Scaling Up Part III: Attention

We learned how to accelerate MLP computation using MoE

- MLP is responsive for knowledge, and knowledge is “very sparse”
- Only a tiny fraction of human knowledge is needed to solve each problem
  - But we need to know the correct “fraction” – Using a Router.

Speeding Up Attention Computation

- We will learn a few tricks to speed up the computation
  - Flash Attention
  - Multi-Query Attention
  - Paged Attention

- The most fundamental layer in the transformer: Multi-head attention (m attention heads).
- Given vectors $v_1, v_2, \ldots, v_n$, each in R^d, a multi-head attention layer is defined as:
- $v_i' = C \times concatenate\left(V_r^T \ \sum_j \alpha_{i,j}^r \, v_j\right)_{r \in [d/m]} + b$
- Where $\left(\alpha_{i,j}^r\right)_{j \in [n]} = \ softmax\left(v_i^T Q_r K_r^T v_j + p_{i,j}^r\right)_{j \in [n]}$
- Here, C is a $d \times d$ trainable matrix.
- Each v_i looks for the “most similar v_j , according to [d/m] many projection matrices Q_r and K_r.

Transformer Architecture

- A (post-layernorm) transformer block is defined as:
- Given input $W \;\; = v_1, v_2, \dots, v_n$, each v_i in R^d.
  - (1). Apply Multi-Head Attention (input dimension d, output dimension d) on W to get $V^{(1)} = v_1^{(1)}, v_2^{(1)}, \dots, v_n^{(1)}$.
  - (2). Apply layer-norm on each of the $v_i^{(1)}$ to get $v_i^{(2)}$.
  - (3). Apply residual link: $v_i^{(3)} = v_i^{(2)} + v_i$.
  - (4). Apply a one hidden layer MLP h (input dimension d, output dimension d) on each $v_i^{(3)}$ to get $v_i^{(4)} = h(v_i^{(3)})$ (all the v_i''' in the uses the same h per layer, different h for different layers).
  - (5). Apply layer-norm on each of the $v_i^{(4)}$ to get $v_i^{(5)}$.
  - (6). Apply residual link: $v_i^{(6)} = v_i^{(5)} + v_i^{(3)}$.
- The output $V^{(6)} \;\; = v_1^{(6)}, v_2^{(6)}, \dots, v_n^{(6)}$, each $v_i^{(6)}$ in R^d.

Transformer Architecture

- A (pre-layernorm) transformer block is defined as:
- Given input W $= v_1, v_2, \ldots, v_n$, each v_i in R^d.
  - (1). Apply layer-norm on each of the v_i to get $v_i^{(1)}$.
  - (2). Apply Multi-Head Attention on $V^{(1)}$ to get $V^{(2)} = v_1^{(2)}, v_2^{(2)}, \ldots, v_n^{(2)}$.
  - (3). Apply residual link: $v_i^{(3)} = v_i^{(2)} + v_i$.
  - (4). Apply layer-norm on each of the $v_i^{(3)}$ to get $v_i^{(4)}$.
  - (5). Apply a one hidden layer MLP h on each $v_i^{(4)}$ to get $v_i^{(5)} = h(v_i^{(4)})$ (all the v_i''' in the uses the same h per layer, different h for different layers).
  - (6). Apply residual link: $v_i^{(6)} = v_i^{(5)} + v_i^{(3)}$.

Computation Time of Transformer Block

- A transformer block = MHA (m heads) + MLP.
- Assuming the context length is n and the embedding dimension is d.
- Forward/Backward time:
  - $nd^2(mlp) + (nd^2 + n^2d)\ (MHA)$
- (Forward) Backward Memory:
  - $nd(mlp) + (nd + n^2m)\ (MHA)$

Reducing Memory Usage of Attention

- Main Memory Usage:
- For each attention head, we need to store the $n \times n$ attention matrix:
- $\left[softmax\left(v_i^T Q_r K_r^T v_j + p_{i,j}^r\right)_{j \in [n]} \right]_{i \in [n]}$
- Let's just consider one row:
  - $softmax\left(v_i^T Q_r K_r^T v_j + p_{i,j}^r\right)_{j \in [n]}$
- Key idea of Flash-Attention:
  - We store $K_r^T v_j, Q_r^T v_j$ for every r and j, this takes memory $d \times n$.
  - We do not store the full softmax matrix, we will "compute them on the fly" to save memory.

- Consider $O = \sum_{i\in[n]} y_i \times softmax(x)_i$
- Where for each x_i, y_i, we need computation time d/m to retrieve it.
- Stupid-Attention computation:
  - For i in range(n):
    - Compute $norm\_factor = norm\_factor + \exp(x_i)$.
    - Compute $O = O + y_i \exp(x_i)$
  - Return O/norm_factor
- This only requires memory O(M), where M = d/m is the dimension of y_i

From Stupid Attention to Flash Attention

- Why is Stupid Attention Stupid?
- Floating Point accuracy. We can not compute $\sum \exp(x_i)$ accurately! No such accuracy.
- Stupid Attention V2:
  - Go through i, compute the max of x_i as m(x)
  - For i in range(n):
    - Compute $norm\_factor = norm\_factor + \exp(x_i - m(x))$.
    - Compute $O = O + y_i \exp(x_i - m(x))$
  - Return O/norm_factor
- But then we need to compute x_i twice, unless we store it in the memory…

From Stupid Attention V2 to Flash Attention

- Stupid Attention V3 is an upgrade of stupid attention v2, where we only compute x_i once and maintain the correct floating-point accuracy.
- For i in range(n):
  - Compute $m_{new}(x) = \max(m(x), x_i)$
  - Compute $norm = \exp\left(m(x) - m_{new}(x)\right) norm + \exp\left(x_i - m_{new}(x)\right)$.
  - Compute $O = \exp(m(x) - m_{new}(x))O + y_i \exp(x_i - m_{new}(x))$
  - Update $m(x) = m_{new}(x)$
- Output O/norm.

From Stupid Attention V3 to Flash Attention

Now the memory usage is good.

Main problem: For i in range(n).

- Cuda operates on the so-called “Thread Block”, so the computation is very fast for operations of “certain sizes”.

In stupid attention v3, the computation inside for loop is:

- Vector of size M = d/m per i. This is typically smaller than the “certain sizes” when m is large.

So we need to do some chunking…

Flash Attention

- Flash attention is a little bit more involved than the previous slides.
- It divides the computation in chunks of R
- For i in range(n//R):
  - Compute the softmax for x[iR:iR +R] using the fastest way, which uses memory R. Then compute
    - $O_i = \sum_{j\in[iR,iR+R)} y_j \times softmax(x[iR:iR+R])_j$ (only store this O_i in SRAM).
  - Store the max of x[j] for j in [iR, iR + R) in memory as m[i].
  - Store the normalization factor of the softmax (after subtracting the max) of $x[iR:iR+R]$ in memory as norm[i].
  - Update $m_{new}(x) = \max(m(x), m[i])$
  - Update $O = O\exp\big(m(x) - m_{new}(x)\big) + \exp\big(m[i] - m_{new}(x)\big)\, O_i \times norm[i]$
  - Update $norm = \exp\big(m(x) - m_{new}(x)\big)\, norm + \mathrm{norm[i]} \times \exp\big(m[i] - m_{new}(x)\big)$.
  - Update $m(x) = m_{new}(x)$

Autoregressive Training

Recall in the autoregressive training objective

Given X[0:i], we want to predict X[i], for every i in [context_length]

Naïve implementation: Treat X[0:i] as a separate input with label X[i].

Total computation time: context_length * computation time on input X[0:context_length]

Can we do it more efficiently in computation time of a single X[0:context_length]?

Attention Mask

- The core of MHA is the soft-max attention score:
- $\left(\alpha_{i,j}^{r}\right)_{j\in[n]} = softmax\left(v_i^T Q_r K_r^T v_j + p_{i,j}^{r}\right)_{j\in[n]}$
- Key observation: We can set $p_{i,j}^{r} = -\infty$ if and only if i < j (attention mask).
- In this way, the new value
  - $v_i' = C\times concatenate\left(V_r^T \sum_j \alpha_{i,j}^{r} v_j\right)_{r\in[d/m]} + b$
- v_i' only depends on v_j for $j \leq i$.

Inference

After autoregressive training, we can use the autoregressive language model to generate texts.

Given a prompt s (text), we can

Tokenize the prompt s into a list of integers S.	* Feed S into the autoregressive language model, and obtain its prediction S_{pred}.	Update S = concatenate(S, S_{pred}).	Repeat Step *.

Multi-Query Attention

- Optimized for inference speed.
- Time-consuming step for inference:
  - Feed S into the autoregressive language model, and obtain its prediction S_{pred}.
  - We do not want to recompute model(S) every time we update S.
- Key observation: Caching.
  - We can cache the past $K_r^T v_j$ and $V_r^T v_j$ values for all j < len(S), and no need to recompute them.
  - However, this requires us to cache
    - $d \times len(S)$ many values.

Multi-Query Attention

- Multi-query attention:
- Instead of using $\left(\alpha_{i,j}^{r}\right)_{j\in[n]} = softmax\left(v_i^T Q_r K_r^T v_j + p_{i,j}^{r}\right)_{j\in[n]}$
- $v_i' = C \times concatenate\left(V_r^T \sum_j \alpha_{i,j}^{r} v_j\right)_{r\in[d/m]} + b$
- We now use $\left(\alpha_{i,j}^{r}\right)_{j\in[n]} = softmax\left(v_i^T Q_r K^T v_j + p_{i,j}^{r}\right)_{j\in[n]}$
- $v_i' = C \times concatenate\left(V^T \sum_j \alpha_{i,j}^{r} v_j\right)_{r\in[d/m]} + b$
- So every head shares the same K, V
    - (of dimension embed_dim x head_dim).