E0294: SYSTEMS FOR MACHINE LEARNING

Assignment 4

AYUSH KUMAR SINGH
SR No: 24851
MTech CSA

Problem 1:

KV cache, batch size, and arithmetic intensity calculation: Write a script (in any language of your choice), that given model, hardware, and input specs enumerated below, computes the size of the KV cache per request and the maximum permissible batch size on the given hardware.
The following are the inputs to the program :
Model Specification
Number of Layers
Number of Query heads
Number of KV heads
Embedding Dimension (h1)
Inner Dimension (h2)
Vocab Size
Device Specification
GPU Memory
Input Specification
Context Length (s)
Quantization
Weights bytes (eg., 2 if 16bit)
KV cache bytes
Consider following parameters
The default values populated are for Llama3-8B.

Number of Layers	32
Number of Query heads	32
Number of KV heads	8
Embedding Dimension (h1)	4096
Inner Dimension (h2)	14336
Vocab Size	128256
GPU Memory	80GB

a) Find the maximum batch size that can fit on a GPU given the above specifications. Please output a table in the below format and show calculation for how you find KV cache per request.

Implementation : ques1a.py

```
import math
from tabulate import tabulate

# Model Specification
NUM_LAYERS      = 32
NUM_QUERY_HEADS  = 32
NUM_KV_HEADS    = 8
EMBED_DIM       = 4096
INNER_DIM       = 14336
VOCAB_SIZE      = 128256
GPU_MEMORY_GB   = 80.0  # 80GB

# Precision & KV cache
WEIGHTS_BYTES   = 2  # 16-bit precision
KV_CACHE_BYTES  = 2  # 16-bit precision
CONTEXT_LENGTH  = 1024
```

```
# Computing total model parameters
embedding_params = VOCAB_SIZE * EMBED_DIM
ffn_params_per_layer = 2 * INNER_DIM * EMBED_DIM + INNER_DIM + EMBED_DIM
total_ffn_params = NUM_LAYERS * ffn_params_per_layer

head_dim = EMBED_DIM // NUM_QUERY_HEADS  # Per-head dimension
kv_dim_total = NUM_KV_HEADS * head_dim   # Total KV dimension

attention_params_per_layer = EMBED_DIM * (EMBED_DIM + 2 * kv_dim_total)
total_attention_params = NUM_LAYERS * attention_params_per_layer

output_layer_params = VOCAB_SIZE * EMBED_DIM

# Summing everything
total_params = embedding_params + total_ffn_params + total_attention_params + output_layer_params

# Computing total parameter memory (GB)
total_param_mem_bytes = total_params * WEIGHTS_BYTES
total_param_mem_gb = total_param_mem_bytes / (1024 ** 3)

# KV Cache memory per request (batch size = 1)
kv_dim_per_token = NUM_KV_HEADS * head_dim * 2  # For K & V
total_kv_bytes = kv_dim_per_token * CONTEXT_LENGTH * NUM_LAYERS * KV_CACHE_BYTES
kv_cache_gb = total_kv_bytes / (1024 ** 3)

# Computing max batch size
leftover_mem_gb = GPU_MEMORY_GB - total_param_mem_gb
max_batch_size = math.floor(leftover_mem_gb / kv_cache_gb) if kv_cache_gb > 0 else 0

# Printing the result in a single table
print("===== Part (a): Single GPU (80 GB) - Llama3–8B =====")
print(tabulate([
    ["Total model parameters", f"{total_params:.2e}"],
    ["Total parameter memory (GB)", f"{total_param_mem_gb:.3f}"],
    ["KV cache per request (GB)", f"{kv_cache_gb:.3f}"],
    ["Max batch size possible", str(max_batch_size)]
], headers=["Item", "Value"], tablefmt="grid"))
```

In this part, we have used the following formulas for calculation –

1. Model Parameters Calculation

1.1. Embedding Layer Parameters

$$Total\ Embedding\ Parameters = Vocabulary\ Size \times Embedding\ Dimension$$

1.2. Feed-Forward Network (FFN) Parameters

$$FFN\ Parameters\ per\ layer = 2 \times (EMBED\_DIM \times INNER\_DIM) + INNER\_DIM + EMBED\_DIM$$

$$Total\ FFN\ Parameters = NUM\_LAYERS \times FFN\ Parameters\ per\ layer$$

1.3. Attention Layer Parameters

$$head\_\dim = \frac{EMBED\_DIM}{NUM\_QUERY\_HEADS}$$

$$kv\_dim\_total = NUM\_KV\_HEADS \times head\_dim$$

$$Attention\ Parameters\ per\ Layer = EMBED\_DIM \times (EMBED\_DIM + 2 \times kv\_dim\_total)$$

$$Total\ Attention\ Parameters = NUM\_LAYERS \times Attention\ Parameters\ per\ Layer$$

1.4. Output Layer Parameters

$$Output\ Layer\ Parameters = VOCAB\_SIZE \times EMBED\_DIM$$

1.5. Total Model Parameters

$$total\ model\ parameters = Total\ embedding\ parameters + total\ FFN\ Parameters + Total\ attention\ Parameters + output\ layer\ parameters$$

2. Total memory calculation

$$Total\ Memory\ (bytes) = Total\ Parameters \times WEIGHTS\_BYTES$$

Converting it to GB:

$$Total\ Memory\ (GB) = \frac{Total\ Memory\ (bytes)}{1024^3}$$

3. KV Cache Memory per Request

$$KV\ per\ token = NUM_{KV_{HEADS}} \times head_{dim} \times 2$$

$$Total\ KV\ Cache\ Memory\ (bytes) = KV\ per\ token \times CONTEXT\_LENGTH \times NUM\_LAYERS \times KV\_CACHE\_BYTES$$

$$KV\ Cache\ Memory\ (GB) = \frac{Total\ KV\ Cache\ Memory\ (bytes)}{1024^3}$$

4. Maximum Batch size Calculation

$$Leftover\ Memory = GPU\ Memory - Total\ Parameter\ Memory$$

$$Max\ Batch\ Size = \frac{Leftover\ Memory\ (GB)}{KV\ Cache\ Memory\ (GB) per\ request}$$

Running the program :

>>> python .\ques1a.py

For context length = 1024

```
PS D:\users\ayush\IISC coursework\sem 2\SML\Assignments\Assignment 4\ques1> python .\ques1a.py
===== Part (a): Single GPU (80 GB) - Llama3-8B =====
+-----------------------------+------------+
| Item                        |      Value |
+=============================+============+
| Total model parameters      |   5.61e+09 |
+-----------------------------+------------+
| Total parameter memory (GB) |  10.458    |
+-----------------------------+------------+
| KV cache per request (GB)   |   0.125    |
+-----------------------------+------------+
| Max batch size possible     | 556        |
+-----------------------------+------------+
```

For context length = 2048

```
PS D:\users\ayush\IISC coursework\sem 2\SML\Assignments\Assignment 4\ques1> python .\ques1a.py
===== Part (a): Single GPU (80 GB) - Llama3-8B =====
+-----------------------------+------------+
| Item                        |      Value |
+=============================+============+
| Total model parameters      |   5.61e+09 |
+-----------------------------+------------+
| Total parameter memory (GB) |  10.458    |
+-----------------------------+------------+
| KV cache per request (GB)   |   0.25     |
+-----------------------------+------------+
| Max batch size possible     | 278        |
+-----------------------------+------------+
```

For context length = 4096

```
PS D:\users\ayush\IISC coursework\sem 2\SML\Assignments\Assignment 4\ques1> python .\ques1a.py
===== Part (a): Single GPU (80 GB) - Llama3-8B =====
+-----------------------------+------------+
| Item                        |      Value |
+=============================+============+
| Total model parameters      |   5.61e+09 |
+-----------------------------+------------+
| Total parameter memory (GB) |  10.458    |
+-----------------------------+------------+
| KV cache per request (GB)   |   0.5      |
+-----------------------------+------------+
| Max batch size possible     | 139        |
+-----------------------------+------------+
```

Results :

Context Length	Total parameter Memory (GB)	KV cache per request (GB)	Max batch size possible
1024	10.458	0.125	556
2048	10.458	0.25	278
4096	10.458	0.5	139

Since the total number of parameters are independent of the context length, hence Total parameter memory remains constant. The results here show that the KV Cache size scales linearly with the context length. Larger context lengths reduce the maximum batch size since more memory per request is

needed. Memory for model parameters is constant (~10.458 GB) and does not depend on batch size. Batch size is dictated by available GPU memory after accounting for the model and KV cache. The mathematical reasoning behind these results is straightforward: As the context length doubles, KV cache per request doubles, and the max batch size halves.

---

b) Repeat the same exercise for a given Tensor Parallelism (TP) dimension 'X'. Assume the model and KV cache memory requirements are divided equally among X GPUs, and recalculate the maximum batch size.

Implementation : ques1b.py

```
import math
from tabulate import tabulate
# ====================== Part (b): Tensor Parallelism ======================

# Defining the Tensor Parallelism (TP) dimension
TP_DIM = 2

# Computing parameter memory per GPU
param_mem_per_gpu = total_param_mem_gb / TP_DIM

# Computing KV cache memory per GPU (for batch size = 1)
kv_cache_per_gpu = kv_cache_gb / TP_DIM

# Computing max batch size per GPU
leftover_mem_gpu = GPU_MEMORY_GB - param_mem_per_gpu
max_batch_size_tp = math.floor(leftover_mem_gpu / kv_cache_per_gpu) if kv_cache_per_gpu > 0 else 0

# Printing results for Part (b)
print(f"\n===== Part (b): Tensor Parallelism (TP={TP_DIM}) - Llama3–8B =====")
print(tabulate([
    ["Total parameter memory per GPU (GB)", f"{param_mem_per_gpu:.3f}"],
    ["KV cache per GPU (GB)", f"{kv_cache_per_gpu:.3f}"],
    ["Max batch size per GPU", str(max_batch_size_tp)]
], headers=["Item", "Value"], tablefmt="grid"))
```

Running the program:

>>> python .\ques1b.py

For tensor parallelism dimension = 2

```
PS D:\users\ayush\IISC coursework\sem 2\SML\Assignments\Assignment 4\ques1> python .\ques1b.py

===== Part (b): Tensor Parallelism (TP=2) - Llama3-8B =====
+-----------------------------------+-----------+
| Item                              |     Value |
+===================================+===========+
| Total parameter memory per GPU (GB) |     5.229 |
+-----------------------------------+-----------+
| KV cache per GPU (GB)             |     0.062 |
+-----------------------------------+-----------+
| Max batch size per GPU            | 1196      |
+-----------------------------------+-----------+
```

For tensor parallelism dimension = 4

```
PS D:\users\ayush\IISC coursework\sem 2\SML\Assignments\Assignment 4\ques1> python .\ques1b.py

===== Part (b): Tensor Parallelism (TP=4) - Llama3-8B =====
+-----------------------------------+-----------+
| Item                              |     Value |
+===================================+===========+
| Total parameter memory per GPU (GB) |     2.615 |
+-----------------------------------+-----------+
| KV cache per GPU (GB)             |     0.031 |
+-----------------------------------+-----------+
| Max batch size per GPU            | 2476      |
+-----------------------------------+-----------+
```

for tensor parallelism dimension = 8

```
PS D:\users\ayush\IISC coursework\sem 2\SML\Assignments\Assignment 4\ques1> python .\ques1b.py

===== Part (b): Tensor Parallelism (TP=8) - Llama3-8B =====
+-----------------------------------+-----------+
| Item                              |     Value |
+===================================+===========+
| Total parameter memory per GPU (GB) |     1.307 |
+-----------------------------------+-----------+
| KV cache per GPU (GB)             |     0.016 |
+-----------------------------------+-----------+
| Max batch size per GPU            | 5036      |
+-----------------------------------+-----------+
```

Results:

TP Dimension	Total parameter memory per GPU (GB)	KV cache per GPU (GB)	Max batch size per GPU
2	5.229	0.062	1196
4	2.615	0.031	2476
8	1.307	0.016	5036

With higher Tensor Parallelism, each GPU handles fewer model parameters which results in the decrease of Model memory per GPU. KV cache is also split across GPUs which again results in the reduction of KV cache per GPU. Since each GPU has more available memory for batch processing, max batch size per GPU increases. The batch size approximately doubles when TP doubles (since KV cache memory requirement per GPU is halved).

---

c) Write down the detailed formula for total flops and total memory access(bytes) to calculate arithmetic intensity(flops/memory_bytes) for prefill and decode attention function. Assume the following attention implementation. Head dimension is d and context length is N.

Algorithm 0 Standard Attention Implementation

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.

Write your calculations, specifically how you arrive at the total flops and memory access for both prefill and decode attention in terms of N and d.

Step 1: Understanding the Given Algorithm (Standard Attention Implementation)

The standard attention implementation in the image follows these steps:

1. Load Q,K from HBM, compute S = QK^T , write S to HBM.
2. Read S from HBM, compute P = softmax(S), write P to HBM.
3. Load P,V from HBM, compute O = PV, write O to HBM.
4. Return O.

We'll now compute the FLOPs and memory access (bytes) for both prefill and decode attention.

Step 2: Compute Total FLOPs

(A) Prefill Attention (Full Context Attention)

1. Computing Attention Scores ($S = Q\,K^T$)

$$Q \in N \times d, K \in N \times d$$

2. Matrix multiplication:

$$(N \times d) \times (d \times N) = N^2 d$$

Since matrix multiplication requires 2 FLOPs per multiply-accumulate:

$$FLOPs = 5\,N^2$$

3. Computing softmax: ($P = softmax(S)$)

$$FLOPs = 5\,N^2$$

4. Computing Output matrix($O = PV$)

$$(N \times N) \times (N \times d) = N^2 d$$

$$FLOPs = 2N^2 d$$

Total FLOPs for prefill attention:

$$4N^2 d + 5N^2 d$$

(B) Decode Attention (One Step Attention)

1. Computing Attention Scores ($S_t = Q_t\,K^T$)

$$Q_t \in N \times d, K \epsilon N \times d$$

$$FLOPs = 2Nd$$

2. Computing softmax: ($P_t = softmax(S_t)$)

$$FLOPs = 5\,N$$

3. Computing Output matrix($O = P_t V$)

$$FLOPs = 2Nd$$

Total FLOPs for decode attention:

$$4Nd + 5N$$

Step 3 : Compute Total Memory Access (Bytes)

(A) Prefill Attention

$$Total\ memory\ Access = 4Nd + 3N^2 + Nd$$

(B) Decode Attention

$$Total\ memory\ Access = (2N + 1)d + 3N$$

Step 4 : Compute Arithmetic Intensity

$$Arithmetic\ Intensity = \frac{FLOPs}{Memory\ Access\ (bytes)}$$

(A) Prefill Attention

$$\frac{4N^2d + 5N^2}{4Nd + 3N^2 + Nd} = O(d)$$

Since O(d) is large for large models, prefill attention is compute-bound.

(B) Decode Attention

$$\frac{4Nd + 5N}{(2N + 1)d + 3N} = O(1)$$

since this is constant, decode attention is memory-bound.

For batch size B, the formulae changes to-

$$= \frac{4BNd + 5BN}{(2N + 1)d + 3NB}$$

---

d) Calculate arithmetic intensity using the formula above, given N=1024 and d=128. Plot the arithmetic intensity as context length (N) varies (for prefill, batch size 1) and as batch size varies (for decode). Crisply explain your observations and reason.

Implementation : ques1d.py

```python
import math
import matplotlib.pyplot as plt

# Computing Arithmetic Intensity for prefill attention (batch=1).
def compute_ai_prefill(seq_length, model_dim, dtype_size=2):
    operations = 4 * (seq_length**2) * model_dim + 5 * (seq_length**2)
    memory_accesses = 4 * seq_length * model_dim + 3 * (seq_length**2) + seq_length * model_dim
    memory_bytes = memory_accesses * dtype_size
    return operations / memory_bytes if memory_bytes > 0 else 0

# Computing Arithmetic Intensity for single-step decode attention.
def compute_ai_decode(seq_length, model_dim, dtype_size=2):
    operations = 4 * seq_length * model_dim + 5 * seq_length
    memory_accesses = (2 * seq_length + 1) * model_dim + 3 * seq_length
    memory_bytes = memory_accesses * dtype_size
    return operations / memory_bytes if memory_bytes > 0 else 0

# Computing Arithmetic Intensity for batched decode attention
def compute_ai_decode_batch(seq_length, model_dim, batch_size, dtype_size=2):
    operations = batch_size * (4 * seq_length * model_dim + 5 * seq_length)
    memory_accesses = (2 * seq_length + 1) * model_dim + 3 * seq_length * batch_size
    memory_bytes = memory_accesses * dtype_size
    return operations / memory_bytes if memory_bytes > 0 else 0

# main function to calculate AI and plot graphs
def analyze_arithmetic_intensity():
    print("========== Arithmetic Intensity Analysis ==========")

    seq_fixed = 1024
    dim_fixed = 128
    data_precision = 2  # 2 bytes per element (FP16)

    #Computing AI for a specific scenario
    prefill_ai_fixed = compute_ai_prefill(seq_fixed, dim_fixed, data_precision)
    decode_ai_fixed  = compute_ai_decode(seq_fixed, dim_fixed, data_precision)

    print(f"For seq_length={seq_fixed}, model_dim={dim_fixed}, FP16 (2 bytes/element):")
    print(f"  Prefill AI = {prefill_ai_fixed:.3f} FLOPs/byte")
    print(f"  Decode  AI = {decode_ai_fixed:.3f} FLOPs/byte")

    #AI vs. Sequence Length for Prefill
    seq_values = [128, 256, 512, 1024, 2048, 4096]
    prefill_ai_values = [compute_ai_prefill(n, dim_fixed, data_precision) for n in seq_values]

    #AI vs. Batch Size for Decode (Exponential Growth)
    batch_sizes = [1, 2, 4, 8, 16, 32, 64, 128, 256]
```

```
    decode_ai_batch = [compute_ai_decode_batch(seq_fixed, dim_fixed, B, data_precision) for B in
batch_sizes]

    # Plot Prefill AI vs. Sequence Length
    plt.figure(figsize=(8,5))
    plt.plot(seq_values, prefill_ai_values, marker='o', linestyle='-', color='blue', label="Prefill AI")
    plt.title("Prefill AI vs. Sequence Length (batch=1)")
    plt.xlabel("Sequence Length (N)")
    plt.ylabel("Arithmetic Intensity (FLOPs/Byte)")
    plt.grid(True)
    plt.legend()
    plt.savefig("prefill_ai_vs_seq_length.png")
    plt.show()

    # Plot Decode AI vs. Batch Size
    plt.figure(figsize=(8,5))
    plt.plot(batch_sizes, decode_ai_batch, marker='s', linestyle='-', color='red', label="Decode AI")
    plt.title("Decode AI vs. Batch Size")
    plt.xlabel("Batch Size (B)")
    plt.ylabel("Arithmetic Intensity (FLOPs/Byte)")
    plt.yscale("log")  # Log scale to highlight exponential growth
    plt.grid(True)
    plt.legend()
    plt.savefig("decode_ai_vs_batch_size.png")
    plt.show()

# Execute the analysis
analyze_arithmetic_intensity()
```

After running the program, the output generated is –

```
PS D:\users\ayush\IISC coursework\sem 2\SML\Assignments\Assignment 4\ques1> python .\ques1d.py
========== Arithmetic Intensity Analysis ==========
For seq_length=1024, model_dim=128, FP16 (2 bytes/element):
  Prefill AI = 71.310 FLOPs/byte
  Decode  AI = 0.998 FLOPs/byte
```

<u>Prefill AI Calculation</u>

Formula:

$$= \frac{4N^2d + 5N^2}{4Nd + 3N^2 + Nd}$$

$$= \frac{4(1024)^2(128) + 5(1024)^2}{4(1024)(128) + 3(1024)^2 + (1024)(128)}$$

$$= \frac{542113792}{3801088} \approx 71.3\ FLOPs/byte$$

Decode AI Calculation

Formula:

$$= \frac{4Nd + 5N}{(2N+1)d + 3N}$$

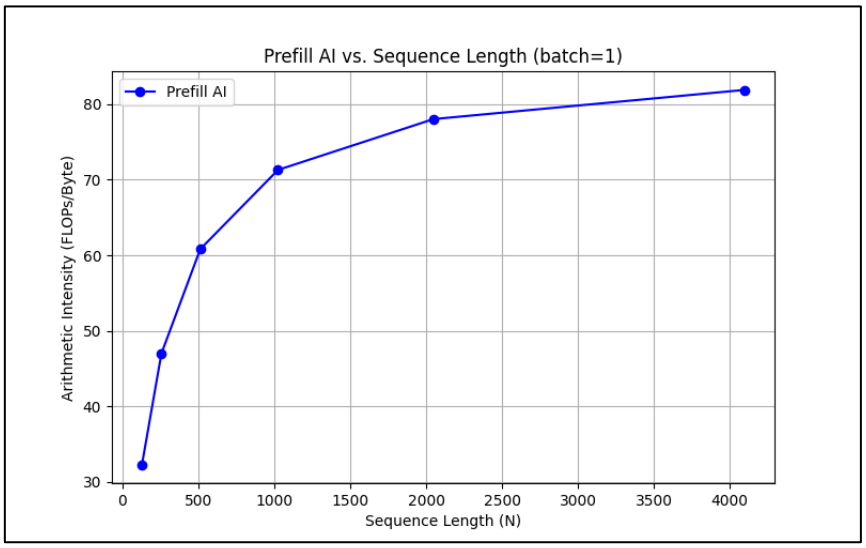
$$= \frac{4(1024)(128) + 5(1024)}{(2(1024)+1)(128) + 3(1024)}$$

$$= \frac{529408}{265344} \approx 1$$

Prefill AI (~71.3) is much higher, confirming that Prefill Attention is compute-bound and the Decode AI (~1.0) is close to a constant, proving that Decode Attention is memory-bound.

The graphs obtained are –

1. **Prefill AI vs Sequence Length (batch = 1)**



$$Prefill\ AI = \frac{4N^2d + 5N^2}{4Nd + 3N^2 + Nd}$$

For small N, the denominator is dominated by 4Nd + (O(Nd)), while the numerator grows as $O(N^2d)$ This leads to:

$$AI \approx O(N)$$

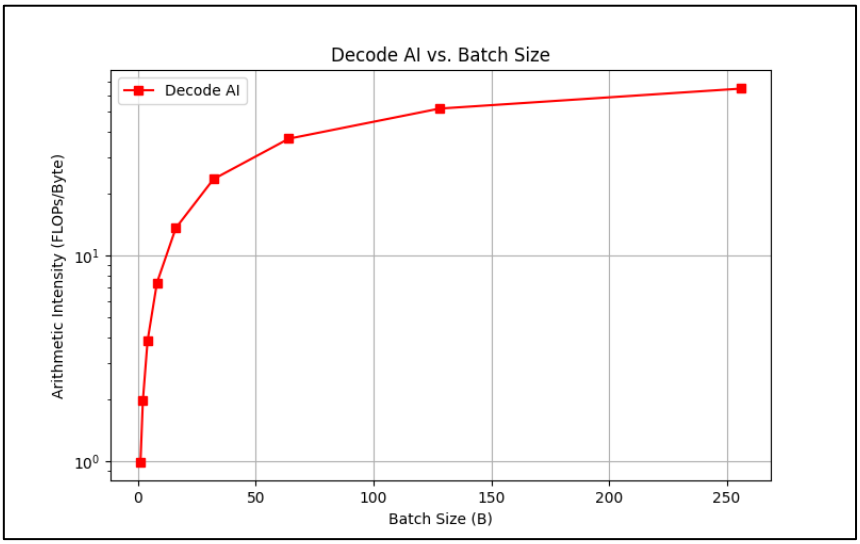
So for small N, AI increases sharply.

For large N, the dominant terms in both numerator and denominator are $O(N^2d)$ and $O(N^2)$, respectively, making AI:

$$AI \approx O(d) = constant$$

Thus, for large N, AI flattens out. Thus result shows that Prefill AI increases with sequence length, stabilizing around O(d) as sequence grows.

2. **Decode AI vs Batch Size**



$$Decode\ AI = \frac{4BNd + 5BN}{(2N+1)d + 3NB}$$

For Small B, the numerator (4BNd + 5BN) and denominator ((2N+1)d + 3NB) are both linear in B. Hence, AI increases initially as FLOPs increase faster than memory accesses. For Large B, the denominator grows due to the 3NB term, leading to AI saturating at a constant value, This suggests that Decode AI first increases and then plateaus. Thus result shows that Decode AI first increases and then becomes constant.

Problem 2:

Implement a simple 2 layer decoder-only transformer model in Python, using the provided skeleton code.
(a) Implement all the TODO sections in the provided code to create a functioning transformer model with the ability to autoregressively generate 'n' tokens.
(b) Correct implementation of the model with logits generated after the initial prompt processing matching the expected values for a given model weight file.
Run the file with mode –evaluate
(c) Implement KV caching to cache the generated KV from previous iterations in every subsequent iter instead of naively processing the entire sequence in each step. Validate that the logits with and without KV caching match.
Run the file with mode –kv_evaluate
(d) Vary the sequence length from [10, 50, 100, 500, 1000] and compare the performance(latency to generate 'n' tokens given a prompt) with and without kv caching of prior tokens. Plot a graph that shows latency with and without KV as a function of sequence length.
Run the file with mode –benchmark (8)
(Partial : Half the points if you are not able to implement the generate method, but can demonstrate the impact of KV cache with just prompt processing)

Implementation : transformer_skeleton.py

Step1 : Importing necessary files

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import time
import json
import argparse
import os
from collections import OrderedDict
```

these files are important for the implementation of the transformer.

Step2 :Single Head Attention class

```
class SingleHeadAttention(nn.Module):
    def __init__(self, d_model):
        super().__init__()
        self.d_model = d_model
        self.W_k = nn.Linear(d_model, d_model, bias=False)
        self.W_v = nn.Linear(d_model, d_model, bias=False)
        self.W_q = nn.Linear(d_model, d_model, bias=False)
        self.W_o = nn.Linear(d_model, d_model, bias=False)

    def forward(self, z, past_kv=None, use_cache=False):
        batch_size, seq_len, _ = z.shape

        key = self.W_k(z)
        value = self.W_v(z)
        query = self.W_q(z)

        if use_cache and past_kv is not None:
            past_k, past_v = past_kv
            key = torch.cat([past_k, key], dim=1)
            value = torch.cat([past_v, value], dim=1)

        attn_scores = torch.bmm(query, key.transpose(1, 2)) / (self.d_model ** 0.5)

        # Create causal mask accounting for past_kv
        if past_kv is not None:
            past_len = past_kv[0].shape[1]
        else:
            past_len = 0

        mask = torch.ones(seq_len, seq_len + past_len, dtype=torch.bool, device=z.device)
        mask = torch.triu(mask, diagonal=1)

        attn_scores = attn_scores.masked_fill(mask, float('-inf'))
        attn_weights = F.softmax(attn_scores, dim=-1)
        context = torch.bmm(attn_weights, value)
        output = self.W_o(context)

        new_kv = (key, value) if use_cache else None
        # Placeholder return - replace with your implementation
        return output, new_kv
```

Implements a single-head self-attention mechanism, a fundamental component in Transformer models. The forward() method computes the key, value, and query projections by applying the respective linear layers to the input z. For KV caching, if caching is enabled (use_cache=True) and previous key-value pairs are provided (past_kv is not None), the current keys and values are concatenated with the past

keys and values along the sequence length dimension. This allows the model to reuse previously computed attention information, enhancing efficiency during generation. The class also involves attention scores calculation by computing the raw attention scores by performing a batch matrix multiplication between the query and the transpose of the key tensors. The result is scaled by the square root of d_model to maintain stable gradients, as suggested in the original Transformer paper. It also involves Creating a causal mask to prevent the model from attending to future tokens. This is achieved by generating an upper triangular matrix (torch.triu) filled with ones above the main diagonal, indicating positions to be masked.

Step 3: Feedforward class

```
class FeedForward(nn.Module):
    def __init__(self, d_model, d_ff):
        super().__init__()
        # TODO: Implement feed-forward network
        # Create two linear layers
        self.fc1 = nn.Linear(d_model, d_ff)
        self.fc2 = nn.Linear(d_ff, d_model)
        # Your code here
        self.dropout = nn.Dropout(0.1)

    def forward(self, z):
        # TODO: Implement the forward pass for feed-forward network
        # Two linear layers with ReLU activation in between
        # Your code here
        z = self.fc1(z)
        z = F.relu(z)
        z = self.dropout(z)
        z = self.fc2(z)
        # Placeholder return - replace with your implementation
        return z
```

It implements a position-wise feed-forward network, applied independently to each position in the sequence. The forward method in this applies the first linear transformation (fc1) to the input z, applies the ReLU activation function to introduce non-linearity, applies dropout to the activated values to prevent overfitting, applies the second linear transformation (fc2) to project back to the original embedding space and returns the transformed tensor.

Step 4 : decoderLayer class

```
class DecoderLayer(nn.Module):
    def __init__(self, d_model, d_ff, dropout=0.1):
        super().__init__()
        # TODO: Implement decoder layer
        # Create attention, feed-forward, layer norms, and dropout

        # Your code here
        self.attention = SingleHeadAttention(d_model)
        self.feed_forward = FeedForward(d_model, d_ff)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, z, past_kv=None, use_cache=False):
        # TODO: Implement the forward pass for decoder layer
        # 1. Apply layer norm before attention
        # 2. Apply attention followed by dropout
        # 3. Apply layer norm before feed-forward
        # 4. Apply feed-forward then residual connection
```

```
        # Your code here
        norm_z = self.norm1(z)
        attn_out, new_kv = self.attention(norm_z, past_kv, use_cache)
        z = z + self.dropout(attn_out)

        # Feed-forward block
        norm_z = self.norm2(z)
        ffn_out = self.feed_forward(norm_z)
        z = z + self.dropout(ffn_out)

        # Placeholder return - replace with your implementation
        return z, new_kv
```

The DecoderLayer class represents a single layer within the decoder stack of a Transformer model. Each decoder layer is responsible for processing input sequences and capturing dependencies through self-attention mechanisms.

Step 5 : DecoderOnlyTransformer class

```
class DecoderOnlyTransformer(nn.Module):
    def __init__(self, vocab_size, d_model, d_ff, num_layers, max_seq_len):
        super().__init__()
        self.d_model = d_model
        self.vocab_size = vocab_size
        self.num_layers = num_layers
        self.max_seq_len = max_seq_len

        self.token_embedding = nn.Embedding(vocab_size, d_model)
        self.pos_embedding = nn.Embedding(max_seq_len, d_model)
        self.layers = nn.ModuleList([DecoderLayer(d_model, d_ff) for _ in range(num_layers)])
        self.norm = nn.LayerNorm(d_model)
        self.output_projection = nn.Linear(d_model, vocab_size)

    def forward(self, input_ids, past_kv=None, use_cache=False):
        batch_size, seq_len = input_ids.shape

        input_ids = torch.clamp(input_ids, min=0, max=self.vocab_size - 1)

        token_embeds = self.token_embedding(input_ids)

        # Handling position offsets for cached generation
        position_offset = past_kv[0][0].shape[1] if (past_kv and past_kv[0]) else 0

        position_ids = torch.arange(position_offset, position_offset + seq_len,
                                    dtype=torch.long, device=input_ids.device).unsqueeze(0)
        position_ids = torch.clamp(position_ids, min=0, max=self.max_seq_len - 1)

        pos_embeds = self.pos_embedding(position_ids)
        x = token_embeds + pos_embeds

        if past_kv is None:
            past_kv = [None] * self.num_layers

        new_past_kv = []
        for i, layer in enumerate(self.layers):
            x, layer_kv = layer(x, past_kv[i], use_cache)
            new_past_kv.append(layer_kv)

        x = self.norm(x)
        logits = self.output_projection(x)

        return logits, new_past_kv if use_cache else None
```

```python
    # Generates new tokens autoregressively
    def generate(self, input_ids, max_new_tokens, temperature=1.0, use_cache=True):

        generated_ids = input_ids.clone()
        past_kv = None

        for _ in range(max_new_tokens):
            logits, past_kv = self(generated_ids, past_kv, use_cache)
            next_token_logits = logits[:, -1, :] / temperature
            probs = F.softmax(next_token_logits, dim=-1)
            next_token = torch.multinomial(probs, num_samples=1)

            generated_ids = torch.cat([generated_ids, next_token], dim=1)

            generated_ids = torch.clamp(generated_ids, min=0, max=self.vocab_size - 1)

        return generated_ids
```

The DecoderOnlyTransformer class defines a Transformer model composed solely of decoder layers, aligning with architectures used in models like GPT (Generative Pre-trained Transformer). Such models are designed for autoregressive tasks, generating sequences by predicting the next token based on previous tokens. The DecoderOnlyTransformer class encapsulates the architecture and methods necessary for tasks that require generating sequences based on given prompts, leveraging the power of self-attention and deep stacking of decoder layers to model complex patterns in data.

Running the scripts :

Evaluating the model

>>> python .\transformer_skeleton.py --mode evaluate

```
PS D:\users\ayush\IISC coursework\sem 2\SML\Assignments\Assignment 4\ques2> python .\transformer_skeleton.py --mode evaluate
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)
```

Evaluating the model with KV cache

>>> python .\transformer_skeleton.py --mode kv_evaluate

```
PS D:\users\ayush\IISC coursework\sem 2\SML\Assignments\Assignment 4\ques2> python .\transformer_skeleton.py --mode kv_evaluate
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)
```

Benchmarking the performance

>>> python .\transformer_skeleton.py --mode benchmark

```
PS D:\users\ayush\IISC coursework\sem 2\SML\Assignments\Assignment 4\ques2> python .\transformer_skeleton.py --mode benchmark
Successfully loaded model state dictionary from model_state_dict.pt
Benchmarking...
Results:
  Without KV cache: 0.0413 seconds
  With KV cache: 0.0323 seconds
  Speedup: 1.28x
```

For plotting the graph a new script is added in the code –

```
elif args.mode == 'plot':
    #for plotting the graph
    if not os.path.exists(args.model_state_dict):
        print(f"Error: Model state dictionary file {args.model_state_dict} not found.")
        return

    # Create model
    model = DecoderOnlyTransformer(args.vocab_size, args.d_model, args.d_ff, args.num_layers, args.max_seq_len)

    try:
        model.load_state_dict(torch.load(args.model_state_dict))
        print(f"Successfully loaded model state dictionary from {args.model_state_dict}")
    except Exception as e:
        print(f"Error loading model state dictionary: {e}")
        return

    # Sequence lengths to test
    sequence_lengths = [10, 50, 100, 500, 1000]
    latencies_without_cache = []
    latencies_with_cache = []

    print("Benchmarking...")
    for seq_len in sequence_lengths:
        input_ids = torch.randint(0, args.vocab_size, (1, seq_len))

        time_without_cache = benchmark_performance(model, input_ids, use_cache=False)
        time_with_cache = benchmark_performance(model, input_ids, use_cache=True)

        latencies_without_cache.append(time_without_cache)
        latencies_with_cache.append(time_with_cache)

        print(f"Seq Len: {seq_len}, Without KV cache: {time_without_cache:.4f}s, With KV cache: {time_with_cache:.4f}s")

    # Plot the results
    plt.figure(figsize=(8, 6))
    plt.plot(sequence_lengths, latencies_without_cache, marker='o', linestyle='-', label='Without KV Cache')
    plt.plot(sequence_lengths, latencies_with_cache, marker='s', linestyle='-', label='With KV Cache')

    plt.xlabel("Sequence Length")
    plt.ylabel("Latency (seconds)")
    plt.title("Transformer Generation Latency vs. Sequence Length")
    plt.legend()
    plt.grid()
    plt.show()
```

In order to run script to plot the graph we simply have to run –

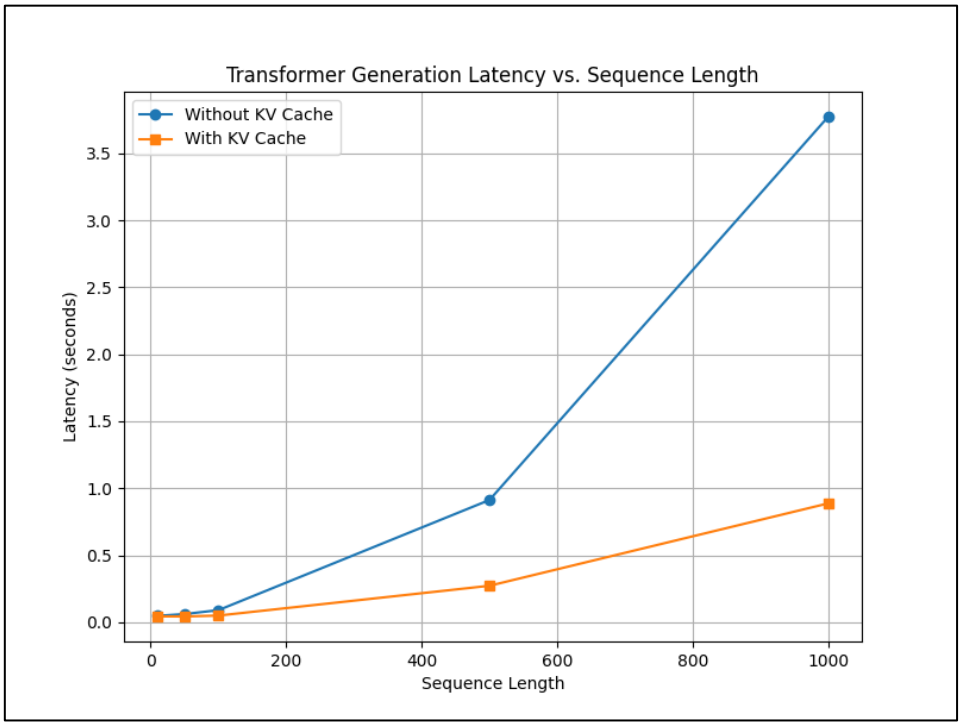
>>> python .\transformer_skeleton.py --mode plot

```
PS D:\users\ayush\IISC coursework\sem 2\SML\Assignments\Assignment 4\ques2> python .\transformer_skeleton.py --mode plot
Successfully loaded model state dictionary from model_state_dict.pt
Benchmarking...
Seq Len: 10, Without KV cache: 0.0477s, With KV cache: 0.0433s
Seq Len: 50, Without KV cache: 0.0607s, With KV cache: 0.0427s
Seq Len: 100, Without KV cache: 0.0900s, With KV cache: 0.0493s
Seq Len: 500, Without KV cache: 0.9123s, With KV cache: 0.2720s
Seq Len: 1000, Without KV cache: 3.7737s, With KV cache: 0.8870s
```

The tabular representation of the data is as follows -

Sequence Length	Without KV cache (s)	With KV cache (s)
10	0.0477	0.0433
50	0.0607	0.0427
100	0.0900	0.0493
500	0.9123	0.2720
1000	3.7737	0.08870

The graph obtained is as follows –



In case of without KV cache (Blue Line) each new token requires recomputing attention over all previous tokens. Hence latency grows quadratically ($O(n^2)$) with sequence length. While in the other case of with KV cache (Orange Line) latency grows linearly ($O(n)$) after initial overhead. In this only the new token's key/value are computed and appended to the cache. Attention is computed only between the new token and cached keys. The orange line (with KV cache) is not perfectly linear at very long sequences because if the KV cache grows too large, memory access can become a bottleneck.