Omkar Yadav (oyadav), Vatsal Joshi (jvatsal), Lohit Kamatham (lohitk), Yoon Sung Ji (ajys)

SUMMARY OF FLASHATTENTION-2: FASTER ATTENTION WITH BETTER PARALLELISM AND WORK PARTITIONING

Problem and Motivation

In recent years, scaling Transformers to longer sequence lengths has been a challenge. Specifically, the attention layer has become a major bottleneck due to its quadratic increase in runtime and memory usage as sequence length increases. Despite the multiple efforts to reduce the computation load of attention on long contexts, a solution could not be found without sacrificing accuracy for optimization. That changed with the introduction of FlashAttention which exploited the asymmetry in GPU memory hierarchy to save memory (from quadratic to linear) and runtime (2-4 times faster than optimized baselines).

However, FlashAttention still falls short to other primitives such as the optimized matrix-multiply (GEMM) operation, achieving only 25-40% of the theoretical maximum FLOPs/s. The authors believe that the gap between FlashAttention and GEMM operations can be explained by the suboptimal work partitioning between different thread blocks and warps on the GPU, which causes low-occupancy or unnecessary shared memory reads/writes.

Related Works

To expand upon FlashAttention, in the forward pass, FlashAttention utilizes classical optimization techniques like tilting to (1) load blocks of inputs from high bandwidth memory (HBM) to SRAM, (2) compute attention to the blocks, (3) and update the output without writing the intermediate matrices S and P to HBM, reducing memory IOs. Since softmax is applied across entire rows or blocks of rows, FlashAttention's online softmax can split the attention computation into smaller blocks and rescale the outputs from each block to ensure correctness. This helps with longer input sequences.

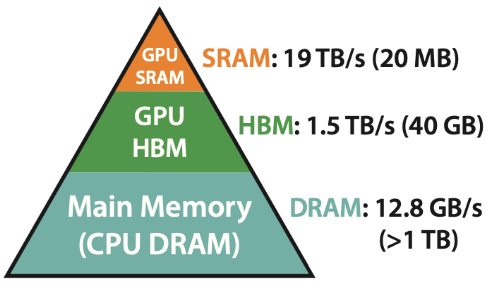


Figure 1: GPU Memory Hierarchy

FlashAttention further reduces memory consumption by re-computing the attention metrics S and P in the backward pass only when blocks of inputs (queries Q, keys K, and values V) are already loaded in SRAM. As a result, FlashAttention avoids having to store large intermediate values, dropping the memory usage 10-20x depending on sequence length and improving the wall-clock speedup by 2-4x. This is why FlashAttention is now widely adopted in large-scale training and inference of Transformers.

In addition to FlashAttention, there is the most recent FlashAttention-3 (brought up in class) and the key online softmax technique which allows FlashAttention. FlashAttention3 is focused more towards the speed-up of attention on Hopper GPUs. They discuss three main techniques: (1) use of warp-specialization to overlap computation and data movement, (2) interleave block-wise matmul and softmax operations, and (3) block quantization and incoherent processing. According to evaluation results, FlashAttention3 is 1.5-2.0x faster on H100 GPUs with FP16, and achieves 2.6x lower numerical error than baseline FP8 attention with GP8. Due to length/complexity, we decided to just mention the method and improved metrics here.

In the case of online softmax, it is a key idea which allows FlashAttention to build upon and achieve its optimizations (Online Softmax to FlashAttention). The online softmax technique to tile the self-attention computation, fusing the multi-head attention layer without GPU global memory for intermediate values and attention scores, is used to derive FlashAttention's computation. It reduces memory footprint while maintaining the accuracy of self-attention. FlashAttention is able to tile, process, and discard segmented intermediate values as the computation moves forward. This leverages the hierarchical memory architecture of modern GPUs.

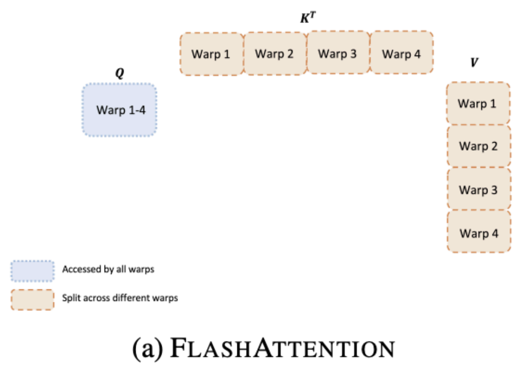
Solution Overview

The authors propose FlashAttention-2, which provides better work partitioning between different thread blocks and warps on the GPU. Specifically, the authors (1) transform the algorithm to reduce the number of non-matmul FLOPs, (2) parallelize the attention computation, and (3) distribute the work between warps within each thread block to reduce communication through shared memory.

First, non-matmul FLOPs are much more expensive (16x) than matmul FLOPs. Therefore, to optimize performance, the goal is to maximize time spent on matmul FLOPs. In order to do so, the authors apply two tweaks to the online softmax in the forward pass: (1) avoid rescaling both terms in the output update and (2) not save both the maximum and the sum of exponentials for the backward pass - only the logsumexp. In the case of the backward pass, it is almost the same as that of FlashAttention. The authors made minor changes to only use the rose-wise logsumexp instead of both the row-rise max and row-wise sum of exponentials in the softmax.

The previous FlashAttention parallelized over batch size and number of heads. In the case of FlashAttention-2, 1 thread block is used to process one attention head, and the number of thread blocks is equal to batch size · number of heads. For instance, this allows each thread block to run on 108 SM on an A100 GPU, providing scheduling efficiency by effectively using almost all of the compute resources. In the case of longer sequence lengths, additional parallelization occurs.

Similar to scheduling thread blocks, the paper discusses the partition method between different warps with the typical size of a thread block being 4 or 8 warps. FlashAttention-1 split K and V across 4 warps and kept Q for each block, utilizing the "split-K" scheme. However, this was inefficient since the warps wrote intermediate results to shared memory. These extra shared memory reads/writes slowed down the forward pass. In FlashAttention-2, Q is split across 4 warps while K and V is kept by all warps. As a result, there is no need for communication between warps and reduces the shared memory reads/writes.



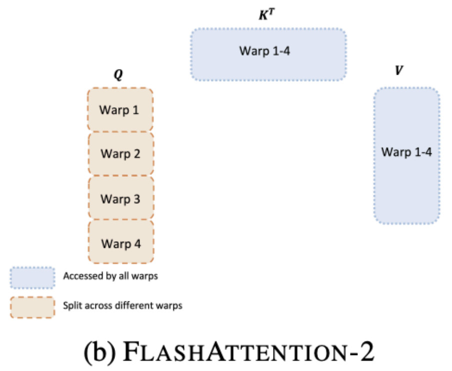


Figure 2: Warps in FlashAttention-1 vs FlashAttention-2

Limitations

1. There are also limitations in its scalability to newer hardware like H100 GPUs, as additional hardware features have yet to be optimized. The paper mentions that just running the same FLASH ATTENTION-2 implementation on H100 GPUs, they obtain only up to 335 TFLOPs/s. They further state that they expect by using new instructions, another speed-up of 1.5x-2x can be made on H100 GPUs.
2. Manual tuning of block sizes was necessary while performing FLASH ATTENTION-2. The authors state that they typically chose blocks of size $\{64,128\}\times\{64,128\}$, depending on the head dimension *d* and the device shared memory size. They manually tuned for each head dimension since there are essentially only 4 choices for block sizes. However, they state that this operation could benefit from auto-tuning to avoid manual labor.
3. While FLASH ATTENTION-2 shows major improvements, it still doesn't fully reach the efficiency of optimized matrix multiply (GEMM) operations possible within A-100 GPUs.
4. FLASH ATTENTION-2 only applies to modern GPUs where matrix multiplication operations have been optimized to run faster as it emphasizes increasing the number of matmul operations compared to FLASH ATTENTION-1.

Future Research Directions

In the near future, the authors plan to implement FLASH ATTENTION-2 in devices such as H100 GPUs and AMD GPUs with the help of other researchers and engineers. As an immediate next step, they plan to optimize FlashAttention-2 for H100 GPUs to use new hardware features (TMA, 4th-gen Tensor Cores, fp8). Combining the low-level optimizations in FlashAttention-2 with high-level algorithmic changes (e.g., local, dilated, block-sparse attention) could allow AI models to be trained with much longer context.

Summary of Class Discussion

Q: Can Flash Attention 2 be used in CPU as well? Since It leverages the matrix multiplication capability of the GPU, can it be used in a similar way in the CPU as well?
A: Most of the advancements from Flash Attention-2 are used within the GPU, because it reduces the memory I/O bottleneck while calculating attention. I suspect that the bottleneck for the CPU is not memory I/O and hence Flash Attention would not be needed here.

SUMMARY OF "SPECINFER: ACCELERATING LARGE LANGUAGE MODEL SERVING WITH TREE-BASED SPECULATIVE INFERENCE AND VERIFICATION"

Problem and Motivation

Naive LLM inference takes a large amount of time due to autoregressive sampling for the output text. The result is a consequence of the innate next word prediction nature of decoder transformer architectures. To mitigate this issue, previous works have leveraged a technique called sequence based speculative inference or speculative decoding. The high level idea is that we sample N tokens from a small model and send them to a large model to verify. The large model will verify these tokens in one forward pass and accept those that match. This process speeds up inference as you need not sample from the large model autoregressive – instead we sample from a quicker smaller model autoregressive and verify in parallel.

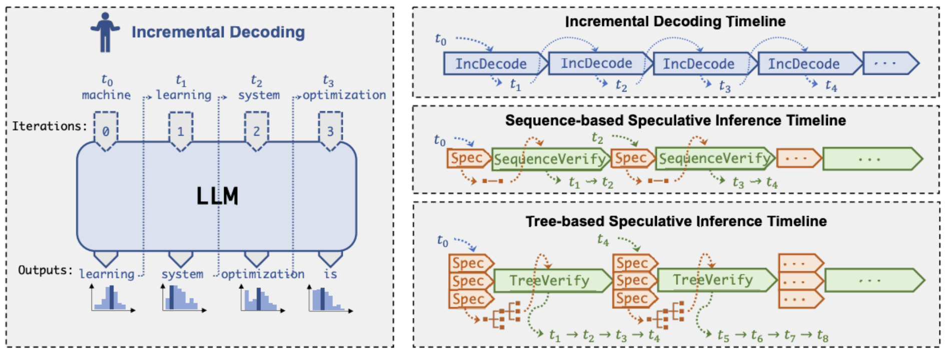
One of the main drawbacks of this approach that this paper aims to solve is that the token acceptance rate of the smaller models is often low because of the difference in model sizes. For reference, the difference in number of parameters between the small and large model is one order of magnitude.

Related Works

1. Lossless Acceleration: Past research has explored methods to enhance LLM inference speed without compromising output quality. The main method in recent works is speculative decoding, where a smaller language model is used to generate token sequences that are verified by an LLM, inspired by speculative execution in processors..
2. Lossy Acceleration: Lossy acceleration trades some degree of output quality for increased inference speed. BiLD is a lossy speculative decoding framework that uses a single SSM, and model compression techniques like weight/activation quantization and structured pruning allow for reduced computational and memory load.

Solution Overview

This paper introduces a system called SpecInfer which improves the efficiency of LLM inference by introducing tree-based speculative inference. The intuition behind this approach is that only having one speculative sequence yields a lower token acceptance rate especially when the sequence grows in length. Thus, tree based speculation asks the small model to come up with multiple possible tokens on each path of the inference process. This improves the token acceptance rate as the small model's output covers many more possibilities as shown by figure 1.



(a) Incremental decoding. **(b)** Timeline Comparison.

Figure 1: Speculative Decoding Architecture

There are two methods to make the token tree. The first is an expansion based architecture on one SSM. At each branch of the tree we decide to sample k words from a subset of branches to regulate the size of the tree. The other method is merge-based where we have multiple SSMs that generate their own trees. Then, these trees are merged into one for the final token tree output. The merge-based method also has individual SSMs that are fine tuned using Adaboost improving and specializing each of their outputs.

The other main step in the SpecInfer system is the verification process called the token tree verifier. The first challenge in this process is that the attention scores for the output tree must be computed. To solve this, the paper introduces tree attention which captures the dependencies between nodes in the tree in the score calculation. Next is the parallel decoding which includes depth-first search (DFS) for the key-value (KV) cache and GPU optimization. At a high level, DFS traverses the token tree and creates a shared KV cache. On the GPU side, the authors introduce a topology-aware causal mask that fuses tree attention for all of the tokens into a single GPU kernel. This ensures that there is not an increase in GPU overhead. For the verification, there are two approaches the paper discusses: greedy and stochastic verification. Greedy decoding simply selects the token with the highest probability at each decoding step while stochastic decoding samples from the LLM probability distribution. Based on the approach, different verification algorithms are run on the output tokens.

Limitations

SpecInfer may face significant challenges in compute and memory-bound applications due to the complexity of managing token trees and parallel verification processes.

- The paper's example of using 4 GPUs to serve 2 SSMs and four requests raises concerns about energy efficiency, cost, and issues like inter-GPU/CPU latency, memory indirection, and non-contiguous memory access patterns, potentially negating inference latency benefits
- There's a lack of detailed exploration of SpecInfer's performance with very large token trees, which would be common in modern language models with huge vocabularies. Wider trees required for these vocabularies could offset many of the promised performance benefits.
- Using a large tree width also increases latency for token tree verification due to fewer spare GPU resources available to SpecInfer.
- As batch sizes increase, available GPU resources decrease, further limiting SpecInfer's ability to perform tree-based decoding effectively.

Future Research Directions

1. Dynamic Token Tree Expansion: The current SpecInfer uses a static strategy for token tree expansion, following a present configuration. Future research could explore dynamic expansion strategies that adapt to input complexity, context, and runtime performance, such as algorithms that adjust the expansion configuration in real-time based on input length, vocabulary distribution, or even feedback from the LLM verification process.
2. Advanced Ensemble Learning for SSMs: In the paper, they used AdaBoost to combine outputs from multiple SSMs, but future research could explore other ensemble learning methods like voting, bagging, and stacking techniques.
3. Integration with Hardware-Specific Optimizations: SpecInfer’s techniques are orthogonal to existing hardware-specific ML computation optimizations. Future research could explore integration with TVM, Ansor, TASO, and PET, which automatically generate optimized kernels.

Summary of Class Discussion

Q: Do the number of verification passes change now that there is a tree tokenizer?
A: No, all the tokens are still verified in one pass of the larger model.

Q: What is the time difference between running speculative inference vs incremental inference
A: In speculative decoding, the verification process is the bottleneck. Overall, the speculative inference process is faster than incremental inference because you usually get more than one token in each pass of the LLM. There are many factors that impact the speedup from speculative inference – the main one being the token acceptance rate. If the smaller model is not well equipped to perform inference on the queries, we will actually observe a decrease in throughput.

Q: What if there is no sensible token generated by the small tree?
A: If none of the SSMs generate any correct tokens, then the larger model's decoded token is used.

Q: Since the LLM still has to do the decoding step, what is the advantage?
A: Normally, we get multiple tokens from a speculative inference pass where a naive approach would only get one token.

Q: Which step of the tree based decoding process is the one where you are saving inference time if the ground truth LLM is still run incrementally?
A: It's not exactly incremental decoding. We send all of the outputs from the smaller model to the LLM and it runs one attention pass, which allows the larger model to output multiple tokens. Otherwise, we would have to run the attention pass multiple times in traditional speculative inference.

Q: Is there any information on the actual throughput of the model? The paper doesn't talk about CPU throughput and bottlenecks with GPU outputs.
A: The throughput is related to the latency graph because a lower latency implies a higher throughput.

Q: If you have a sequence of probability distributions, can you just do incremental decoding from this?
A: The large model generates its own probability distribution for the tokens that will be output. The sampling compares the probability of a given token t from both the SSM and the large model to determine the output.

Q: Does knowledge distillation take place for this tree-based speculation similar to how it's done for incremental decoding?
A: It seems like the only distillation that takes place is when they train the ensemble SSM models. No distillation takes place in the inference passes.

Comment: It seems like as the tree width increases, there is a point where the generation from the SSM takes longer than incremental speculative decoding