ML Sys Project Report: Enhancing Barenet for CUDA-based GPT2 Inference

Xu Zhang[1], **Amy Kim**[1]

[1]Courant Institute, New York University
{xz4863, jk8901}@nyu.edu

Abstract

Modern deep learning frameworks such as Pytorch and Tensorflow play an important role in the development of machine learning systems. However, the implementation details of many crucial operations are hidden from the framework users. To gain further understanding of how autoregressive LLM inference works at the GPU level, we implement a GPT2 inference model by extending 2D CUDA tensor implementation from our previous labs to 4D and implementing advanced kernels required by GPT2 such as permutation and layernorm. Our model outputs text generation results that are mathematically equivalent to that of a Pytorch implementation without using any deep learning framework such as Pytorch and reveals great insight into how GPT2 inference is handled on the GPU level.

1 Introduction

Widely used deep learning frameworks provide highly optimized and easy-to-use abstractions for building neural networks. While this usability has lowered the barrier to deploying complex models, it also makes much of the underlying computations a "black box" for users. Even seemingly simple operators such as matrix multiplication, layer normalization, or softmax rely on specialized kernel implementations and memory layouts that are usually hidden.

In our course, earlier lab assignments focused on uncovering this abstraction by building a minimal deep learning framework, barenet, and implementing basic CUDA kernels from scratch. By the end of these labs, the framework supported several fundamental operators, basic gradient computation, and simple models such as MNIST classifiers. However, these implementations were insufficient for large language models, which rely heavily on higher-dimensional tensor representations and more advanced operators.

This project addresses this gap by extending the barenet framework to support GPT-2 inference without relying on any external deep learning libraries such as PyTorch. Our contributions include generalizing barenet's tensor abstraction from 2D to generic 4D tensors, preserving backward compatibility with existing 2D code, and implementing key operators required for transformer-based models. Using these extensions, we successfully re-implemented GPT-2 inference end-to-end and validated both correctness and performance against PyTorch's implementation.

2 Related Works

Language Models are Unsupervised Multitask Learners by Radford et al. [1] introduced GPT-2 and its decoder-only transformer architecture, which forms the basis of our work. The paper defines the multi-head self-attention, feed-forward layers, layer normalization, and autoregressive decoding procedure that our system re-implements. Rather than modifying the model architecture, our project focuses on reproducing GPT-2 inference using custom CUDA kernels.

PyTorch, introduced by Paszke et al. [2], is a popular framework used to deploy transformer models and provides highly optimized GPU kernels built on CUDA libraries such as cuBLAS and cuDNN,

along with kernel fusion and memory optimizations. In our experiments, PyTorch serves as the baseline implementation for both numerical correctness and performance.

We also draw on open-source reimplementations of GPT-2, such as the PyTorch GPT-2 codebase by Jung et al. [3], which provides a concise and readable reference for the model's forward pass and tensor transformations. This implementation informed our understanding of layer composition and weight layout and guided the process of importing pretrained parameters into our framework.

Finally, FlashAttention by Dao et al. [4] demonstrates that transformer inference performance can be significantly improved through specialized GPU kernels that reduce memory access and fuse multiple operations. Although our work does not target such optimizations, FlashAttention highlights the importance of kernel design in achieving high performance. The performance gap between our implementation and PyTorch further emphasizes the role of these optimizations and motivates future extensions such as kernel fusion and memory-efficient attention.

3 Methods

Our method follows a modular approach to develop our pipeline. We split the pipeline into independent parts: tensor generalization, kernel implementations, GPT2 layers, weight import, overall GPT2 forward pass, and experiments. This helps the debugging of each individual part.

Our main contributions mainly lie in enhanced tensor implementation, and advanced kernels.

3.1 4D Tensors

The Bten tensor class from our course lab is 2D and is not applicable to large-scale deep learning tasks that require training in batch. We generalize the CUDA tensor class into 4D with dimension b, d, h, and w. In particular, our 4D tensor implementation features backward compatibility of all the code from our previous lab and an efficient 3D tensor representation.

Backward Compatibility

There were many legacy codes from previous lab that assume the used tensor to be 2D. To make our 4D tensor compatible to those code, we hide the details about 4D implementation for 2D tensor users. In particular, we keep the attributes h and w the same as 2D implementation. For 4D tensor, we assign two new variables $true\_h$ and $true\_w$. When operating with 4D tensor, we assign memory and manipulate the tensor by consider $true\_h$ and $true\_w$ as the last two dimension of the 4D tensor (the height and width dimension). When we work with 2D tensor code, we adapt a flattened view of 4D tensors: the memory is assigned as $height = b * d * true\_h$ and $width = w$. In this way, 2D tensors are effectively treated as 4D tensor with $b = d = 1$ and the code is backward compatible with all the previous code.

Easy 3D Tensor

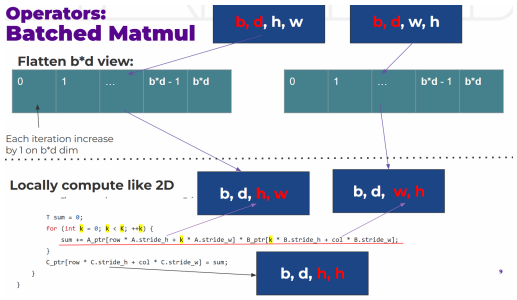
During the inference process of GPT2, 3D tensors are used as the intermediate results. 3D tensors utilities such as to_numpy need special treatment since numpy and various other frameworks manipulate 3D tensors differently (for example shape of numpy array returns different result for 3D tensor and 4D tensor of b = 1). To address this, we give an extra attribute is_3d to 4D tensor object, so that whenever the operators need special treatment for 3d tensors, the operators can simply check the boolean variable of is_3d. Our easy 3D tensor implementation saved massive effort of implementing an individual 3D tensor class and make our framework much concise.

3.2 Advanced Kernels

With our 4D tensor class, we were able implement advanced kernels that are required by GPT2 inference but not available in our previous labs.

Matrix-Multiply in Batch (BMM)

As the figure shows, our BMM kernel (op_bmm) enables matrix multiplication on 4D tensors [Batch, Sequence_len, Head, Dim] (i.e. b,d,h,w) without physical reshaping. The kernel calculates a flattened



(a) Workflow of BMM

batch index batch_idx from the GPU grid's z-dimension (blockIdx.z). By utilizing the 4D stride metadata stored in the tensor class, the kernel computes precise memory offsets (batch_idx * stride_b) to locate the start of each individual matrix in the batch stack. This allows us to perform $Q \cdot K^T$ and $W \cdot V$ operations by launching a single kernel that iterates over all heads and batches in parallel, avoiding the need to physically copy or reorder memory between operations.

Layer Normalization

We implemented a custom LayerNorm kernel (op_layernorm) that mirrors GPT-2's specific normalization logic (using biased variance). The kernel operates row-wise: each thread block processes a single row (token vector). It first performs a reduction loop to compute the mean, followed by a second pass to compute the variance $\sigma^2 = \frac{1}{N}\sum(x_i - \mu)^2$. Finally, it normalizes each element using the inverse square root rsqrtf(var + eps) and applies the affine transformation. To ensure numerical stability and compiler compatibility across float/int templates, we enforce float precision for the rsqrt calculation.

Permutation

We implemented two specialized CUDA kernels to handle the dimension permutations required by Multi-Head Attention: permute_0213 (for transforming linear outputs into head-separated format and merging heads) and permute_0231 (for preparing the Key matrix by transposing its last two dimensions). Each thread decodes its linear index into 4D output coordinates (b, d, h, w) based on the target shape, and then maps these coordinates back to the source tensor's memory offset using the input tensor's strides. This "scatter-gather" approach allows for arbitrary dimension swapping without intermediate buffers or complex shared memory tiling, directly leveraging the stride metadata stored in our 4D tensor class to handle non-contiguous reads efficiently.

4 Experiments

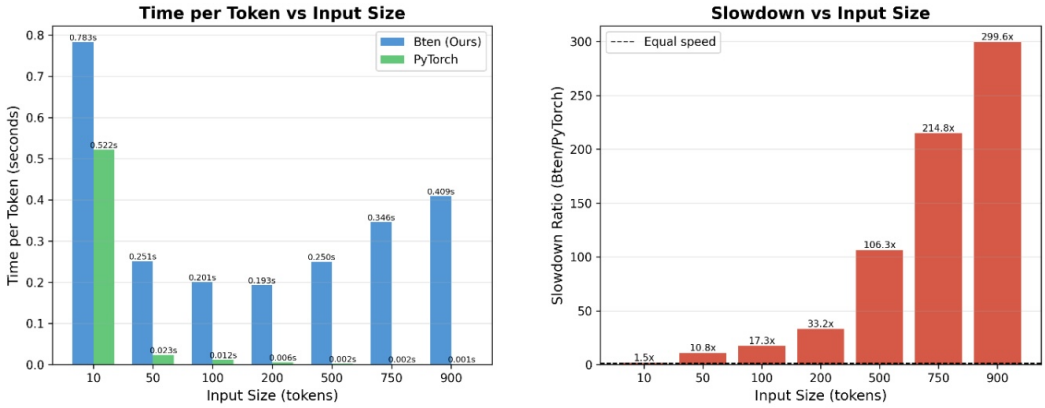
We evaluated our implementation through different experiments to validate the correctness and performance relative to that of PyTorch's GPT2. All experiments used the same GPT-2 configuration and pretrained weights for both implementations. Model weights were imported from PyTorch into barenet by matching tensor layouts, and to ensure deterministic behavior, we used top-k sampling with k=1, which restricts token selection to the single most probable token.

4.1 Implementation Details

We compare our implementation with the reference Pytorch GPT2 implementation from [3]. We import the same pretrained weight by converting the weight into numpy and then into bten tensors. We did not implement KV-cache. Each time the context is updated as prev, we fill the whole context variable prev into the forward pass.

4.2 Module-wise output comparison

To verify numerical correctness, we performed module-level comparisons between barenet and PyTorch outputs. Specifically, we compared the outputs of individual components: multi-head



(a) Time per token vs. input length

(b) Slowdown factor relative to PyTorch GPT-2

Figure 2: Inference Speed performance of Bten GPT-2 compared to PyTorch GPT-2.

self-attention, feed-forward (MLP) layers, layer normalization, full transformer blocks, and the complete GPT-2 model. For each module, outputs were matched element-wise against PyTorch results, confirming that our custom CUDA kernels and tensor abstractions produced mathematically equivalent results.

4.3 Text Generation Validation

Beyond intermediate activations, we tested end-to-end behavior by generating text from identical prompts using both implementations. Given the same input prompt, our barenet GPT-2 implementation produced exactly identical token sequences to PyTorch GPT-2 across all tested cases. This confirms that not only individual operators but also the full autoregressive generation pipeline behaves correctly.

4.4 Speed Performance Evaluation

For our final evaluation, we measured inference speed by recording time per generated token as a function of input sequence length. As shown in Figure 2, while barenet GPT-2 is functionally correct, it is slower than PyTorch's highly optimized implementation, particularly for longer input sequences. While the time per token decreases at first due to GPU utilization, this performance gap grows with sequence length, reflecting the absence of advanced optimizations such as kernel fusion and other kernel optimization techniques. These results highlight both the correctness of our implementation and the importance of kernel optimizations in deep learning frameworks.

5 Conclusion

In this project, we successfully extended the barenet framework to support inference of a modern autoregressive language model, GPT-2, using only custom CUDA kernels and without relying on any deep learning libraries. By generalizing tensor representations to 4D while maintaining backward compatibility, and by implementing the necessary transformer operators, we demonstrated that barenet can reproduce PyTorch GPT-2 outputs at both the module and sequence generation levels.

Although our implementation is slower than PyTorch's optimized kernels, this was an expected outcome and reinforces the value of optimizations such as fused kernels and memory-efficient attention mechanisms. Future work could extend this system by adding autograd support to enable training and fine-tuning, incorporating KV caching, and introducing kernel fusion to improve performance, as well as exploring more advanced sampling strategies.

Overall, this work demonstrates that the core components of large language model inference can be re-implemented using custom CUDA kernels and fundamental tensor operations, providing an end-to-end view of how GPT-2 inference executes at the operator and kernel level on GPUs.

References

[1] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019) Language Models are Unsupervised Multitask Learners. *OpenAI Technical Report*.

[2] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., & Chintala, S. (2019) PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*.

[3] Jung, T.H. (2019) GPT-2 PyTorch Implementation. Available at *https://github.com/graykode/gpt-2-Pytorch*.

[4] Dao, T., Fu, D.Y., Ermon, S., Rudra, A., & Ré, C. (2022) FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems*.