EE451 Project Report
LLM Inference Comparison

Introduction

Large language models (LLMs) have revolutionized the field of natural language processing, enabling a range of advanced applications from text generation to machine translation. Despite its usefulness, LLM is both computation- and memory-intensive. There are significant challenges for real-time inference and deployment at scale. The computational cost is often driven by the attention mechanism, a core component of these models, which computes contextual relationships between tokens within input sequences.

This project explores the efficiency of LLM inference by comparing the performance of a standard attention mechanism against a more recent optimization technique, FlashAttention. FlashAttention aims to address the bottlenecks associated with traditional attention by reducing memory usage and computational overhead, potentially leading to faster and more efficient inference.

In this project, several work tasks will be done to identify bottlenecks in LLM inference and explore strategies to accelerate it. The key works are as follows:

- Profiling LLM Inference in PyTorch
- Implementation of Standard Attention in PyTorch
- Implementation of FlashAttention in CUDA
- Evaluation and Performance Testing
- Comparison of Average Runtime

Algorithm

The attention mechanism is a key component of LLMs, enabling models to focus on specific parts of input sequences based on contextual relevance. Standard Attention involves computing scaled dot-product attention, which can become computationally and memory expensive with larger models and sequences. Below is the pseudocode of a Standard Attention implementation [1].

Algorithm 0 Standard Attention Implementation

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.

Profiling LLM Inference

- Model: LLaMA-2 [2]
- Testing environment: A100, 32GB VRAM
- Test File:
  - example_text_completion.py (provided by Meta)
  - Added PyTorch profiler

- Key Findings:

  During the profiling of LLaMA-2 inference, we found that functions such as aten::copy_, aten::_to_copy, and aten::to were among the most time-consuming and were called repeatedly during the inference process as shown in the graph below. These functions are associated with data copying and memory management, indicating that a considerable portion of the computational time is spent on memory operations rather than core inference tasks.

  This observation aligns with the hypothesis that memory-bound operations can be a significant bottleneck in LLM inference. The high

frequency of these function calls suggests that the model's internal operations involve extensive data transfers and memory copying, which can lead to increased execution time and reduced efficiency.

Identifying these functions as bottlenecks underscores the importance of optimizing memory access and minimizing redundant data movements in the inference process. This is why we decided to explore techniques like FlashAttention, which aim to reduce memory access and streamline computations.

aten::copy_	3.01%	270.029ms	6.29%	564.694ms	11.607us	282.436ms	17.21%	333.597ms	6.857us	48653
aten::_to_copy	1.61%	144.890ms	7.25%	650.793ms	18.769us	0.000us	0.00%	203.698ms	5.875us	34674
aten::to	0.73%	65.738ms	7.77%	697.672ms	14.328us	0.000us	0.00%	200.473ms	4.117us	48692

Context

The memory-bound nature of standard attention mechanisms results in high bandwidth memory (HBM) usage and frequent data transfers, impacting performance and scalability. These bottlenecks are especially concerning in real-time applications and environments with limited resources, where efficient inference is crucial.

Various techniques have been proposed to improve attention mechanisms' efficiency, but many still face limitations. Some rely on approximations that can affect accuracy, while others require specialized hardware that isn't always accessible. These limitations have driven the search for new methods that can offer better performance without compromising accuracy.

FlashAttention emerges as a promising solution to these challenges. By reducing memory access and optimizing computational operations, FlashAttention aims to streamline the attention process, leading to faster inference times and lower memory usage.

Acceleration Strategy

The traditional attention mechanism involves significant memory access, leading to bottlenecks in computation. FlashAttention addresses these challenges by focusing on reducing memory access and optimizing the computation process. Below are the core concepts and advantages of FlashAttention:

- Reduced Memory Access: FlashAttention aims to minimize HBM access, a primary bottleneck in standard attention due to the large data transfer requirements. By reducing the frequency and volume of HBM read and write operations, FlashAttention achieves significant performance gains [1].

- Softmax Reduction Without Full Input: The softmax reduction, a crucial step in the attention mechanism, is computed without the need to access the entire input at once. This selective data access reduces memory consumption and speeds up computation [1].

- Block-wise Computation: FlashAttention adopts a block-wise computation approach, where the input is split into smaller blocks. This tiling technique allows the system to perform several passes over these blocks, incrementally computing the softmax reduction. By managing memory in this way, FlashAttention reduces overhead and enhances efficiency [1].

- Performance Improvements: Although FlashAttention introduces more floating-point operations (FLOPS) due to recomputation, it still demonstrates better performance compared to standard attention. The reduced HBM access is the key driver for this improvement, as attention is known to be a memory-bound algorithm [1].

- Potential for Approximate Attention Algorithms: FlashAttention's optimization of memory access has implications for approximate attention algorithms. With its block-wise computation and reduced overhead, FlashAttention paves the way for block-sparse versions that are even faster, scaling to very long sequences (up to 64k). This

scalability is critical for handling complex and large-scale language models [1].

Algorithm of FlashAttention [1]:

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

Hypothesis

The hypothesis for this project is that memory-efficient attention methods, such as FlashAttention, will reduce memory access and improve inference performance. This reduction in memory access can lead to lower overhead, faster execution times, and potentially fewer function calls due to optimized computational pathways.

The reason behind this hypothesis is that by using techniques like block-wise computation, FlashAttention minimizes the need for repeated memory operations. This optimized approach is expected to contribute to a more efficient inference process, reducing the computational cost and potentially lowering the number of function calls.

Experimental Setup

- Implementation: Implement standard attention in PyTorch and FlashAttention in CUDA
- Input Data: Random tensors were generated for query, key, and value with the given batch size, number of heads, sequence length, and model dimensions.
- Testing environment: A100 on CARC
- Memory: 32 GB VRAM
- Batch Sizes tested: 1, 4, 16, 64
- Sequence length: 64
- Evaluation Metrics: Measure the average runtime for LLM inference with varying batch sizes

Result and Analysis

The following results demonstrate the difference in CUDA execution time between standard attention and FlashAttention across four batch sizes. The results consistently show that FlashAttention outperforms standard attention, indicating its efficiency in reducing inference time.

- Batch Size = 1: The total CUDA time for standard attention was 624.202 ms, whereas FlashAttention completed in 135.383 ms. This represents an approximately 78% reduction in execution time, with FlashAttention offering a speedup factor of about 4.61 times.

- Batch Size = 4: Standard attention required 978.146 ms, while FlashAttention took 262.702 ms, marking a 73% reduction. This translates to a speedup factor of about 3.72 times.

- Batch Size = 16: The total CUDA time for standard attention was 423.332 ms, compared to 332.725 ms for FlashAttention, a 21% reduction. FlashAttention achieved a speedup of about 1.27 times compared to standard attention.

- Batch Size = 64: The most significant improvement is seen with this batch size, with standard attention taking 26.460 ms and FlashAttention only 1.358 ms. This represents a 95% reduction and a speedup factor of about 19.48 times.

These results clearly demonstrate the advantages of FlashAttention in terms of reducing inference time and achieving higher speedup compared to standard attention. The most dramatic reduction in execution time occurs at batch size 64, where the speedup factor is nearly 20 times. This indicates that FlashAttention's approach becomes increasingly efficient as the workload grows, providing a scalable solution for inference tasks requiring high throughput.

Conclusion

In this project, we profiled the inference performance of LLaMA-2 to identify computational bottlenecks and inefficiencies. This initial analysis revealed that standard attention mechanisms were prone to overhead, with frequent memory-bound operations like aten::copy_ and aten::_to_copy.

To address these inefficiencies, we implemented FlashAttention in CUDA, a memory-efficient mechanism designed to reduce HBM access and optimize computation. We then compared FlashAttention with standard attention, using varying batch sizes to assess their performance.

The results showed significant speedups with FlashAttention across all batch sizes. At batch size 1, FlashAttention was 4.61 times faster than standard attention, and this speedup grew to nearly 20 times at batch size 64. This substantial improvement suggests that FlashAttention's block-wise computation and minimized memory operations can play a key role in accelerating large language model inference.

Reference

[1] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In NeurIPS, 2022.

[2] Touvron, Hugo, et al. "Llama 2: Open Foundation and Fine-Tuned Chat Models." arXiv preprint arXiv:2307.09288 (2023).