Case Western Reserve University

IMPLEMENTING FLASH ATTENTION-2 USING TRITON

Anonymous authors
Paper under double-blind review

Pranav Balabhadra, Arohi Mehta, Ali Puccio

ABSTRACT

Large Language Models (LLMs) like OpenAI's GPT-3/4 and Meta's LLaMA require immense computational resources for training and inference, resulting in significant financial costs. OpenAI reportedly incurred half a billion dollars in GPU expenses by mid-2023, while Meta's LLaMA cost several million dollars per training run. Reducing these costs by even 1% could translate into substantial savings. This project aims to explore cost-efficient methods for training and inference by implementing flash attention first introduced in*FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning* by Tri Dao, using Triton, a Python-like language designed for GPU programming that promises more optimized kernels compared to standard frameworks like PyTorch. By re-implementing the FlashAttention-2 mechanism in Triton and comparing its performance to the PyTorch implementation, we aim to assess potential improvements in both computational efficiency and cost-effectiveness, providing valuable insights for large-scale model training.

1 INTRODUCTION

The primary aim of this project is to implement and benchmark the scaled flash attention mechanism as described in *FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning* by Tri Dao. Specifically, the focus will be on using Triton to optimize flash attention, evaluating its potential to improve performance and reduce computational costs compared to a PyTorch implementation.

To achieve this goal, the project involves the following steps:

- Develop a baseline implementation of the scaled flash attention mechanism in PyTorch for reference and benchmarking.
- Leverage Triton to write a highly optimized version of the flash attention layer, exploring its parallelism and efficiency benefits.
- Conduct computational benchmarks to compare the Triton and PyTorch implementations, focusing on metrics such as latency and performance. Additionally, comparing these metrics on the forward and backward passes.

Through this project, we aimed to assess whether Triton can outperform PyTorch in implementing Flash Attention, providing insights into its utility for optimizing critical components of deep learning models. This evaluation will contribute to understanding Triton's potential role in enhancing the scalability and efficiency of modern AI systems.

2 BACKGROUND

2.1 FLASH ATTENTION-1

FlashAttention-1 introduces a novel approach to compute attention mechanisms more efficiently. It achieves better efficiency by leveraging tiling and recomputation strategies to reduce memory usage. The key innovations of FlashAttention-1 include:

1. Breaking down the attention computation into smaller tiles that fit in fast GPU memory.
2. Avoiding storing the full attention matrix in GPU high-bandwidth memory (HBM).
3. Recomputing certain intermediate values during the backward pass instead of storing them.

- Triton exposes intra-instance parallelism through operations on blocks, abstracting away complexities of CUDA thread block management.

Triton aims to enable researchers with minimal CUDA experience to write highly efficient GPU code. It has shown promise in optimizing various deep learning operations, including attention mechanisms, potentially offering performance improvements over standard PyTorch implementations.

3 METHOD

3.1 EXPERIMENT DESIGN

The goal of this experiment is to evaluate the performance and efficiency of the flash attention mechanism implemented in Triton, with PyTorch serving as a baseline for comparison. The experiment is divided into two phases: implementation and performance benchmarking. Each phase is designed to assess both the correctness and computational efficiency of the Triton implementation across various configurations.

3.2 CODE OVERVIEW

The project began by setting up an environment with PyTorch and Triton installed, ensuring GPU support was properly configured. First, we implemented the Flash Attention-2 mechanism in Triton, leveraging Triton's ability to optimize GPU kernels for memory-efficient attention. Flash Attention-2 reduces memory usage by computing attention in blocks, avoiding the need to explicitly materialize large intermediate matrices. This mechanism involves partitioning queries, keys, and values into smaller chunks, performing attention computations within these chunks, and writing results back to global memory incrementally. Key operations, including block-wise matrix multiplication, Softmax, and normalization, were written in Triton using custom GPU kernels to maximize parallelization and memory efficiency. After completing the Triton implementation, we tested it with synthetic data to ensure correctness and stability. Next, we implemented Flash Attention-2 using PyTorch as a baseline for benchmarking. Both implementations included custom forward and backward pass kernels, ensuring that gradients were computed efficiently during backpropagation. Following the implementation, we conducted performance benchmarking to measure execution time, memory usage, and computational efficiency across varying input sizes and conditions. We used PyTorch's CUDA functions for precise timing of GPU execution and performed multiple iterations to obtain reliable and consistent results. Finally, the performance results were analyzed to compare the Triton and PyTorch implementations. The findings highlighted the extent to which Triton outperformed PyTorch, particularly in terms of execution speed and memory efficiency. These results were visualized to demonstrate the computational advantages of using Triton for Flash Attention-2. The primary goal of this project was to deepen our understanding of Triton programming and its application to memory-efficient attention mechanisms in LLM architectures. By implementing the Triton version first and referring to the PyTorch implementation for comparison, we successfully evaluated the potential of Triton in optimizing critical components of LLMs.

3.3 TRITON IMPLEMENTATION

Due to the length of these functions, we are summarizing the key functions and what is accomplished in each portion. Note: we did take heavy influence from Tri Dao's Flash Attention implementation using Triton to perform our experiment.

The function `_attn_fwd_inner` implements the core computations for the attention mechanism, iterating over KV blocks and updating the accumulator.

In this implementation:

- The function processes different computation stages, controlled by the `STAGE` parameter.
- Masks are employed to enforce causality for autoregressive models.
- Maximum and sum operations are used to maintain numerical stability when updating the softmax normalization terms.

3.3.1 FORWARD PASS FUNCTION

The `_attn_fwd` function is the main Triton kernel, orchestrating memory loading, block-wise computation, and result storage.

In this implementation:

- The grid is dynamically defined for parallel execution based on the input shape.
- `tl.make_block_ptr` is leveraged to optimize memory access patterns.
- Specific stages are executed depending on the causal masking requirement.

3.3.2 PYTHON WRAPPER FOR FORWARD PASS

The static method `forward` provides a Python-level interface, integrating the Triton kernel with PyTorch.

In this implementation:

- Kernel parameters configure and launch the `_attn_fwd` kernel.
- The `STAGE` variable is created to adjust and enforce causality as needed.
- The computed output is returns and stores intermediate results for the backward pass.

3.3.3 PREPROCESSING DELTA FOR BACKPROPAGATION

The kernel `_attn_bwd_preprocess` calculates the intermediate `Delta` tensor, used to normalize gradients during the backward pass. In this implementation:

- The inputs are: `O` (output), `DO` (gradient of output), and tensor dimensions.
- Compute $\Delta_m = \sum(o \cdot do)$ for each row, where o and do are slices of `O` and `DO`, respectively.
- The outputs are: Δ values are stored for later use in gradient normalization.

3.3.4 GRADIENT COMPUTATION FOR K AND V

The kernel `_attn_bwd_dkdv` computes gradients for K and V. In this implementation:

- The inputs are: `Q`, `K`, `V`, `DO`, and attention scaling factor.
- Iteratively, the gradients of V ($\frac{\partial V}{\partial L}$) are computed by applying the softmax attention scaling and backward projection. Additionally the gradients of K ($\frac{\partial K}{\partial L}$) by propagating DO through $QK^\top$ interactions, adjusted by precomputed Δ values.
- Autoregressive masking is handled to ensure causality in attention.

3.3.5 GRADIENT COMPUTATION FOR Q

The kernel `_attn_bwd_dq` computes gradients for Q. In this implementation:

- The inputs are: `Q`, `K`, `V`, `DO`, and normalized gradients from `_attn_bwd_preprocess`.
- backward pass logic is used for $QK^\top$ to compute $\frac{\partial Q}{\partial L}$. The kernel adjusts for scaling and iterates over masked and unmasked sections of Q.

3.3.6 MAIN BACKWARD KERNEL

The main kernel `_attn_bwd` orchestrates the backward computation. In this implementation:

- Memory offsets are set for batch-head operations and initializes loop pointers.
- `_attn_bwd_dkdv` is called to compute K and V gradients. Additionally, `_attn_bwd_dq` is called for Q gradients.
- Computed gradients for Q, K, and V are stored to their respective tensors.

3.3.7 PYTHON WRAPPER FOR BACKWARD PASS

This is the static `backward` function. In this implementation:

- Δ is prepared using the `_attn_bwd_preprocess`.
- grid configuration invokes `_attn_bwd` with based on tensor dimensions.
- Gradients are returned using `dq`, `dk`, and `dv`.

3.4 PYTORCH IMPLEMENTATION

For the PyTorch implementation we implemented a forward and backwards pass for Flash Attention.

3.4.1 FORWARD PASS

The attention output is the weighted sum of values based on the similarity between queries and keys. In the PyTorch implementation, the forward pass was responsible for computing this.

- The query and key matrices are multiplied using the `torch.matmul()` function, which computes the dot product.
- When `causal=True`, a lower triangular mask is applied to the scores to prevent attention to future tokens in causal self-attention. The mask is created using `torch.tril()` and then applied to the score tensor with `masked_fill()`, setting future tokens' scores to `float('-inf')`, which makes their attention weights zero after the softmax operation.
- The softmax function is applied to the scores to convert them into probability distributions.
- The attention weights are then used to compute the weighted sum of the value vectors (`v`) by performing another matrix multiplication with `torch.matmul()`. The output of this function is the final output of this attention mechanism.

3.4.2 BACKWARD PASS

The backward pass is responsible for computing the gradients of the attention weights, query, key, and value tensors. This allows the model to adjust its parameters through gradient descent to minimize the loss during training.

In this implementation:

- After computing the attention output, the backward pass is called using `out.backward(torch.randn_like(out))`. This simulates the gradient computation from a loss function during training.
- During the backward pass, PyTorch automatically computes the gradients of the input tensors (`q`, `k`, and `v`) with respect to the output using the chain rule. These gradients are then stored in the respective tensors' `grad` attributes and can be used to update the model's weights.
- After the backward pass, an optimizer (e.g., `torch.optim.Adam`) is used to update the model parameters based on the computed gradients.

4 EXPERIMENT

For the results we measured computing performance and latency. The following section describes our results for forward pass only, backward pass only, and forward and backward pass.

4.1 FORWARD PASS ONLY

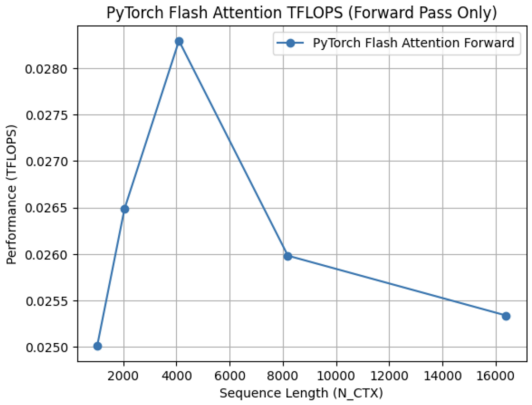

Figure 1: Measuring forward pass TFLOPS for PyTorch Flash Attention-2.

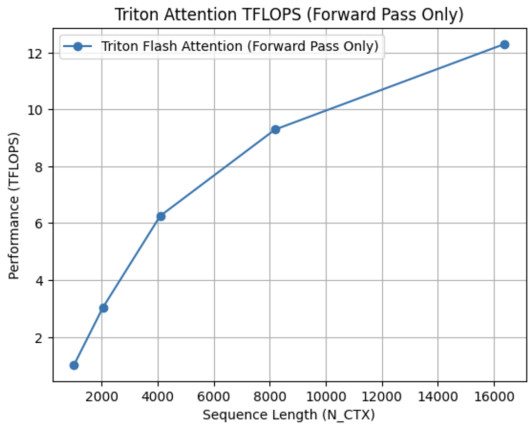

Figure 2: Measuring forward pass TFLOPS for Triton Flash Attention-2.

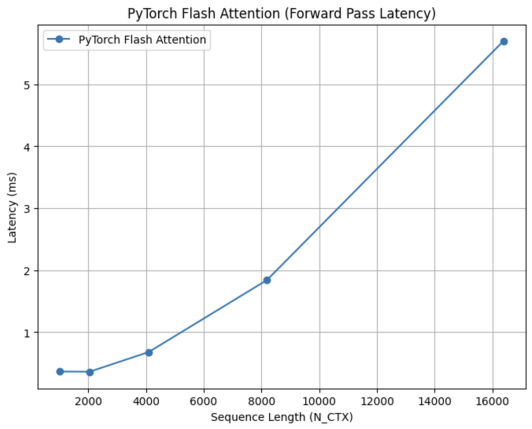

Figure 3: Measuring latency for PyTorch Flash Attention-2.

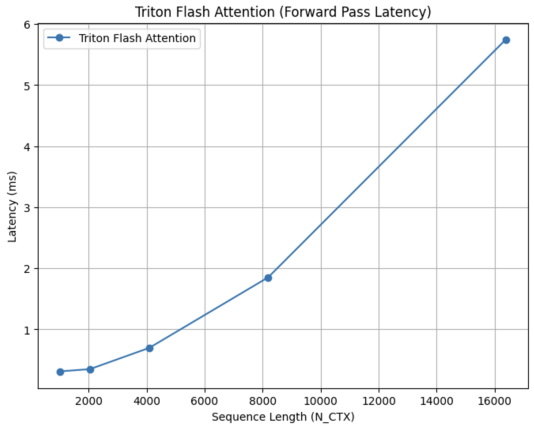

Figure 4: Measuring forward pass latency for Triton Flash Attention-2.

4.2 Backward Pass Only

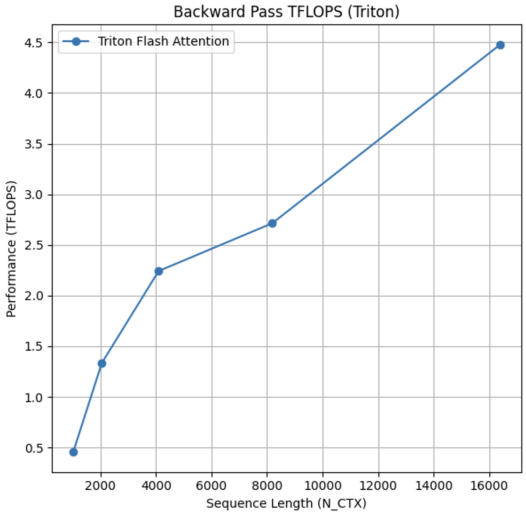

Figure 5: Measuring backward pass TFLOPS for Triton Flash Attention-2.

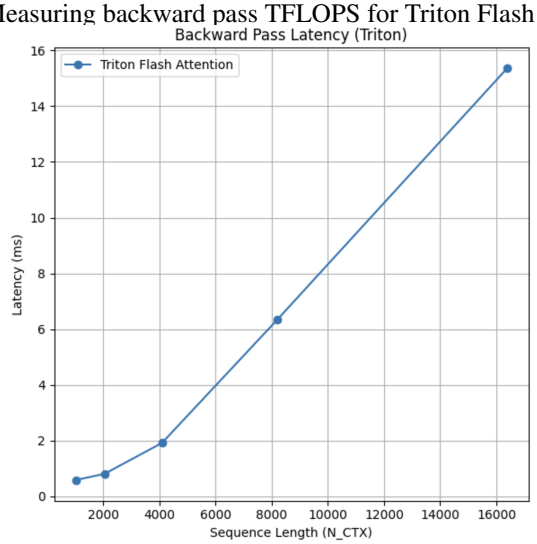

Figure 6: Measuring backward pass latency for Triton Flash Attention-2.

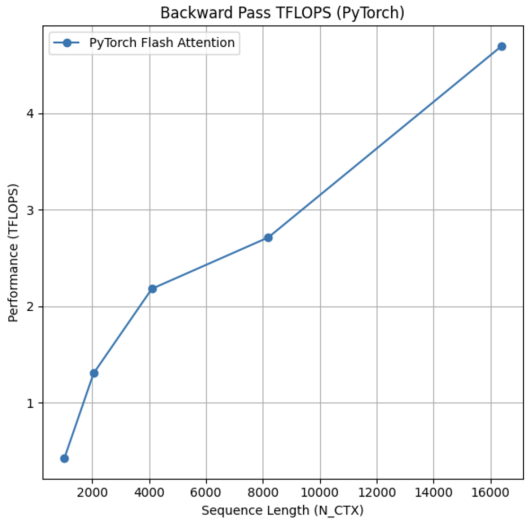

Figure 7: Measuring backward pass TFLOPS for PyTorch Flash Attention-2.

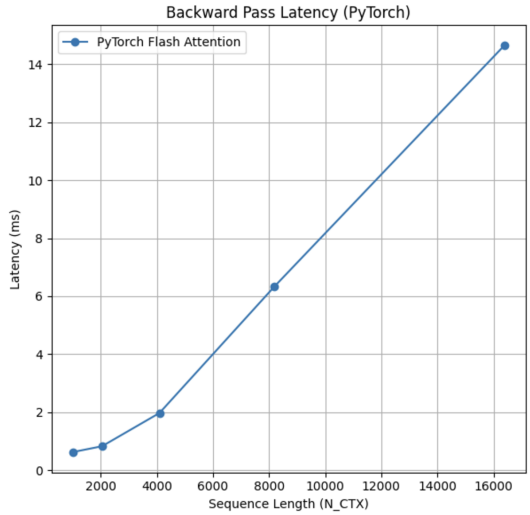

Figure 8: Measuring backward pass latency for PyTorch Flash Attention-2.

4.3 Forward and Backward Pass

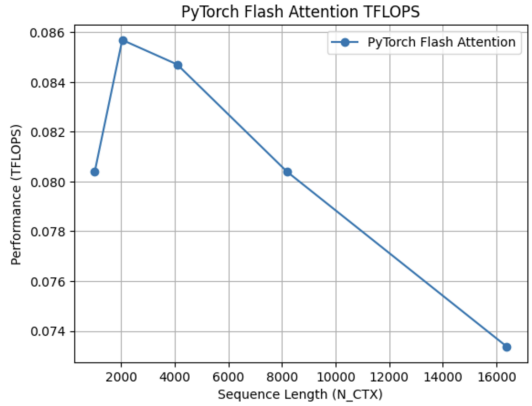

Figure 9: Measuring forward and backward pass TFLOPS for PyTorch Flash Attention-2.

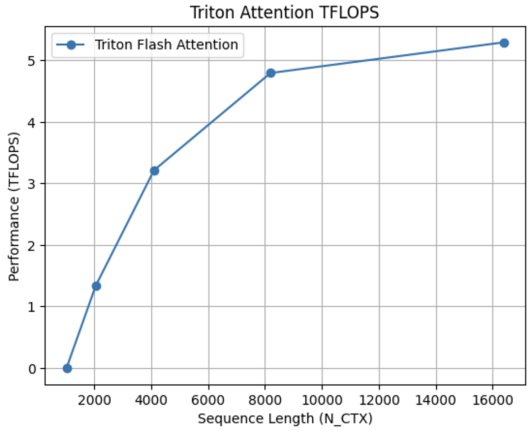

Figure 10: Measuring forward and backward pass TFLOPS for Triton Flash Attention-2.

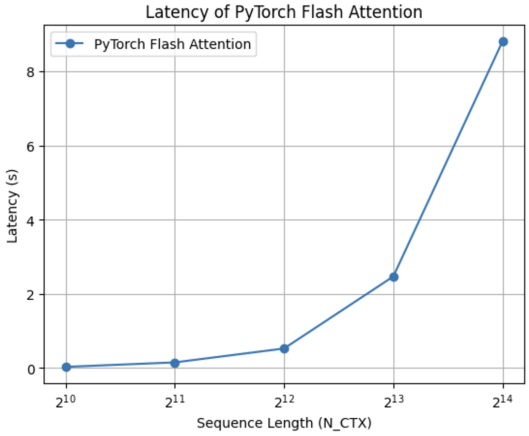

Figure 11: Measuring forward and backward pass latency for PyTorch Flash Attention-2.

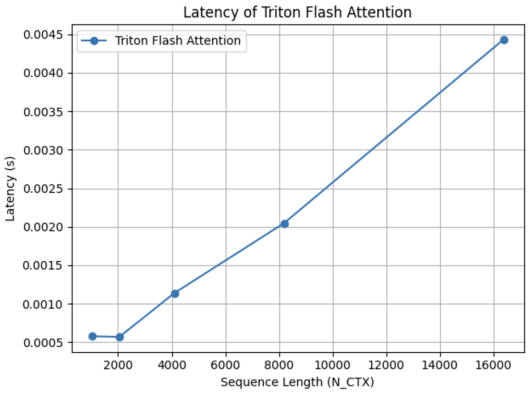

Figure 12: Measuring forward and backward pass latency for Triton Flash Attention-2.

5 Conclusion

Based on the results of our experimentation, we can see that when observing solely the forward pass, Triton Flash Attention outperforms PyTorch Flash Attention significantly in terms of TFLOPs.

When looking at the latency, although minimal, Triton Flash Attention has lower latency than PyTorch Flash Attention. Surprisingly, when looking solely at backward pass, there is almost no difference in latency or TFLOPS between the two implementations. When analyzing the combined TFLOPS of running both the forward and backward pass, we see once again that the TFLOPS for Triton Flash Attention is significantly higher than that of the PyTorch implementation whereas the latency is again, not much different between the two. We believe that we are getting these results because Triton is specifically optimized for GPU compute workloads. It uses kernel fusion, efficient memory access patterns, and customized GPU thread-block configurations to achieve high FLOP utilization. In the forward pass, Triton can fully exploit its optimizations because the operations are simpler (matrix multiplications, softmax, etc.), leading to higher throughput. PyTorch's Flash Attention implementation might rely more on general-purpose GPU kernels (e.g., cuBLAS), which are not always as tightly optimized as Triton for this specific workload. We believe the reason we see similar results in the backward pass is because the backward pass involves more complex operations, including the gradient computations for the input tensors. These computations involve more branching and memory-bound operations, where the benefit of Triton's optimizations may diminish. The backward pass is often memory-bound rather than compute-bound due to the need to read and write large amounts of data during gradient propagation. Since both Triton and PyTorch implementations likely hit similar memory bandwidth limitations, the performance (TFLOPS) becomes similar. The memory-bound nature of the backward pass also means that latency is governed by data transfer speeds rather than computational efficiency. As a result, both Triton and PyTorch implementations exhibit similar latencies. These results in the forward and backward-only testing carry over to when we test the performance and latency of both, explaining why we are seeing better TFLOPS in Triton than in the PyTorch implementation.

6 Future Work

In the future, we would like to extend our findings to LLMs by integrating the optimized Flash Attention mechanism implemented in Triton. We would investigate its performance on various LLM architectures, assessing both training and inference efficiency. This would include benchmarking against existing implementations in frameworks like PyTorch and exploring the scalability of our approach across different model sizes and datasets.

References

[1] Isamu Isozaki. *Understanding the Triton Tutorials, Part 1*. Medium, 2023. Available at: https://isamu-website.medium.com/understanding-the-triton-tutorials-part-1-6191b59ba4c. Accessed: October 4, 2024.

[2] Soohwan Kim. *Attentions - A Collection of Attention Mechanisms Implemented in PyTorch*. GitHub, 2023. Available at: https://github.com/sooftware/attentions?tab=readme-ov-file. Accessed: October 4, 2024.

[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. *Attention is All You Need*. In Advances in Neural Information Processing Systems, pp. 5998–6008, 2017.

[4] Tri Dao. *FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning*. International Conference on Learning Representations, July 17, 2023.

[5] Clint Greene. *Developing Triton Kernels on AMD GPUs*. ROCm Blogs, April 15, 2024. Available at: https://rocm.blogs.amd.com/artificial-intelligence/triton/README.html. Accessed: December 8, 2023.

[6] Philippe Tillet. *Introducing Triton: Open-Source GPU Programming for Neural Networks*. OpenAI, July 28, 2021. Available at: https://openai.com/index/triton/. Accessed: December 6, 2024.