Balanced Diagonal Caching for Long-sequence LLMs

Dookyung Kang
st00ne@snu.ac.kr
Graduate School of Data Science

1 Introduction

Recent advancements in large language models (LLMs) have significantly increased the demand for long-context sequence processing. In particular, the *prefill* phase of autoregressive generation involves large intermediate tensors in attention layers, introducing substantial memory bottlenecks due to matrix multiplications and reduction operations.

Modern attention variants for long-context LLMs - such as H2O [9], Keyformer [1], RoCo [5], and SnapKV [3], intensify this challenge by requiring both row-wise reductions for softmax normalization and column-wise reductions to score token-level importance for KV caching. These dual-axis dependencies complicate parallelization, causing many tensor compiler frameworks — such as MetaSchedule [7], Ansor [10], TensorRT [6], and TorchInductor [2] — often fall back on schedules that repeatedly move large intermediate tensors between global and shared memory, thereby introducing persistent memory bottlenecks.
To address this, recent compiler research has proposed graph-level transformations to reduce memory movement. For instance, Welder [8] introduces a tile-graph abstraction for flexible scheduling, while IntelliGen [4] performs graph rewrites based on monotonic memory access patterns. Notably, FlashTensor [11] proposes a non-convex kernel mapping strategy that splits memory-bound attention into two disjoint but sequentially scheduled kernels. This design leverages **tensor-level recomputation** to avoid materializing large intermediates, achieving significant speedups by trading memory reuse for recompute. Compared to previous fusion-based methods, this tradeoff represents a paradigm shift in how tensor programs are structured for memory-bound workloads. However, this raises a key question: what if we could find optimal performance by recomputing some parts while reusing others?

In this work, we hypothesize that further performance gains are achievable by rebalancing compute and memory workloads through hybrid recompute-reuse strategy to design efficient schedules for LLM prefill computation characteristics. Implemented codebase can be accessed via github: https://github.com/stonerdk/streamforge

2 Background and Motivation

As shown in Figure 1, TorchInductor dispatches H_2O core module into four fused kernels - separately handling the GEMM, elementwise, softmax and H_2O score computations - while FlashTensor uses only two kernels: **Kernel 0** parallelizes across rows to compute the softmax denominator and the attention outputs, while **Kernel 1** parallelizes across columns to compute the H_2O scores using the denominators produced in Kernel 0. Counterintuitively, despite this extra recomputation, FlashTensor achieves a 2.3× speedup over TorchInductor. This clearly demonstrates that the global memory communication overhead of large intermediate tensors between Kernel 2 and Kernel 3 generated by TorchInductor is a critical performance bottleneck. Moreover, even though FlashTensor runs faster, it exhibits lower warp occupancy and lower DRAM bandwidth as shown in Figure 1(c) and (d). By eliminating intensive memory traffic, FlashTensor shifts the workload into a compute-bound regime. This observation is particularly meaningful for workloads where DRAM bandwidth is the primary bottleneck rather than compute throughput, such as in autoregressive generation during LLM inference.

Prior contributions in turn indicate that there remain further opportunities for optimization on both the compute and memory fronts. Specifically, this opens up the idea of a **hybrid recompute-reuse** strategy: what if we recompute some tiles while reusing others? Such a design could both reuse DRAM bandwidth saved from eliminating full tensor writes and avoid redundant computation in Kernel 1 for reused tiles—thereby improving overall latency. To realize this, we propose an approach where the recomputed and reused tiles are partitioned in a sliding window fashion. Specifically, we can design an algorithm that balances the number of tiles each thread stores in Kernel 0 and reuses in Kernel 1. To mitigate the increased memory pressure caused by this hybrid strategy, we consider leveraging asynchronous memory transfer functionalities in modern GPUs, such as `cp.async` and TMA (Tensor Memory Accelerator).

Consequently, we explore alternative scheduling and memory optimization strategies to accelerate long-context prefill attention workloads by maximizing overlap between memory transfer and computation. Specifically, we aim to design and evaluate a set of memory-efficient kernel schedules hybrid recompute-reuse strategy, generalizing the idea of balancing and overlapping and exploring how it can be extended to existing autotuning-supported deep learning frameworks.

3 Implementation

Looking back at the mechanism by which FlashTensor opti-

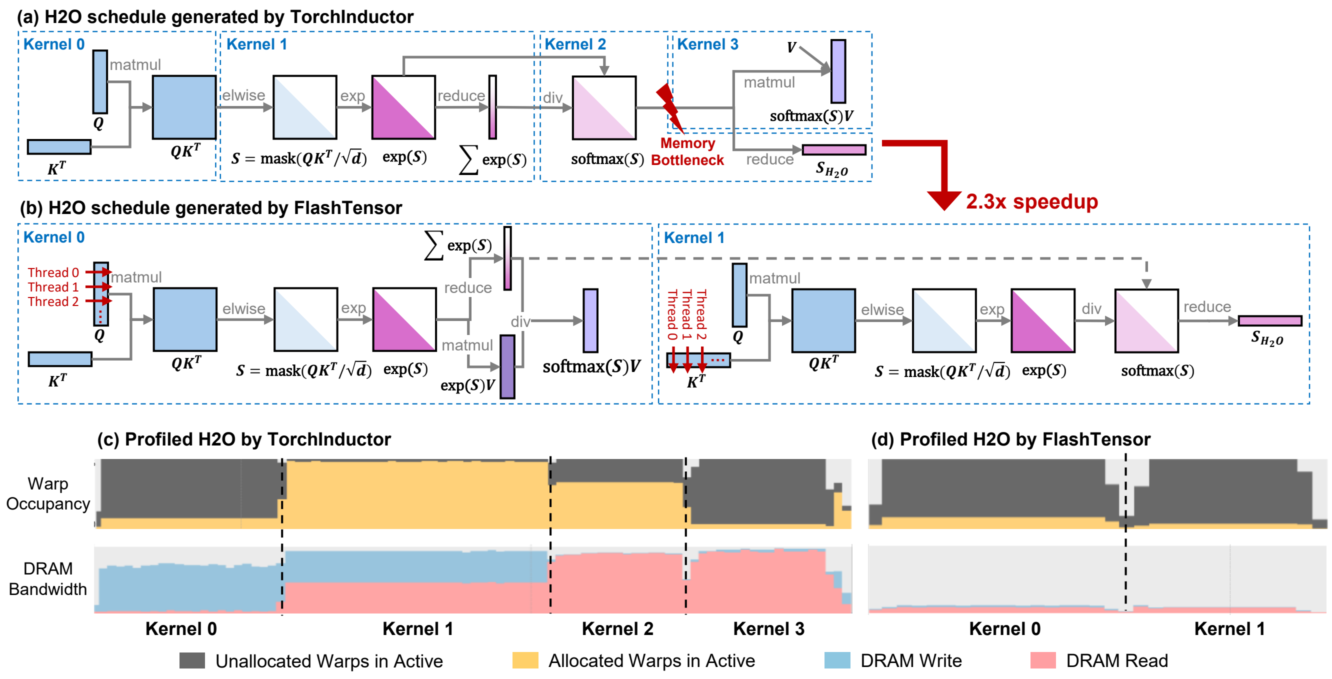
(a) H2O schedule generated by TorchInductor
Kernel 0
Kernel 1
Kernel 2
Kernel 3
matmul
elwise
exp
reduce
div
Memory Bottleneck
2.3x speedup
(b) H2O schedule generated by FlashTensor
Thread 0
Thread 1
Thread 2
(c) Profiled H2O by TorchInductor
(d) Profiled H2O by FlashTensor
Warp Occupancy
DRAM Bandwidth
Unallocated Warps in Active
Allocated Warps in Active
DRAM Write
DRAM Read

Figure 1: Found schedules and profiled result of H_2O core module by TorchInductor and FlashTensor.

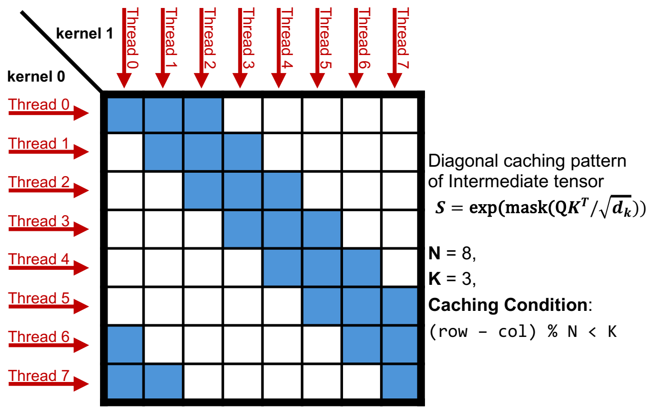

Figure 2: Balanced diagonal caching pattern.

mizes the H2O kernel in Figure 1(b), we can observe that both kernels compute the intermediate tensor $S = \exp(\text{mask}(\frac{QK^T}{\sqrt{d}}))$ but differ in their thread parallelism and loop direction. Kernel 0 has threads operating row-wise, while Kernel 1 operates column-wise in parallel. The key to caching parts of the intermediate tensor is enabling uniform caching access, regardless of whether the access pattern is row-wise or column-wise.

We propose a **balanced diagonal caching** technique to mitigate this issue. By caching the diagonal portions of the large intermediate tensor, both row-wise and column-wise parallel access patterns can uniformly store or load cached values.

Specifically, one thread block computes and loads one block (e.g. 128×128), and caching follows this granularity. For batched operations, we assume the caching pattern is identical across all batches with complete parallelism. For example, in Figure 2, when the row and column dimensions are tiled into 8 blocks and we want to cache 3 blocks, we cache those blocks (r, c) that satisfy the following condition:

$$(r - c) \mod N < K$$

With this approach, threads 6 and 7 in kernel 0 and threads 0 and 1 in kernel 1 will have their caching loops interleaved at opposite ends. Our hypothesis is that there exists an optimal K that yields the best performance, and the implementation overhead can be offset by various techniques such as asynchronous memory execution.

Now, our implementation aims to modify the two Triton-based H_2O kernels generated by FlashTensor to apply balanced diagonal caching. *Kernel 0'* focuses on storing SV and softmax denominator while simultaneously caching partial values of the intermediate tensor, and *Kernel 1'* aims to reuse these cached values to compute the H_2O score.

3.1 Diagonal Caching in Forward Kernel

The key is to cache parts of the intermediate tensor S for reuse during computation in p1. Instead of allocating memory space for the entire large tensor in global memory, the cache uses only as much space as needed in a compact manner. As shown in Algorithm 1, while iterating through the loop to compute each block, we attempt to cache it if its index satisfies the caching condition.

Algorithm 1 Kernel 0'

```
1:  Load block $\mathbf{Q}_{b,h,q} \in \mathbb{R}^{128\times128}$
2:  $\mathbf{Q} \leftarrow \mathbf{Q} \cdot c_{\text{scale}}$
3:  Initialize accumulators $\mathbf{acc}_{\text{SV}}, \mathbf{acc}_{\text{Z}}$
4:  for $k \leftarrow 0$ to $q + h$ step 128 do
5:      $\mathbf{K}, \mathbf{V} \leftarrow$ load block from $k$
6:      $\mathbf{S} \leftarrow \mathbf{Q} \cdot \mathbf{K}^T$
7:      $\mathbf{S} \leftarrow$ apply triangular mask to $\mathbf{S}$
8:      $\mathbf{P} \leftarrow \exp_2(\mathbf{S})$
9:      if $((\frac{k}{128} - q) \bmod \text{num\_blocks}) < k$ then
10:         Compute cache offset based on $(q, h, k)$
11:         diagonal_cache[...] $\leftarrow \mathbf{P}$
12:     end if
13:     $\mathbf{acc}_Z \mathrel{+}= \sum \mathbf{P}$
14:     $\mathbf{acc}_{SV} \mathrel{+}= \mathbf{P} \cdot \mathbf{V}$
15: end for
16: Write $\mathbf{acc}_Z, \mathbf{acc}_{SV}$ to memory
```

3.2 Diagonal Reusing for Score Calculation

For kernel 1, we implemented a strategy where blocks in the caching region are cached, while others are recalculated. The implementation pseudocode is shown in Algorithm 2.

Algorithm 2 Kernel 1'

```
1:  Load block $\mathbf{K}_{b,h,k} \in \mathbb{R}^{128\times128}$
2:  $\mathbf{K} \leftarrow \mathbf{K} \cdot c_{\text{scale}}$
3:  Initialize accumulator $\mathbf{acc}_{\text{score}} \in \mathbb{R}^{128}$
4:  for $q \leftarrow h \times 128$ to 4096 step 128 do
5:      $\mathbf{Q}_{b,h,q} \leftarrow$ load block from $q$
6:      $\mathbf{L}_{b,h,q} \leftarrow$ load normalization block from $q$
7:      if $((k - \frac{q}{128}) \bmod \text{num\_blocks}) < k$ then
8:          Compute cache offset based on $(q, h, k)$
9:          $\mathbf{P} \leftarrow$ diagonal_cache[...]
10:     else
11:         $\mathbf{S} \leftarrow \mathbf{Q} \cdot \mathbf{K}^T$
12:         $\mathbf{S} \leftarrow$ apply triangular mask to $\mathbf{S}$
13:         $\mathbf{P} \leftarrow \exp_2(\mathbf{S})$
14:     end if
15:     $\mathbf{W} \leftarrow \mathbf{P} \oslash \mathbf{L}$                ▷ element-wise division
16:     $\mathbf{acc}_{\text{score}} \mathrel{+}= \sum \mathbf{W}$ over rows
17: end for
18: Write $\mathbf{acc}_{\text{score}}$ to memory
```

4 Evaluation

The experiments were conducted on an AMD EPYC 7702 64 core x2 CPU environment with an A6000 GPU. We target the core kernels described in Figure 1 for H_2O operations with attributes of head = 32, sequence length = 4096, and embedding dimension = 128. We use torch, torchInductor, tensorrt, and flashtensor as baselines. Due to errors in the MLIR lowering pass while reproducing FlashTensor compilation and autotuning, we instead reused the triton kernels from the A100-target logs in the artifact codebase.

One significant issue is that FlashTensor effectively addresses the redundancy of masked upper triangular values in the intermediate tensor by linearly determining the loop extent based on the parallel unit number in the kernel, which has a significant positive impact on performance. However, we have left masking-aware diagonal caching as future work and deliberately disabled this feature in our baseline FlashTensor, modifying it to iterate through the entire loop, resulting in a slightly downgraded performance version.

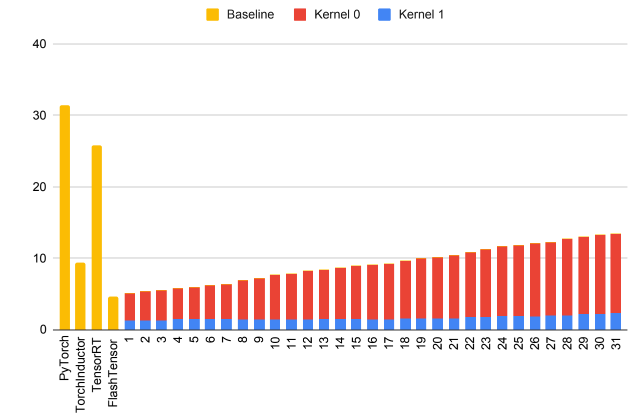

Figure 3: Evaluation result of H2O kernel.

As shown in Figure 3, our implementation demonstrates that as k increases (the number of cached blocks per row and column), the overall latency monotonically increases, indicating that k = 1 is optimal. While this performance is decreased compared to the original FlashTensor results, it still shows speedups of 6.16×, 1.83×, and 5.05× compared to PyTorch, TorchInductor, and FlashTensor respectively. Kernel 0 consumes more time than Kernel 1, which can be attributed to the fact that Kernel 1 does not perform calculations for cached blocks, creating a trade-off relationship, whereas Kernel 0 both stores and separately calculates for cached blocks, and performs more calculations than Kernel 1. The experimental results indicate that our proposed strategy for finding a balance between data reuse and recomputation was not effective in practice.

5 Discussion and Limitations

FlashTensor computes the same intermediate values repeatedly in a serial manner without materializing them. We hypothesized that reusing some of these values would improve performance and proceeded with implementation. However, our approach failed to enhance performance in practice. We observed that for both kernel 0 and kernel 1, latency increased monotonically as the number of cached blocks grew. We expected that various techniques could offset the overhead caused by data reuse, storage, loading, branching, and index calculations. Contrary to our expectations, these overheads had a more critical impact on performance.

Branch-Induced Recalculation Tradeoff. For Kernel 1, we reached the counterintuitive conclusion that recalculation was less expensive than reuse. This was primarily due to the kernel implementation with branches inside inefficient loops rather than memory movement costs. This structure prevented Triton from performing aggressive loop optimizations, resulting in greater performance degradation and overhead. We attempted to resolve this by partitioning loops into reuse loops and recalculation loops to eliminate branches within loops and encourage compiler optimization, but we were unsuccessful and could not confirm the possibility of substantial performance improvements.

Limitations of Triton's Asynchronous Memory Support. Regarding memory storage/loading costs, we thought we could address them using asynchronous memory movement techniques (`cp.async`) provided by Ampere to overlap computation and memory movement. However, the Triton kernel implementation did not expose this functionality. Instead, it was implemented to automatically tune through internal paths by adjusting decorator parameters like `num_stage`. From an engineering perspective, it was challenging to manually manipulate this and verify proper operation by modifying only the Triton kernel code.

Diagonal Caching with Mask Awareness. Additionally, the actual FlashTensor is optimized to avoid iterating through the masked upper triangular matrix portion in kernel 0, making our implementation perform worse compared to the real FlashTensor. Making diagonal caching mask-aware requires a different strategy since caching exactly the same amount in row-wise and column-wise approaches becomes meaningless. Naively, we believe this presents a non-trivial problem with an exponentially large search space depending on which diagonals are cached.

Triton's Abstraction Gap and Portability Issues. Fundamentally, we attempted optimization based on the Triton code from FlashTensor's artifact logs. While Triton is a powerful intermediate representation, it was unsuitable both for easily handling schedules at a high level and for directly calling target-aware low-level functions. This created methodological issues. Scaling to various shapes and algorithms beyond H2O proved difficult, suggesting that leveraging higher-level compiler stacks from the beginning might have been beneficial. For instance, using TVM with Metaschedule for autotuning while maintaining FlashTensor's k0, k1 semantics could have enabled more scalable development.

6 Conclusion

While our balanced diagonal caching implementation did not deliver the expected performance gains, it revealed key challenges in optimizing tensor workloads for LLMs—particularly in managing branching overhead and limited control over asynchronous memory in Triton. These issues underscore the importance of compiler infrastructure in enabling flexible optimization.

Despite the constraints, the idea of diagonal caching remains promising, especially for architectures where memory access patterns are more favorable. Future work may benefit from frameworks like TVM for more explicit scheduling and from exploring mask-aware caching tailored to attention sparsity.

References

[1] Muhammad Adnan, Akhil Arunkumar, Gaurav Jain, Prashant J Nair, Ilya Soloveychik, and Purushotham Kamath. 2024. Keyformer: Kv cache reduction through key tokens selection for efficient generative inference. *Proceedings of Machine Learning and Systems* 6 (2024), 114–127.

[2] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. 2024. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 929–947.

[3] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. 2024. Snapkv: Llm knows what you are looking for before generation. *Advances in Neural Information Processing Systems* 37 (2024), 22947–22970.

[4] Zixuan Ma, Haojie Wang, Jingze Xing, Shuhong Huang, Liyan Zheng, Chen Zhang, Huanqi Cao, Kezhao Huang, Mingshu Zhai, Shizhi Tang, et al. 2025. IntelliGen: Instruction-Level Auto-tuning for Tensor Program with Monotonic Memory Optimization. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. 107–122.

[5] Zhao Mandi, Shreeya Jain, and Shuran Song. 2024. Roco: Dialectic multi-robot collaboration with large language models. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 286–299.

[6] NVIDIA. 2017. https://developer.nvidia.com/tensorrt

[7] Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. 2022. Tensor program optimization with probabilistic programs. *Advances in Neural Information Processing Systems* 35 (2022), 35783–35796.

[8] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. 2023. Welder: Scheduling deep learning memory access via tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 701–718.

[9] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. 2023. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems* 36 (2023), 34661–34710.

[10] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 863–879.

[11] Runxin Zhong, Yuyang Jin, Chen Zhang, Kinman Lei, Shuangyu Li, and Jidong Zhai. 2025. FlashTensor: Optimizing Tensor Programs by Leveraging Fine-grained Tensor Property. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 183–196.