[Re] FlashAttention Three Years On

Kia Ghods[*] **Jishnu Roychoudhury**[*] **Addison Wu**[*]
{kia.ghods, jishnu.roy, addisonwu}@princeton.edu
GitHub Repository: https://github.com/kiaghods/FlashAttentionTriton_NLP

Abstract

In this paper, we reproduce FLASHATTENTION (Dao et al., 2022), an IO-aware method for calculating exact attention that addresses limitations in the memory usage and bandwidth inefficiencies of standard attention mechanisms. We recreate the memory-optimized attention kernels in Triton (Tillet et al., 2019) and evaluate the runtime and memory usage against several baselines, including the standard PyTorch scaled dot-product attention (SDPA), a naive baseline implemented using explicit matrix multiplications and softmax operations in PyTorch, and a CUDA implementation implemented by Dao et al. (2022). Furthermore, we conduct a realistic next-token-prediction benchmarking evaluation of our kernel implementation on WikiText-103 (Merity et al., 2017) and introduce an implementation of our Triton FLASHATTENTION kernel in the form of Multi-Query Attention (Shazeer, 2019).

1 Introduction

1.1 Background and Related Work

Transformers have revolutionized the way that models process sequential data by replacing recurrent layers in RNNs with self-attention, enabling parallel computation and improved contextual representation across entire input sequences enabling breakthrough improvements in applications in fields like text and audio processing (Vaswani et al., 2017). However, the default self-attention kernel used in transformers faces a significant limitation in its scalability to longer context lengths due to quadratic asymptotic growth in terms of both computation and memory usage.

Many approximate attention methods have been proposed to reduce the compute and memory requirements of attention. These methods range from low-rank approximation (Choromanski et al., 2021), to sparse attention (Beltagy et al., 2020), learned compression (DeepSeek-AI et al., 2024), chunking (Dai et al., 2019), and combinations (Chen et al., 2021). However, in part due to the increased implementation complexity, these methods have not gained widespread adoption, and the majority of large training runs still use exact attention. Moreover, these methods often produce subtle inaccuracies that degrade model quality (Vyas et al., 2020), and while they often improve the computation complexity to linear or near-linear in sequence length, many of them do not provide a wall-clock speedup over standard attention in practice (Daras et al., 2020).

One significant reason for the lack of meaningful wall-clock runtime reduction is because approximate methods focus on reducing the compute operations required while not adequately addressing overheads from I/O operations. However, compute operations have outpaced memory operations on modern GPUs, and as such, most transformer architectures are memory-bottlenecked (Dong et al., 2025). In particular, repeated reads from relatively slow high-bandwidth memory (HBM) and writes to fast on-chip shared random access memory (SRAM) incur substantial latency and bandwidth costs, leading to underutilization of compute units and limiting overall speedup, even when the theoretical compute complexity is reduced (Leroux et al., 2024). As recent models scale to handle context lengths of tens or even hundreds of thousands of tokens, the inefficiencies of both standard and approximate attention become increasingly prohibitive, both in training and inference settings (Yazdanbakhsh et al., 2022; Duman Keles et al., 2023).

In response, Dao et al. (2022) propose FLASHATTENTION, an exact attention algorithm which calculates attention with a constant factor more computations than a naive algorithm, but far fewer (subquadratic) memory accesses. The primary goal of FLASHATTENTION is to eliminate the quadratic memory reads and writes between high-bandwidth memory (HBM) and on-chip SRAM that are required by naive implementations of self-attention. While the standard approach materializes the entire attention matrix and stores intermediate values (such as softmax-normalized scores) in HBM for reuse during the backward pass, FLASHATTENTION avoids this by recomputing key intermediate quantities in a memory-efficient way. We outline the algorithm in **Section 2**.

FLASHATTENTION has gained widespread traction in both research and production environments due to its ability to significantly reduce memory usage while preserving the exactness of standard attention. Its efficiency has enabled faster training and inference, particularly for models operating on long sequences, where traditional attention mechanisms become prohibitively expensive. Building on the original algorithm, FlashAttention-2 introduced further improvements, such as support for variable-length sequences, dropout, and enhanced numerical stability (Dao, 2024). These enhancements have led to FlashAttention-2's integration into the PyTorch core library as the backend for `torch.nn.functional.scaled_dot_product_attention`, making it a default choice for efficient attention in modern transformer architectures (PyTorch Foundation, 2024). Given its growing impact and widespread deployment, a careful replication of FLASHATTENTION is important for validating its claims, understanding its performance characteristics, and assessing its practical implications under different hardware and workload conditions.

1.2 Our Contributions

Following **Algorithm 1** in Dao et al. (2022), we implement the full FLASHATTENTION kernel (forward and backward passes) in the Triton programming language (Tillet et al., 2019), which provides a high-level interface for writing GPU kernels. Our contributions are:

- **Triton-based implementation.** A complete and modifiable Triton reimplementation of the original FLASHATTENTION kernel, designed to be easily integrable into PyTorch-based transformer architectures.
- **Further improvements.** We introduce additional optimizations over the original kernel, including pipelining and device-specific autotuning for improved throughput and memory efficiency.

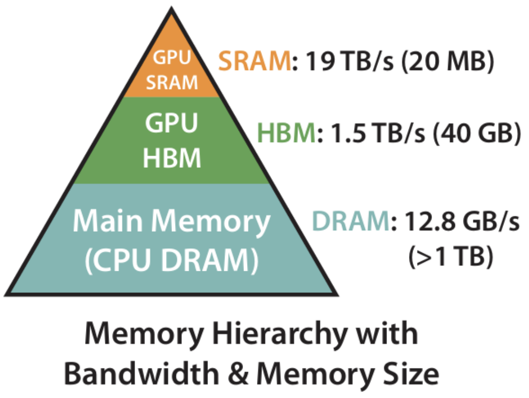


Figure 1: (Dao et al., 2022) GPU memory hierarchy illustrating the latency and bandwidth characteristics of different tiers, from on-chip SRAM to off-chip GPU HBM and CPU DRAM. The widening gap between compute throughput and memory access cost is a key bottleneck in transformer attention mechanisms. Numbers chosen are those for the A100 GPU.

- **Benchmarking on A100.** We provide comprehensive benchmarks for various FLASHATTENTION implementations and show performance for our kernel forward pass better than the original CUDA implementation and competitive to PyTorch's scaled dot product attention.
- **Multi-query attention (MQA) support.** We extend the kernel to support MQA, a variant where keys and values are shared across heads to reduce memory and compute. Unlike most off-the-shelf implementations, which do not natively support MQA, our kernel includes full support along with, to our knowledge, the first set of performance and memory benchmarks for FlashAttention-style MQA.

2 FlashAttention

2.1 Hardware Characteristics

We describe a few key characteristics of modern GPUs that motivate FLASHATTENTION.

Memory hierarchy. Modern GPUs feature a hierarchical memory architecture comprising fast, low-capacity on-chip SRAM (including registers and shared memory) and slower, high-capacity off-chip High Bandwidth Memory (HBM). In Figure 1 we list compute throughput and memory capacity for the A100. Given that compute throughput on GPUs far exceeds memory access bandwidth (Kao et al., 2023), it is crucial to maximize data reuse within SRAM and minimize costly transfers between SRAM and HBM.

Kernel execution model. In the GPU execution model, each kernel is designed to perform a single operation by loading input data from High Bandwidth Memory (HBM) into on-chip SRAM, executing computations, and writing the results back to HBM. *Kernel fusion* converts multiple operations into a single kernel, thus saving HBM reads/writes that occur between the operations.

Performance regimes. GPU workloads generally fall into two performance regimes: compute-bound and memory-bound. Compute-bound operations are limited by the availability of arithmetic units and their throughput, whereas memory-bound operations are constrained by memory bandwidth and latency. Attention mechanisms, as used in transformer architectures, are predominantly memory-bound due to their high demand for memory access relative to computation, particularly during the key-query dot product and subsequent softmax operations. FLASHATTENTION serves to rectify this issue.

2.2 Standard Attention Implementation

The self-attention mechanism in transformers computes the output as a weighted sum of values, where the weights are determined by the scaled dot-product of queries and keys (Vaswani et al., 2017). Given an input matrix in $\mathbf{X} \in \mathbb{R}^{N \times d}$, where N is the sequence length and d the hidden dimension, the inputs are projected into query, key, and value matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$. The standard attention formula for each head is

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N \times N}, \quad \mathbf{P} = \mathrm{softmax}(\mathbf{S}) \in \mathbb{R}^{N \times N}, \\ \mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N \times d}$$

where the $\mathrm{softmax}$ is applied row-wise. While the number of FLOPs is $O(N^2 d)$, the dominant performance bottleneck on modern GPUs arises from memory usage and bandwidth. In the standard implementation of attention, intermediate matrices such as the attention scores $\mathbf{S}$ and softmax outputs $\mathbf{P}$ are of size $O(N^2)$ and cannot fully reside in fast on-chip SRAM due to its limited capacity. Instead, they are computed in tiles in SRAM and repeatedly written to and read from HBM during both the forward and backward passes. This results in $O(N^2 + Nd)$ HBM memory accesses (Theorem 1), which often exceed the cost of the actual computations. As a result, the standard implementation is memory-bound, with compute units frequently stalling while waiting for data to be transferred; this overhead becomes particularly prohibitive as sequence lengths increase (Ivanov et al., 2021).

2.3 FlashAttention: Algorithm

FLASHATTENTION aims to reduce the number of expensive HBM accesses by fusing attention score computation, softmax normalization, and value aggregation into a single pass over the input data, using tiling to fit intermediate results into SRAM (Dao et al., 2022). The input matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ are assumed to reside in HBM, while computations are performed block-by-block using SRAM.

Tiling. The input sequence is partitioned into blocks of size B, and attention is computed in tiles. Rather than forming the full attention matrix $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, the algorithm processes blocks of $\mathbf{Q}_i$ against all blocks $\mathbf{K}_j$, $\mathbf{V}_j$. For each such pair, we (1) compute partial scores

$$\mathbf{S}_{ij} = \frac{\mathbf{Q}_i \mathbf{K}_j^\top}{\sqrt{d}} \in \mathbb{R}^{B \times B},$$

and (2) maintain row-wise softmax statistics for numerical stability:

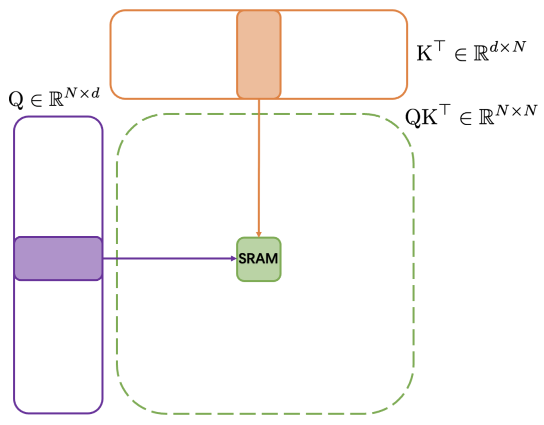
$$m(\mathbf{x}) := \max_i x_i, \quad \ell(\mathbf{x}) := \sum_i e^{x_i - m(\mathbf{x})}, \\ \mathrm{softmax}(\mathbf{x}) = \frac{e^{x_i - m(\mathbf{x})}}{\ell(\mathbf{x})}$$

As softmax couples all keys (columns of $\mathbf{K}$), the algorithm incrementally computes the output by maintaining: (1) the running max m_i for each row of scores, (2) the scaled softmax sum ℓ_i across blocks, and (3) the accumulated output $\mathbf{O}_i$ (Milakov and Gimelshein, 2018; Rabe and Staats, 2022).

When processing multiple score blocks sequentially, say for inputs $\mathbf{x}^{(1)}, \mathbf{x}^{(2)} \in \mathbb{R}^B$, we concatenate $\mathbf{x} = \left[\mathbf{x}^{(1)} \quad \mathbf{x}^{(2)}\right] \in \mathbb{R}^{2B}$. The softmax is then decomposed as:

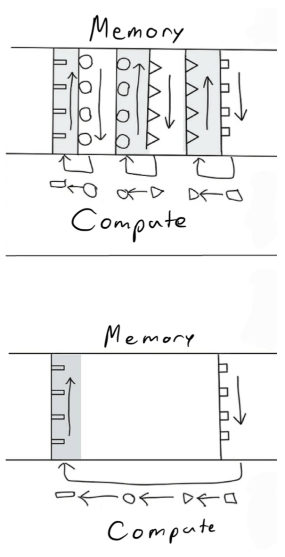
$$m(\mathbf{x}) = \max(m(\mathbf{x}^{(1)}), m(\mathbf{x}^{(2)})), \\ \ell(\mathbf{x}) = e^{m(\mathbf{x}^{(1)}) - m(\mathbf{x})} \cdot \ell(\mathbf{x}^{(1)}) \\ + e^{m(\mathbf{x}^{(2)}) - m(\mathbf{x})} \cdot \ell(\mathbf{x}^{(2)})$$

which ensures that scaling across multiple blocks yields correct results. The output block is then updated with the scaled values: $\mathbf{O}_i \leftarrow \mathbf{O}_i + e^{\mathbf{S}_{ij} - m_i}\mathbf{V}_j$ and normalized at the end as $\mathbf{O}_i \leftarrow \frac{\mathbf{O}_i}{\ell_i}$.



$Q \in \mathbb{R}^{N\times d}$
$K^\top \in \mathbb{R}^{d\times N}$
$QK^\top \in \mathbb{R}^{N\times N}$
SRAM

(a) Tiling



(b) Fusion

Figure 2: **Left:** FLASHATTENTION uses *tiling* to load blocks of $\mathbf{Q}$ and $\mathbf{K}^\top$ into SRAM, avoiding materialization of $\mathbf{QK}^\top \in \mathbb{R}^{N\times N}$ in HBM. **Right**: FLASHATTENTION fuses multiple memory-bound kernels into a single compute pass; Top: standard attention performs separate memory loads and stores between compute stages; Bottom: FLASHATTENTION fuses operations into one kernel, minimizing HBM traffic. Figure 2b is heavily inspired by He (2022).

The tiling process is illustrated in Figure 2a, where only a block of queries $\mathbf{Q}_i$ and keys $\mathbf{K}_j$ are loaded into on-chip SRAM at a time. The attention scores $\mathbf{S}_{ij}$ are computed in SRAM, and softmax statistics (m_i, ℓ_i) are incrementally updated across tiles. This avoids materializing the full attention matrix $\mathbf{QK}^\top$ in memory, reducing HBM traffic from $O(N^2 + Nd)$ to $O(N^2d^2/M)$.

Recomputation and Backward Pass. Rather than storing the full attention matrix $\mathbf{S}$ or softmax matrix $\mathbf{P}$, FLASHATTENTION caches only the row-wise softmax statistics (m_i, ℓ_i) for each query block. During the backward pass, these statistics are used to recompute $\mathbf{S}$ and $\mathbf{P}$ on-the-fly in SRAM from the original $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ blocks. This approach eliminates the need to store $O(N^2)$ intermediate values in HBM and serves as a targeted form of gradient checkpointing, specifically optimized for the structure of the attention mechanism.

Kernel fusion. A major performance benefit of FLASHATTENTION comes from kernel fusion, which fuses multiple stages of the attention computation – including the matrix multiplication, softmax computation, and weighted value aggregation – into a single GPU kernel. In contrast, standard implementations invoke separate kernels for each stage, requiring intermediate results like the attention matrix and softmax outputs to be written and read between each step. By fusing these operations, FLASHATTENTION avoids these intermediate memory transfers and instead keeps all intermediate data in on-chip SRAM throughout the computation of each tile. This dramatically reduces kernel launch overhead and HBM traffic, enabling a much higher effective throughput. This strategy is illustrated in Figure 2b, where standard attention (top) incurs repeated memory accesses between stages, while FLASHATTENTION (bottom) fuses the computation into a single pass, drastically reducing memory traffic and improving throughput. Note that without kernel fusion, the FLASHATTENTION algorithm is entirely not workable: each of the tile-by-tile computations of the attention matrix and softmax would be written back to HBM, thus maintaining $O(N^2)$ HBM memory traffic and defeating the purpose of the algorithm's IO-aware design.

Masking. Masking is done tile-wise: each tile of attention scores is multiplied by the corresponding mask slice before softmax. Most masks used in practice (e.g. causal and key padding) exhibit strong spatial contiguity, and thus a major performance boost (1.8x for causal attention, see Fig. 3) can be achieved by skipping computations of tiles which are fully masked.

IO-Complexity of FlashAttention. With the FlashAttention algorithm already laid out in full, we introduce a key theorem regarding the efficiency of FlashAttention in comparison to the standard matrix multiplication attention algorithm in regards to memory accesses.

Theorem 1. *(Dao et al., 2022) Let N be the sequence length, d be the attention head dimension, and M be the size of GPU SRAM where $d \leq M \leq Nd$. Then, we have that the standard matrix multiplication attention algorithm requires $\Theta(Nd + N^2)$ HBM accesses, whereas FlashAttention only requires $\Theta(N^2 d^2 M^{-1})$ HBM accesses.*

Proof. See Appendix **A: Algorithm Details.** □

In fact, $O(N^2 d^2 M^{-1})$ is asymptotically optimal; proof can be found in Dao et al. (2022).

2.4 Extension: Further Improvements

We implement two improvements over the existing FlashAttention architecture: fine-grained pipelining and a Triton-driven autotuner (Tillet et al., 2019). In pipelining, the scheduler partitions the attention matrix into tiles and overlaps three stages—loading Q K V blocks, computing dot-products, and writing outputs—so memory fetch for tile $t{+}1$ is issued while tile t computes, hiding latency. Concurrently, the autotuner samples a grid of kernels that vary tile dimensions, pipeline depth, warp count, and optional value-prefetch, measures runtime, and retains the fastest candidate for subsequent calls. To keep the search tractable, configurations are pruned by a rule-based filter keyed on head dimension and data type (plus tile/warp limits), and we seed the search with a few hand-picked A100/H100 configurations to avoid cold-start penalties. The chosen kernel is memoized and reused whenever later calls share the same signature, ensuring steady speed-ups on both NVIDIA A100 and H100 GPUs without hand tuning.

3 Experiments and Results

We evaluate the run a suite of experiments on a MIG-partitioned A100 GPU. Note that a MIG slice only has access to $1/7$-th of the GPU's compute and memory capability, and thus achieves substantially lower throughput compared to full-GPU benchmarks reported in prior work.

Experimental Setup. All runtime and memory experiments were conducted with a batch size of 4 and head dimension of 64, unless otherwise specified. These settings were held constant across all experiments to ensure fair comparisons. We use 16 attention heads for MHA and 16 query heads for MQA (with shared keys and values).

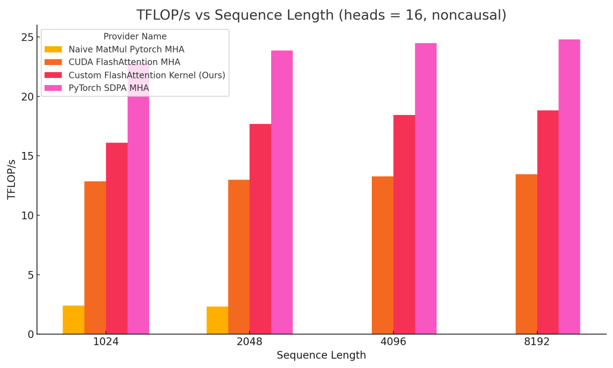
3.1 Runtime Analysis

We validate the performance of our implementation of FlashAttention for multi-head attention. Benchmarks on different settings show that our FlashAttention implementation achieves significant speedup compared to naive implementations and comparable with current optimized implementations in the PyTorch library (PyTorch Foundation, 2024). Notably, our implementation is easily modifiable to suit subtly different attention mechanisms (for instance, arbitrary attention masks – we already provide support for key padding masks), while PyTorch's implementation only supports causal masking. We present the FLOPs/s. We calculate FLOPs by the formula:

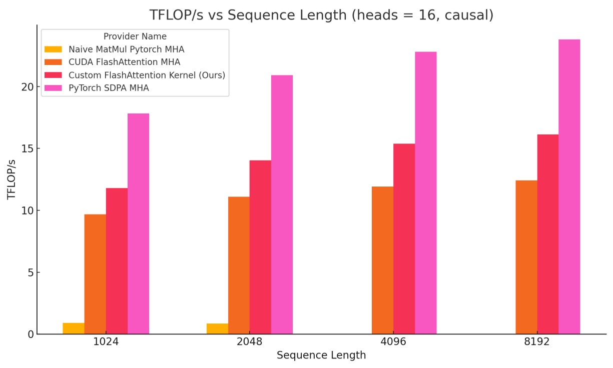
$$4 \cdot \text{seqlen}^2 \cdot \text{head dimension} \cdot \text{number of heads}$$

With a causal mask, we divide this number by 2 to account for the fact that approximately only half of the entries are calculated.

Our results, shown in Fig. 3, are consistent with previous work such as Dao (2024). In particular, the original CUDA implementation of FlashAttention has a throughput of 13 TFLOPs/s for non-causal and 10-11 TFLOPs/s for causal attention. Noting that throughput typically scales linearly with the number of streaming multiprocessors (SMs), our numbers correspond to 91 TFLOPs/s and 70-77 TFLOPs/s on a full A100 GPU, which matches the 91 TFLOPs/s and 76 TFLOPs/s reported by Dao (2024). Our implementation is competitive with existing implementations, performing better than the CUDA kernel and less than 1.5x slower than Pytorch's scaled dot product attention, which uses the much improved FlashAttention-2. Our improvement over the original CUDA kernel is likely due to the additional performance optimizations of pipelining and autotuning for the A100 GPU. Notably, our kernel is implemented in Triton, which typically does not match the performance of hand-optimized CUDA kernels; this highlights



(a) Non-causal



(b) Causal

Figure 3: Runtime (TFLOP/s) comparison of 16-head attention forward pass in non-causal and causal settings.

the effectiveness of our additional optimizations despite the higher-level abstraction.

3.2 Memory Usage Analysis

We compare the maximum memory usage of our implementation to a standard matrix multiplication implementation, the contemporary PyTorch scaled dot product implementation, and the CUDA implementation of FLASHATTENTION, shown in Figure 4. As expected, all FlashAttention-style methods consume significantly less memory than the naive baseline, thanks to their sub-quadratic space complexity. In addition, our implementation outperforms that of the original CUDA FLASHATTENTION kernel and equal to PyTorch's scaled-dot product attention, despite being implemented at a higher abstraction level. Our slight improvement over the CUDA kernel is likely due to our autotuning selecting the optimal configurations for our GPU, in particular the choice of tiling sizes.

3.3 Language Modeling

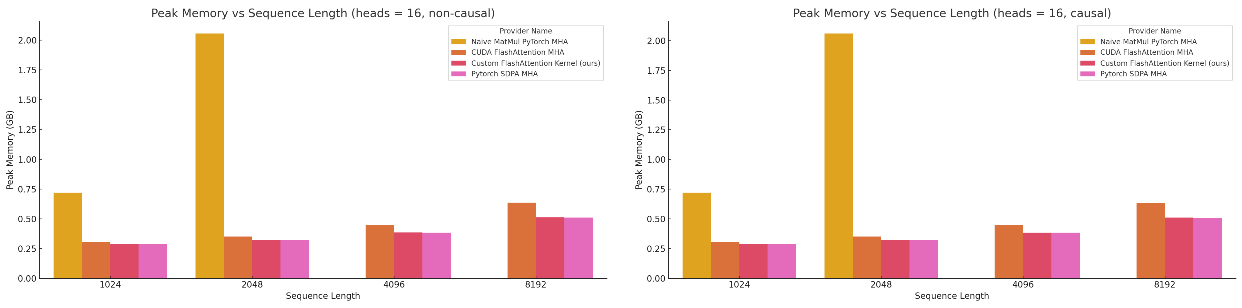
We evaluate how well our FLASHATTENTION kernel programmed in Triton (Tillet et al., 2019) performs in a realistic setting. Namely, we measure the perplexity, over the course of epochs, achieved by a multi-head transformer architecture (Vaswani et al., 2017) that uses either naive PyTorch attention with standard `matmul` operations for the forward and backward passes or our Triton kernel, as it learns the task of next-token prediction over the WikiText-103 corpus (Merity et al., 2017). The specific transformer architecture we implemented consisted of 12 layers with 8 attention heads. We tokenize the WikiText-103 dataset using OpenAI tiktoken (OpenAI, 2022) and subject to our compute requirements, we train the transformer on 10% of the WikiText-103 training corpus over 5 epochs. We present our results for the validation perplexity achieved at the end of each epoch by both transformers in Figure 5. Despite the initial divergence at epoch 0, with the default attention transformer achieving a noticeably lower perplexity than the Triton FLASHATTENTION transformer at 155.06 compared to 202.60, respectively, we have that convergence of validation perplexities is nearly achieved at the fifth epoch (epoch 4) with minimally-differing perplexities of 70.85 and 73.33, respectively.

3.4 Extension: Multi-Query Attention

We extend FLASHATTENTION to handle multi-query attention (MQA), a variant of the standard multi-head attention mechanism. In MQA, while each attention head has its own query projection, all heads share a single set of key and value projections, thus reducing the size of the KV cache during inference as displayed in Figure 8. This design significantly reduces memory usage and computational overhead.

While multi-query attention has been previously implemented with FlashAttention-2 (Dao, 2024), it is not currently integrated into PyTorch's native attention modules, and, to our knowledge, there has been no publicly available experimentation or analysis of FLASHATTENTION's performance with MQA, nor any implementation of MQA for the original FLASHATTENTION. To address these gaps, we have extended our Triton FLASHATTENTION kernel to implement multi-query attention.

To retain the memory and compute benefits of MQA in our Triton-based kernel, we avoid broadcasting the key and value tensors across all heads by manipulating the indexing logic within the at-



(a) Non-causal (b) Causal

Figure 4: Peak memory usage comparison of 16-head MHA forward pass in non-causal and causal settings

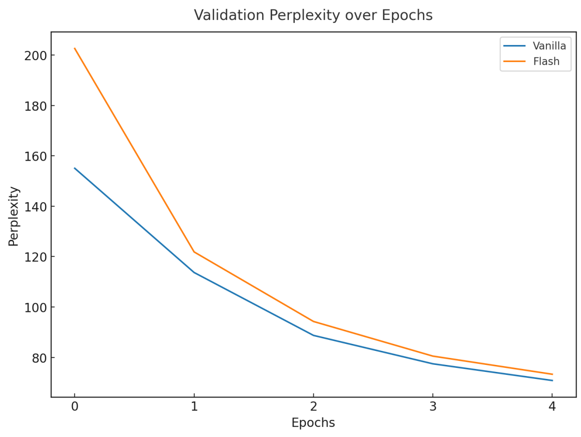


Figure 5: Validation perplexity over five epochs from a transformer trained on WikiText-103 with both `matmul` attention and Triton FLASHATTENTION kernels

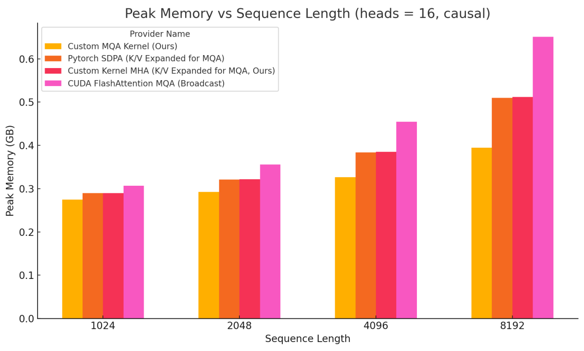


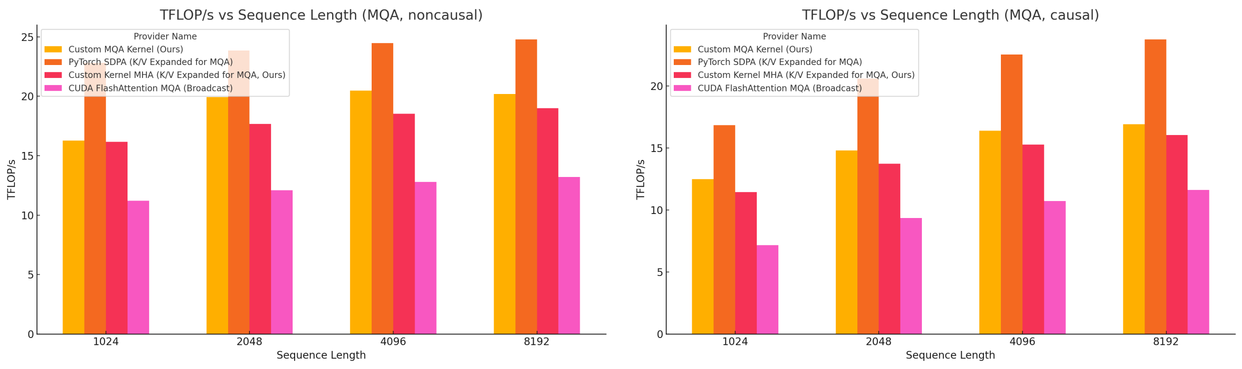
Figure 6: Peak memory usage comparison of 16-head multi-query attention in causal setting, forward pass.

tention kernel. Specifically, we configure the memory layout such that the key and value tensors are treated as shared across heads by assigning a zero stride along the head dimension, effectively reusing the same data without explicit replication.

We compare our MQA implementation against off-the-shelf implementations as well as our own MHA implementation. Most off-the-shelf implementations (such as the original FLASHATTENTION paper and PyTorch scaled dot product implementation) do not inherently support MQA and thus must be run with expanded K/V or broadcasting.

We find that our implementation is slightly better than existing implementations. In particular, as shown in Fig. 6, we use the least memory out of all implementations due to the absence of explicit key/value duplication across heads. This optimization also minimizes memory bandwidth pressure, contributing to the superior compute throughput over our MHA implementation shown in Fig. 7.

We note one surprising aspect of our result: the memory reduction associated with MQA compared to our MHA implementation is less significant than expected. Specifically, in the MHA setting, memory usage at peak includes storage for the $\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$, and output ($\mathbf{O}$) tensors, whereas in MQA, the $\mathbf{K}$ and $\mathbf{V}$ tensors are shared across heads and thus contribute negligibly to the overall footprint. Thus, we would expect memory usage to approximately halve, but we observe much more modest savings. Inspecting the graph, we observe that there is a fixed overhead across all implementations which limits the visible benefit of removing per-head $\mathbf{K}$ and $\mathbf{V}$. We hypothesize that this overhead originates from the PyTorch allocator's strategy of preemptively reserving large memory blocks to satisfy future allocations, overwhelming the actual memory allocation at small scales. This hypothesis is validated by running a memory profiler for our MQA and MHA implementations, finding that the memory reserved far exceeds memory allocated. Table 1 shows that MQA with FLASHATTENTION attains allocated memory savings of 40-45%, near the theoretical maximum.



(a) Non-causal (b) Causal

Figure 7: Runtime (TFLOP/s) comparison of 16-head MQA forward pass in non-causal and causal settings.

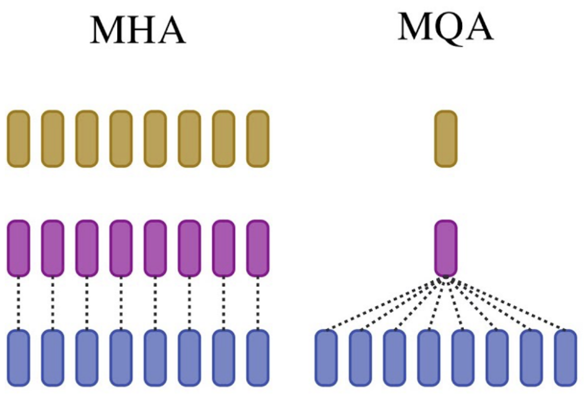


Figure 8: Comparison of Multi-Head Attention (MHA) and Multi-Query Attention (MQA). In MHA (left), each head has distinct query, key, and value projections. In MQA (right), while each head retains its own query projection, all heads share a single set of key and value projections, reducing memory and computation overhead during inference.

Seq Len	Allocated		Reserved	
	MQA	MHA	MQA	MHA
1024	0.02	0.03	0.27	0.29
2048	0.04	0.07	0.29	0.32
4096	0.08	0.13	0.30	0.37
8192	0.15	0.27	0.35	0.46

Table 1: Peak GPU memory allocated and reserved (GB) for our MQA and MHA implementations.

4 Discussion

Our reproduction and extension of the original FLASHATTENTION algorithm, three years after its introduction, demonstrates the importance of IO-aware attention mechanisms in modern transformer systems. Despite being implemented in Triton, our kernel achieves throughput and memory efficiency that are not far behind highly optimized implementations such as PyTorch's current scaled dot-product attention, which uses FlashAttention-2 as its backend. This performance is especially notable given the constraints of our MIG-partitioned A100 setup, and underscores the strength of the original algorithm's design principles.

These results reaffirm a key insight from prior work: memory bandwidth, not raw compute, is the primary bottleneck in attention. By fusing multiple memory-bound operations into a single pass and minimizing reads and writes to high-bandwidth memory (HBM), FLASHATTENTION reduces both latency and peak memory consumption. Our experiments suggest that this advantage persists on modern hardware, and our use of Triton's autotuning further shows how such kernels can be made portable and performant across diverse GPU configurations.

Our extension of the kernel to support multi-query attention (MQA) also fills a practical gap. While MQA is widely adopted in modern large language models for inference efficiency, it has not been natively supported in some popular implementations of the original FLASHATTENTION. Our adaptation preserves the core memory and compute benefits by avoiding key-value duplication across heads and highlights the flexibility of our implementation in adapting to emerging model architectures.

Empirically, we also show that training dynamics remain stable. Our transformer trained on WikiText-103 converges to a comparable validation perplexity whether it uses standard matmul-based attention or our Triton-based FLASHATTENTION kernel. This indicates that IO-aware optimizations can be adopted without sacrificing model quality or convergence behavior.

References

Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The long-document transformer.

Han Cai, Junyan Li, Muyan Hu, Chuang Gan, and Song Han. 2024. Efficientvit: Multi-scale linear attention for high-resolution dense prediction.

Beidi Chen, Tri Dao, Eric Winsor, Zhao Song, Atri Rudra, and Christopher Ré. 2021. Scatterbrain: Unifying sparse and low-rank attention. In *Advances in Neural Information Processing Systems*.

Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, David Belanger, Lucy Colwell, and Adrian Weller. 2022. Rethinking attention with performers.

Krzysztof Marcin Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Quincy Davis, Afroz Mohiuddin, Lukasz Kaiser, David Benjamin Belanger, Lucy J Colwell, and Adrian Weller. 2021. Rethinking attention with performers. In *International Conference on Learning Representations*.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. 2019. Transformer-XL: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2978–2988, Florence, Italy. Association for Computational Linguistics.

Tri Dao. 2024. Flashattention-2: Faster attention with better parallelism and work partitioning. In *The Twelfth International Conference on Learning Representations*.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Giannis Daras, Nikita Kitaev, Augustus Odena, and Alexandros G. Dimakis. 2020. SMYRF: Efficient attention using asymmetric clustering. In *Advances in Neural Information Processing Systems*, volume 33, pages 6476–6488.

DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Yang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jin Chen, Jingyang Yuan, Junjie Qiu, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruizhe Pan, Runxin Xu, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Size Zheng, T. Wang, Tian Pei, Tian Yuan, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Liu, Xin Xie, Xingkai Yu, Xinnan Song, Xinyi Zhou, Xinyu Yang, Xuan Lu, Xuecheng Su, Y. Wu, Y. K. Li, Y. X. Wei, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Zheng, Yichao Zhang, Yiliang Xiong, Yilong Zhao, Ying He, Ying Tang, Yishi Piao, Yixin Dong, Yixuan Tan, Yiyuan Liu, Yongji Wang, Yongqiang Guo, Yuchen Zhu, Yuduan Wang, Yuheng Zou, Yukun Zha, Yunxian Ma, Yuting Yan, Yuxiang You, Yuxuan Liu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhewen Hao, Zhihong Shao, Zhiniu Wen, Zhipeng Xu, Zhongyu Zhang, Zhuoshu Li, Zihan Wang, Zihui Gu, Zilin Li, and Ziwei Xie. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model.

Yanhao Dong, Yubo Miao, Weinan Li, Xiao Zheng, Chao Wang, and Feng Lyu. 2025. Accelerating llm inference throughput via asynchronous kv cache prefetching. *arXiv preprint arXiv:2504.06319*.

Feyza Duman Keles, Pruthuvi Mahesakya Wijewardena, and Chinmay Hegde. 2023. On the computational complexity of self-attention. In *Proceedings of The 34th International Conference on Algorithmic Learning Theory*, volume 201 of *Proceedings of Machine Learning Research*, pages 597–619. PMLR.

Horace He. 2022. Making deep learning go brrrr from first principles.

Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Kangdi Chen, Yuhan Dong, and Yu Wang. 2024. Flashdecoding++: Faster large language model inference on gpus.

Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. 2021. Data movement is all you need: A case study on optimizing transformers.

Sheng-Chun Kao, Suvinay Subramanian, Gaurav Agrawal, Amir Yazdanbakhsh, and Tushar Krishna. 2023. Flat: An optimized dataflow for mitigating attention bottlenecks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 295–310, New York, NY, USA. Association for Computing Machinery.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention.

Nathan Leroux, Paul-Philipp Manea, Chirag Sudarshan, Jan Finkbeiner, Sebastian Siegel, John Paul Strachan, and Emre Neftci. 2024. Analog in-memory computing attention mechanism for fast and energy-efficient large language models. *arXiv preprint arXiv:2409.19315*.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2017. Pointer sentinel mixture models. In *International Conference on Learning Representations*.

Maxim Milakov and Natalia Gimelshein. 2018. Online normalizer calculation for softmax.

OpenAI. 2022. tiktoken: A fast bpe tokenizer for use with openai's models.

PyTorch Foundation. 2024. Pytorch 2.2: Flashattention-v2 integration, aotinductor.

Markus N. Rabe and Charles Staats. 2022. Self-attention does not need $o(n^2)$ memory.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. Flashattention-3: Fast and accurate attention with asynchrony and low-precision.

Noam Shazeer. 2019. Fast transformer decoding: One write-head is all you need.

David R. So, Wojciech Mańke, Hanxiao Liu, Zihang Dai, Noam Shazeer, and Quoc V. Le. 2022. Primer: Searching for efficient transformers for language modeling.

Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, page 10–19, New York, NY, USA. Association for Computing Machinery.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008.

Apoorv Vyas, Angelos Katharopoulos, and François Fleuret. 2020. Fast transformers with clustered attention. In *Advances in Neural Information Processing Systems*, volume 33, pages 21665–21674.

Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. 2020. Linformer: Self-attention with linear complexity.

Xiaoxia Wu, Haojun Xia, Stephen Youn, Zhen Zheng, Shiyang Chen, Arash Bakhtiari, Michael Wyatt, Reza Yazdani Aminabadi, Yuxiong He, Olatunji Ruwase, Leon Song, and Zhewei Yao. 2023. Zeroquant(4+2): Redefining llms quantization with a new fp6-centric strategy for diverse generative tasks.

Yunyang Xiong, Zhanpeng Zeng, Rudrasis Chakraborty, Mingxing Tan, Glenn Fung, Yin Li, and Vikas Singh. 2021. Nyströmformer: A nyström-based algorithm for approximating self-attention.

Amir Yazdanbakhsh, Ashkan Moradifirouzabadi, Zheng Li, and Mingu Kang. 2022. Sparse attention acceleration with synergistic in-memory pruning and on-chip recomputation. In *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 744–762. IEEE.

Zhihang Yuan, Chenhao Xue, Yiqi Chen, Qiang Wu, and Guangyu Sun. 2024. Ptq4vit: Post-training quantization for vision transformers with twin uniform quantization.

5 Acknowledgments

This work was completed as a final project for *COS 484 - Natural Language Processing* at Princeton University. We thank Alex Dremov for his Triton tutorials, which helped inspire this project. We are also greatful to Professor Tri Dao for valuable discussions and clarifications related to the FLASHATTENTION implementation. This work was supported in part by computational resources provided by Princeton University's Della cluster.

A Related Work

IO-Aware Runtime Optimization. Transformer models are often bottlenecked by memory bandwidth rather than compute. FLASHATTENTION addresses this by reducing reads and writes to high-bandwidth memory through tiling and kernel fusion (Dao et al., 2022). FlashAttention-2 improves on this by better partitioning work across threads and supporting variable-length sequences (Dao, 2024), while FlashAttention-3 introduces asynchronous execution and FP8 support for further speedups (Shah et al., 2024). Other IO-aware efforts include vLLM, which introduces PagedAttention to treat key-value memory as a paged structure, minimizing fragmentation and improving KV cache management during inference (Kwon et al., 2023). Additionally, FlashDecoding++ proposes GPU-specific optimizations for autoregressive decoding, such as asynchronous softmax and pipelined memory access, to reduce latency and boost throughput (Hong et al., 2024).

Efficient ML Models via Structured and Low-Rank Matrices. Reducing the complexity of core matrix operations has become a key area of optimization for large models. Performer replaces softmax attention with kernelized linear attention, lowering complexity to linear in sequence length (Choromanski et al., 2022). Linformer and Nyströmformer utilize low-rank projections or approximations of the attention matrix to compress representation power while preserving expressiveness (Wang et al., 2020; Xiong et al., 2021). Scatterbrain combines sparse and low-rank structures for better tradeoffs in speed and accuracy (Chen et al., 2021).

Efficient Transformers. Transformer models have become the backbone of many state-of-the-art applications but are often resource-intensive. Efforts to optimize these models include architectural modifications and training strategies. For instance, the Primer architecture introduces squared ReLU activations and depthwise convolutions, resulting in reduced training costs without compromising performance (So et al., 2022). EfficientViT proposes a memory-efficient vision transformer with cascaded group attention, enhancing throughput while maintaining accuracy (Cai et al., 2024).

Quantization and Sparse Training. To deploy machine learning models on resource-constrained devices, quantization and sparsity are employed to reduce model size and computation. PTQ4ViT offers a method to quantize models after training while considering sparsity patterns to maintain accuracy (Yuan et al., 2024). ZeroQuant(4+2) introduces a novel FP6-centric post-training quantization strategy that bridges the accuracy gap between INT4 and FP16, enabling efficient deployment of large generative models without fine-tuning (Wu et al., 2023).

B Algorithm Details

Forward-Pass Pseudocode

Algorithm 1 FLASHATTENTION (Forward Pass)

```
Require: $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N\times d}$ in HBM; on-chip SRAM of size $M$
1:  $B_c \leftarrow \lceil \frac{M}{4d} \rceil$,   $B_r \leftarrow \min\left(\lceil \frac{M}{4d} \rceil, d\right)$
2:  Initialize $\mathbf{O} \leftarrow \mathbf{0}_{N\times d}$, $\ell \leftarrow \mathbf{0}_N$, $m \leftarrow (-\infty)_N$
3:  Partition:
4:      $\mathbf{Q} \rightarrow T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \ldots, \mathbf{Q}_{T_r}$
5:      $\mathbf{K}, \mathbf{V} \rightarrow T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_j, \mathbf{V}_j$
6:      $\mathbf{O}, \ell, m \rightarrow T_r$ blocks $\mathbf{O}_i, \ell_i, m_i$
7:  for $j = 1$ to $T_c$ do
8:      Load $\mathbf{K}_j, \mathbf{V}_j$ to SRAM
9:      for $i = 1$ to $T_r$ do
10:         Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ to SRAM
11:         $\mathbf{S}_{ij} \leftarrow \mathbf{Q}_i \mathbf{K}_j^\top$
12:         $\tilde{m}_{ij} \leftarrow \text{rowmax}(\mathbf{S}_{ij})$
13:         $\tilde{\mathbf{P}}_{ij} \leftarrow \exp(\mathbf{S}_{ij} - \tilde{m}_{ij})$
14:         $\tilde{\ell}_{ij} \leftarrow \text{rowsum}(\tilde{\mathbf{P}}_{ij})$
15:         $m_i^{\text{new}} \leftarrow \max(m_i, \tilde{m}_{ij})$
16:         $\ell_i^{\text{new}} \leftarrow e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij}$
17:         $\mathbf{O}_i \leftarrow e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j$
18:         $\ell_i \leftarrow \ell_i^{\text{new}}$,   $m_i \leftarrow m_i^{\text{new}}$
19:         Write $\mathbf{O}_i, \ell_i, m_i$ to HBM
20:     end for
21: end for
22: return $\mathbf{O}$
```

In this algorithm, $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N\times d}$ are the query, key, and value matrices, respectively, stored in high-bandwidth memory (HBM). The output matrix $\mathbf{O} \in \mathbb{R}^{N\times d}$ holds the result of the attention computation and is initialized to zero. The vector $\ell \in \mathbb{R}^N$ stores cumulative normalization factors for each row, while $m \in \mathbb{R}^N$ tracks the running row-wise maximum of the attention logits for numerical stability. The block sizes B_c and B_r determine how many rows or columns are processed in each iteration, constrained by the available on-chip SRAM of size M. For efficient streaming, matrices are partitioned into T_r row blocks (indexed by i) and T_c column blocks (indexed by j), allowing the algorithm to load and compute on small chunks of data sequentially. During each step, $\tilde{\mathbf{P}}ij$ contains unnormalized attention weights for block (i, j), while $\tilde{m}_{ij}$ and $\tilde{\ell}_{ij}$ contain the row-wise maximum and the sum of the exponentiated logits, respectively. These are used to compute new normalization constants $m_i^{\text{new}}, \ell_i^{\text{new}}$ and to update the output block $\mathbf{O}_i$ accordingly.

Proof of IO-Complexity of FlashAttention

Proof. We model the IO complexity in terms of memory transfers between fast on-chip memory (SRAM of size M) and slow off-chip memory (HBM). The objective is to count the number of words transferred from HBM to SRAM during the computation of attention.

In the standard attention mechanism, we compute

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\mathbf{Q}\mathbf{K}^\top)\mathbf{V},$$

where $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N\times d}$. The computation involves forming $\mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N\times N}$, which requires reading all of $\mathbf{Q}$ and $\mathbf{K}$, each of size Nd, followed by computing softmax over each row and multiplying the result with $\mathbf{V}$. Assuming that $Nd > M$, the SRAM cannot hold the full $\mathbf{Q}$, $\mathbf{K}$, or $\mathbf{V}$, nor the intermediate $\mathbf{Q}\mathbf{K}^\top$ matrix, which has size N^2. As a result, data must be frequently loaded from HBM. In an idealized streaming model, each of $\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$, and $\mathbf{Q}\mathbf{K}^\top$ contributes to the IO cost. Hence, the total IO complexity is $\Theta(Nd+N^2)$ HBM accesses.

In contrast, FLASHATTENTION avoids explicitly materializing $\mathbf{Q}\mathbf{K}^\top$ by computing attention using a tiling strategy that fits intermediate computations into SRAM. It partitions $\mathbf{Q}$ and $\mathbf{K}$ into blocks of size $B\times d$, such that $Bd \le M$, and processes attention blockwise. For each block of $\mathbf{Q}$, the algorithm iterates over all blocks of $\mathbf{K}$ and computes partial results, which are normalized and accumulated on the fly. Each block of $\mathbf{Q}$ and $\mathbf{K}$ is loaded from HBM only once per pairwise interaction, leading to an IO cost proportional to the number of block pairs.

The total number of block pairs is $(N/B)^2$, and each pair involves reading $O(Bd)$ data. Thus, the total IO cost is

$$\Theta\left(\left(\frac{N}{B}\right)^2 \cdot Bd\right) = \Theta\left(\frac{N^2 d}{B}\right).$$

Since B is chosen to be the largest value such that $Bd \le M$, we have $B = \Theta(M/d)$, giving an overall IO cost of

$$\Theta\left(\frac{N^2 d}{M/d}\right) = \Theta\left(\frac{N^2 d^2}{M}\right).$$

Therefore, FLASHATTENTION requires $\Theta(N^2 d^2 M^{-1})$ HBM accesses, completing the proof. ■

C Backward-Pass

Algorithm 2 FLASHATTENTION Backward Pass

```
Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{O}, \mathbf{dO} \in \mathbb{R}^{N\times d}$ in HBM; vectors $\ell, m \in \mathbb{R}^N$ in HBM; SRAM of size $M$; softmax scaling $\tau \in \mathbb{R}$; masking function $\mathrm{mask}$; dropout probability $p_{\text{drop}}$; RNG state $\mathcal{R}$ from forward pass
1: Set RNG state to $\mathcal{R}$
2: $B_c \leftarrow \lceil \frac{M}{4d} \rceil$,   $B_r \leftarrow \min\left(\lceil \frac{M}{4d} \rceil, d\right)$
3: Partition:
4:     $\mathbf{Q} \to T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_i$,   $\mathbf{K}, \mathbf{V} \to T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_j, \mathbf{V}_j$
5:     $\mathbf{O}, \mathbf{dO} \to T_r$ blocks $\mathbf{O}_i, \mathbf{dO}_i$, and $\ell, m \to \ell_i, m_i$
6: Initialize $\mathbf{dQ} \leftarrow \mathbf{0}_{N\times d}$, partitioned into blocks $\mathbf{dQ}_i$
7: Initialize $\mathbf{dK}, \mathbf{dV} \leftarrow \mathbf{0}_{N\times d}$, partitioned into blocks $\mathbf{dK}_j, \mathbf{dV}_j$
8: for $j = 1$ to $T_c$ do
9:     Load $\mathbf{K}_j, \mathbf{V}_j$ to SRAM
10:    Initialize temporary accumulators: $\mathbf{dK}'_j, \mathbf{dV}'_j \leftarrow \mathbf{0}_{B_c\times d}$
11:    for $i = 1$ to $T_r$ do
12:        Load $\mathbf{Q}_i, \mathbf{O}_i, \mathbf{dO}_i, \ell_i, m_i$ to SRAM
13:        $\mathbf{S}_{ij} \leftarrow \tau \cdot \mathbf{Q}_i \mathbf{K}_j^\top$
14:        $\mathbf{S}_{ij}^{\text{masked}} \leftarrow \mathrm{mask}(\mathbf{S}_{ij})$
15:        $\mathbf{P}_{ij} \leftarrow \mathrm{diag}(\ell_i)^{-1} \cdot \exp(\mathbf{S}_{ij}^{\text{masked}} - m_i)$
16:        Sample dropout mask $\mathbf{Z}_{ij} \in \mathbb{R}^{B_r\times B_c}$ with values $\frac{1}{1-p_{\text{drop}}}$ w.p. $1 - p_{\text{drop}}$, else 0
17:        $\mathbf{P}_{ij}^{\text{dropped}} \leftarrow \mathbf{P}_{ij} \circ \mathbf{Z}_{ij}$   ▷ Hadamard product
18:        $\mathbf{dV}'_j \leftarrow \mathbf{dV}'_j + (\mathbf{P}_{ij}^{\text{dropped}})^\top \mathbf{dO}_i$
19:        $\mathbf{dO}'_i \leftarrow \mathbf{dO}_i \mathbf{V}_j^\top$
20:        $\mathbf{D}_i \leftarrow \mathrm{rowsum}(\mathbf{dO}_i \circ \mathbf{O}_i)$
21:        $\tilde{\mathbf{S}}_{ij} \leftarrow \mathbf{P}_{ij} \circ (\mathbf{dO}'_i - \mathbf{D}_i)$
22:        Write $\mathbf{dQ}_i \leftarrow \mathbf{dQ}_i + \tau \cdot \tilde{\mathbf{S}}_{ij} \mathbf{K}_j$ to HBM
23:        $\mathbf{dK}'_j \leftarrow \mathbf{dK}'_j + \tau \cdot \tilde{\mathbf{S}}_{ij}^\top \mathbf{Q}_i$
24:    end for
25:    Write $\mathbf{dK}_j \leftarrow \mathbf{dK}'_j$,   $\mathbf{dV}_j \leftarrow \mathbf{dV}'_j$ to HBM
26: end for
27: return $\mathbf{dQ}, \mathbf{dK}, \mathbf{dV}$
```

Variable Descriptions. $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ are the query, key, and value matrices from the forward pass, while $\mathbf{O}$ is the output matrix and $\mathbf{dO}$ is its upstream gradient. Vectors ℓ and m are the normalization constants computed in the forward pass to ensure numerical stability. τ is the softmax scaling factor (generally $d^{-1/2}$). $\mathrm{mask}(\cdot)$ applies an attention mask (e.g., causal or key-padding). p_{drop} is the dropout rate, and $\mathcal{R}$ is the saved random number generator state from the forward pass to ensure consistent dropout behavior. Matrices are partitioned into row blocks (B_r) and column blocks (B_c) to fit in SRAM. Intermediate variables such as $\mathbf{P}_{ij}$ represent attention weights, $\mathbf{Z}_{ij}$ is the sampled dropout mask, and $\mathbf{dV}'_j, \mathbf{dK}'_j$ are partial gradients that accumulate over row blocks i.

Theorem 2 (IO Complexity of FLASHATTENTION Backward Pass)**.** *Let N be the sequence length, d the attention head dimension, and M the size of on-chip SRAM, where $d \leq M \leq Nd$. Then the standard matrix multiplication-based attention backward pass requires*

$$\Theta(Nd + N^2)$$

HBM (high-bandwidth memory) accesses, whereas the FLASHATTENTION *backward pass requires only*

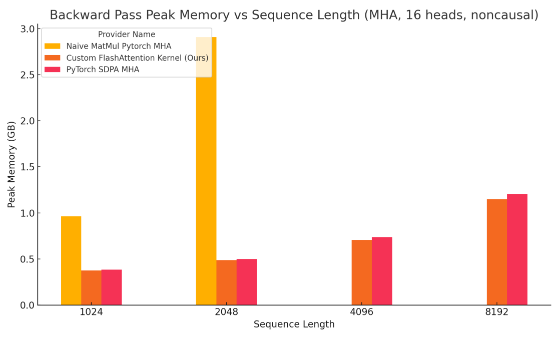
$$\Theta\left(\frac{N^2 d^2}{M}\right)$$

HBM accesses.

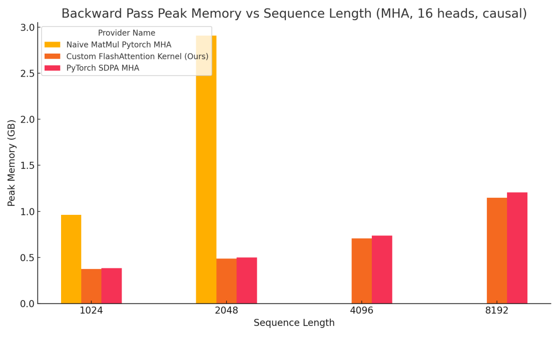
Results. We extend our evaluation to include the backward pass of multi-head attention. As shown in Figure 9, our Triton-based FLASHATTENTION kernel remains highly memory-efficient in the backward setting. Across all sequence lengths and both masking regimes, our implementation consistently uses less memory than naive baselines.

This is attributable to the recomputation strategy adopted in the FLASHATTENTION backward algorithm, which avoids materializing the full attention or softmax matrices during the backward pass. Instead, key statistics from the forward pass (the row-wise max and normalization constants) are used to rederive gradients on the fly. This recomputation significantly reduces high-bandwidth memory (HBM) traffic and minimizes intermediate buffer allocation.

Additionally, our kernel fuses gradient computation across $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ within a single pass over the SRAM tile, avoiding redundant loads and improving memory locality. These optimizations preserve the sub-quadratic memory footprint of FLASHATTENTION, even when the full backward graph is evaluated.



(a) Non-causal



(b) Causal

Figure 9: Peak memory usage of 16-head attention backward pass in non-causal and causal settings.