EO 294: Systems for Machine Learning Assignment #4

Rohit Jorige (22166)
Indian Institute of Science, Bangalore

March 30, 2025

1 Problem 1: Let's do some math

(a)

As mentioned in the question, LLaMA model has two key dimensions:

- $\boldsymbol{h_1 = 4096}$: This is the embedding dimension.
- $\boldsymbol{h_2 = 14336}$: This is the inner dimension of the feed-forward network, since this is for computing the feed forward network we will not use this for calculating the size of the kv cache, this will be used in calculating the model parameters.

The formula for the space taken by the KV cache per request is:

$$\text{KV Cache Size (bytes)} = 2 \times \text{precision} \times N_{\text{layers}} \times h_1 \times s \times B,$$

where:

- The factor 2 is for storing both key and value matrices.
- precision = 2 bytes since the model on hugging face is stored in BF16 format which means 16 bits or 2 bytes.
- $N_{\text{layers}} = 32$.
- $h_1 = 4096$.
- $s = 8196$ (LLaMA 3 8B has a 8k context length on the huggingface card)
- B is the batch size.

Hence, we have:

$$\text{KV Cache Size} = 2 \times 2 \times 32 \times 4096 \times 8196 \times B = 4294967296 \times B \quad \text{bytes.}$$

Converting to gigabytes:

$$\text{KV Cache Size (GB)} = \frac{4294967296 \times s \times B}{2^{30}} = 4\text{GB per batch.}$$

From the hugginng face card, LLaMA 3-8B has 8.03B parameters, with 2 bytes per paramter the size of the model is:

$$\text{Total parameter memory (GB)} = \frac{8.03\text{B} \times 2}{2^{30}} = 14.957040548 \approx 15GB$$

So the left our memory for the kv cache is **80-15 = 65GB**. The max batch size possible with this is: $65/4 \approx 16$

Metric	**Value/Expression**
Total model parameters	8.03B
Total parameter memory (GB)	15 GB
KV Cache per request (GB)	$4 * B$ GB
Max batch size possible	16

(b)

So now every GPU still has 80 GB of memory, but now handles only a fraction of the total memory requirements.

Model Parameter Memory per GPU Total parameter memory $M = 15$ GB is split across X GPUs. Memory per GPU for parameters $= \frac{M}{X} = \frac{15}{X}$ GB.

KV Cache Memory per GPU The total KV cache size for the entire batch is $4 \times B$ GB. With tensor parallelism, we get KV cache memory per GPU $= \frac{4 \times B}{X}$ GB.

Note: The KV cache size depends on the batch size B, which is the same across all GPUs, but the memory for keys and values is split across GPUs, typically along the head or embedding dimension in transformer models. But in the question it is given assume eequal split among all GPUs so I am taking an equal split.

Total Memory Constraint per GPU Each GPU's memory usage must not exceed its 80 GB capacity:

$$\text{Memory per GPU} = \frac{M}{X} + \frac{\text{KV Cache Size}}{X} \leq 80$$

Substitute the values:

$$\frac{15}{X} + \frac{4 \times B}{X} \leq 80$$

We calculate the maximum batch size as

$$\frac{4 \times B}{X} \leq 80 - \frac{15}{X}$$

$$4B \leq X \times \left(80 - \frac{15}{X}\right)$$

$$B \leq \frac{X \times \left(80 - \frac{15}{X}\right)}{4}$$

$$B_{\max} = \left\lfloor \frac{X \times \left(80 - \frac{15}{X}\right)}{4} \right\rfloor$$

If we are to write the general formula we get:

$$B_{\max} = \left\lfloor \frac{X \times \left(\text{GPU\_memory} - \frac{Model\_size}{X}\right)}{K} \right\rfloor$$

where K is the memory of kv cache for a single batch.

(c)

Prefill Phase

In the prefill phase, we compute attention over the full sequence, so $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$. Here a single floating point number takes a value of b bytes.

Step 1: $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$

- **Matrix Dimensions**: $\mathbf{Q}$ is $N \times d$, $\mathbf{K}^\top$ is $d \times N$, so $\mathbf{S}$ is $N \times N$.
- **Flops**: For a matrix multiplication $A \times B$ producing a $p \times q$ matrix, with A of shape $p \times r$ and B of shape $r \times q$, the flops are $2 \cdot p \cdot r \cdot q$. here we have , $p = N$, $r = d$, $q = N$, so we get total flosp as :

$$\text{Flops} = 2 \cdot N \cdot d \cdot N = 2N^2 d$$

- **Memory Access**:
  - **Read**: Load $\mathbf{Q}$ ($N \cdot d \cdot b$ bytes) and $\mathbf{K}$ ($N \cdot d \cdot b$ bytes), total $2Ndb$ bytes.
  - **Write**: Write $\mathbf{S}$ ($N \cdot N \cdot b = N^2 b$ bytes).
  - **Total**: $2Ndb + N^2 b$.

Step 2: $\mathbf{P} = \textbf{softmax}(\mathbf{S})$

- **Matrix Dimensions**: $\mathbf{S}$ is $N \times N$, softmax is applied row-wise, so $\mathbf{P}$ is $N \times N$.
- **Flops**: For each of the N rows of length N:
  - Exponentials: N operations.
  - Sum of exponentials: $N - 1$ additions (approximately N).
  - Divisions by the sum: N operations.
  - Total per row: $\approx 3N$ flops.
  - Total for N rows: $3N \cdot N = 3N^2$ flops.
  - Final divide by sum of final results in the column : $N - 1 + N = 2N - 1 \approx 2N$
  - Total Number of Flops $\approx 3N^2 + 2N$
- **Memory Access**:
  - **Read**: Read $\mathbf{S}$ ($N^2 b$ bytes).
  - **Write**: Write $\mathbf{P}$ ($N^2 b$ bytes).
  - **Total**: $N^2 b + N^2 b = 2N^2 b$.

Step 3: $\mathbf{O} = \mathbf{PV}$

- **Matrix Dimensions**: $\mathbf{P}$ is $N \times N$, $\mathbf{V}$ is $N \times d$, so $\mathbf{O}$ is $N \times d$.
- **flops**: $p = N$, $r = N$, $q = d$, so:

$$\text{Flops} = 2 \cdot N \cdot N \cdot d = 2N^2 d$$

- **Memoyr Access**:
  - **Read**: Load $\mathbf{P}$ ($N^2 b$ bytes) and $\mathbf{V}$ (Ndb bytes), total $N^2 b + Ndb$.
  - **Write**: Write $\mathbf{O}$ (Ndb bytes).
  - **Total**: $N^2 b + Ndb + Ndb = N^2 b + 2Ndb$.

Total for Prefill

- **Total Flops**:

$$2N^2 d + 3N^2 + 2N + 2N^2 d = 4N^2 d + 3N^2 + 2N$$

- **Total Memory Access**:
  - Reads: $2Ndb$ (step 1) + $N^2 b$ (step 2) + $N^2 b + Ndb$ (step 3) = $3Ndb + 2N^2 b$.
  - Writes: $N^2 b$ (step 1) + $N^2 b$ (step 2) + Ndb (step 3) = $Ndb + 2N^2 b$.
  - Total: $4Ndb + 4N^2 b$.

Thus, for the prefill phase:

- **Total Flops**: $4N^2 d + 3N^2 + 2N$
- **Total Memory Access**: $4Ndb + 4N^2 b$

Decode Phase

In the decode phase, ew compute attention for a single new token , with $\mathbf{Q} \in \mathbb{R}^{1 \times d}$ and $\mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$, N over here is the context length of accumulated keys and values.

Step 1: $\mathbf{S} = \mathbf{QK}^\top$

- **Matrix Dimensions**: $\mathbf{Q}$ is $1 \times d$, $\mathbf{K}^\top$ is $d \times N$, so $\mathbf{S}$ is $1 \times N$.
- **Flops**: $p = 1$, $r = d$, $q = N$, so:

$$\text{Flops} = 2 \cdot 1 \cdot d \cdot N = 2Nd$$

- **Memory Access**:
  - **Read**: Load $\mathbf{Q}$ (db bytes) and $\mathbf{K}$ (Ndb bytes), total $db + Ndb$.
  - **Write**: Write $\mathbf{S}$ (Nb bytes).
  - **Total**: $db + Ndb + Nb$.

Step 2: $\mathbf{P} = \textbf{softmax}(\mathbf{S})$

- **Matrix Dimensions**: $\mathbf{S}$ is $1 \times N$, so $\mathbf{P}$ is $1 \times N$.
- **Flops**: Softmax over one row of length N: $\approx 3N$ flops (as above).
- **Memory Access**:
  - **Read**: Read $\mathbf{S}$ (Nb bytes).
  - **Write**: Write $\mathbf{P}$ (Nb bytes).
  - **Total**: $Nb + Nb = 2Nb$.

Step 3: $\mathbf{O} = \mathbf{PV}$

- **Matrix Dimensions**: $\mathbf{P}$ is $1 \times N$, $\mathbf{V}$ is $N \times d$, so $\mathbf{O}$ is $1 \times d$.
- **Flops**: $p = 1$, $r = N$, $q = d$, so:

$$\text{Flops} = 2 \cdot 1 \cdot N \cdot d = 2Nd$$

- **Memory Access**:
  - **Read**: Load $\mathbf{P}$ (Nb bytes) and $\mathbf{V}$ (Ndb bytes), total $Nb + Ndb$.
  - **Write**: Write $\mathbf{O}$ (db bytes).
  - **Toal**: $Nb + Ndb + db$.

Total for Decode

- **Total flops**:

$$2Nd + 3N + 2Nd = 4Nd + 3N$$

- **Total memmory access**:
  - Reads: $db+Ndb$ (step 1) + Nb (step 2) + $Nb+Ndb$ (step 3) = $db+2Ndb+2Nb$.
  - Writes: Nb (step 1) + Nb (step 2) + db (step 3) = $2Nb + db$.
  - Total: $2db + 2Ndb + 4Nb$.

Final results for the decode phase:

- **Total Flops**: $4Nd + 3N$
- **Total Memory Access**: $(2d + 2Nd + 4N)b$

(d)

As we can see form the plot below the arithmetic intensity increases, initially as the sequence length increases, as it slows down due to increase in memory access as the sequence length increases. The behaviour can be explained by the formula that I have derived above. But we can say that the prefill phase becomes compute-bound at large N, we can say that its performance is limited by available compute power rather than memory bandwidth, so we can sort of optimize it by parallelization. For the decode phase it increases slighly and dies down as it is memory-bound and its computational demands per token remain largely unaffected by context size,as its autoregressive processing of one token at a time.

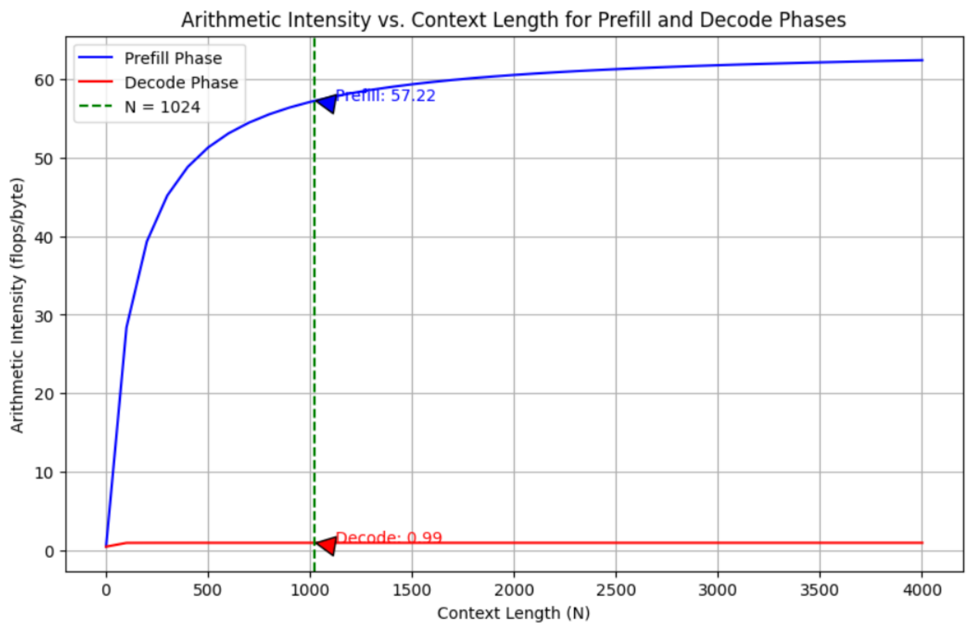

Figure 1: arithmetic intensity

2 Problem 2: Transformer implementation with KV caching

(a)

Implemented and code file is attached.

(b)

Implemented and code file is attached. A key issue I faced here was the numerical precision. With an absolute tolerance of `1e-5` I observed a pass rate of 20%, with a tolerance of `1e-4` I observed a pass rate of 70% and with a tolerance of `1e-3` I got a pass rate of 100% on all test cases. Reducing the numerical precision allowed me to match with the test cases.

```
Evaluation Results:
  Num test cases: 10
  All tests passed: True
  Pass rate: 100.00% (10/10)
```

(c)

Implemented in the code file. Observed the same results as without kv caching as above (in terms of test cases passed).

```
Evaluation Results:
  Num test cases: 10
  All tests passed: True
  Pass rate: 100.00% (10/10)
```

(d)

Different speedups were observed, since this was running on my machine on CPU, we can expect the caching patterns to be different for each run, but the general idea that kv caching causes speedups can be observed, from one of the runs we see that,

```
Results:
  Without KV cache: 0.0165 seconds
  With KV cache: 0.0112 seconds
  Speedup: 1.47x
```

For the benchmark, I tested with sequence lenghts of [10,50,100,1000,10000]. I have plotted the results below, from the diagram it is clear that kv caching does help in significantly reducing inference time for longer sequence lengths. (the below results are for **n=20**.)

```
Sequence length 10: Without KV: 0.0133 s, With KV: 0.0115 s
Sequence length 50: Without KV: 0.0186 s, With KV: 0.0124 s
Sequence length 100: Without KV: 0.0189 s, With KV: 0.0121 s
Sequence length 500: Without KV: 0.0530 s, With KV: 0.0154 s
Sequence length 1000: Without KV: 0.1573 s, With KV: 0.0237 s
Sequence length 10000: Without KV: 27.9321 s, With KV: 1.4681 s
```

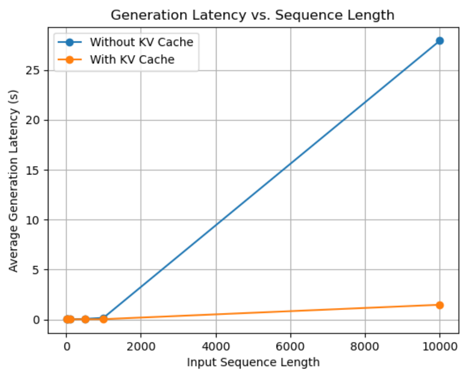

Figure 2: latency plot including seq_len=10000, n=20 case

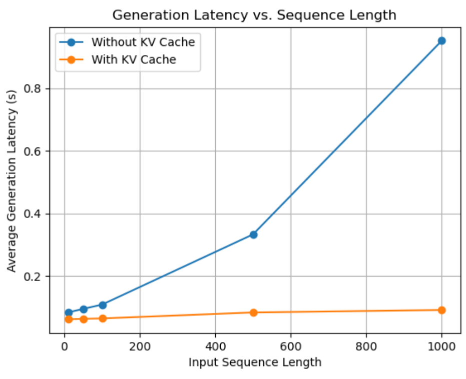

Figure 4: latency plot without seq_len=10000, n=100 case

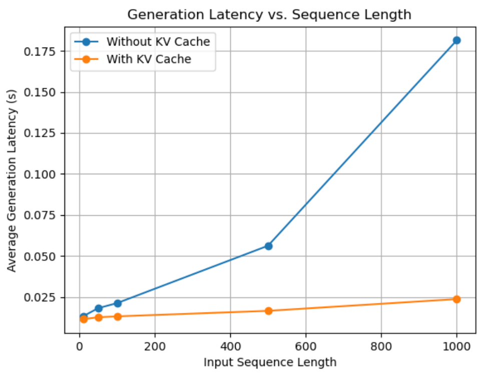

Figure 3: latency plot without seq_len=10000, n=20 case

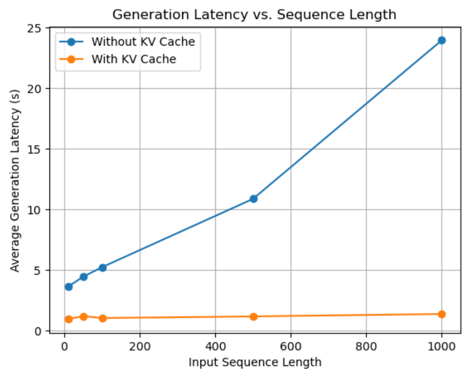

Figure 5: latency plot without seq_len=10000, n=1000 case

As expected, increasing the length of generated token increases latency time. (Also a key reason why the Pricing for a larger context length model for ChatGPT is almost double the lower context length model.)