FLASHSPECINFER: MINIMAL AND MEMORY-EFFICIENT IMPLEMENTATION OF SPECINFER

Vineeth Kada [1] **Shrey Gupta** [1] **Koushik Guntakanti** [1]

1 INTRODUCTION

In our paper, we aim to address the pressing need for efficient inference techniques for Large Language Models (LLMs). The motivation behind our work emerges from the ever-growing demand for scalable LLM inference across various applications, ranging from natural language understanding to generation. Our work is crucial in this field as it tackles this fundamental issue while being memory efficient.

Our main contributions lie in two key areas. Firstly, we propose a novel approach to SpecInfer to optimize memory consumption significantly, thereby also improving runtime performance. Secondly, we present a very minimal implementation of our idea using plain CUDA code without any libraries like cuBLAS and cuDNN. The motivation behind that is to learn the concepts behind CUDA programming and to see how close we can come to PyTorch implementation of FlashAttention 2 using around 150 lines of CUDA code. Our code can be found at: **https://github.com/Vineeth-Kada/flash-specinfer**

Before going into the specifics of the problem, we would like to briefly introduce the recent advances in the field of LLM inference. LLMs generate text using an incremental decoding strategy which is very slow as it generates only one token per iteration and also requires a large number of memory accesses. A strategy called speculative decoding (Chen et al., 2023; Leviathan et al., 2023) was introduced to speed up LLM inference by using a small speculative model (SSM) to generate a sequence of tokens incrementally and use an LLM to verify them in parallel and accept more than one token by rejection sampling. This method can produce speed-ups in the range of 2-2.5x but using only a single SSM can result in low acceptance rates in the verification phase as the SSMs are generally orders of magnitude smaller than LLMs and may not be aligned well. SpecInfer (Miao et al., 2023) proposes a tree-based speculative inference that uses multiple SSMs to generate token sequences which are then organized as a token tree with each path from root to leaf representing a potential token sequence. This tree of sequences is verified by an LLM in parallel by using a causal mask. Leveraging these tree structures has produced a 96-97% acceptance rate of tokens as compared to 52-57% of sequence-based speculative inference.

[*]Equal contribution [1]Carnegie Mellon University, USA. Correspondence to: Vineeth Kada <skada@andrew.cmu.edu>, Shrey Gupta <shreygup@andrew.cmu.edu>, Koushik Guntakanti <sguntaka@andrew.cmu.edu>.

MLSys course project

2 PROBLEM

As stated in the previous section, SpecInfer combines the speculations from multiple SSMs into a token tree, as there are commonalities across speculations. One of the key bottlenecks in SpecInfer that prevents it from scaling to large trees is the explicit computation of the N x N causal mask, as illustrated in Figure 1, where N denotes the number of nodes in the speculated tree. For instance, it restricts us from incorporating additional SSMs or generating more speculations through various sampling strategies from the same SSM. A scalable method for computing this causal attention would enable us to compare the model output against a large number of speculations. This project aims to enhance the scalability of SpecInfer by compressing this causal mask using the tree structure. The second problem we are tackling is the minimal attention computation CUDA implementation from scratch. Most FlashAttention implementations on Github rely on libraries. The rest of them are very inefficient. By starting with one such popular minimal implementation cited by Andrej Karapathy in him llm.c project, we provide a simple CUDA-only implementation of our concept without relying on libraries like cuBLAS and cuDNN. This approach aims to explore how closely we can replicate PyTorch's FlashAttention 2 in approximately 150 lines of CUDA code.

3 RELATED WORK

FlashAttention (Dao et al., 2022) introduces techniques aimed at reducing memory usage and enhancing runtime efficiency within the context of self-attention mechanisms. It achieves these improvements by bypassing the explicit computation of the weight matrix QK^T. Instead, it updates the attention directly using blocks of Q, K, and V, and utilizes an online softmax algorithm. To minimize data

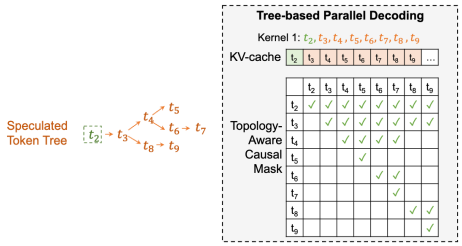

Figure 1. Causal mask used by SpecInfer for the token tree shown on the left

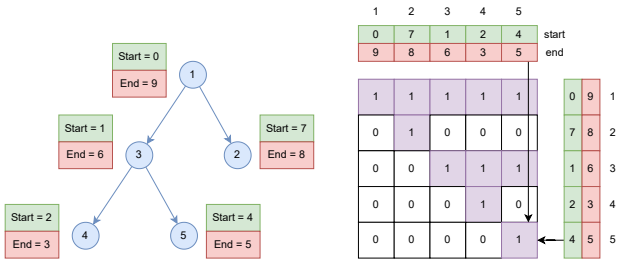

Figure 2. An example graph and its causal mask are shown, where the j^{th} column denotes all the nodes that the j^{th} token attends to. This representation is the transposed version of how we typically depict the causal mask. We can also observe how this relationship is captured by just the start and end times of a Depth First Search (DFS) traversal. The start and end time interval of an ancestor is a superset of the interval of the descendant.

movement between the GPU's main memory and its fast cache, FlashAttention employs classical tiling techniques. This strategy reduces the memory footprint, decreases global memory accesses, and speeds up execution. FlashAttention-2 (Dao, 2023) builds on this foundation by further reducing non-matrix multiplication (non-matmul) floating-point operations (FLOPs), introducing parallelism across the sequence length, and altering the workload partitioning among warps within each thread block. These changes aim to minimize synchronization and communication between warps, enhancing performance.

While FlashAttention and FlashAttention-2 focus primarily on optimizing self-attention computation, our proposed method draws inspiration from their memory-efficient techniques. We apply these principles to develop efficient kernels specifically for the computation of causal masks in tree-attention mechanisms, which are part of tree-based speculative decoding in SpecInfer. Our approach takes into account the dependency of the causal mask on the tree structure, aiming to reduce both the memory footprint and the runtime of SpecInfer.

4 Overview

As stated in the previous section, we are tackling two related problems in our course project. Consequently, we have included two figures instead of one, as specified in the instructions. The first problem involves compressing the tree causal mask, as shown in Figure 2. The second problem focuses on developing a minimal yet fast FlashAttention implementation, starting with the CUDA code from https://github.com/tspeterkim/flash-attention-minimal as our baseline, as illustrated in Figure 3. Our solutions are detailed in the following section.

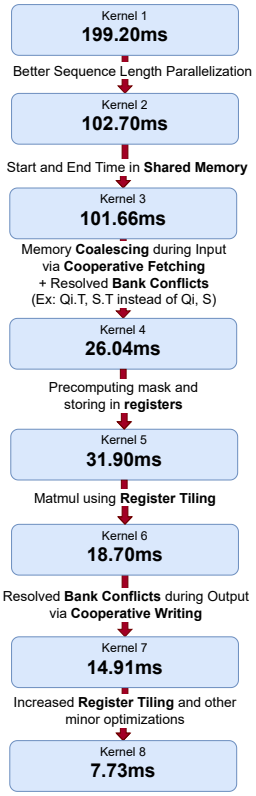

Figure 3. For the configuration—Batch = 10, Heads = 10, Sequence Length = 1024, Head Embed Dim = 64—the figure illustrates the techniques we used in the CUDA attention kernel implementation and the improvements we achieved compared to the baseline implementation. For comparison, PyTorch runtime (FlashAttention2) for this configuration is 5 ms.

5 Method

In this section, we will go into our methods in a detailed fashion. First, we will discuss our causal mask compression technique that reduces memory consumption from O(N x N) to O(N) per prompt. Then, we will talk about how we came

up with a CUDA-only minimal implementation of attention computation with only 150 lines that is competitive with PyTorch in many cases and even beats PyTorch in some cases. We added the tree attention computation (SpecInfer) to that attention kernel. Note that we didn't use any libraries like cuBLAS or cuDNN.

5.1 Compressing O(N * N) tree causal mask to O(N)

In FlashAttention the weight matrix, QK^T, isn't computed explicitly. Rather it is computed on demand using Q and K with an online softmax algorithm thus requiring only $O(N)$ memory instead of $O(N * N)$ which translates to less memory requirement and thus facilitating scalability as CUDA will not throw out of memory error easily in $O(N)$ case unlike $O(N*N)$ case. But the problem with SpecInfer is that even if don't compute QK^T explicity, the causal mask becomes a memory bottle neck requiring $O(N * N)$ memory. In this subsection let's see how to compress that mask.

In contrast to simpler sequence models where the causal mask is straightforwardly defined as $\text{mask}[i, j] = 1$ if $i \leq j$, otherwise 0, the SpecInfer framework introduces complexity due to dependencies on tree structures. This results in non-uniform causal masks that vary depending on the input tree's topology, as illustrated in Figure 1.

To address this, we employ a depth-first search (DFS) strategy to determine the critical timestamps for each node within the tree: the start time (when a node is first visited) and the end time (when the subtree rooted at the node is fully explored). These times are pivotal for constructing the causal mask. Specifically, the causal mask's entry (i, j), which indicates whether node i is an ancestor of node j, is defined by the following conditions:

$$\text{mask}[i, j] = \begin{cases} 1 & \text{if start}[i] \leq \text{start}[j] \text{ and end}[i] \geq \text{end}[j], \\ 0 & \text{otherwise.} \end{cases}$$

This approach is depicted in Figure 2, where the mask is constructed based on the traversal times obtained from the DFS, ensuring that the causal dependencies conform to the ancestral relations inherent in the tree structure.

Why DFS Start and End Times Determine Ancestry
During a DFS traversal, the *start* time for each node is recorded when the node is first visited. This timestamp indicates the initiation of exploration for that node and its subsequent subtree. Conversely, the *end* time is marked when all nodes within the subtree rooted at that node have been fully explored and the traversal backtracks to its predecessor.

These timestamps are key to determining ancestry because:

- A node i is an ancestor of a node j if and only if i's traversal is initiated before j's (*start*[i] $\leq$ *start*[j]) and i's subtree includes j, which is completed before i's exploration ends (*end*[i] $\geq$ *end*[j]).
- This relationship holds as DFS ensures that once a node is visited, all its descendants are explored before the traversal returns to explore any of i's siblings or ancestors, thus encasing the entire descendant subtree within i's start and end times.

5.2 Minimal SpecInfer implementation

Kernel 1
To demonstrate the effectiveness of our approach, we aimed to create a minimal implementation of the tree attention computation kernel. We were searching online for minimal implementation of FlashAttention-2 that doesn't use any libraries like cuBLAS but couldn't find any optimal ones. The closest we came across was https://github.com/tspeterkim/flash-attention-minimal cited by Andrej Karapathy in his llm.c project. Instead of relying on complex and hard-to-understand code from the FlashAttention 2 repository, we decided to write everything from scratch using only the techniques learned in class, without employing any CUDA libraries. This approach allowed us to conduct ablation studies and gain a deeper understanding of the underlying mechanisms. Our ultimate target to write a FlashAttention kernel using just plain CUDA code and see how close we can come to PyTorch, then later include SpecInfer into that code. In this subsection we will talk about that attention kernel implementation starting with the above method as our baseline kernel.

Notation
Before going further, in Table 1, we the present notation used. We will use the same notation as FlashAttention-2.

Symbol	Meaning
B	Batch Size
N	Sequence Length (Tree Nodes)
nH	Number of Heads
d	Head Dimension
Br, Bc	Tile Size (row, col)
Tr, Tc	# Tiles (rows, cols)
Qi, Kj, Vj	Dim: (Bc x D), (Br x D), (Bc x D)
S	Qi @ Kj.T
StartT, EndT	DFS Start and End Times

Table 1. Notation

Kernel 2
We made several incremental improvements to optimize the performance of our kernels. As the first step, we introduced more parallelization over the sequence length by removing a for loop in our baseline implementation. The reduced the runtime from 199 ms to 103 ms, which is nearly a two-fold

improvement with minimal modifications. All the timings mentioned in this section uses the configuration of (B, nH, N, d) = (10, 10, 1024, 64). They were using a (B x nH) grid in GPU with 32 threads per block. Note that in this project we took Br = Bc = 32 as static values same as the baseline repository. Hence each thread is processing (N = Tr * Br) / 32 = Tr number of queries. Now let's answer the question: Why didn't they take (B x nH x Tr) grid? We hypothesize that it might be because if B x nH is exactly sufficient to occupy all the SMs then there is no point in using (B x nH x Tr) as it might result in additional overhead. But that is a problem when the SMs are capable of processing more than (B x nH) blocks at a time like for instance in edge inference where B = 1. So we decided to go with the (B x nH x Tr) config and as we show in the experiemental section kernel 2 is consistenly outperforming kernel 1 in all the extensive list of configurations we tested.

```
dim3 grid_dim(B, nh);        // Kernel 1
dim3 grid_dim(B, nh, Tr);    // Kernel 2
```

Kernel 3

For the next optimization, we are storing DFS start and end times in shared memory instead of fetching them from global memory every time we use the causal mask. By fetching these values into shared memory, we aimed to reduce GPU RAM accesses, as the causal mask is calculated in every matrix multiplication operation. This optimization provided only a small improvement of approximately 2 milliseconds. It is worth noting that when the causal mask fits into the GPU memory, it does not act as a bottleneck since the complex matrix multiplication operations dominate the runtime as noted in our experiments.

Kernel 4

This introduces a significant optimization by altering the memory access patterns. This access pattern is proposed by us. Unlike the FlashAttention-2 paper that suggests a naive approach of assigning each row of attention matrix computation to one thread we show that there is a better way of splitting the workload. First let's talk about the work division among threads in populating the shared memory then in the later parts we will talk about work division in attention matrix computation. Fetching one row of Q, K, V by one thread lacks memory coalescing and results in bank conflicts. To address this issue, we employ a pattern, depicted in Figure 4, where all threads cooperatively fetch the first 32 elements in the first row, followed by the next 32 elements in the first row, and so on. Once the first row is complete, the threads move on to the second row. This approach eliminates bank conflicts and enables consecutive threads in a warp to access consecutive GPU RAM locations and store them in consecutive shared memory locations. Additionally, we discovered that storing query tile, Qi, and the attention matrix tile, S, in transposed form (QiT, ST) facilitates better memory coalescing and further reduces bank conflicts. Overall, this change in access patterns yielded a four-fold improvement, reducing the runtime from 100 milliseconds to 26 milliseconds. With that access pattern we were able to achieve 100% memory coalescing and resolved all the bank conflicts except fetching QiT but that is not a bottle neck so it didn't make much difference. The code snippets below illustrate this optimization. tx is threadIdx.x and if it occurs at the end it facilitates memory coalescing and resolves bank conflicts as Bc = 32 and we are using number of threads per block as 32 and a warp contains 32 threads. Also note how transpose of Qi, S are helping to get tx at the end of indexing.

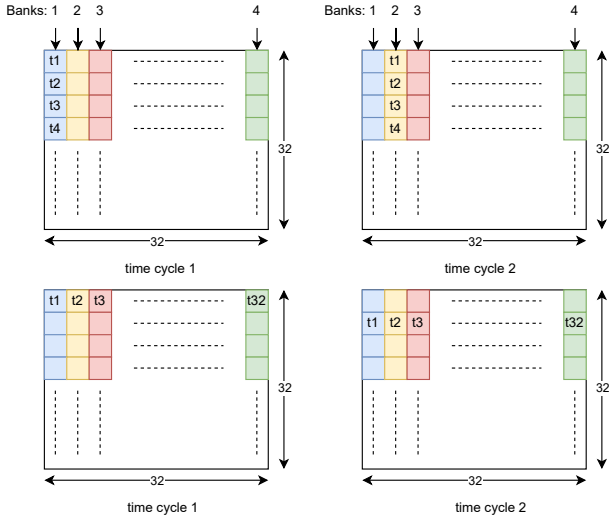

Figure 4. Illustration of access pattern in a simple case where d = 32. The top figure shows the naive access pattern of one thread one row of Q, K, V and the bottom figure shows our new proposed access pattern.

```
<------------ Kernel 3 ------------>
for (int x = 0; x < d; x++) {
  Kj[(tx * d) + x] = K[... + (tx * d) + x];
  Vj[(tx * d) + x] = V[... + (tx * d) + x];
}
...
sum += Qi[(tx * d) + x] * Kj[...];
...
S[(Bc * tx) + y] = sum;

<------------ Kernel 4 ------------>
for(int start = 0; start < Bc*d; start+=Bc){
  Kj[start + tx] = K[... + start + tx];
  Vj[start + tx] = V[... + start + tx];
}
...
sum += QiT[(x * 32) + tx] * Kj[...]
...
ST[... + tx] = sum;
```

Kernel 5
Moving on to Kernel 5, we introduce a small optimization by precomputing the mask and storing it in registers instead of explicitly computing it whenever needed using shared memory. While this optimization comes with additional overheads due to the GPU having to create and maintain significantly more registers, it provides slightly better performance in some configurations and slightly worse performance in others.

Kernel 6
This focuses on improving matrix multiplication using register tiling, a technique discussed in class. This optimization yields significant improvements over the previous iteration, reducing the runtime from 31 milliseconds to 18 milliseconds. This again deviates from the FlashAttention-2 way of doing things where we assign one thread to one row. In register tiled matrix multiplication we use all the threads to compute a small tile of the result. In this kernel we only did this optimization for the softmax(S)@V multiplication just to test how this works and in the later iterations we implemented register tiling to compute S itself.

Kernel 7
We apply the similar memory access patterns discussed in Kernel 4 to the write the output of matmul to shared memory. By resolving bank conflicts and changing the work allocation algorithm for different threads, we ensure that threads operate on good locations and store the results in consecutive shared memory locations, facilitating efficient cooperative writing. An illustrative code snippet is shown below where as you can see tx = threadIdx.x is at the end of indexing.

```
<------------ Kernel 6 ------------>
for (int x = 0; x < d; x++) {
  O[... +(tx*d)+x] = row_m_exp * \
    O[... +(tx*d)+x] + PVi[(tx*d)+x];
}

<------------ Kernel 7 ------------>
row_m_exp_shared[tx] = row_m_exp;
for(int start = 0; start < Bc*d; start+=Bc){
  Oi[start+tx] = row_m_exp_shared[start/d] \
    * Oi[start+tx] + PVi[start+tx];
}
```

Kernel 8
We employ register tiling for the matrix multiplication used to compute attention matrix $S = QiKj^T$. The attention kernel involves two matrix multiplications, and by applying register tiling to both of them, along with other minor optimizations, we achieve a runtime of 7 milliseconds, starting from an initial runtime of 200 milliseconds. This performance is very close to what PyTorch achieves (4ms) using Flash Attention-2.

These iterative refinements highlight the potential of straightforward CUDA kernels to speed up computations and attain performance levels comparable to well-established libraries such as PyTorch. By meticulously identifying and mitigating the performance bottlenecks throughout the process, we managed to secure substantial speed improvements, narrowing the performance gap with the finely tuned PyTorch implementations.

6 EVALUATION

In our experimental setup, we compared our method, which uses DFS-based start and end times, against the PyTorch implementation that employs the full causal mask. When the sequence length (tree size) is large, PyTorch fails to allocate sufficient memory for the causal mask on the GPU, resulting in a CUDA out-of-memory error. Consequently, our method is clearly superior in such cases and scales well with the sequence length (tree size in SpecInfer). In such cases our approach yields significant runtime improvements, as we must store the causal mask in RAM or disk and transfer it to GPU memory in segments as needed. In such cases, we observed improvements of more than 10x, even compared to our inefficient kernels. Hence, to see interesting results, let's focus on the scenario where the causal mask fits into memory. Before moving on to the runtime experiments, let's discuss the consequences on memory consumption. For all the experiments we are using a single A6000 GPU or L4 GPU. If nothing is specified under the figure assume A6000.

6.1 Memory Gains

Our approach significantly reduces the memory footprint compared to the original SpecInfer implementation. Instead of performing N^2 global memory accesses multiplied by the batch size and head dimension (as each head and batch is processed by a different thread block), our method only requires $2N$ global memory accesses as well as storage. Although this may not be the primary bottleneck in terms of runtime compared to matrix multiplication when the causal mask fits into GPU memory, it becomes crucial when considering storage limitations, particularly when the sequence length N is large. For instance, if we have a sequence length of 4096 and a batch size of 128, storing the full causal mask would require 2 Gbits of memory. In contrast, our method, which utilizes DFS start and end times, only requires 32 Mbits of storage, resulting in a substantial reduction in memory usage. One point to note here is that the causal mask is just boolean so each entry is 1bit but start and end times are integers so they are 4bytes = 32bits. Even then we still get memory gains as N becomes more than 64. This memory efficiency allows our approach to handle larger sequence lengths that would otherwise be practically infeasible due to GPU memory constraints.

6.2 Runtime Ablation

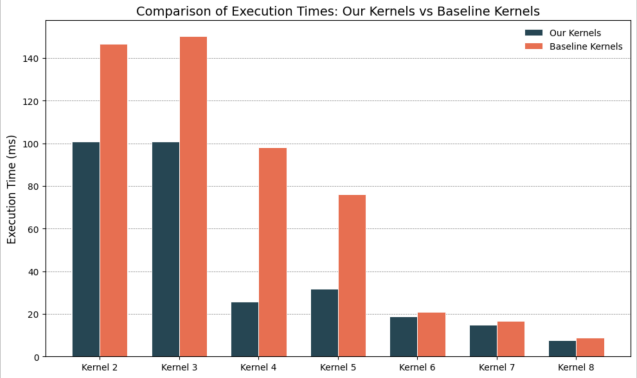

Figure 5. Full vs Start-End time mask runtime ablation study for the competitive case where mask fits into GPU memory.

In the previous subsection, we posited that if the causal mask fits entirely within GPU memory, it does not become a runtime bottleneck, given the presence of complex matrix multiplication operations which dominate computational overhead. To empirically examine this, we carried out an ablation study. This study compared our kernel with a baseline variant of the exact same kernel, which differs primarily in that it employs a full causal mask instead of selective start-Time and endTime masking. Ideally, the baseline should represent scenarios where the causal mask exceeds memory capacity, though such conditions would predictably yield significantly poor performance. Instead, we chose a more competitive case by assuming the causal mask is entirely accommodated within the GPU's memory.

As illustrated in Figure 5, as kernel optimization progresses—particularly through more efficient handling of the mask in shared memory—the reading of the causal mask ceases to be a bottleneck. Nevertheless, our method, which uses startTime and endTime based causal masking, continues to outperform the baseline, albeit by a slight margin. This advantage is intuitive as matmul is a considerable bottle neck compared to the shared memory population. Once again, it is crucial to note that when the causal mask exceeds the available GPU memory, it becomes a serious bottleneck, as the CPU would need to retrieve data from CPU RAM or even persistent storage.

6.3 Runtime vs PyTorch

As we showed in the previous subsection, when the causal mask fits into the GPU memory, it won't create a runtime bottleneck as matmul is much more expensive. Hence, we further compare the runtime of our kernels against the PyTorch implementation (which uses FlashAttention-2) across different sequence lengths and batch sizes. This comparison is one of our goals, as we aim to assess how closely we can match PyTorch's performance with plain CUDA code, without using any libraries like cuBLAS.

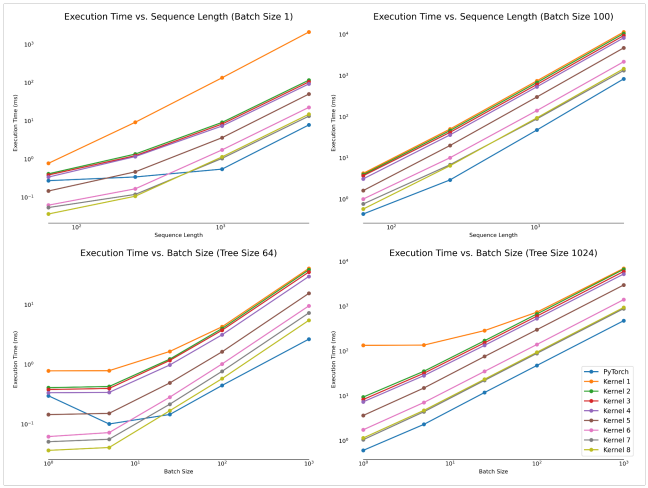

Figure 6. Results - Runtime vs PyTorch (A6000 GPU)

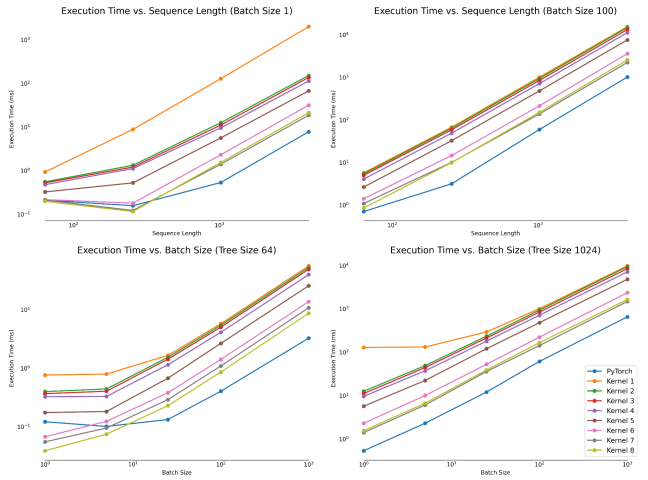

Figure 7. Results - Runtime vs PyTorch (Colab L4 GPU)

We conducted experiments on two different GPUs and observed consistent results across both setups without any fine-tuning, demonstrating the robustness of our approach. The results are presented in Figures 6 and 7. Notably, when the sequence length is small, as is typically the case in SpecInfer where tree sizes are relatively small, our method outperforms PyTorch. Additionally, one very interesting case occurs when the batch size is one (the top left plot), which is particularly relevant for edge devices. This setup is useful for projects like MLC-LLM, where the batch size is usually set to one since the user provides one prompt at

a time, unlike in industrial settings. Our method performs very well in terms of runtime in such cases. This is because while writing kernels, we focused mainly on sequence parallelization and just processed batches and the number of heads naively by assigning each batch to a block, hence we achieved good results in the batch size 1 case. One last point we would like to make is that we are not using any hyperparameter optimizations; we just used a single configuration for all the experiments and still came very close to PyTorch with just 150 lines of plain CUDA code, and also we beat the baseline we took from GitHub by more than 30 times.

6.4 SRAM Usage

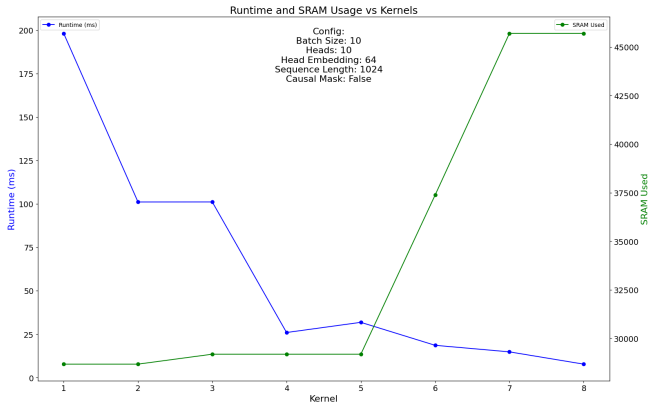

Figure 8. Results - SRAM Usage vs. Kernel Runtime. A6000 has 48kB of SRAM per block and we are using 45kB in kernel 8.

Efficient utilization of SRAM is essential for optimizing the performance of our kernels. As we made iterative optimizations to kernels, we utilized more and more SRAM, which resulted in a reduction of runtime. This underscores the significance of meticulously managing SRAM resources to achieve peak performance, as demonstrated in Figure 8. It is noteworthy that, given the availability of multiple kernels that use varying amounts of SRAM, based on specific use cases we can select an appropriate kernel based on available SRAM. For example, if a device has limited SRAM, we might opt for Kernel 4, which performs comparably to Kernel 8 but significantly better than Kernel 1. This also reflects the principle of diminishing returns in SRAM utilization in our methods.

7 Conclusion

Summary

In this work, we present a very minimal and memory-efficient implementation of SpecInfer. By leveraging depth-first search (DFS) to compute start and end times for each node in the token tree, we eliminate the need for explicitly storing the $N \times N$ causal mask, resulting in significant memory savings. Our approach reduces the memory footprint from $O(N^2)$ to $O(N)$, enabling the handling of larger sequence lengths that would otherwise be practically infeasible due to GPU memory constraints. Through iterative optimizations and careful memory access pattern design, we achieve runtime performance close to that of the highly optimized PyTorch implementation. Runtime wise, our method outperforms PyTorch for small batch sizes which is particularly relevant for edge devices and scenarios where the batch size is limited to one. The simplicity of our implementation, requiring only around 150 lines of CUDA code, showcases the effectiveness of our approach in developing a minimal and efficient version of FlashSpecInfer without relying on complex libraries like cuBLAS. This work highlights the potential for further optimizations in the field of efficient transformer inference, opening up avenues for future research and development.

Lessons Learnt

In the course of conducting experiments on GPU programming, several critical lessons were learned regarding efficient memory usage and hardware utilization. These lessons are how we got great improvements compared to the baseline repository. Firstly, it is essential to minimize memory reads from High Bandwidth Memories (HBMs) and to maximize the efficiency of shared memory and registers. This involves careful allocation and access patterns that reduce the need for frequent data fetching from slower memory tiers. Secondly, memory coalescing is crucial; it is beneficial to avoid bank conflicts by ensuring that consecutive threads access contiguous memory segments. This alignment enhances access speed and overall performance. Regarding hardware efficiency, an intimate understanding of the GPU architecture is necessary, such as the amount of shared memory, the number of threads it supports, and registers per Streaming Multiprocessor (SM). Such knowledge facilitates tailored optimizations. Moreover, avoiding register spilling is vital; spilling forces registers to offload data to global memory, significantly hindering performance. Thus, maintaining an optimal register usage that prevents overflow is crucial for maximizing GPU efficiency.

8 Limitations and Future Work

While our current implementation has shown promising results, there are several avenues for further optimization and exploration. 1) Parameter Tuning: We are currently using a fixed configuration of 32 threads per block. Fine-tuning this parameter could potentially lead to additional performance improvements. 2) Kernel Generation: Implementing a code generator kernel that generates CUDA code dynamically could help eliminate static computations performed by every thread, reducing redundant operations and improving efficiency. 3) Leveraging NVIDIA Libraries: Exploring the use of CUDA libraries like cuBLAS or other NVIDIA libraries

for accelerating matrix multiplications and other operations could provide further performance gains.

REFERENCES

Chen, C., Borgeaud, S., Irving, G., Lespiau, J.-B., Sifre, L., and Jumper, J. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

Dao, T. FlashAttention-2: Faster attention with better parallelism and work partitioning. 2023.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, 2022.

Leviathan, Y., Kalman, M., and Matias, Y. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pp. 19274–19286. PMLR, 2023.

Miao, X., Oliaro, G., Zhang, Z., Cheng, X., Wang, Z., Wong, R. Y. Y., Chen, Z., Arfeen, D., Abhyankar, R., and Jia, Z. Specinfer: Accelerating generative llm serving with speculative inference and token tree verification. *ArXiv*, abs/2305.09781, 2023. URL https://api.semanticscholar.org/CorpusID:258740799.