E0294: SYSTEMS FOR MACHINE LEARNING

Assignment 4

Submitted By:

Anshika Modi
SR: 24095
CSA Dept.
IISc, Bangalore
2024-26

March 28, 2025

1 Problem 1:

Given Model, Hardware and Input Specifications are as follows:

Specifications	**Llama3-8B**
Number of Layers	L = 32
Number of Query Heads	QH = 32
Number of KV Heads	KVH = 8
Embedding Dimension	h_1 = 4096
Inner Dimension	h_2 = 14336
Vocab Size	V = 128256
GPU Memory	GPU = 80GB
Context Length	S = 4096
Weight Bytes	WB = 2 Bytes
KV cache Bytes	KVB = 2 Bytes
Total Number of Parameters	Param = 8 Billion

1.1

In `Llama3-8B`, there are total appoximately 8 Billion Parameters. Each Parameter is a FP-16 i.e. 2 Bytes. So, Total Parameter Memory (in GB) is given by:

$$\frac{8,000,000,000 * 2}{1024^3} GB \approx 15GB$$

For one token, KV Cache stores a pair of tensor, key-value, for each layer and attention head. The size of these tensors is governed by the dimension of attention head. Thus, total memory consumption (in Bytes) for one token is given by:

$$2 \times \#Layers \times \#KVH \times (\text{Attention head Dimension}) \times KVB$$

where, Attention Head Dim $= \frac{\text{Emb Dim}}{\text{Num of Heads}} = \frac{h_1}{QH} = \frac{4096}{32} = 128$
In formula, It is multiplied by 2 as both keys and values are stored.

$$= 2 * 32 * 8 * 128 * 2 \text{ Bytes}$$

$$= 1,31,072 \text{ Bytes}$$

So, Total KV Cache per request for S tokens is given by:

$$= 131072 \times S \text{ Bytes}$$

$$= 131072 * 4096 \text{ Bytes}$$

$$= \frac{536,870,912}{1024^3} = 0.5GB$$

Hence, Maximum permissible batch size possible with 80 GB GPU memory is given by:

$$\text{Max Batch Size} = \frac{(\text{GPU memory} - \text{Parameter memory})}{\text{KV cache per request}}$$

$$\text{Max Batch Size} = \frac{(80GB - 15GB)}{0.5GB} = 130$$

Hence, the final table is as follows:

Total Model Parameters	8 Billion
Total Parameter Memory (GB)	15 GB
KV Cache Per Request (GB)	0.5 GB
Max Batch Size Possible	130

1.2

With Tensor Parallelism of dimension X, Each GPU will hold 1/X of model parameters as well as of KV cache. Hence, Model parameter memory and KV cache memory per request shrinks by $\frac{1}{X}$.

$$\text{Total Parameters Memory Per GPU } = \frac{15}{X}GB$$
$$\text{KV Cache per request Per GPU } = \frac{0.5}{X}GB$$

So, Maximum permissible batch size per GPU possible with Tensor Parallelism is given by:

$$\text{Max Batch Size Per GPU} = \frac{(80GB - \frac{15}{X}GB)}{\frac{0.5}{X}GB}$$

If we take $X = 3$, Maximum Permissible Batch Size Per GPU will be 450! And combined batch Size over all 3 GPUs will be 1350!
Hence, the final table for Tensor Parallelism of dimension 3 is as follows:

Total Model Parameters	8 Billion
Total Parameter Memory per GPU (GB)	5 GB
KV Cache Per Request per GPU (GB)	$\frac{0.5}{3}$ GB
Max Batch Size Possible per GPU	450
Max Batch Size Possible	450*3 = 1350

The plot of Variation of Maximum Permissible Batch Size Per GPU with Tensor Dimension (X) is shown below:

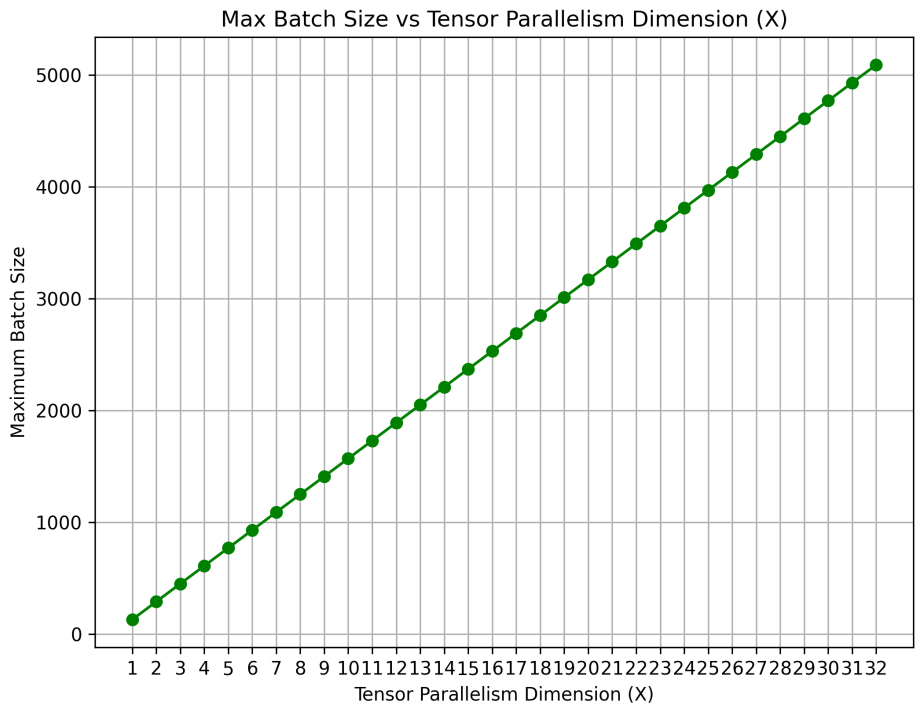

Figure 1: Max Batch Size Per GPU Vs Tensor Dimension(X)

1.3

Head Dimension is d. And, Context Length is N.
Attention Matrix Computation is given by:

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

Its Pseudocode is given below:

Algorithm 1 Standard Attention Implementation

```
Require: Matrices Q, K, V $\in R^{N,d}$ in HBM.
1: Load Q, K, compute $S = QK^T$ and write back S.
2: Read S, compute P = softmax(S) and write P back.
3: Load P, V, compute $O = PV$ and write it back.
4: return O.
```

Each of Q, K, V is of shape (N, d).
Arithmetic Intensity is given by:

$$AI = \frac{\#\text{FLOPs}}{\#\text{bytes accessed}}$$

1.3.1 Prefill Attention Computation

In prefill, whole sequence N is processed. Compute the entire score matrix $(N \times N)$, softmax it, and multiply by V to get $(N \times d)$.

Total FLOPs in Prefill Attention Computation are calculated as follows:

1. Compute $S = QK^T$. Both Q, K are of dimensions $(N \times d)$ and S is of dimension $(N \times N)$. For 1 element of S i.e S_{ij}, the corresponding elements of i^{th} row of Q and j^{th} column of K^T are multiplied and then added. So, total number of FLOPs done to get 1 element of S are $2d$. Thus, for $N \times N$ elements of S, total FLOPs done are $2N^2d$.

2. Compute $P = \text{softmax(S)}$. To do the softmax of S, each element is first exponentiated. Hence, N^2 FLOPs for exponentiating. The exponentiated values are added across each row i.e N additions in 1 row and N^2 additions for N rows. Then, each exponentiated entry is divided by the sum of corresponding row. So, again N^2 FLOPs for this. Hence, Total number of FLOPs done to get P is given by:

$$N^2 + N^2 + N^2 = 3N^2$$

3. Compute $O = PV$. P is a $(N \times N)$ matrix, V is of dimensions $(N \times d)$ and O will be of dimension $(N \times d)$. For 1 element of O i.e O_{ij}, the corresponding elements of i^{th} row of P and j^{th} column of V are multiplied and then added. So, total number of FLOPs done to get 1 element of O are $2N$. Thus, for $N \times d$ elements of O, total FLOPs done are $2N^2d$.

4. Therefore, total number of FLOPs in Prefill attention computation is given by:

$$2N^2d + 3N^2 + 2N^2d = N^2(4d + 3).......(1)$$

Total Memory Accesses (in Bytes, assuming each element is FP16 i.e 2 Bytes) in Prefill Attention Computation are calculated as follows:

1. Read Q, K from HBM. Both Q, K are of dimensions $(N \times d)$. So, $2Nd$ memory accesses

2. Write S back to HBM. S is of dimension $(N \times N)$. So, N^2 memory accessed.

3. Load S again, compute P and write P to HBM. $2N^2$ memory accesses.

4. Load P and V. P is of dimension $(N \times N)$ and V is of dimension $(N \times d)$. So, $N^2 + Nd$ memory accesses.

5. Compute O of dimension $(N \times d)$ and write it to HBM, Thus, Nd memory-accesses.

6. Therefore, total bytes of memory accesses in Prefill attention computation is given by:

$$2 * (2Nd + N^2 + 2N^2 + N^2 + Nd + Nd)B$$

$$= 8Nd + 8N^2 = 8N(N + d) \text{ Bytes.......(2)}$$

From Equation (1) and (2), the Arithmetic Intensity for Prefill Attention Computation is given by:

$$AI = \frac{\#\text{FLOPs}}{\#\text{bytes accessed}} = \frac{N^2(4d+3)}{8N(N+d)} \text{ FLOPs/Bytes}$$

$$AI = \frac{N(4d+3)}{8(N+d)} \text{ FLOPs/Bytes}$$

1.3.2 Decode Attention Computation

In decode, processing is done only for one query vector(new token) i.e compute attention for the new token against the cached KV.

Total FLOPs in Decode Attention Computation are calculated as follows:

1. Compute $S = QK^T$. Here, Q is $1 \times d$ vector and K is a $N \times d$ matrix. S obtained is of dimension $1 \times N$. For 1 element of S i.e S_{1j}, the corresponding elements Q and j^{th} column of K^T are multiplied and then added. So, total number of FLOPs done to get 1 element of S are $2d$. Thus, for N elements of S, total FLOPs done are $2Nd$.

2. Compute $P = \text{softmax(S)}$. To do the softmax of S, each element is first exponentiated. Hence, N FLOPs for exponentiating. The exponentiated values are added across the row i.e N additions as there is only 1 row. Then, each exponentiated entry is divided by the sum.. So, again N FLOPs for this. Hence, Total number of FLOPs done to get P is given by:
$$N + N + N = 3N$$

3. Compute $O = PV$. P is a $(1 \times N)$ matrix, V is of dimensions $(N \times d)$ and O will be of dimension $(1 \times d)$. For 1 element of O i.e O_{1j}, the corresponding elements of P and j^{th} column of V are multiplied and then added. So, total number of FLOPs done to get 1 element of O are $2N$. Thus, for $1 \times d$ elements of O, total FLOPs done are $2Nd$.

4. Therefore, total number of FLOPs in Decode attention computation is given by:
$$2Nd + 3N + 2Nd = N(4d+3).......(3)$$

5. If we have a batch size of B i.e B new tokens are generated at once i.e Q is a $B \times d$ matrix then, S becomes $B \times N$ matrix and O becomes $B \times d$ matrix. So, total number of FLOPs for a batch size of B will be:
$$2BNd + 3BN + 2BNd = NB(4d+3).......(4)$$

Total Memory Accesses (in Bytes, assuming each element is FP16 i.e 2 Bytes) in Decode Attention Computation are calculated as follows:

1. Read Q, K from HBM. Q, K are of dimensions $(1 \times d)$ & $(N \times d)$ respectively. So, $d + Nd$ memory accesses to read them.

2. Write S back to HBM. S is of dimension $(1 \times N)$. So, N memory accesses.

3. Load S again, compute P and write P to HBM. $2N$ memory accesses.
4. Load P and V. P is of dimension $(1 \times N)$ and V is of dimension $(N \times d)$. So, $N + Nd$ memory accesses.
5. Compute O of dimension $(1 \times d)$ and write it to HBM, Thus, d memory-accesses.
6. Here, we can see that the major computation is of repeatedly loading the KV cache to decode each token.
7. Therefore, total bytes of memory accesses in Decode attention computation is given by:
$$2 * (d + Nd + N + 2N + N + Nd + d)$$
$$= 4Nd + 8N + 4d = 4(Nd + d + 2N)\text{Bytes.......(5)}$$
8. If we have a batch size of B i.e B new tokens are generated at once then the loading of K and V cache is done once for all batches (to generate B tokens), but Q is $B \times d$, S is $B \times N$, P is $B \times N$ and O is $B \times d$. So, bytes of memory accesses for batch size B becomes:
$$2 * (Bd + Nd + BN + 2BN + BN + Nd + Bd)$$
$$= 4Nd + 8BN + 4Bd\text{Bytes.......(6)}$$

From Equation (4) and (6), the Arithmetic Intensity for Decode Attention Computation for batch size B is given by:
$$AI = \frac{\#\text{FLOPs}}{\#\text{bytes accessed}}$$
$$AI = \frac{NB(4d+3)}{4Nd + 8BN + 4Bd} \text{ FLOPs/Bytes}$$

1.3.3

Following table shows the FLOPs, Memory Accesses (Bytes) and the Arithmetic Intensity (FLOPs/Bytes) for both Prefill and Decode Attention for Batch Size 1 is given below:

	Prefill Attention	**Decode Attention**
Total Operations (FLOPs)	$N^2(4d+3)$	$N(4d+3)$
Memory Accesses (Bytes)	$8N(N+d)$	$4(Nd+d+2N)$
Arithmetic Intensity (FLOPs/Bytes)	$\frac{N(4d+3)}{8(N+d)}$	$\frac{N(4d+3)}{4Nd+8N+4d}$

1.4

- For $N = 1024$ & $d = 128$, the Arithmetic Intensity for Prefill Attention Computation using formula obtained above is calculated below:
$$AI = \frac{N(4d+3)}{8(N+d)} \text{ FLOPs/Bytes}$$

$$AI = \frac{1024((4*128)+3)}{8(1024+128)} \text{ FLOPs/Bytes}$$

$$AI = \frac{527,360}{9,216} \text{ FLOPs/Bytes}$$

$$AI = 57 \text{ FLOPs/Bytes}$$

- For $N = 1024$ & $d = 128$ with batch size $B = 1$, the Arithmetic Intensity for Decode Attention Computation using formula obtained above is calculated below:

$$AI = \frac{NB(4d+3)}{4(Nd+Bd+2BN)} \text{ FLOPs/Bytes}$$

$$AI = \frac{1024((4*128)+3)}{4((1024*128)+2048+128)} \text{ FLOPs/Bytes}$$

$$AI = \frac{527,360}{532,992} \text{ FLOPs/Bytes}$$

$$AI \approx 1 \text{ FLOPs/Bytes}$$

- The plot of variation of Arithmetic Intensity(AI) with Context Length (N) for Prefill Attention Computation is shown below:

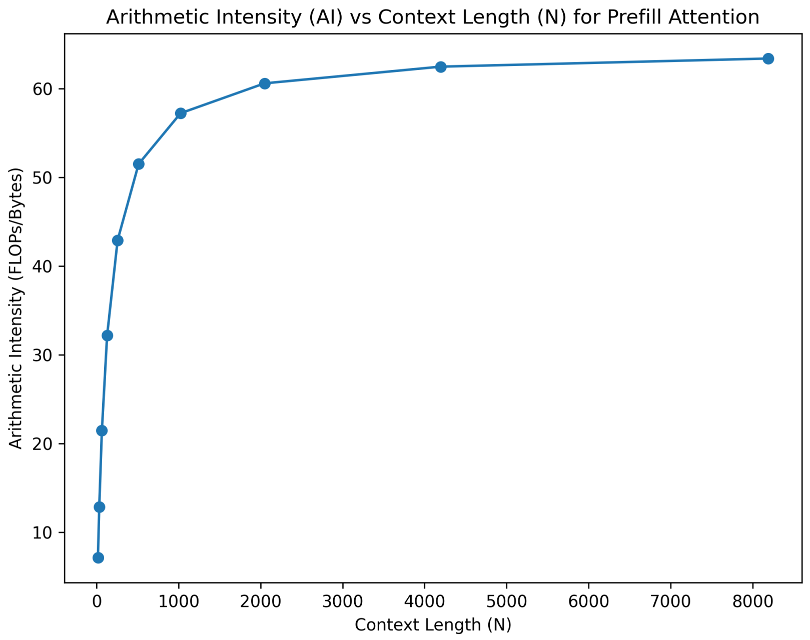

Figure 2: AI Vs Context Length (N) For Prefill Attention

As the Context Length increases, the Arithmetic Intensity first increases for smaller values of N then start becoming constant. This is because from the arithmetic intensity formula derived above:

$$AI = \frac{N(4d+3)}{8(N+d)} \text{ FLOPs/Bytes}$$

As N increases, the term d becomes negligible in front of it, so, we can remove it and the formula becomes:

$$AI = \frac{4Nd}{8N} \approx \frac{d}{2} \text{ FLOPs/Bytes}$$

Hence, here, the graph seems to be saturated at 64 AI, since $d = 128$.

- The plot of variation of Arithmetic Intensity(AI) with Context Length (N) on Batch Size $B = 1$ for Decode Attention Computation is shown below:

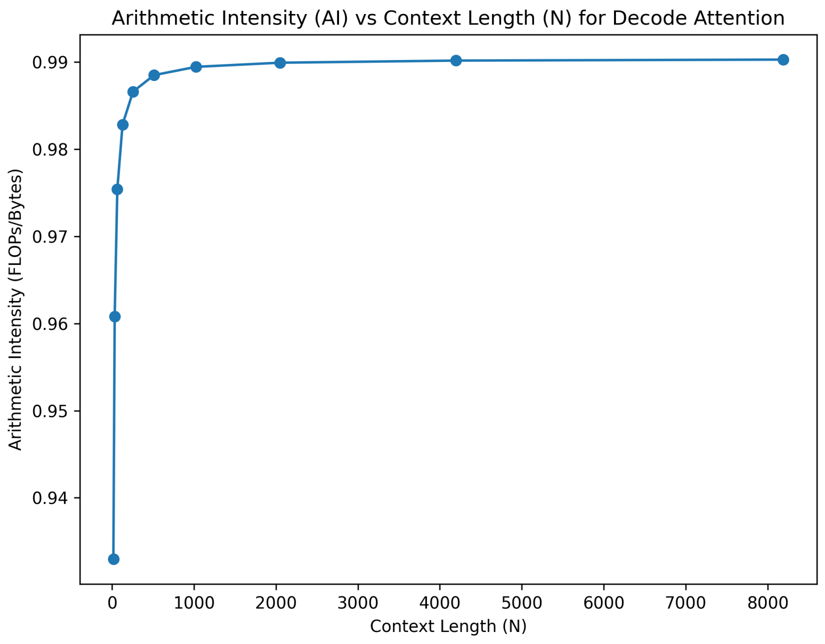

Figure 3: AI Vs Context Length(N) For Decode Attention

- The plot of variation of Arithmetic Intensity(AI) with Batch Size(B) for Decode Attention Computation is shown below:
  As Batch Size B increases (for large B), then for a fixed N and d (here $N = 1024$ & $d = 128$) AI intensity becomes:

$$AI = \frac{NB(4d)}{4B(2N+d)} \text{ FLOPs/Bytes}$$

$$AI = \frac{Nd}{2N+d} \text{ FLOPs/Bytes}$$

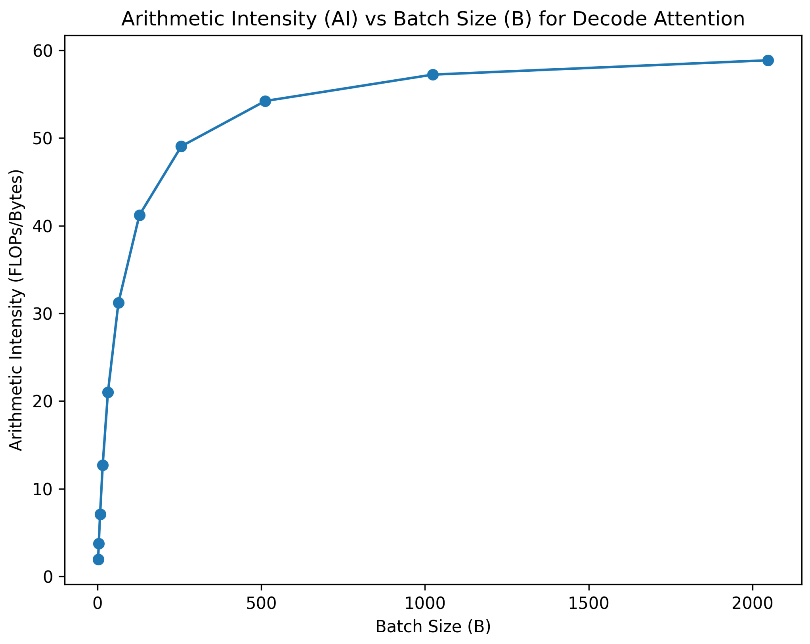

Figure 4: AI Vs Batch Size(B) For Decode Attention

$$AI = \frac{1024 * 128}{2048 + 128} \approx 60 \text{ FLOPs/Bytes}$$

Hence, the graph seems to be saturated at ≈ 60 FLOPs/bytes.

2 Problem 2

2.1

2-Layer Decoder-Only Transformer model is fully implemented in the file `transformer.py`. It has the ability to autoregressively generate n tokens. On running the file, following output was visible:

```
Successfully loaded model state dictionary from model_state_dict.pt
Weights loaded from model_state_dict.pt
Weights match for layer 0 Wq:  True
Weights match for layer 1 Wk:  True
Test cases loaded from test_cases.json
Evaluating model...
Evaluation Results:
All tests passed:  True
Pass rate:  100.00% (10/10)
Benchmarking performance...
Performance Results:
Without KV cache:  0.0089 seconds
With KV cache:  0.0065 seconds
Speedup:  1.38x
```

2.2

File is successfully executed with `--mode evaluate`. Logits generated after the initial prompt processing matched the expected values for a given model weight file. All the 10 test cases were passed. Output seen on the terminal was as follows:

```
Successfully loaded model state dictionary from model_state_dict.pt
Test cases loaded from test_cases.json
Evaluation Results:
Num test cases:  10
All tests passed:  True
Pass rate:  100.00% (10/10)
```

2.3

File is successfully executed with `--mode kv_evaluate`. Logits are validated with and without KV Caching. The speedup observed on using KV Cache as compared to without KV Cache observed was $1.4x$. Output seen on the terminal was as follows:

```
Successfully loaded model state dictionary from model_state_dict.pt
Test cases loaded from test_cases.json
Evaluation Results:
```

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

2.4

The Sequence length was varied from $[10, 50, 100, 500, 1000]$ and the latency with and without using KV Caching of prior tokens using the command`--mode benchmark` is as follows:

```
Successfully loaded model state dictionary from model_state_dict.pt
Benchmarking...
Results:
Without KV cache:  0.0088 seconds
With KV cache:  0.0066 seconds
Speedup:  1.34x
Benchmarking at different sequence lengths...
SeqLen=10 NoCache=0.0181s WithCache=0.0128s
SeqLen=50 NoCache=0.0218s WithCache=0.0130s
SeqLen=100 NoCache=0.0259s WithCache=0.0132s
SeqLen=500 NoCache=0.0894s WithCache=0.0174s
SeqLen=1000 NoCache=0.2559s WithCache=0.0251s
```

The plot of Latency with and Without KV Caching of Prior Tokens as a function of Sequence Length (N) is given below:

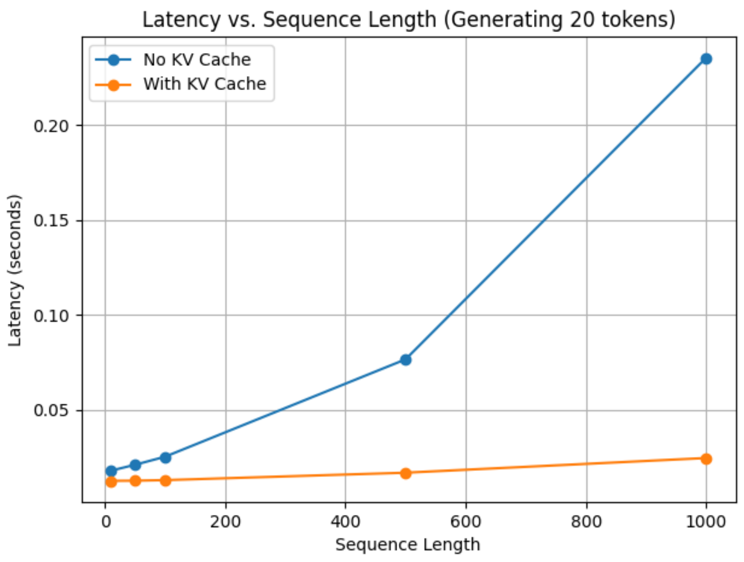

Figure 5: AI Vs Batch Size For Decode Attention

References

- https://huggingface.co/meta-llama/Meta-Llama-3-8B-Instruct
- https://adithyask.medium.com/from-7b-to-8b-parameters-understanding-weight-matrix-changes-i