Beating PyTorch: Optimizing GPT-2 Small Model Inference through Custom CUDA Kernel on NVIDIA T4 GPUs

Shubham Ojha (so 2754) Mathew Martin (mm14460)

Summary of the slides :-

The standard PyTorch implementation of the **GPT-2 124M** model is not optimized for high-performance inference. During inference, flexibility is far less important than raw speed, but PyTorch eager-mode execution introduces overhead that limits throughput. To improve performance on specific GPU hardware, practitioners often implement **custom CUDA kernels** for critical operations, enabling hardware-aware optimizations.

In our case, we focus on **NVIDIA T4 GPUs**, which are widely available on platforms like Google Colab and Kaggle, especially among beginners experimenting with LLMs. However, the T4 is an older GPU with significantly lower memory bandwidth compared to modern accelerators such as the A100 or H100. Popular kernels like **Flash Attention** were designed with newer architectures in mind and do not fully exploit T4-specific constraints or capabilities.

Because GPT-2 124M is the most commonly used model by newcomers and is frequently deployed on T4 GPUs, there is a strong need for a **T4-optimized inference solution** that maximizes performance while remaining accessible. Our goal is to develop such an optimized implementation by designing custom kernels tailored to the T4's architecture, enabling faster and more efficient GPT-2 inference for the broader community.

Technical challenges

1. Writing fused kernels is easier, but to write performant fused kernels in cuda is difficult.

2. Limited bandwidth capacity compared to A100/H100 GPUs

3. No implementation of Flash Attention for T4GPUs

Approach

To address the performance limitations of the standard GPT-2 124M PyTorch implementation on T4 GPUs, we followed a systematic, profiling-driven optimization workflow. Our goal was to identify the true bottlenecks—compute, memory, or kernel-launch overhead—and design T4-specific optimizations using custom kernels and precision tuning.

1. Profiling the Baseline GPT-2 124M Implementation

We began by profiling the naïve PyTorch inference pipeline. The profiling was performed in a **hierarchical manner**:

(a) High-Level Profiling

At the module level, we measured end-to-end timings of each component (attention blocks, MLP layers, layer norms, embeddings, etc.). This helped us identify **which layers dominated latency during the forward pass**.

(b) Kernel-Level Profiling

For the most time-consuming modules, we performed a deeper kernel-level analysis. Here, we examined:

- which CUDA kernels were launched
- how many kernels were launched per layer
- whether kernels were compute-bound or memory-bound
- the presence of excessive, small-granularity kernel launches (kernel-launch overhead)

This dual-level profiling gave us a clear picture of where optimizations would have maximum impact.

2. Classifying GPT-2 Operations

We categorized all operations used in the GPT-2 computation graph into three fundamental types:

1. **GEMM operations** (matrix multiplications – dominant in attention and MLP)
2. **BLAS-style reductions** (softmax, layer norm, scaling, etc.)
3. **Elementwise operations** (bias add, activation functions, residual connections)

This classification allowed us to systematically evaluate which type of operation was hitting which bottleneck—compute, memory, or kernel overhead.

3. Understanding T4 GPU Architectural Constraints

We analyzed key T4 hardware characteristics to guide optimization choices, including:

- **Memory bandwidth:** ~320 GB/s (much lower than A100/H100)
- **Shared memory size and SM count (40 SMs with 64kb L1/shared memory)**
- **Supported Tensor Core data types:** FP16, BF16, INT8

This understanding was essential because many modern kernels (e.g., Flash Attention) are optimized for Ampere/Hopper GPUs and do not fully leverage T4 architecture.

4. Optimization Strategies

Based on insights from profiling and hardware analysis, we applied the following targeted optimizations:

(a) Improving Compute-Bound Operations

- We migrated key GEMM operations to **Tensor Cores** using FP16/BF16 formats.

- This significantly increased throughput, especially for MLP and QKV projections.

(b) Reducing Kernel-Launch Overhead

- We performed **kernel fusion** using CUTLASS to replace multiple small kernels with fewer, larger fused kernels.

- This reduced launch overhead and improved kernel occupancy on T4.

(c) Optimizing Memory-Bound Operations

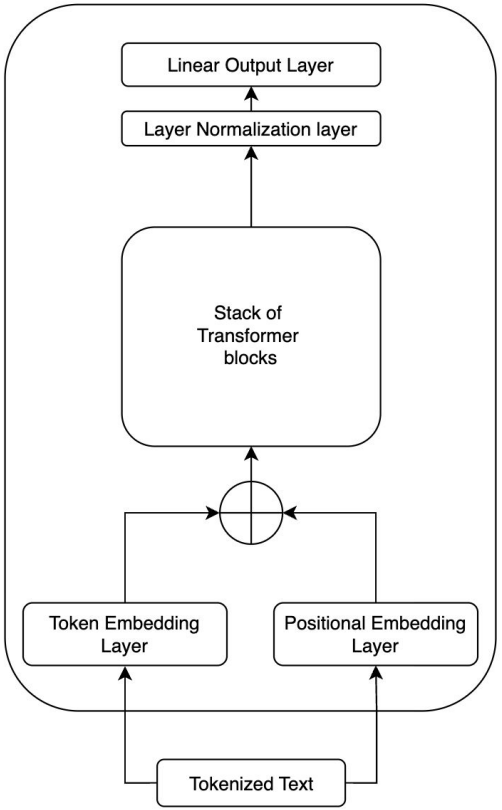
- We lowered precision (FP16/BF16) to reduce memory traffic.

- We ensured data reuse by structuring computations to keep operands in **shared memory or registers**, reducing global memory access.

- We fused memory-bound elementwise operations to minimize redundant reads/writes.
- Together, these optimizations directly addressed the three core latency contributors: **compute bottlenecks, memory bandwidth limitations, and kernel-launch overhead**.

Experimental evaluation

GPT2_Module

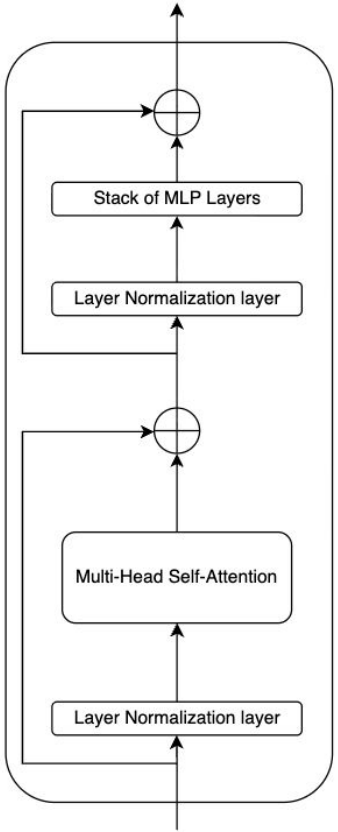

Linear Output Layer
Layer Normalization layer
Stack of Transformer blocks
Token Embedding Layer
Positional Embedding Layer
Tokenized Text

High level Profiling results

GPT2_Module :-

Layers	**Breakdown %**
Stack of Transformer blocks	89.8%
Linear Output Layer	6.9%
Layer Normalization Layer	1.1 %
Token embedding layer	0.5%
Additional OP [Token embedding + Positional Embedding layer]	0.4%
Positional Embedding layer	0.3%

Transformer Block

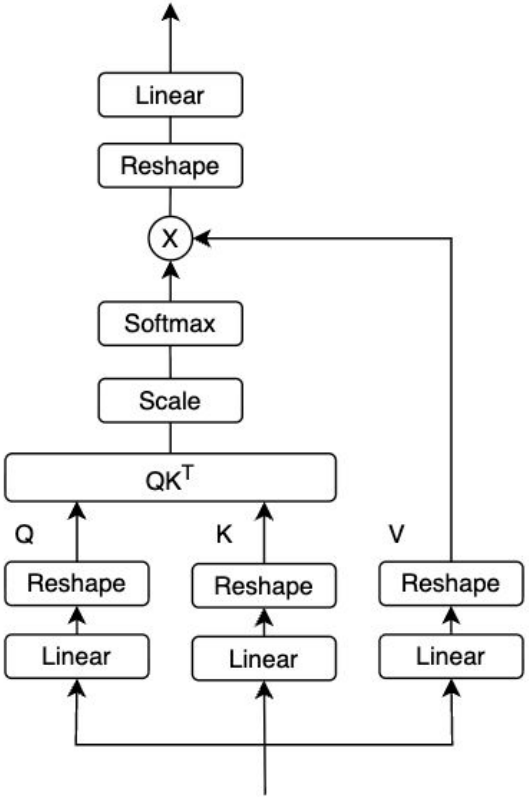


High level Profiling results

Transformer Block

Layers	Breakdown %
Multi-head self attention layer	37.6%
Stack of MLP layers	25.6%
Layer normalization layer 1	11.3%
Layer normalization layer 2	11.2%
Residual connection 1	2.7%
Residual connection 2	2.6%

Multi-head Self-Attention

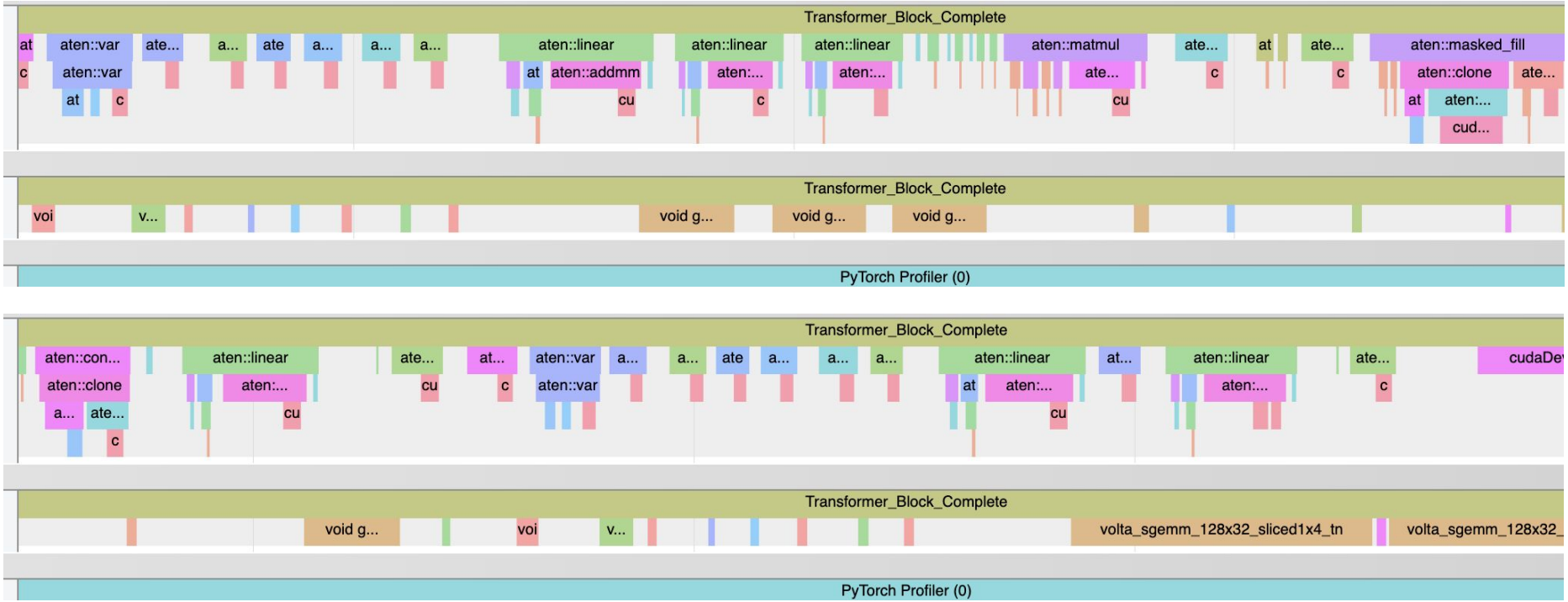


High level Profiling results

Multi-head Self-Attention

Layers	Breakdown %
Attn_QKV_Projection	22.6%
Attn_Mask_Apply	13.6%
Attn_Scores_Compute	12.5%
Attn_Output_Projection	11.3%
Attn_Values_Apply	8.5%
Attn_Output_Reshape	6.5%
Attn_QKV_Reshape	4.2%
Attn_Softmax	3.8%

Kernel level Profiling results (Multi-head Self-Attention)



Kernel level Profiling results (Multi-head Self-Attention)

QKV Projection (Q, K, V Linear Layers) → gemmSN_TN_kernel

Attention Score Computation → gemmSN_TN_kernel(BMM), vectorized_elementwise_kernel (sqrt, reciprocal)

Causal Mask & Softmax → softmax_warp_forward, masked_fill_

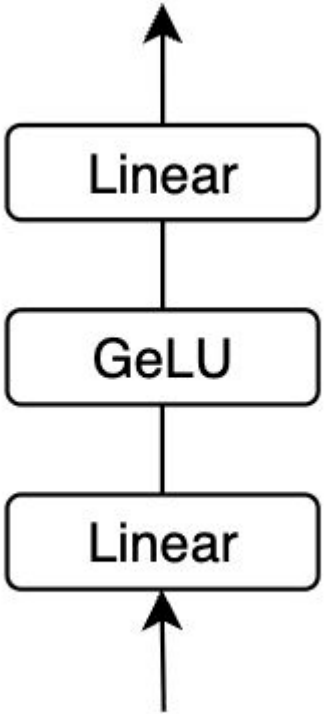
Weighted Sum (Attention @ Values) → gemmSN_NN_kernel

Output Projection → gemmSN_TN_kernel

Following optimization recommend:-

1) Switch from FP32 to FP16(in order to use Tensor cores of the GPUs)
2) Fused Attention Kernels → QKV Projection + Score Computation + Softmax + Weighted Sum (as a lot of small kernels are being launched)

Stack of MLP layers

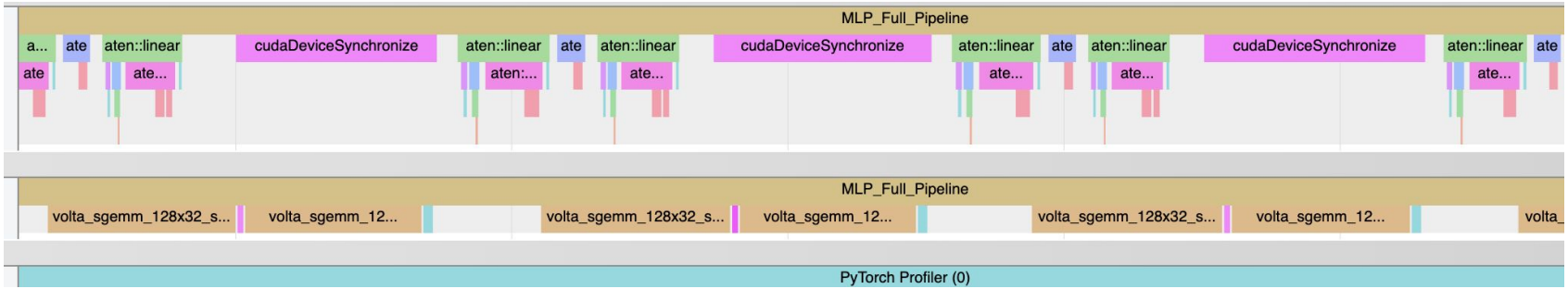


High level Profiling results

Stack of MLP layers

Layers	Breakdown %
MLP_Activation	31.4%
MLP_Projection	29.9%
MLP_Expansion	29.2%

Kernel level Profiling results (Stack of MLP layers)



Following 3 kernels were launched :-

1) volta_sgemm_128x32_sliced1x4_tn(for MLP_Projection and MLP_Expansion)
2) cublas Lt::splitKreduce_kernel (for MLP_Projection)
3) void at::native::vectorized_elementwise_kernel (MLP_Activation GeLU)

Following optimization recommend:-

Since GPU line is maximally dominated by volta_sgemm_128x32_sliced1x4_tn kernel, which is GEMM op, we can optimise it to get max performance. So in order to do so we can try following:

A. Switch dtype form Fp32 to Fp16(which uses Tensor cores of T4GPUs) **[High impact]**
B. Fuse Kernel for 3ops **[Low Impact]**

FlashAttention on T4

- Officially no support for Flash Attention in T4 GPUs
- Targeted towards A100/H100 GPUs
- T4 (Turing (SM75)) lacks lacks Ampere/Hopper tensor core instructions (LDMatrix, MMA) required by Flash Attention
- Smaller shared memory per SM
- Lower HBM memory bandwidth (320 GB/s vs 1.5 TB/s on A100) limits FA's I/O-aware throughput gains
- Official Flash Attention library fails to compile for compute capability 7.5

Conclusion

- From the profiling of a naïve GPT-2 124M PyTorch implementation, we observed several inefficiencies during inference. To improve inference performance, we need systematic profiling to identify bottlenecks at both the model and kernel levels.
- Before applying optimizations, it is essential to understand the underlying GPU architecture—such as the number of CUDA cores, available Tensor Cores, memory hierarchy, and bandwidth characteristics. This architectural knowledge helps us reason about which operations are compute-bound or memory-bound.
- Finally, writing custom CUDA kernels (or fusing existing ones) provides the greatest control over execution and allows us to extract maximum performance during inference.

Link to GitHub repo

Link to github repo :- https://github.com/shubhamOjha1000/HPML_final_project.git