Implementing NestedTensors for Transformer Models

Kushaan Gowda (kg3081), Harsh Benahalkar (hb2776), and Siddarth Ijju (si2462)

***Abstract*—In this paper, we present an enhancement to IBM's Foundation Model Stack, specifically the LLaMA architecture, by modifying its source code to also support nested tensors. Nested tensors, which are jagged tensors with irregular shapes, offer significant memory efficiency by eliminating the need for padding, resulting in a nearly 50% reduction in memory usage when dealing with variable-length sequences or inputs. While this modification does not lead to significant improvements in computational time efficiency, it enhances memory utilization, which is particularly advantageous for tasks involving multi-modal data. We perform a detailed time analysis on several fundamental functions and identify certain operations that could not yet be vectorized for nested tensors, highlighting areas where future optimizations may be necessary. Through empirical results, we demonstrate the potential of nested tensors to improve the scalability and memory efficiency of large-scale transformer models in real-world applications.**

I. Introduction

A. Background and Motivation

Large language models (LLMs) are becoming increasingly integral to the daily lives of consumers worldwide. However, as the adoption of LLMs grows, the associated costs of hosting and serving these models are expected to rise significantly. A key challenge in achieving cost-effective inference for LLMs lies in their reliance on a fixed input sequence length, necessitating the padding of shorter sequences to match the specified length. At the scales at which many large models are deployed, the cumulative padding overhead becomes substantial, especially given that this added padding represents unused information for most practical applications [9]. Implementing a transformer architecture capable of handling irregular batches of unpadded inputs could significantly reduce the memory footprint, with these benefits becoming increasingly pronounced as the scale of deployment grows.

B. Problem Statement

The increasing adoption of transformer-based models has highlighted inefficiencies in handling variable-length inputs, often requiring padded tensors that lead to significant memory overhead and computational inefficiencies. While PyTorch's introduction of nested tensors[3] provides a promising framework for processing ragged sequential inputs efficiently, their integration with transformer architectures remains largely unexplored. This work aims to address this gap by implementing the transformer architecture using nested tensors, evaluating the feasibility of this approach, and benchmarking its performance in terms of memory utilization and computational efficiency.

C. Objectives and Scope

In particular, we focused the scope of our work on the IBM Foundation Model Stack[2] repository. Our focused objective was to reimplement the LLaMA implementation within the Foundation Model Stack to support nested tensors, and then to compare and benchmark the performance gains derived thereof. For the purposes of time, we additionally limited our work to implementing solely the forward pass of LLaMA. Furthermore, because the functionality of nested tensors is still limited in scope because the project is currently being developed, we focused our time on functionality that was feasible for us to implement without having to request significant new features from PyTorch itself. On this point, we have made note of some features which we believe may be useful to integrate directly into the nested tensor library in the future, which we will discuss in a later section.

II. Literature Review

A. Review of Relevant Literature

Arguably, the most critical innovation for the function of modern large language models is the transformer and the attention mechanism [7]. The highly parallel nature of attention and the capability to process large amounts of text data through the transformer have led to incredible advances in natural language generation and processing. However, even with the advances afforded by attention, the transformer represents a significant bottleneck for fast inference for large language models, specifically because the self-attention mechanism has quadratic complexity relative to the length of the input sequence [5]. In general, large language model inference is quite expensive due to the size of the models and the memory requirements to load their parameters, which make any optimizations to the memory involved for processing very important.

This issue has been known for a while, which has resulted in several other approaches to the underlying memory constraints for large language models. One of the first to be widely accepted and integrated in practical systems was continuous batching [8]. Continuous batching addressed the issue with the autoregressive nature of large language models, in that sequences had to be repeatedly processed in batches until completion. However, processing power could be wasted on shorter sequences while longer ones were still being processed. Continuous batching allows for sequences with finished computations to be hot swapped with new sequences every iteration by dynamically computing a batch size per

iteration. This approach is very successful, but still suffers from the underlying issue related to sequence lengths - namely, because each sequence can have a different length it is difficult to process them all in one batch in a memory efficient manner.

The majority of other approaches to the memory efficiency of the transformer revolve around the attention mechanism itself. One of the most celebrated advances in this category is FlashAttention [4]. FlashAttention introduced a few new novelties, namely that of a fused kernel for some attention operation and improved I/O operations with the matrices involved underneath the hood. Other research in a similar vein has focused on improving the quadratic complexity of naive self-attention by changing the number of tokens each other token must attend to within the mechanism [1]. Another recent development, PagedAttention, also attempted to address the memory inefficiency of normal attention through smart paging mechanism for the KV-cache and other memory intensive operations [6].

B. Identification of Gaps in Existing Research

Although memory optimizations are clearly an incredibly important issue, most modern research is focused on optimizing the attention mechanism itself. As aspect of attention that seems untouched is that of the fixed sequence length required for a large language model, and the relationship that occurs in practice with batching for those sequence lengths. This represents a significant opportunity for research, in that if we can develop a method that works around this requirement, we can significantly improve the memory efficiency of large language models in practice without having to modify the actual attention implementation mechanisms themselves.

III. Methodology

A. Data Collection and Preprocessing

Our approach to this project was designed to eliminate the need for a pre-collected dataset. Since the proposed changes were intended to function on any tensor input, we used random inputs for testing. To simulate the varying lengths of inputs in a real batch, we developed a custom data collator and dataset generator capable of producing tensors with randomized values and lengths, supporting both nested and real tensor datasets. The number of words per sentence was randomly sampled from a range of 10 to 256, and the length of each word was sampled from a range of 1 to 10 characters. On average, each generated sentence contained approximately 600 tokens.

B. Model Selection

The initial scope of this project was focused on LLaMA models. While several models are available in the IBM Foundation Model Stack, LLaMA was selected as the starting point due to its open-source nature and the accessibility of its model weights. To ensure consistency and feasibility, we restricted our computational environment to L4 and T4 GPUs, as these were readily available for experimentation. Consequently, we chose LLaMA models that could fit within the memory constraints of these GPUs and conducted our testing on the IBM LLaMA 160M Accelerator implementation.

C. Optimization Procedure(s)

Our project occupies a unique niche by focusing on optimizing the memory requirements for model passes rather than directly improving optimization procedures. Specifically, our efforts were centered on implementing nested tensors correctly, rather than employing conventional speed or memory optimization techniques. The rationale behind this approach is that the inherent structure of nested tensors—designed to handle variable-length data efficiently—would naturally reduce memory overhead compared to traditional tensor implementations. This optimization stems from the properties of nested tensors rather than any specific code modifications made to the LLaMA implementation.

D. Profiling Tools and Methods

The application was comprehensively profiled for both time and memory usage. A custom profiler was developed to analyze key LLaMA operations, including residual connections, rotary position embeddings, self-attention calculations, layer normalization, and lower triangular mask computations for the decoder. The time profiler measured the execution time of each operation, enabling the identification and analysis of performance bottlenecks when comparing PyTorch operations on nested and padded tensors.

For GPU memory profiling, we employed PyTorch's built-in profiler. This tool provided detailed insights into memory consumption during the execution of various operations. By analyzing memory allocation patterns for nested and padded tensors, we were able to further evaluate the memory optimization achieved through our implementation. The combination of custom and PyTorch profiling tools offered a thorough understanding of both time and memory performance characteristics.

E. Evaluation Metrics

To evaluate our implementation, we compared the tensor distributions produced with nested tensors against those generated with padded tensors at every computational step, using absolute tolerance measured via the L^∞ norm. This ensured that the output logits of nested tensors closely matched those of padded tensors, validating numerical consistency throughout the computation.

Experiments involved saving tensors from computations performed with both padded and nested tensors at each step. Due to dimensional differences, padded logits were truncated (Algorithm 1) to align with nested logits before comparison.

Algorithm 1 Truncating Padded to Match Nested Tensors

Require: Padded tensor $\mathbf{P}$, Nested tensor $\mathbf{N}$
Ensure: Truncated padded tensor $\mathbf{P}'$
  Initialize $\mathbf{P}' \leftarrow []$ ▷ Empty structure for truncated tensors
  for all $\mathbf{n}_i$ in $\mathbf{N}$ **do**
    $\mathbf{d} \leftarrow \dim(\mathbf{n}_i)$ ▷ Get dimensions of nested tensor $\mathbf{n}_i$
    $\mathbf{p}'_i \leftarrow \mathbf{P}_i[:\mathbf{d}]$ ▷ Truncate $\mathbf{P}_i$ to match $\mathbf{d}$
    $\mathbf{P}'$.Append($\mathbf{p}'_i$)
  end for
  return $\mathbf{P}'$

The L^∞ norm, which captures the maximum absolute deviation between tensors, was used to robustly identify significant discrepancies.

$$\text{Tolerance} = \|\mathbf{N} - \mathbf{P}'\|_\infty = \max \left|\mathbf{N}_{i_1,i_2,\ldots,i_k} - \mathbf{P}'_{i_1,i_2,\ldots,i_k}\right|$$

This stepwise evaluation confirmed the numerical stability of our implementation, demonstrating that nested tensors maintained consistency with padded tensors without introducing significant deviations.

IV. Experimental Results

A. Experimental Setup

All inference experiments for the proof of concept were conducted on a Google Cloud Platform (GCP) virtual machine configured with g2-standard-8. The VM featured 8 virtual CPUs, 32 GB of RAM, and a single NVIDIA L4 GPU (24 GB). A total of 300 samples were generated from the dataset, grouped into batches of 8, and processed over a single epoch. IBM's Llama-micro model was initialized using the *ibm-fms/llama-160m-accelerator* tokenizer and weights available through Hugging Face[1]. To ensure reproducibility, random seeds were set to 555 for CUDA, Hugging Face, PyTorch, and python.random.

B. Performance Comparison

This section compares the performance of our implementation across inference time, GPU memory utilization, and numerical accuracy, evaluating the efficiency and reliability of using nested tensors.

1) Inference Time Analysis: From Figure 1, nested tensors took longer to process than padded tensors, possibly due to the overhead of handling irregular shapes and dynamic memory access patterns. Unlike padded tensors, nested tensors may require additional effort to manage varying dimensions efficiently.

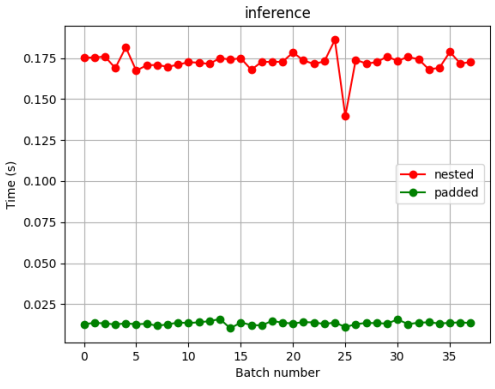
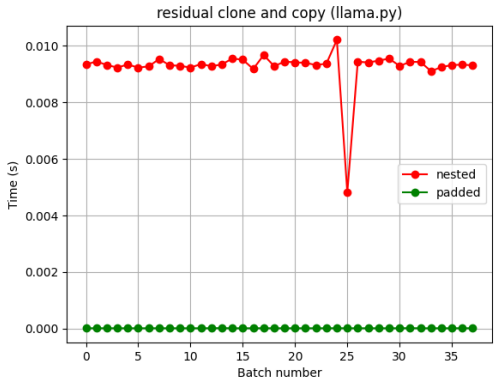


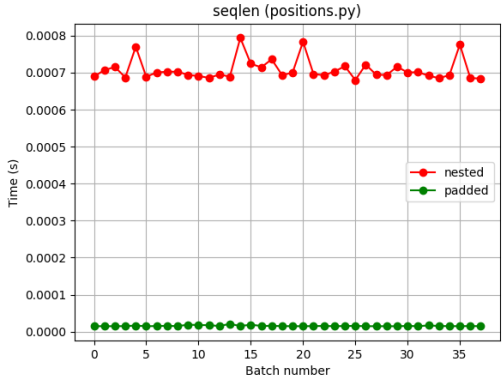
Fig. 1: Inference times for the entire model

We conducted additional profiling on specific sections of the code where vectorization of operations was not feasible (Figures 2a, 2b).

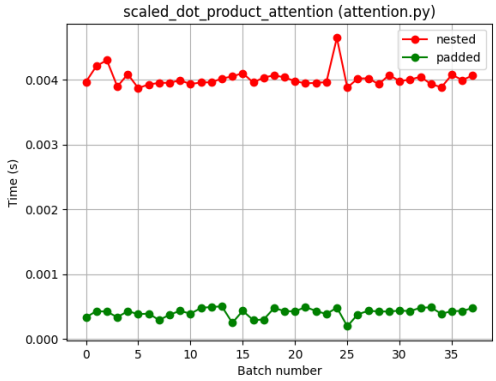
[1]https://huggingface.co/ibm-fms/llama-160m-accelerator



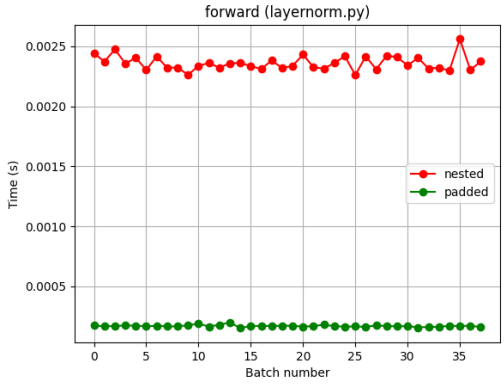
(a) Clone and copy function



(b) Computing seqlen



(c) Self attention

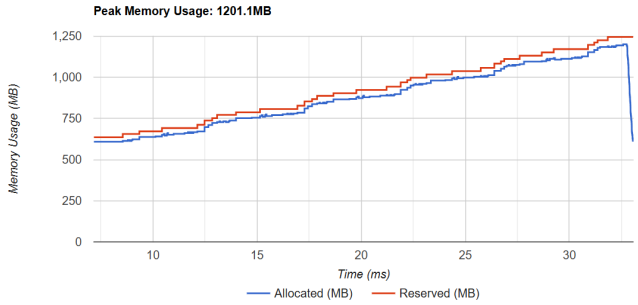


(d) Layernorm

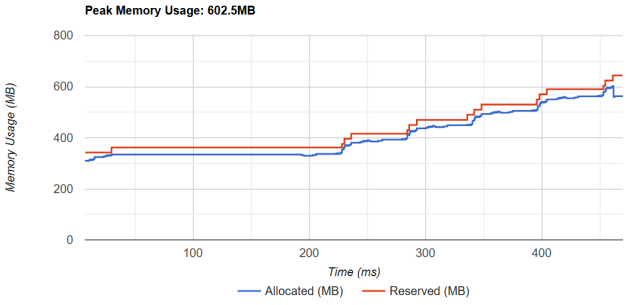
Fig. 2: Comparison of inference times for non vectorized operations

Finally, we profiled self-attention (Figure 2c) and layer normalization (Figure 2d), where no changes were made to the existing code base.

2) GPU Memory Utilization: In our analysis of GPU memory utilization (Figure 3), nested tensors demonstrated significantly improved efficiency, using nearly half the memory required by padded tensors. This reduction is likely due to the elimination of padding, which avoids allocating unnecessary memory for uniform tensor shapes. By dynamically managing irregular tensor dimensions, nested tensors optimize memory usage, making them particularly advantageous for variable-length input sequences.



(a) Padded tensors



(b) Nested tensors

Fig. 3: Comparison of GPU memory usage

3) Numerical Accuracy Validation: We performed numerical accuracy validation multiple times across several batches (Figure 4), comparing the outputs of nested and padded tensors. In all cases, the tolerance values were less than 1×10^{-6}, confirming the consistency of our implementation.

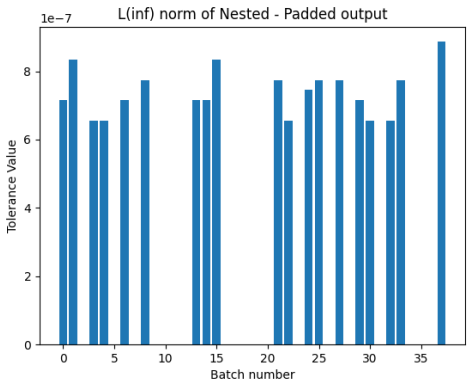


Fig. 4: Tolerance values

Figure 5 presents a Kernel Density Estimate (KDE) plot comparing the output distributions of nested and padded tensors. The close overlap between the two distributions demonstrates that nested tensors preserve numerical consistency with padded tensors, with only minimal differences observed.

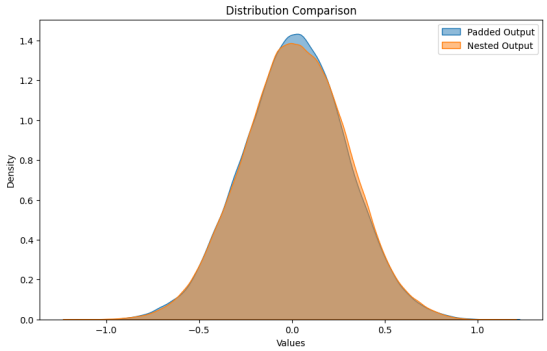


Fig. 5: KDE plot comparing the outputs of padded and nested tensors

C. Analysis of Results

The experiments demonstrated that nested tensors required approximately 10 times more inference time compared to padded tensors. The increase in time was consistent across various batch sizes and persisted even after profiling different sections of the code. However, GPU memory utilization showed significant efficiency, with nested tensors using nearly 50% less memory than padded tensors.

V. Discussion

A. Interpretation of Results

We began by comparing batch inference times using padded tensors and nested tensors (Figure 1). As expected, the nested tensor implementation exhibited slower batch times due to the presence of non-vectorized operations utilizing `for` loops, which become a bottleneck for larger batches. However, the observed slowdowns, approximately 10 times, were significantly higher than anticipated. To further investigate, we profiled sections of the code that were explicitly modified (Figures 2a, 2b). Although these sections showed slowdowns, as expected, the magnitude of the observed delays still did not fully account for the 10-fold slowdown in overall batch inference times. Subsequently, we profiled unmodified sections of the code (Figures 2c, 2d). Surprisingly, similar trends of slowdown were observed. After profiling the entire codebase, we concluded that native PyTorch operations were inherently less efficient when handling nested tensors compared to padded tensors, contributing significantly to the overall slowdown.

Despite the observed slowdowns in inference time, our analysis of GPU memory utilization revealed significant improvements when using nested tensors. Specifically, nested tensors demonstrated nearly a 50% reduction in memory usage compared to padded tensors. This reduction highlights the potential of nested tensors to optimize memory efficiency, particularly for applications involving large-scale models and

variable-length inputs, even when some computational overhead is introduced.

B. Challenges and Limitations

Our implementation faced several challenges, particularly with operations that could not be vectorized. Key difficulties included dynamically creating masks for irregular data and handling shape mismatch errors between tensors. Resolving these mismatches often required cloning one tensor and manually copying values from another, leading to additional computational overhead. Similarly, operations such as finding the maximum length along an unregular dimension of a nested tensor, slicing nested structures, and performing an unsqueeze operation in the 0th dimension proved cumbersome due to the irregular nature of nested tensors.

Another significant limitation was that the scaled dot-product operation, a core component of attention mechanisms, does not natively support nested tensor masks. We were unable to implement nested tensors for rotary position embeddings (RoPE) in a computationally efficient manner, further highlighting the challenges of adapting nested tensors to established architectural components.

These challenges were compounded by the inherent complexity of managing irregular data structures, which contributed to the observed inference slowdown. The lack of support for optimized operations in current frameworks further emphasized the difficulty of integrating nested tensors into high-performance workflows. While the memory efficiency gains were significant, these limitations underline the trade-offs between computational and memory efficiency when using nested tensors.

C. Future Directions

Building on the findings and challenges of this work, several avenues for future improvements can be pursued. A key focus is the development of efficient, vectorized implementations for operations such as dynamic mask creation, shape mismatch handling, slicing, and dimension manipulations, which currently hinder the computational performance of nested tensors.

Another promising direction is optimizing the use of nested tensors on GPUs by addressing native inefficiencies. Modern attention implementations are highly reliant on the batched nature of inputs, particularly sequence length, which creates challenges for irregular data structures like nested tensors. Enhancing GPU support for nested tensor operations, such as scaled dot products and rotary position embeddings (RoPE), could significantly improve their compatibility with core transformer components. This would involve leveraging low-level optimizations and specialized GPU kernels to handle irregular data structures more efficiently.

By improving the native efficiency of nested tensors on GPUs, future work can help bridge the gap between their memory advantages and computational overhead, making them more suitable for large-scale applications and diverse workloads.

VI. Conclusion

A. Summary of Findings

This study assessed the performance of nested tensors in large language models (LLMs) compared to padded tensors, with the following key findings:

1) Inference time: Nested tensors were approximately 10x slower than padded tensors due to inefficiencies in handling irregular data structures.
2) GPU memory utilization: Nested tensors used nearly 50% less GPU memory, highlighting significant memory efficiency.
3) Numerical accuracy: Output differences between nested and padded tensors were consistently within 1×10^{-6}, confirming numerical reliability.
4) Challenges: Issues included the lack of vectorization for operations such as dynamic mask creation, shape mismatch handling, and slicing, along with inefficiencies in native PyTorch operations like scaled dot-product attention and rotary position embeddings.

B. Contributions

Name	Contributions
Harsh Benahalkar	Dataset class and Datacollator class for HF and IBM inference pipeline. Baseline NestedTensor forward inference implementation. Tensor eval and time profiling for tolerance verification. Report.
Kushaan Gowda	Preliminary experiments on decoder models. Optimized implementation for NestedTensor on forward inference. Performed memory profiling on nested and padded tensors. Report.
Siddarth Ijju	Benchmarking inference setup and runs. Debug key issues in fms. Developed truncation methods and comparisons. Slides and report.

TABLE I: User Contributions

C. Recommendations for Future Research

Several key directions could enhance compatibility and efficiency in this line of research. Since nested tensors are still under development in PyTorch, further integration across the library, particularly for functional attention operations, is essential. Collaborating with the PyTorch team could significantly streamline future work.

Algorithmic improvements in attention mechanisms, such as enabling smaller, per-item attention masks instead of broadcasting a single matrix, represent a promising area for optimization. Similarly, developing vectorized implementations for operations like dynamic mask creation, shape mismatch handling, and slicing could address existing computational overheads.

Finally, optimizing GPU performance for nested tensors through specialized kernels and low-level improvements would

bridge the gap between their memory efficiency and computational performance, enabling broader adoption in large-scale applications.

ACKNOWLEDGMENT

We would like to thank Dr. Antoni Viros Martin and Professor Kaoutar El Maghraoui for their guidance throughout our work.

REFERENCES

[1] Rewon Child et al. *Generating Long Sequences with Sparse Transformers*. 2019. arXiv: 1904.10509 [cs.LG]. URL: https://arxiv.org/abs/1904.10509.

[2] IBM Foundation Model Stack Contributors. *Foundation Model Stack*. 2024. URL: https://github.com/foundation-model-stack/foundation-model-stack/.

[3] PyTorch Contributors. *torch.nested*. 2023. URL: https://pytorch.org/docs/stable/nested.html.

[4] Tri Dao et al. *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*. 2022. arXiv: 2205.14135 [cs.LG]. URL: https://arxiv.org/abs/2205.14135.

[5] Feyza Duman Keles, Pruthuvi Mahesakya Wijewardena, and Chinmay Hegde. *On The Computational Complexity of Self-Attention*. 2022. arXiv: 2209.04881 [cs.LG]. URL: https://arxiv.org/abs/2209.04881.

[6] Woosuk Kwon et al. *Efficient Memory Management for Large Language Model Serving with PagedAttention*. 2023. arXiv: 2309.06180 [cs.LG]. URL: https://arxiv.org/abs/2309.06180.

[7] Ashish Vaswani et al. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL].

[8] Gyeong-In Yu et al. "Orca: A Distributed Serving System for Transformer-Based Generative Models". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 521–538. ISBN: 978-1-939133-28-1. URL: https://www.usenix.org/conference/osdi22/presentation/yu.

[9] Yujia Zhai et al. "ByteTransformer: A high-performance transformer boosted for variable-length inputs". In: *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2023, pp. 344–355.