CS 229S Project 1

October 25, 2023

By the end of this project, you'll have real experience making transformer inference go brr. Specifically, we're taking Karpathy's wonderful nanoGPT implementation,[1] and improving GPT2 throughput by 10x and latency by 2x! But first, we'll do some theory.

At the end, you'll need to submit the following files:

- model.py to the code section of Project-1 on Gradescope
- Pdf file with your written responses to the written part of Project-1 on Gradescope.

The assignment should be completable with Google Colab. We understand that some students may require financial assistance for Colab. Please reach out to the teaching staff if this is a major challenge for you and we will try our best to resolve.

1 Preliminaries

We will study two main efficient inference techniques in this assignment:

1. KV caching
2. Speculative decoding

In this section, we will recap the preliminaries and resources for each of these two techniques.

1.1 KV Caching

Recall that in KV caching, we store the keys and values from previously processed tokens so that we do not need to recompute them for every next-token generation. We have discussed this technique a few times in class.

Resources You can reference Lectures 2-4 (including discussion of Transformers, KV caching, and FLOPs computations), the assignment 1 solutions (released after late days for all students expire), and this additional resource on KV caching `https://kipp.ly/transformer-inference-arithmetic/`.

1.2 Speculative decoding

Speculative decoding is another cutting edge efficient inference technique. We briefly discussed this technique in Lecture 4 (see the last few slides) and point out this resource `https://jaykmody.com/blog/speculative-sampling/`. In addition to this, we provide the preliminaries here:

Setup We have a larger (and thus slower, but often generally higher quality) model M_l and a smaller (and thus faster, but also generally lower quality) model M_s. The overall objective is: suppose we have tokens $\{x_1, ..., x_k\}$ and want to generate the next tokens $\{x_{k+1}, ..., x_n\}$ with M_l. Suppose M_l takes t_l time to generate these tokens. The objective of speculative decoding is: can we leverage the smaller, faster M_s to perform the generation of the $n-(k+1)$ tokens in $t_s < t_l$.

[1] `https://github.com/karpathy/nanoGPT`

Algorithm The vanilla speculative decoding algorithm is as follows. Suppose we want our **final sequence** to be of length n. While the number of total tokens is less than n (since we want to go up until x_n), repeat the following steps:

1. Use M_s to generate m tokens (e.g. $\{x_{k+1}, ..., x_{k+m}\}$), conditioned on the prior tokens P (e.g. $\{x_1, ..., x_k\}$ initially).
2. Construct m unique prefixes, by plugging in the predictions from M_s:

$$\text{prefix}_1 = \{x_1, ..., x_k\}$$

$$\text{prefix}_2 = \{x_1, ..., x_{k+1}\}$$

$$...$$

$$\text{prefix}_m = \{x_1, ..., x_{k+m-1}\}$$

3. Use M_l to perform generation for the single next token, given each of these prefixes, in parallel (i.e. we can *batch* the prefixes). Now we have $S = \{x_{k+1}, ..., x_{k+m}\}$ as the predictions from M_s and $L = \{x_{k+1}, ..., x_{k+m}\}$ as the predictions from M_l.
4. Iterate through the two lists S and L. While the items $x_i \in S$ and $x_i \in L$ match, keep going and add x_i to the **final sequence**. If the length of the **final sequence** is now n, we can stop. Upon a mismatch, throw away the remaining tokens in each of the lists. Return to step 1, using the current **final sequence** as the new P.

Analysis Note that every token in the **final sequence** is what M_l would exactly generate if we were to use standard decoding / KV caching. Note that in the worst case if M_s gets every generation incorrect, we degrade to KV caching. This is because using prefix_1 above and plugging this into M_l, one loop of the above algorithm at least yields a keepable x_{k+1} next-token prediction.

M_s speeds up the generation process more if M_s is increasingly faster than M_l at predicting the next token **and** M_s tends to make several predictions that match M_l (i.e. quality is not terrible). We also are batching the prefixes in step 2 when we pass them to M_l, so the batch needs to fit within our system's memory (we won't grapple with this piece for our assignment).

2 Inference Math (50 pts)

We're going to build on our understanding of Transformers from the last assignment to better understand when and where certain optimizations make sense.

2.1 Naive GPT2-XL Inference (17 pts)

Problem specifications GPT-2-XL has the following architectural details:

- $n\_vocab = 50257$
- $n\_embed = 1600$
- $n\_heads = 25$
- $n\_layers = 48$
- $mlp\_hidden\_dim = 6400$

Some additional notes for this problem:

- We'll use B as the batch size and N as the sequence length.
- We're going to ignore the embedding, LayerNorms, skip connections, and language modeling head as they're all relatively inconsequential compared to the core transformer loop.
- Assume all computations are done with 16-bit precision.
- We are focused on inference, so the forward pass.

2.1.1 MHA (4 pts)

How many FLOPs and how much memory bandwidth is required to run MHA on a $B \times N$ batch of tokens? (Use your results from last time, but don't forget about the final output projection!)

Hint: your answer will take the form:
$FLOPS = C_1 \times BN + C_2 \times BN^2$
$MEM = C_3 + C_4 \times BN + C_5 \times BN^2 + C_6 \times N^2$
Answer:

2.1.2 MLP (3 pts)

The typical MLP in a transformer includes a first projection from the $N \times n\_embed$ input X to a value that is $N \times 4(n\_embed)$, by multiplying the input with a $n\_embed \times 4(n\_embed)$ matrix. Next, there is a second projection from the $N \times 4(n\_embed)$ matrix to $N \times n\_embed$, by multiplying the with a $4(n\_embed) \times d$ matrix. How many FLOPs and how much memory bandwidth is required to run an MLP on a $B \times N$ batch of tokens?
Answer:

2.1.3 GPT2-XL (3 pts)

How many FLOPs and how much memory bandwidth is required to run the full GPT2-XL model?
Answer:

2.1.4 Building Intuition (7 pts)

As with the previous assignment, we'll assume an NVIDIA T4 GPU with 300 GB/s of bandwidth and 65 TFLOPs of compute. Let's look at a few scenarios to try to understand inference bottlenecks in practice:

In each case: how much compute and bandwidth are required for a forward pass? Is the GPU memory bound or compute bound?
Answer:

2.2 Decoding GPT2-XL with KV caching! (17 pts)

We're now going to assume that we've already filled our KV cache with $B \times N$ tokens, and we want to decode B new tokens in parallel using that KV cache. Let's do the same math and see what changes!

2.2.1 Warmup (1 pt)

How big is the KV cache at each layer? Assume it's also stored in 16-bit precision.
Answer:

2.2.2 MHA (3 pts)

How many FLOPs and how much memory bandwidth is required to run MHA to decode this batch of B tokens? *Hint: there's no masking required, and no N^2 terms anymore!*
Answer:

2.2.3 MLP (3 pts)

How many FLOPs and how much memory bandwidth is required to run the MLP on this batch of B tokens?
Answer:

2.2.4 GPT2-XL (3 pts)

How many FLOPs and how much memory bandwidth is required to decode this batch of B tokens through the full GPT2-XL model?

Answer:

2.2.5 Comparing with Naive Inference (7 pts)

Let's look at the same cases we examined before (N-1 since this is now the size of the KV cache).

- B=1 and N=31
- B=1 and N=1023
- B=64 and N=31
- B=64 and N=1023

How much compute and bandwidth are now required for a forward pass with the KV cache in place? Are we memory or compute bound? And (roughly) how much faster would we expect case each to run on the T4 GPU?

Answer:

2.3 Speculative Decoding (16 pts)

Problem specifications We're now going to turn our attention to the $B = 1$ case, where we can really focus on reducing latency. To do this, we'll be speculatively decoding GPT2-XL with GPT2, which has the following architectural details. Again, as with the previous assignment, we'll assume an NVIDIA T4 GPU with 300 GB/s of bandwidth and 65 TFLOPs of compute.

- $n\_vocab = 50257$
- $n\_embed = 768$
- $n\_heads = 12$
- $n\_layers = 12$
- $mlp\_hidden\_dim = 3072$

2.3.1 Shortcut to $B = 1$ Inference Latency – GPT2 (4 pts)

Take a look back at your result for decoding the $B = 1$ case with KV caching enabled. What's (by far) the dominant factor in the latency of decoding a batch? Use that to derive a constant approximation for the cost of decoding a token for GPT2 on a T4 GPU. Also, in this case don't neglect the 768×50257 language modeling head, as it does matter quite a bit for the smaller GPT2.

Your answer should be in microseconds.

Answer:

2.3.2 Shortcut to $B = 1$ Inference Latency – GPT2-XL (4 pts)

What does this same approximation give you for decoding the latency of decoding a single batch with GPT2-XL? What's the relative cost of decoding the two models?

Your answer should be in milliseconds.

Answer:

2.3.3 Optimizing Tokens (8 pts)

Let's assume that GPT2-XL and GPT2 agree on their predictions with greedy decoding 75% of the time. How many tokens should we decode at once with speculative decoding for the optimal speedup? And how would that change if we sampled with temperature $T > 0$?

Answer:

3 Implementation (50 pts)

In this section, you'll take Karpathy's nanoGPT implementation and improve its GPT2 inference throughput by 10x with KV caching and its inference latency by 2x with speculative decoding. The teaching staff has added some scaffolding to help guide your implementation, but the code is essentially as Karpathy wrote it.

All of the changes you'll need to make are in model.py.

3.0.1 KV Caching KV Caching (30 pts)

For this section, you'll implement a KV cache for GPT2. There are five sections to fill in:

- Register a buffer for the KV cache with the right dimensions. You can see examples of register_buffers in the "model.py" file.
- Add prefilling the KV cache to the forward pass of CausalSelfAttention.
- Write the decode CausalSelfAttention implementation, which uses the KV cache.
- Write the decode implementation for the main GPT model class. (This one's easy.)
- Write generate_kv to generate many sequential tokens using KV cache based decoding.

We've had some trouble getting gpt2-xl to run on the Colab T4 instances, so gpt2-large will be fine for this assignment. A good implementation should increase throughput on the provided `run.sh` test by $> 10\times$. This can be run on colab using the provided `inference.ipynb` notebook.

3.0.2 Speculative Decoding (20 pts)

For this section, you'll implement speculative decoding for GPT2. For this, all you need to do is write generate_speculative in model.py. A good implementation should reduce latencies on the provided run.sh test by around $2\times$. Good luck!

3.0.3 Tips

Some tips to guide your implementation:

- Get everything working on gpt2 (small) first – it will make your development cycle much faster! Only once everything is working should you run on the bigger models.
- Don't worry about the exact match passing as long as you're getting good outputs – enforcing numerically identical results is actually extremely difficult due to how CuBLAS is implemented by NVIDIA. But, you can set DEBUG=True in model.py, and that will make things significantly *more* deterministic than they would otherwise be, at the cost of speed. You may need to run

  ```
  export CUBLAS_WORKSPACE_CONFIG=:4096:8
  ```

  in order to get debug mode to work. Note that debug True hurts speed because it disables the use of tensor cores and uses FP32 instead of BF16.
- Don't bother with trying to get your speculative decoding to use KV caching. It's unlikely to have a significant effect unless you want to run extremely long prompts on weak GPUs.
- Speculative decoding may look simpler than the KV cache but can be deceptively tricky. We'd recommend simulating it by hand on paper, and then using that to guide your implementation.