11868 PROJECT FINAL REPORT - FLASH ATTENTION

Vedant Bhasin [*1] **Nasrin Kalanat** [*1]

ABSTRACT

Transformers struggle with inefficiency when processing long sequences due to the quadratic scaling of time and memory requirements associated with self-attention. This limitation has led to the development of approximate attention methods aimed at reducing computational complexity. However, these approaches typically fail to achieve significant improvements in real-time performance. A crucial oversight in existing solutions is the lack of attention to input/output (IO) dynamics, specifically the interactions between various levels of GPU memory. To address this gap, the authors introduce FlashAttention (Dao et al., 2022), an exact attention algorithm that incorporates an IO-aware strategy. By optimizing memory interactions, particularly between the GPU's high bandwidth memory (HBM) and on-chip SRAM. For this project we incorporate flash attention with minitorch.

1 INTRODUCTION & MOTIVATION

[1] The main problem Flash Attention aims to solve is that of attention not scaling well with increased context length. The standard scaled dot product attention formula is as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Where Q has dimensions $[N, d_q]$, K has dimensions $[N, d_k]$, V has dimensions $[N, d_v]$, d_k is the dimensionality of the keys and queries, d_v is the dimensionality of the values, N is the common sequence length for queries, keys, and values. Note that the intermediate values before multiplication with V have shape $N \times N$ which means that the complexity increases quadratically with context length. The authors of the Flash Attention paper identify the bottleneck as GPU High Bandwidth Memory (HBM) reads/writes. Therefore the key challenges are: softmax reduction without access to full input and Backward without storing the large attention matrix from forward.

Improving the latency and memory footprint of the attention mechanism is crucial for several reasons. Firstly, it directly enhances the scalability of models to handle longer sequences, which is essential for tasks involving more extensive context, such as document summarization or complex question answering. By reducing latency, models can process information faster, making them more practical for real-time applications like machine translation and interactive chatbots. Secondly, optimizing memory usage allows for the deployment of more sophisticated models on resource-constrained environments, such as mobile devices and edge computing platforms. This democratizes access to advanced AI capabilities, enabling broader adoption and integration into everyday applications. Furthermore, more efficient memory management contributes to energy conservation and reduces the carbon footprint associated with training and deploying large models. As the demand for AI grows, sustainability becomes increasingly important, making efficient computing an imperative rather than an option. By addressing these aspects, FlashAttention not only pushes the boundaries of model performance but also aligns with the broader goals of making AI more accessible, responsive, and sustainable.

2 RELATED WORK

The concept of IO-Aware Runtime Optimization relates to strategies designed to improve the efficiency of reading and writing operations to different types of memory, which can vary significantly in speed. This optimization is crucial because it deals with I/O complexity—a core challenge in both traditional computing and modern deep learning architectures. These efforts can be tied to several foundational theories and models:

Memory Hierarchies: Understanding how data moves across different levels of memory (from fast, small caches to slower, larger storages) is fundamental to optimizing any computer process.

Working Set Model: This model helps predict the amount

[*]Equal contribution [1]Carnegie Mellon University, Pittsburgh, Pennsylvania. Correspondence to: Vedant Bhasin <vedantbhasin@cmu.edu>, Nasrin Kalanat <nkalanat@andrew.cmu.edu>.

Proceedings of the 5^{th} MLSys Conference, Santa Clara, CA, USA, 2024.

[1]All code can be found at https://github.com/NasrinKalanat/minitorch_flash_attention

of memory a program will require during execution, thereby optimizing memory allocation.

Data Locality: Optimizing data locality involves arranging data so that data elements accessed close together in time are also stored close together in memory, reducing delay.

Roofline Model: A visual model that serves as a tool to analyze the performance of HPC systems and applications by examining how computational intensity (operations count per data movement) interacts with memory bandwidth and computational peaks.

Analyses of Scalability: These involve understanding how systems perform under an increasing load, which can help in designing systems that scale effectively as data volume grows. The goal of such optimizations is not just theoretical improvement but also practical enhancements in the deep learning stack, encouraging the community to integrate these optimized processes more broadly.

2.1 Efficient ML Models with Structured Matrices

This topic addresses the use of structured matrices—matrices with a specific pattern or structure that reduces computational demands—as a solution to the computational bottlenecks in machine learning, particularly in matrix multiplication operations. Examples include:

Sparse and Low-rank Matrices: These matrices are simplified forms where many elements are zero (sparse) or where the matrix can be approximated by a product of two smaller matrices (low-rank).

Fast Transforms: Such as Fourier and sine/cosine transforms, which can significantly accelerate operations on data.

Butterfly Matrices: A type of structured matrix that allows for efficient computation and storage, capable of representing complex matrix operations in a compressed form. These structured matrices are theoretically efficient, but there are practical challenges (the ”hardware lottery”) in adapting these theoretical gains into actual performance improvements due to the high optimization of traditional dense matrix operations.

2.2 Sparse Training

Training methodologies for sparse models—models where the connections are selectively pruned to reduce complexity without significantly compromising performance. The ”lottery tickets hypothesis” is a key theory here, suggesting that there exist small, highly-efficient sub-networks within larger networks that can achieve comparable performance. The block-sparse FlashAttention method applies this principle by using a fixed sparse pattern (butterfly pattern) throughout training, demonstrating effectiveness similar to denser models in specific tasks.

2.3 Efficient Transformer

Transformers, due to their architecture, face scaling issues as their computational and memory requirements increase quadratically with sequence length. Various techniques aim to mitigate these bottlenecks:

Approximation Techniques: Including using sparsity (hashing) and low-rank approximations to simplify the computations.

Combination of Sparse and Low-Rank Methods: For enhanced efficiency and accuracy.

Sequence Compression: To handle more data points simultaneously and extend contextual awareness without linear increases in resource demands. These optimizations are intended to maintain or improve the performance of transformers in processing long sequences more efficiently, crucial for applications in NLP and computer vision.

3 METHODOLOGY

The original Flash Attention implementation incorporates two primary techniques, tiling and recomputation.

3.1 Tiling

The authors make use of shared memory tiling to reduce the accesses to GPU HBM. This relies on the principle that a softmax over a contiguous input can be computed for separate sections and then multiplied with a scaling factor to give the same overall result. This enables softmax calculations using shared GPU memory without accessing HBM for individual components.

3.2 Recomputation

Rather than storing the large output from the attention forward, it is recomputed in the backward pass.

We integrate flash attention into minitorch as follows:

Fused CUDA Kernels: The fused kernels written according to the algorithms described in the Flash Attention paper.

Functions in cuda_kernel_ops: This module contains a collection of functions responsible for launching the CUDA kernels. These functions are the interface which connects the python code and the CUDA kernels.

Flash Attention Class in tensor_functions: The Flash Attention functionality is encapsulated within a class in the tensor_functions module. This class defines the high-level

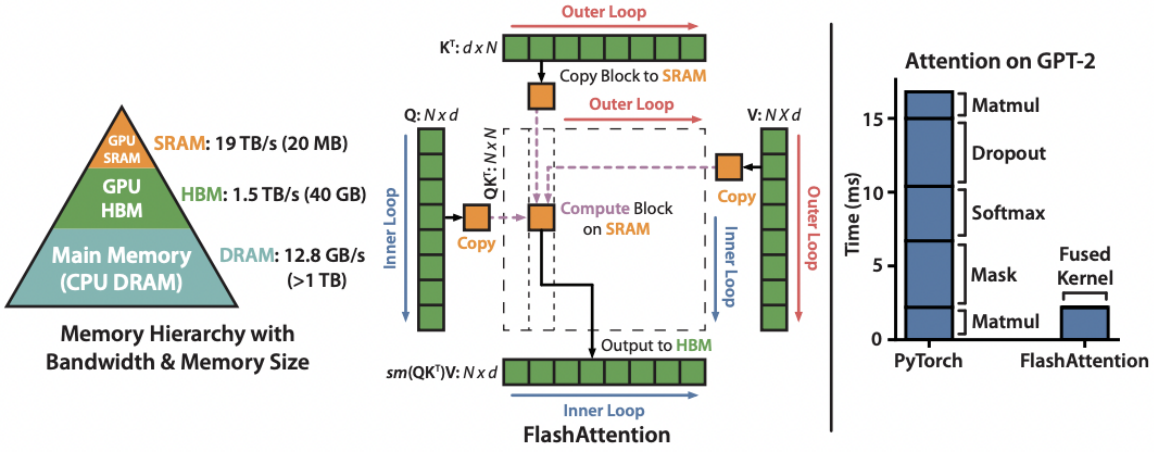

Figure 1. The key idea behind the algorithm. Accesses to HBM are minimized through kernel fusion (Dao et al., 2022)

Algorithm 1 FLASHATTENTION

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size M.

Set block sizes $B_c = \lceil \frac{M}{4d} \rceil$, $B_r = \min\left(\lceil \frac{M}{4d} \rceil, d\right)$.

Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}$, $\ell = (0)_N \in \mathbb{R}^N$, $m = (-\infty)_N \in \mathbb{R}^N$ in HBM.

Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \ldots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_1, \ldots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \ldots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.

Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \ldots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_i, \ldots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1, \ldots, m_{T_r}$ of size B_r each.

for $1 \le j \le T_c$ **do**

  Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.

  for $1 \le i \le T_r$ **do**

    Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.

    On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.

    On chip, compute $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.

    On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.

    Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1}(\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.

    Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.

  end for

end for

Return $\mathbf{O}$.

Figure 2. The Flash Attention algorithm described in detail (Dao et al., 2022)

interface for applying the Flash Attention mechanism within the MiniTorch framework. It abstracts the underlying CUDA operations in a class, and is called by a tensor method.

Tensor Method: To integrate Flash Attention into the typical workflow of MiniTorch, a specific tensor method is implemented. This method extends the MiniTorch tensor class, allowing users to directly apply Flash Attention to tensors as part of their model architectures. The method internally calls the Flash Attention function from the tensor_functions class, simplifying the process.

4 EVALUATION

We will use the following measures to evaluate the implementation performance.

4.1 Correctness

To evaluate the correctness of the implementation we design test cases that compare the output of our flash attention implementation with a ground truth attention mechanism. Our output matches the reference output.

4.2 Speedup

To measure performance gains we profile the forward and backward pass for different sequence lengths on different GPUs. We compare the time taken for our Flash Attention kernel with a minitorch baseline implementation.

5 EXPERIMENTS

5.1 Speedup over different sequence lengths

The experiment measures the computational efficiency gains from Flash Attention across varying sequence lengths. This is critical to understand because sequence length can greatly impact the time complexity of attention mechanisms, and improvements might scale differently as sequence lengths change.

5.2 Speedup over different modes

The focus would be on comparing the acceleration Flash Attention provides during training versus inference. It's important since training often requires backpropagation and may involve different computational paths than inference, affecting how much speedup is observed.

5.3 Speedup over different GPUs

We measure the speedup over three different GPUs: A100, T4, V100 to understand the different gains that can be made across different machines.

5.4 Results

In Figure 3 we can see that the time complexity for flash attention grows significantly slower across all GPUs in with sequence length compared with vanilla attention. This shows how tiling helps curb the quadratic time complexity of the attention mechanism. Figure 4 helps visualize the speedup per mode for every sequence length. For vanilla attention the execution time is dominated by the backward pass. Compared to our minitorch baseline we were able to achieve up to almost 40x speedup for the forward pass. Figures 5 and 6 directly show the speedups achieved for different GPUs at different sequence lengths. The highest speedups achieved were for the highest sequence length tested — 2048 on a V100 GPU we were able to achieve a speedup of almost 40x for forward and almost 16x for the backward pass.

6 ANALYSIS

The extent of speedup we were able to achieve is quite surprising, upon closer inspection this is likely due to our choice of baseline. The minitorch implementation of attention is far less efficient than popular auto differentiation frameworks, this is likely due to the mechanisms by which tensors are stored and updated. This leads to our baseline being quite slow, therefore the speedup from the fused kernel is quite large.

7 CONCLUSION

The Flash Attention algorithm marks a pivotal advancement in transformer optimizations, particularly in reducing latency and memory usage through IO-aware strategies such as tiling and optimized memory access. A key insight from this project is the critical role of IO-awareness in enhancing computational efficiency, which has proven essential for managing large-scale models effectively. Integrating advanced mechanisms like Flash Attention into autodiff frameworks such as MiniTorch has shown that sophisticated computational techniques can be seamlessly adopted, broadening their application and facilitating innovation. This integration not only supports the development of more efficient AI models but also underscores the importance of aligning algorithm design with hardware capabilities. These developments suggest a promising direction for future AI technologies, emphasizing sustainable, efficient, and powerful systems.

REFERENCES

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

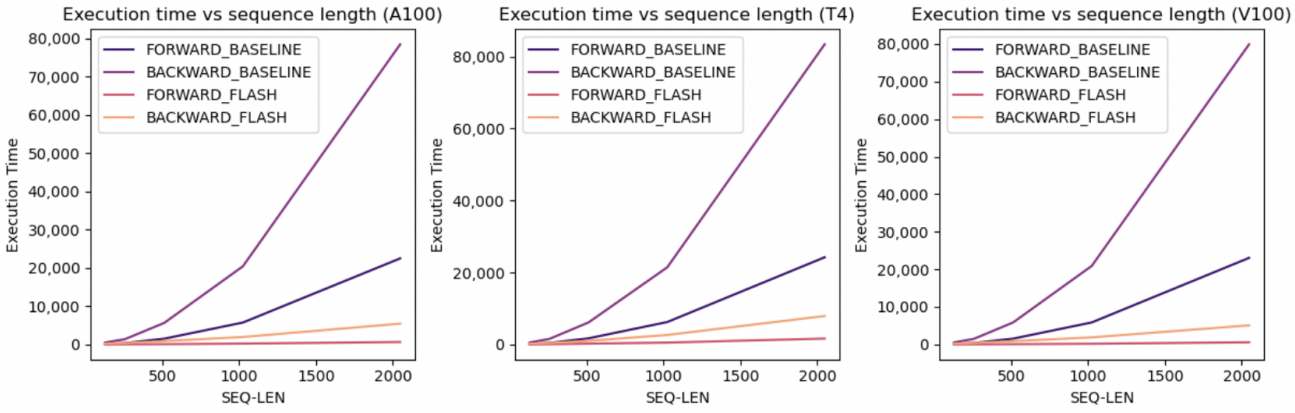

Figure 3. Execution time trends with different sequence lengths

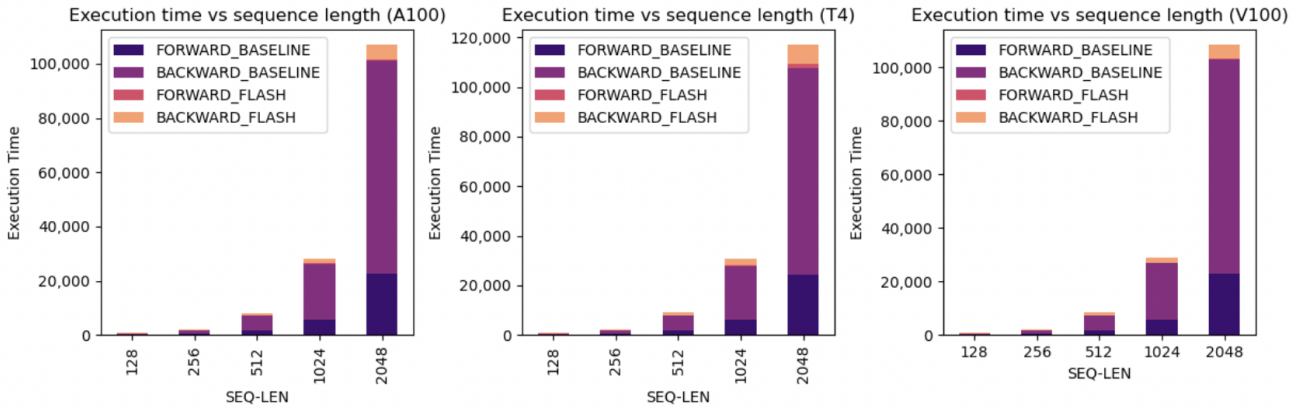

Figure 4. Execution time over different modes

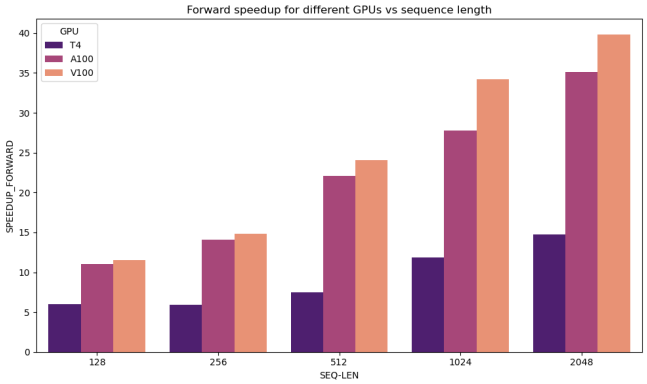

Figure 5. Forward speedup for different GPUs over different sequence lengths

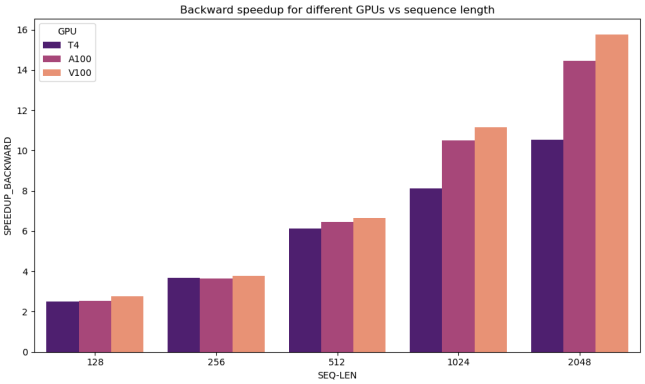

Figure 6. Backward speedup for different GPUs over different sequence lengths