Parallelizing GPT-2 -"It works on my GPU"

CSCI 563

Siggy Sigler, Sam Abderholden, Dawson Matthews

Colorado School of Mines

May 5, 2026

Introduction

- We focus on the GPT-2 transformer architecture proposed in *Language Models are Unsupervised Multitask Learners*
  - decoder only, each layer is self-attention + MLP
  - causal self attention, only attend to the previous tokens in a sequence
  - autoregressive, $p(x_t \mid x_1, \ldots, x_{t-1})$
- *microgpt* is a 200 line, dependency free, Python implementation of GPT-2 created by Andrej Karpathy
  - trained on a corpus names in order to produce new names at inference time
  - follows a similar architecture as GPT-2 with far less parameters (4,192 vs 1.6 billion)
  - *microgpt* uses RMSNorm and ReLU instead of LayerNorm and GELU used in GPT-2
- We create a C++ serial version matching *microgpt*, in order to provide a better comparison for our parallel version
- Parallelizing ONLY the forward pass of MicroGPT

Serial Implementation

`name_list: [[BOS, E, L, L, A, BOS], [name_B], [name_C]]`

`BOS = Beginning of Sequence`

Serial

```
for name in name_list:
  for token in name:
    embedding(token)
    for layer in layers:
      attention(token)
      mlp(token)
    calc_loss(token)
sum_loss(tokens)
```

Serial Analysis

Big-O:

- T = sequence length, E = embedding size, L = layer, N = names
- $O(N \cdot L \cdot (C_{\text{attn}} + C_{\text{mlp}}))$
- $O(C_{\text{attn}}) = T^2 \cdot E$
- $O(C_{\text{mlp}}) = T \cdot E^2$
- Since names are mostly constant size it becomes $\approx O(N \cdot L \cdot E^2)$

Memory vs CPU Bound:

- Total Time: 6.41 s, CPU Time: 6.19
- CPU utilization $\approx \frac{6.19}{6.41} \approx 96.6\% \Rightarrow$ CPU-bound

Static Analysis:

- **99.4%** of work spent doing matrix mult, (0.4% for rmnsnorm, SM, emb)
- **71.1%** for MLP **28.3%** for ATTN
- By Amdahl's Law, with $p = 0.998$, the theoretical maximum speedup is $\frac{1}{1-p} \approx 500\times$

Serial vs. Parallel Implementation

`name_list: [[BOS, E, L, L, A, BOS], [name_B], [name_C]]`

`BOS = Beginning of Sequence`

Serial

```
for name in name_list:
    for token in name:
        embedding(token)
        for layer in layers:
            attention(token)
            mlp(token)
        calc_loss(token)
sum_loss(tokens)
```

Parallel (per batch)

```
calculate_embeddings(name_list)

for layer in layers:
    calculate_attention_scores(
    name_list)
    calculate_next_hidden_state(
    name_list)

calc_loss(name_list)
```

Parallelization Strategy

- **Batching**
  - For each sequence launch a transformer $\rightarrow$ run transformer on all sequences at once
- Store all of our information in a flattened array (B, T, d)
  - Allows assignment of work easily by batch to threads
- Store a "workspace" on the GPU that holds weights, hidden states, etc.
- Large portions of the forward pass are parallelizable
  - **Embedding calculations**
  - **RMS normalization and linear calculations**
  - **Attention**
- **Layers must be performed sequentially**

Performance Evaluation (Forward Pass)

	PyTorch Baseline	**C++ Serial**	**CUDA Parallel**
Names: 2500 Layers: 4 Embeddings: 256 Attn. Heads: 8	5.2244 seconds (8.919x)	46.5978 seconds	0.1949 seconds (239.0856x)
Names: 5000 Layers: 6 Embeddings: 384 Attn. Heads: 12	14.5409 seconds (22.589x)	328.4692 seconds	0.7374 seconds (445.44x)
Names: 10,000 Layers: 8 Embeddings: 512 Attn. Heads: 16	38.6489 seconds (40.946x)	1582.5279 seconds	2.9164 seconds (542.63x)