task_1 - Soham Kalburgi

Q1
This question is about the Feed-forward neural network (FFNN) block present in the transformer architecture.
You've noticed that the FFNN has two types of layers, namely Linear and ReLU (besides LayerNorm and other layers, which we omit here).
The neural network applies weights and biases in the linear layer, and ReLU layer introduces a non-linearity.
Why do you think this non-linearity is needed?
Are we better off not using ReLU (and thus saving some compute)?
If not, are there any alternatives to ReLU that could be thought of? (they need not be more efficient, just alternatives are enough)

A1
Without ReLU (or any other activation fn/non-linearity), f=W2*W1*x, say W3 = W2*W1, f=W3*x, which is again linear -> only representing linear fns, which is not good for complex patterns in data. Better off?, absolutely - can save ~0% compute for ~0% accuracy xD (this is a joke). Other alternatives - sigmoid, tanh, GELU, Leaky ReLU, SiLU, ELU.

Q2
During your reading for the assignment, you come across quantization, a way to reduce the LLM model's size by expressing its weights as 16-bit Floating-Point numbers or 8-bit integers.
You decide to apply quantization to the LLM you're working with, so that you can fit a larger LLM into a smaller GPU VRAM and get better responses.
What could be the possible issues you face with this approach?

A2
I think the responses would surely be *faster* but not necessarily *better* as there can be potential loss in accuracy due to squeezing of high precision weights to lower precision.

Q3
You've read about the attention mechanism for this assignment.
Current LLM models implement a lot of attention heads (multi-headed attention or MHA) instead of having one big attention head.
Why do you think this is the case?

A3
MHA allows the model to attend to different positions and relationships in the input simultaneously, hence learning diverse contextual patterns, and providing a better understanding of the data. It also allows efficient parallel computation across heads.

Q4 – Q5

Based on the following case study:
Pranav and Soham are new to machine learning, and have recently learned about transformers and neural networks.
They want to start off by implementing a neural network that can help identify handwritten digits, with training data obtained from the MNIST dataset.
They decide to have a neural network architecture that looks like this:

28 × 28 image → 784-dimensional vector → passes through a layer of 784 neurons and compresses it to 256 dimensions → apply ReLU →
Pass through another layer, compressing it to 128 dimensions → ReLU →
Another layer that compresses it to 10 dimensions → apply softmax → Make a prediction

Q4

They've now decided to begin training on the dataset, and split it into train, validation and test sets.
For the loss function, they'll be using softmax regression, and optimizer would be the SGD optimizer.
Pranav decides that the learning rate should be set to 1 so that the training ends quicker, and begins training the NN.
However, the training seems to continue for a long time, with no significant improvements in both training and validation accuracy.
Moreover, the values for accuracy seem to be oscillatory (e.g., the accuracy would jump from 56% to 59%, and back to 56% again after successive iterations).
Why do you think this could be the case?

A4

The updates are too large and thus the model bounces around the optimum, preventing meaningful convergence. High LR doesn't necessarily mean low training times. In this case the network wastes iterations as it cannot make stable progress towards reducing the loss. Can be fixed by lower LR or an LR scheduler.

Q5

Thanks to your help, they were able to figure out the problem!
However, their NN seems to perform very poorly (~60% accuracy) on the prediction task.
Soham suggests that adding more neurons to the intermediate layers would help the model capture relationships better.
Using this suggestion, Pranav and Soham notice that the model's performance on the training set has improved, but the model is performing poorly on the validation set.
Why has this happened, and how could this be mitigated?

A5

Overfitting - high variance, low bias. Use regularization, dropout, data augmentation etc.

Q6 – Q7
Based on the following case study:
Vimarsh is hosting a LLM called DaSHGPT on his small cluster of GPUs for about 70 people.
The LLM uses the classic transformer architecture that you've learnt about (embedding, attention heads, FFNN, softmax).
Vimarsh also knows that GPUs are good at processing things in parallel, and decides to leverage this by parallelizing the pipeline as much as he can.

Q6
What are some parallelization techniques he could use to speed up inference on his small GPU cluster?

A6
https://huggingface.co/docs/transformers/en/perf_train_gpu_many
1. Data Parallelism - distribute data across multiple GPUs, each GPU processes their batch of data, results are then collected.
2. Model Parallelism - distribute the model (typically its layers) across multiple GPUs, quote "On the forward pass, the first GPU processes a batch of data and passes it to the next group of layers on the next GPU. For the backward pass, the data is sent backward from the final layer to the first layer".
3. Pipeline Parallelism - similar to model parallelism, micro-batches of data, pass these micro-batches b/w GPUs, thus reducing idle wait time
4. Tensor Parallelism - distribute tensor computations, slice horizontally or vertically and each GPU operates on its tensor slice - good for models that don't fit in the memory of a single GPU
5. Hybrid Parallelism
Also a good read - https://soumith.ch/blog/2024-10-02-training-10k-scale.md.html

---

Q7
Vimarsh recently came across what is known as "batching" prompts together.
He decides to save compute and memory resources, and sets the batch size to 64.
This approach works well, as a lot of compute power is saved due to GPUs working less frequently, and in parallel.
However, users complain of long waiting time after giving a prompt to DaSHGPT.
Why has this happened?
What would you do to fix this?

A7
GPU waits to fill the batch before inference - high latency - increased wait times. One very obvious fix, reduce batch size. Another fix - dynamic batching i.e. process whatever prompts are available after a timeout.

Q8 – Q10
Based on the following case study:
Atharva has recently read the paper on attention.
Thinking that *"Attention is all He Needs"*, he begins working on his own transformer/LLM.
He's able to code up almost all aspects of the neural networks, but needs help in some other ones.

Q8
Assume that you have a sentence that is of the form:
"The quick brown fox jumps over the lazy ___"
You want the LLM to predict the next token in the sequence.
Atharva's LLM does this by tokenizing the input sentence (assume one token = one word in the sentence),
pushing it into the transformer's attention layers, computes all the attention dot products for the sentence,
then multiplies it by the value matrix, finds the linear combination and updates the embedded vector,
and finally passes it through the FFNN, repeating this procedure from the beginning each time a new token is added,
thus getting the result using softmax regression.
However, Atharva's LLM seems to be very slow at producing the output.
What could be the issue?

A8
Not using cache. Attention is O((n^2)*d) (n=seq_len, d=hid_dim). KV cache reuses the keys and values from previous steps, so the new Q interacts with the cached K, V; making TC O(n*d).

Q9
Atharva has realized his mistake now, and is set to correct it.
He now stores the required computations in his GPU VRAM.
Assuming that Atharva is using only 1 GPU with 16GB VRAM, a 7B parameter model with FP16 precision consumes about 14 GB of his VRAM.
Happy that his LLM fits in the GPU, Atharva begins inference, but soon realises that the time taken to produce an output keeps increasing as he gives more and more prompts, eventually piling up to an unacceptable delay.
Why has this happened?
What is the fix that comes to your mind?
Assume that the prompts are interdependent (i.e. prompt 1 uses context from prompt 2, and so on).
(Hint: Think in terms of where and how the pre-computed values are being stored and accessed. You ideally do not want the values to consume GPU VRAM.)

A9
Caching K, V in the GPU VRAM is the problem - the cache accumulates linearly b/w prompts, eventually OOMing. Fix is to store the cache off the GPU. Keep only current token's computation on the GPU, store previous K, V in CPU RAM, transfer to & fro GPU when required. Or you can use a sliding window approach, don't keep all previous tokens, keep a fixed number of most recent tokens' Ks, Vs in the cache, drop the old ones - but this could degrade accuracy.

Q10
Great, Atharva has listened to your advice and implemented "the fix" you've mentioned above.
However, instead of an eventual, unacceptable delay, Atharva now has a fixed but noticeable delay in producing an output for each token of every prompt.
Why has this happened?
(Hint: think about where the KV cache is stored now, and what happens each time a new token is generated. Can this data movement be reduced?)
Can you think of a simple way to reduce this delay?

A10
Because of the to & fro b/w GPU and CPU. Naive brute force - spend more money, get a better GPU, store cache on GPU itself, problem solved. Otherwise, I think SWA should be a good way. PagedAttention (arxiv:2309.06180) is also of interest.

– BONUS –

Q11
You have a 4-node setup:
C1, C2, C3 → 1 GPU each, 8 GB VRAM
C4 → 2 GPUs (A1, A2), 12 GB VRAM each
The network between nodes is slow (considerable communication delay).
NVLink is present within C4 (transfer between A1 ↔ A2 is fast).
You have a 3B parameter model in FP16 (6 GB VRAM per model instance).

Task:
Design a scheduling algorithm to assign prompts to GPUs based on their total length (input + output), classified as short / medium / large.
(Assume output length is known beforehand — a very unrealistic one, but fits for this case.)
You may draw a graph or diagram to explain your approach.
Proper justification MUST be provided for your approach.
Hints:

- Consider VRAM constraints: a GPU must fit the model + KV cache for the assigned prompt.
- Prefer fast local connections for prompts that require multiple GPUs.
- Assign short prompts to smaller GPUs, and long prompts to larger GPUs.
- Aim for load balancing while minimizing communication overhead.

If you're able to figure out an optimal scheduling algorithm for the above case, list two problems you'd face if you were to drop the unrealistic assumption made above.

A11
From, https://developer.nvidia.com/blog/mastering-llm-techniques-inference-optimization/:
size of KV cache per token in bytes = 2 * (num_layers) * (num_heads * dim_head) * precision_in_bytes

Let's consider llama3.2:3b (~3.21B params). Taking config from https://huggingface.co/meta-llama/Llama-3.2-3B/blob/main/config.json (or metadata from https://ollama.com/library/llama3.2:3b/blobs/dde5aa3fc5ff), we have:
num_layers = 28, num_heads = 24, dim_head = 128, precision_in_bytes = 2 (as FP16)
(Note: Llama is only used as an approximation for the KV cache calculations, although it's 3.21B params; it's just 2GB in size).

KV cache / token = 2*28*24*128*2 = 344064 bytes = 0.000344064 GB

Back to Q, as given, the model itself is 6GB.
For C1, C2, C3 (8GB VRAM) -> 2GB available for KV cache after loading model
For C4 (A1, A2) (12GB VRAM each) -> 6GB available for KV cache / GPU

Now, let's classify the prompts based on token size.

Small - fits completely in C1/C2/C3 (and thus ofc C4). KV cache <= 2GB.
Tokens = 2/0.000344064 = 5812.87 ~ can fit 5812 tokens
0 <= small <= 5812 tokens

Medium - fits only on A1/A2. 2GB < KV cache <= 6GB.
Tokens = 6/0.000344064 = 17438.61 ~ can fit 17438 tokens
5812 < medium <= 17438 tokens

Large - > 17438 tokens

Algorithm:
Each prompt arrives with a known total token length (input+output). Determine small/ medium/ large based on the above ^.

For small prompts, assign to C1, C2, C3 in a cyclic manner based on current VRAM utilization (assign to lowest VRAM usage; cyclically if tied). If all C1-3 are busy, queue FIFO small prompts until one frees up.

For medium prompts, assign to C4's A1 or A2; whichever has more free VRAM. If both busy, or C4 locked (by large prompt); queue.

For large prompts, use both A1 and A2 together via NVLink and lock C4. Load the model on A1 and split/shard the KV cache across both GPUs. So A1 holds model + partial KV, A2 holds only the remaining KV. NVLink interconnect ensures efficient token streaming b/w A1 and A2. If A1/A2 is busy, queue.

De-queuing: Poll queues on a time period (say 100ms); re-attempt assignment when resources are free. Or a better option is to wake up the scheduler on job completion. Prioritize large > medium > small to minimize wait for bandwidth-heavy jobs.

We are not using preemption cuz it's gonna require saving/restoring of KV cache, thus adding significant overhead.

Problems:
If the assumption that the output length is known (and hence the total token count), the scheduler doesn't know how much KV cache will be required or how long a GPU will remain busy. This leads to the following -

- Unpredictable VRAM usage
    - KV cache grows dynamically during decoding as each new token is generated.
    - Without knowing the output length in advance, a small prompt may gradually exceed the VRAM budget of the GPU it was placed on (a small input doesn't necessarily have a small output) - can lead to OOMs.
- Unpredictable scheduling
    - The scheduler can no longer predict when a GPU will free up.
    - One short prompt may finish quickly, while another (with similar input) could generate thousands of extra tokens and block that GPU for much longer - thus the completion times become stochastic.

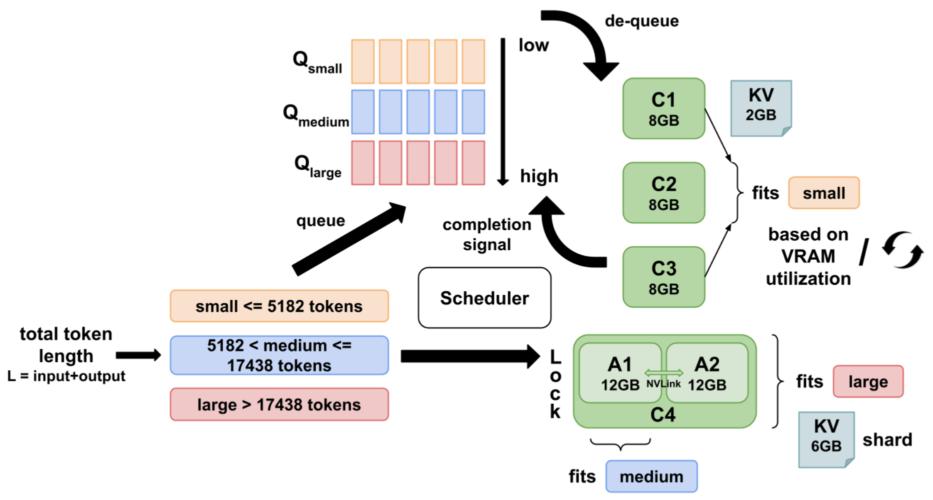

Figure: Overview of the Scheduling Algorithm