CS 229S Project 0

October 7, 2023

We are excited to release Project 0, constructed specifically for this new course! We hope you learn a ton about how Transformers work, how to think about and measure the efficiency of different parts of an architecture, and how to go about speeding up the computations. These are incredibly useful tools! You will submit the following files for each section. Please submit the zipped files to Gradescope.

- Part 1: src/lm.py and programming_transformer_ar.ipynb with your completed implementations.
- Part 3: "conv1d_cuda.cu" with your completed implementation.
- Your latex file with the filled out written responses.

The due date for this assignment is **Friday October 20 at Midnight**.

1 Understanding Transformers

The goal of this section is to learn how to implement Transformers in detail. We have provided scaffolding code for training a Transformer model.

Here we describe the training data included in the provided code: We are going to focus on training with synthetic language modeling data that follows a simple pattern. An example X constructed from a *prefix* of key-value pairs and a *query* key. The model needs to output the *value* corresponding to the query by referring to the prefix. This type of task is often called Associative Recall (AR).

$$X = [\text{a1c6a1e2f3} \rightarrow \text{e}], Y = 2$$

1.1 Training Transformers for Language Modeling (15 points)

First complete the implementation in "src/lm.py" at the positions marked with TODOs. You will need to implement Multihead Self Attention. We have provided a script "run.sh" that you can use to launch the code once implemented. The code should reach 90%+ accuarcy on the default Associative Recall dataset within 50 epochs (likely sooner) of training when correct. You should not need to modify any other files.

Below we will go through questions about the Transformer implementation and its training behavior.

1. To build intuition for the number of required layers of Attention to solve this task, include a plot to demonstrate when the model starts solving the task with 90%+ accuracy as a function of the number of Transformer layers. You can control the number of model layers in the arguments in "src/experiments.py". The minimum number of layers at which the Transformer starts solving to 90+% accuracy is?
2. Provide a brief explanation for the "causal mask" in Self Attention. What is it's purpose in 1-3 sentences?
3. Provide a brief (1-3 sentence) explanation for the relationship between "inputs" and "targets" for the language modeling data that is being passed to the model.

1.2 Hard Coding the Attention Weights to Solve Associaitve Recall (15 points)

In this problem, we will write out how a Transformer would solve AR by hard coding the Transformer weights (rather than learning them) in a Notebook. The goal is for you to get very comfortable with the attention operation. We can solve AR using two layers of attention (as you hopefully found in **Part 1**). We are providing a notebook with scaffolding and test data for you to evaluate your solution.

The Notebook includes the following three steps covered in class: (1) converting raw tokens to *embeddings* (real-valued vectors that represent the tokens in the input data), (2) computing attention at each layer with residual connections, and (3) mapping the final attention outputs to predictions. Note that we will ignore the Transformer MLPs, Multi-headed attention, and LayerNorms for this exercise, to maintain simplicity. We define the embeddings via *one-hot* encodings for this exercise.[1]

Submit code in the notebook (look for "# TODO" comments) that maximizes the score of the function "score_solution". Your implementation will be reviewed for its correctness.

2 Understanding Transformer Attention Performance (40 points)

Training Large Language models requires a ton of compute resources which are expensive, we want to use these resources as efficiently as possible so as to cut cost and iterate faster. In this section, we shall dive into the performance aspects of a Transformer from a training perspective. We shall investigate the forward pass of the Transformer (attention and MLP).

As a first step, we want to understand the properties of the different operations in a Transformer. For this problem, we focus on decoder-only Transformers (GPT-2, LLAMA etc.), as opposed to the encoder-decoder architecture or encoder-only architectures like BERT. A key to high-performance implementations is understanding the compute and IO-characteristics of the algorithm. In this section, you will compute the number of floating-point operations (FLOPS) required for different portions of the Transformer layer.

For the following questions, you may make the following assumptions:

- In class we have discussed that there is a memory hierarchy that includes registers (fastest), L1 cache, L2 cache, and Global Memory (slowest). For this problem, ignore the hierarchy and assume there is only Global Memory.
- The number of bytes used to represent a datatype can be denoted as n_{bytes}. Recall that 1 byte is 8 bits. Popular data types in machine learning are BFloat16, FP16 and FP32, which are 2 bytes, 2 bytes, and 4 bytes respectively.
- For simplicity ignore all biases in linear layers.

2.1 Input Projections (5 points)

The first operation in the Transformer block is a linear transformation of the input to obtain 3 projections, Q (typically referred to as query), K (typically referred to as key), and V (typically referred to as value). The input X, to the Transformer block is a tensor of shape (B, N, D), where B is the batch dimension, N is the context (sequence) length and D is the embedding dimension. Given learnable weight matrices W_Q, W_K and W_V each of shape (D, D), Q, K, and V are obtained as follows.

$$Q = XW_Q \tag{1}$$

$$K = XW_K \tag{2}$$

$$V = XW_V \tag{3}$$

[1] `https://www.geeksforgeeks.org/ml-one-hot-encoding-of-datasets-in-python/`

1. Derive an expression for the number of floating point operations required to perform the input projections.

2. Derive an expression for the total number of bytes read/written to memory in order to perform the input projections.

2.2 Scaled Dot-Product Attention (20 points)

Recall that scaled-dot product attention is given by the equation below. The overall steps are (1) **Dot Product Scores** computing dot-product similarities between queries and keys. (2) **Masking** masking the output matrix to maintain causality (i.e. exclude entries of the matrix that are between queries and keys that are at future positions in the matrix). (3) **Softmax** taking the row-wise softmax of the matrix and (4) **Computing the Output** multiplying the attention scores with the values.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

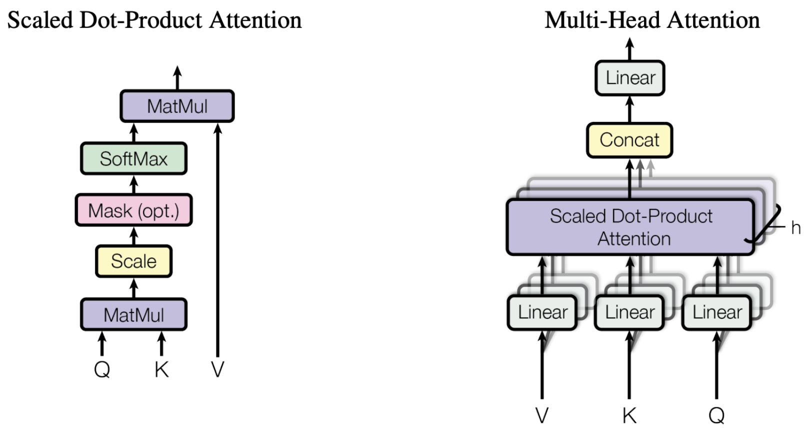

Figure 1: Multihead attention from [1].

1. **Computing $\mathbf{QK^T}$ Dot-Product Scores**

   For this question, you may assume that $\sqrt{d}$ is provided and you do not need to read it from memory or compute it.

   (a) Derive an expression for the number of floating point operations required to perform the $\frac{QK^T}{\sqrt{d_k}}$ operation.

   (b) How many bytes are read from/written to memory for the operation? (You may assume that $\frac{QK^T}{\sqrt{d_k}}$ is a fused operation i.e both the QK^T multiplication and the division by $\sqrt{d_k}$ happen before any result is written to memory.

2. **Causal Masking**

   Masking is used to enforce a "causal" relationship where the output at each step does not depend on future steps, thereby preserving the temporal order of the input data. Masking can be implemented in multiple ways. For the purpose of this question, you may assume that masking is done via elementwise addition of the input and a tensor **M** containing the mask.

   To understand how **M** is constructed (note this is a hint for Part 1 of this assignment!), note that Softmax is computed as $\frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}$. If we set a value in the vector z_i to be very small, for instance -10000, taking the softmax results in a negligibly small value. Therefore we construct **M** to have -10000 at the positions we want to mask out in a matrix that has the same shape as our QK^T matrix, we can add the two matrices to perform masking.

   (a) What is the number of floating point operations required for Masking
   (b) How many bytes are read from/written to memory for the Masking?

3. **Softmax**

   The softmax function is used in machine learning, especially in the field of deep learning, to convert a vector of real numbers into a probability distribution. Each output of the softmax function is between zero and one, and the sum of the outputs is equal to one. The softmax function S for a vector z of length K is defined as follows:

   (a) What is the number of floating point operations required for Softmax
   (b) How many bytes are read from/written to memory to perform masking?

4. **Computing the Output**

   Denote the output of the softmax stage as $A_{mat} = softmax(\frac{QK^T}{\sqrt{d_k}})$. The final stage of scaled dot product attention (SDPA) can be written as

   $$Attention(Q, K, V) = A_{mat}V$$

   .

   (a) What is the number of floating point operations required for this step?

   (b) How many bytes are read from/written to memory for this step?

5. Putting everything together

   Now that we have computed the number of FLOPS required for each sub-operation in scaled dot-product attention, we can compute the following:

   (a) What is the total number of floating point operations required for scaled-dot product attention?
   (b) What is the total amount of IO reads/writes in bytes?

2.3 Multi-Head Attention (MHA) (10 points)

The computations in the previous question are usually done in parallel across H heads.

1. Derive an expression for the number of floating point operations required for MHA (do not include the projections in your calculation).
2. Derive an expression for the total number of bytes read/written to memory in order to perform the input projections.
3. How does the number of FLOPS for MHA scale with sequence length?
4. How does the total memory access in bytes scale with sequence length?
5. Given a batch size $B = 16$, model dimension $D = 4096$, sequence length $N = 16384$, and using half-precision (FP16) i.e 2 bytes per element, is MHA compute or memory bound on an A100-80GB SXM? Datasheet for an A100 can be found on this link.

2.4 Multilayer Perceptron (5 points)

Next we will compute for the position-wise feed-forward networks (MLPs) of the Transformer, which are applied after the attention computation. The typical MLP in a transformer includes a first projection from the $N \times d$ input X to a value that is $N \times 4d$, by multiplying the input with a $d \times 4d$ matrix. Next, there is a second projection from the $N \times 4d$ matrix to $N \times d$, by multiplying the with a $4d \times d$ matrix. Assuming that the input X is $B \times N \times d$ where B is the batch size, N is the sequence length, and B is the batch size, answer the following questions.

1. Derive an expression for the number of floating point operations for a forward pass of the MLP.
2. Derive an expression for the total number of bytes read/written to memory for a single forward pass through the MLP.
3. How does the arithmetic intensity of an MLP compare to that of multi-head attention?

3 Fast Depthwise 1D Convolutions! (30 pts)

In this problem, you will see how creating custom implementations for operators using a tool like CUDA can be powerful. There are several reasons why you might want to do this: you might be creating a new operator that is not available in any existing libraries like PyTorch or the existing implementation is not very optimized for your specific use case. This problem focuses on depthwise 1D convolutions with short filter lengths. Depthwise 1D convolutions are used in some new language model architectures like Hyena [2] and RWKV [3]. You will implement and benchmark a CUDA kernel for the forward pass of a depthwise 1D convolution that is several times faster than the current implementation in PyTorch.

- **Inputs**: We are given a signal u of shape (B, L, D) , a filter of shape (D, K) and a bias of shape (D). Where B is the batch dimension, L is the sequence length, D is the channel dimension and K is the filter length. For this example we will consider a specific filter length $K = 3$, although it should be relatively easy to extend your solution to work for other filter lengths. We want the input signal in shape (B, L, D), because in some of these models, the Conv1D module operates on output from upstream modules which come in that shape. We also only consider strides and dilations which are equal to 1 (if you don't know what these are, you can safely ignore for the purpose of this problem).
- **Output**: Our output signal is of shape (B, L, D). We pad our input signal with 0's on both ends, so that the shape of the input signal is exactly that of the output signal. The output signal is then computed by sliding the filter over the padded input signal, and adding the bias. This is done independently for each channel dimension.

  As a concrete example, consider an input signal u of shape $(1, 5, 16)$, a filter k_f of shape $(16, 3)$, and bias b of shape (16). Suppose we have:

  $$u[0,\ :,\ 0] = [2, -3, 1, -1, 1]$$

  $$k_f[0,\ :] = [1, -1, 1]$$

  $$b[0] = [6]$$

  We compute the output for $d = 0,\ b = 0$ as follows:

  1. Pad the input along the l dimension with 4 zeros, 2 on each end to ensure that the input is the same shape as the output.

     $$u[0,\ :,\ 0] = [0, 0, 2, -3, 1, -1, 1, 0, 0]$$

  2. Compute the output by doing a 1D convolution of the filter and signal along the l dimension, independently for each b and d dimension. For example:

     $$out[0,\ 0,\ 0] = 0 \times 1 + 0 \times -1 + 2 \times 1 + 6$$

     $$out[0,\ 1,\ 0] = 0 \times -1 + 2 \times -1 - 3 \times 1 + 6$$

     $$out[0,\ 2,\ 0] = 2 \times -1 - 3 \times -1 + 1 \times 1 + 6$$

3.1 Performance of Depthwise 1D convolution (5pts)

As a first step, we want to understand the performance of the 1D Depthwise convolution on a Nvidia T4 GPU. A datasheet for the Nvidia T4 GPU can be found at this link. Assume you are given an input signal u of shape $(1, 8192, 8192)$, filter of shape $(8192, 3)$ and bias of shape (8192) and we need to compute the 1D depthwise convolution as described above so that the output is of shape $(1, 8192, 8192)$. To do this we pad the original input along the L dimension with 4 zeros, 2 on both sides. Assume half-precision (each element is 2 bytes).

1. What is the GPU memory bandwidth for the T4? **(0.25pts)**
2. What is the mixed precision compute bandwidth for the T4? **(0.25pts)**
3. What is the compute bandwidth to memory bandwidth ratio for the T4, using the information above? **(0.5pts)**
4. What is the total number of floating point operations required to compute the convolution? **(0.5pts)**
5. What is the total number of bytes accessed (read + written) to compute the convolution? (The input is first read from memory, padded, and the padded input is written back to memory. The padded input, filter, and bias are then read from memory and the convolution is computed and the output is written back to memory). **(0.5pts)**
6. What is the arithmetic intensity of the operation? **(0.5pts)**
7. Is this operation compute or memory bound on a T4 ? justify your answer. **(0.5pts)**
8. Given the responses from the above section, why is having two separate kernels (instead of a single one): one for padding and another for computing the convolution a bad idea ? **(0.5pts)**
9. In Lecture 3, we saw a few simple principles that can help us achieve good performance (fusion, tiling, pipelining, caching/recomputation, etc.). What opportunities exist to apply the following techniques to our problem: **(1pts)**
   (a) Fusion
   (b) Tiling
10. In Lecture 3, we saw that exposing enough parallel work to the GPU is important to achieve good performance. How will you parallelize the work in the algorithm? **(0.5pts)**

3.2 Depthwise 1D convolution in CUDA (25pts)

In class we touched on GPUs and how they can be programmed. In this section you will complete Complete the kernel in `conv1d.cu`, you can use `dev.py` through out your development to validate the correctness of your solution. The readme file included in the problem set is helpful on how to run your solution using free GPU credits on a T4 GPU from google cloud. We will be evaluating your solution using the same setup. Here are a few great resources that could be helpful:

- An even easier introduction to CUDA
- The first two sections of the CUDA c-programming guide

To compile your kernel, run the `build.sh` script.

1. Once you are done with your kernel, run `test.py` and include the table generated below. **(23pts)**
2. From the results above, what are the effective memory bandwidth and effective compute bandwidth utilization for sequence length 8192 and channel dimension 8192? **(1pts)**
3. Plot a roofline for the T4 showing where the depthwise 1D convolution operation for the given problem size lies. **(2pts)**

References

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. volume 30, 2017.

[2] Michael Poli, Stefano Massaroli, Eric Nguyen, Daniel Y Fu, Tri Dao, Stephen Baccus, Yoshua Bengio, Stefano Ermon, and Christopher Ré. Hyena hierarchy: Towards larger convolutional language models. *arXiv preprint arXiv:2302.10866*, 2023.

[3] Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, Kranthi Kiran GV, et al. Rwkv: Reinventing rnns for the transformer era. *arXiv preprint arXiv:2305.13048*, 2023.