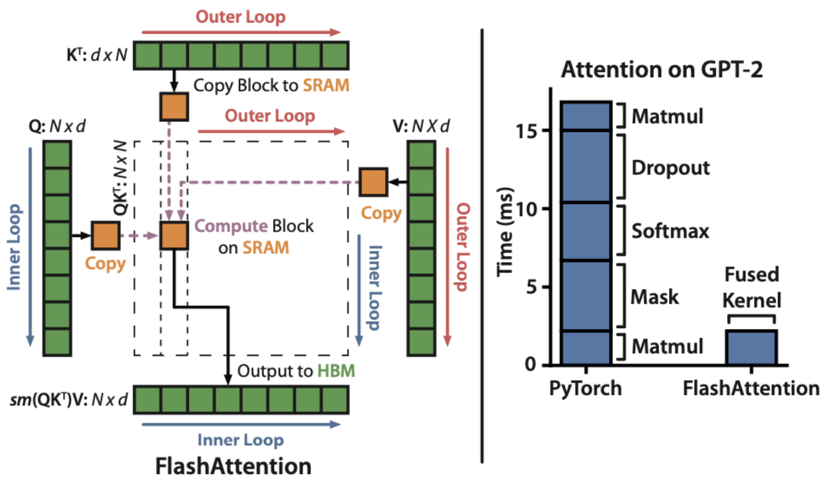

Figure 1: Overview of FlashAttention.

Block-sparse FlashAttention: is an approximate variation of FlashAttention that achieves 2-4× speedup while efficiently scaling to sequence lengths up to **64k**. It reduces computation and memory overhead by selectively computing attention only for specific blocks of tokens, optimizing performance for extremely long sequences.

1.3 Performance Evaluation

Although FlashAttention has a higher FLOP count than standard attention (due to recomputation in the backward pass), it significantly reduces HBM accesses, leading to a much faster runtime.

Attention	Standard	FlashAttention
GFLOPs	66.6	75.2
HBM R/W (GB)	40.3	4.4
Runtime (ms)	41.7	7.3

Figure 2: Forward + backward runtime of standard attention and FlashAttention for GPT-2 medium on A100.

Effect of Block Size: As block size increases, HBM accesses decrease since fewer passes over the input are needed, leading to **lower runtime**—but only up to a certain point. Beyond a block size of 256, performance becomes bottlenecked by other factors, such as arithmetic operations. Additionally, excessively large block sizes may exceed SRAM capacity, limiting their feasibility.

Higher Quality Models by modeling longer contexts/sequences: FlashAttention scales Transformers to handle longer sequences, improving model quality and unlocking new capabilities. For example, the authors observe a 0.7 improvement in perplexity on GPT-2 and a 6.4-point increase in performance for long-document classification. FlashAttention enables the first Transformer to achieve better-than-chance performance on the Path-X challenge solely by increasing sequence length to **16K**. Furthermore, Block-Sparse FlashAttention allows Transformers to scale to even longer sequences (up to **64K**), enabling the first model to achieve better-than-chance performance on Path-256.

Faster Model Training: FlashAttention accelerates Transformer model training, reducing wall-clock time significantly. It trains **BERT-large** (sequence length 512) **15%** faster than the MLPerf 1.1 training speed record from NVIDIA and **GPT-2** (sequence length 1K) **3×** faster than baseline implementations from HuggingFace and Megatron-LM (Here, the goal is to reach the target accuracy of 72.0% on masked language modeling).

Benchmarking Attention: The authors evaluate the runtime and memory performance of FlashAttention and Block-Sparse FlashAttention. They find FlashAttention's memory footprint **scales linearly with sequence length** and is up to **3× faster** than standard attention for sequences up to 2K. Block-Sparse FlashAttention also scales linearly and outperforms all existing approximate attention baselines.

1.4 Limitations

1. **Compiling to CUDA**: The IO aware implementation requires writing the attention algorithm in a lower-level language than PyTorch, and requires significant engineering effort. These implementations may also not be transferrable across GPU architectures.
2. **IO-aware deep learning**: While FlashAttention optimizes memory access for attention layers, other layers in deep learning models may still involve heavy data transfers with GPU HBM, leading to potential bottlenecks in overall performance.
3. **Multi-GPU IO-Aware Methods**: This paper talks about optimizing attention on a single GPU. However, the attention computation may be parallelizable across multiple GPUs, which might incur added complexities, such as accounting for data transfer between GPUs.

2 Efficient Memory Management for LLM Serving with PagedAttention

In this paper, the authors develop a new attention algorithm, **PagedAttention**, and build a high-throughput distributed LLM serving engine called **vLLM** on top of PagedAttention that achieves near-zero waste in KV cache memory. Managing the KV cache well is key to handling lots of requests in LLMs since it affects how many batch sizes we can process at once. A bigger batch size means more work is done at the same cost, improving efficiency. vLLM flattens the fast-rising memory demands of the KV cache seen in other systems, giving a big lift to how many requests it can handle.

KV Cache: Whenever you generate a new token, the values for K and V will not change for the previous positions. Hence, we can cache them instead of re-computing them for every generation. KV cache is essentially a memory space to store intermediate vector representation of tokens.

- **Without KV Caching**: For a sequence of length n, the model recomputes K and V for all n tokens every time it generates a new token, leading to a computational cost that grows **quadratically** with sequence length.
- **With KV Caching**: The model computes K and V once per token as it's processed, stores them in memory (the cache), and reuses them for all future steps. This reduces the computation to just the new token's Q, K, V, and the attention operation, making it much faster.

However, the size of the KV cache **dynamically grows** and **shrinks**, and its lifetime and length are not known a priori. Every time we generate a new token, we need to append a row in the KV cache table. Tokens are deleted once the sequence finishes. Improving the throughput is possible by batching multiple requests together. However, to process many requests in a batch, the memory space for each request should be efficiently managed.

2.1 Memory waste in KV Cache

To store the KV cache of a request in a contiguous space, systems pre-allocate a contiguous chunk of memory with the request's maximum length (e.g., 2048 tokens).

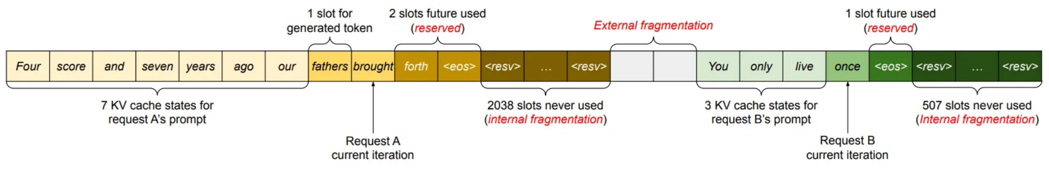
1 slot for generated token
2 slots future used (reserved)
External fragmentation
1 slot future used (reserved)
Four
score
and
seven
years
ago
our
fathers
brought
forth
<eos>
<resv>
...
<resv>
You
only
live
once
<eos>
<resv>
...
<resv>
7 KV cache states for request A's prompt
Request A current iteration
2038 slots never used (internal fragmentation)
3 KV cache states for request B's prompt
Request B current iteration
507 slots never used (Internal fragmentation)

Figure 3: KV cache memory management in existing systems.

- **Reservation**: refers to **pre-allocating memory** for the KV cache to handle the expected sequence length during inference. Since transformers generate text autoregressively, the KV cache grows as more tokens are processed, storing the keys (K) and values (V) for all previous tokens. To avoid constant reallocation of memory—which is slow and inefficient—systems often reserve a fixed block of memory upfront. This pre-allocated memory is not used at the current step, but used in the future. If the reserved size is too large (e.g., expecting 2048 tokens, but only 50 are used), then the memory is wasted.
- **Internal fragmentation**: Occurs when the reserved memory for the KV cache is not fully utilized, leaving unused gaps within the allocated space. Here, the memory is over-allocated due to the unknown output length.
- **External fragmentation**: This is when free memory is spread out in small, separated pieces across the system. Even if there's enough total free memory, it's tough to find one big, unbroken chunk for a new task. This happens because each request might reserve a different amount of memory, leaving gaps between used blocks.

Although the reserved memory is eventually used, reserving this space for the entire request's duration, especially when the reserved space is large, occupies the space that could otherwise be used to process other requests.

2.2 Paged Attention

PagedAttention is an attention algorithm inspired by the operating system's (**OS**) solution to memory fragmentation and sharing: **virtual memory** with **paging**. Here, the KV cache is stored and managed in **discontiguous**, **fixed-size pages** (called token blocks) rather than as a single, contiguous block of memory. (**Token block**: A fixed-size contiguous chunk of memory that can store token states from left to right.)

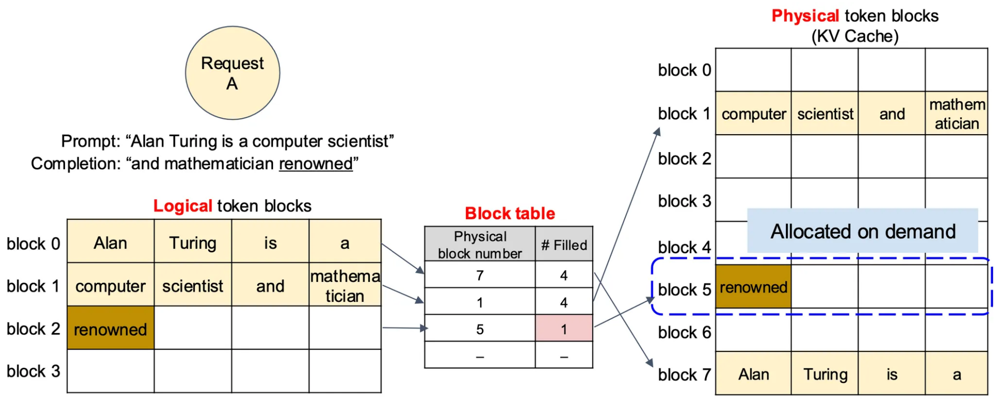

Figure 4: Block table translation in vLLM.

- PagedAttention divides the request's KV cache into token blocks, each of which can contain the attention keys and values of a fixed number of tokens. Previous systems required storing the KV cache in a continuous chunk of memory, whereas in PagedAttention, the blocks for the KV cache are not necessarily stored in a contiguous space. Hence, we can manage the KV cache in a more flexible way, as in **OS's virtual memory**: one can think of blocks as pages, tokens as bytes, and requests as processes. This design alleviates internal fragmentation by using relatively small blocks and allocating them on demand.
- The authors virtualize the KV cache into Logical & physical token blocks. In the logical view, the tokens are stored in contiguous blocks, and their order is preserved. In the physical view, the tokens might not be stored in contiguous blocks, and their ordering can be arbitrary.
- They also maintain a block table (like a page table in OS) that preserves a mapping between logical blocks and physical blocks. This mapping tracks which pages belong to which sequence and in what order. This allows the model to access K and V across non-contiguous memory locations. Each request has its own block table.

Attention Computation: During attention, the model uses the page/block table to fetch K and V from the relevant pages, concatenating them logically as if they were contiguous. The query (Q) for the current token is computed as usual and matched against the paged K and V.

Memory Management: Pages can be **allocated** and **deallocated** independently. When a request finishes, its pages are freed back to a pool for reuse by other requests. New pages/blocks are allocated only as the sequence grows. This **avoids reserving** a huge block upfront. This reduces memory pressure and fragmentation by working with smaller, reusable units. These memory savings translate directly into a higher batch size, which means higher throughput and cheaper serving.

In earlier systems, only **20–40%** of the KV cache is utilized to store token states. This approach results in **96.3%** KV cache utilization.

Table 1: **Summarizing the Pros & Cons of PagedAttention**

Pros	
1.	Paged Attention has minimal internal fragmentation as it only happens at the last block of a sequence. The number of tokens wasted per sequence will be less than the block size.
2.	It eliminates external fragmentation as all blocks have the same size.
3.	It also enables memory sharing at the granularity of a block across the different sequences associated with the same request or even across different requests.
4.	In multi-user systems, Paged Attention allows better memory sharing. Pages freed by one request can be reused by another, even if their sequence lengths differ.
Cons	
1.	Implementation Complexity: Requires more sophisticated memory management than a simple contiguous KV cache.
2.	Overhead: Managing a page table and accessing non-contiguous pages adds slight computational complexity (e.g., indirection lookups).