Inference Optimization

Inference on GPU

- Compute bound
- Memory bound
  - Large transformer inference is memory bound.
  - Improve FLOP utilization => improve efficiency

GPU Architecture

- Basics: DRAM, L2 cache and SM
- Comparison to CPU
  - SM is similar to CPU cores, but with larger level of parallelism. This is also called the SRAM.
  - L2 cache and DRAM is similar to CPU L2 and dram
  - In Flash Attention(FA), L1 is called SRAM
- A100 80G SXM
  - 108 SM
  - DRAM 80G
  - 40M L2 cache
- What's inside an SM?
  - L1 cache: instruction and data
  - Tensor core => matrix multiply i.e all computations of NN happen here

GPU Programming Basics

When performing `model.generate(prompt)` we do

- Memory Access
  - Load model weights from HBM to L2 cache to SM
- Compute
  - perform a matrix mul in SM, SM asks tensor core to do it
  - Which op takes more time? MEMORY ACCESS!
  - when running inference on transformers, most of the time is spent on moving model params/activation from/to the memory, rather than the actual computation
- A100:
  - 108 SM, DRAM 80G, 40 M L2 cache
  - bf16 tensor core: 312 trillion FLOPS (tFLOPS)
  - DRAM memory bandwidth 2039 GB/sec = 2.039 TB / sec
- If the model is large, split into multiple GPUs, connected by NVLink
  - NVLink 300 GB / sec = 0.3 T / sec
- We get speed hierachy.
  - 312 T (SM Compute) > 2.03 T (DRAM Memory Access) > 0.3 = 300 G (NVlink cross-device communication) > 60G PCIe cross-device communication
- If we want speed:
  - fully utilize the SM
  - reduce memory access in one GPU (because it's much slower than compute)
  - and reduce comms between GPUs (becauses it's even slower than memory access)

How to determine if we have fully utilized the SMs?

- Check whether operation is compute bound or memory bound
- **GPU operations per byte:** flop / memory bandwidth
  - A100 = 312 / 2.039 = 153 operations per byte(or FLOPS per byte)
- **Arithimetic internsity**
  - If approx equal to ops per byte then compute bound
  - If smaller, then memory bound
- **Increasing batch size will change behavior from memory bound to compute bound**
- **Kernel Fusion** : reduce memory access operations => fuse multiple ops into one

1.2 Transformer Inference Basics

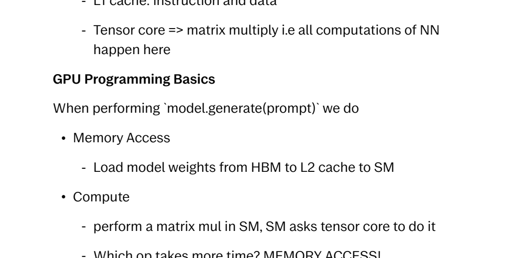
Prefilling and Decoding

Two steps when calling model.generate(prompt)

- Prefilling:
  - Compute kv cache for prompt
  - Compute bound, because we compute for a sequence of tokens in parallel
- Decoding
  - sample next tokens autoregressively i.e select token with max prob
  - memory bound => we can only compute one token. Can't fully utilize SMs

Transformer inference is memory bound

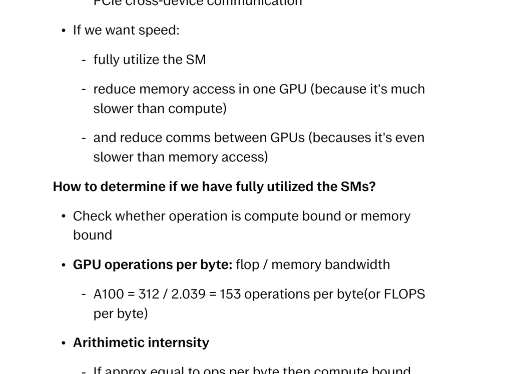
↑ batch size changes pattern from memory to compute bound because decoding only sample one token in one pass



→ ↑ batch size ↑ hardware efficiency
- → because we compute multiple tokens at a time
- → large batches changes from memory bound to Flop bound.
- → HOWEVER, BATCH SIZE HAS LIMIT. GPU MEMORY IS ONLY 80GB (A100)

Auto-regression is different from initial computation (i.e training) in terms of arithmetic intensity.

Memory Layout



Nvidia A100 40 GB

To serve 13b parameter model in BF16, we only have 10G of memory to store the KV cache.
- → cannot have too large batch size (even though we want large size to improve efficiency)
- → cannot have too long sequence (though we want to serve 100K)

Online vs offline inference i.e throughput vs latency

→ Offline: throughput optimization
- → may need this to run ckpt of model on benchmark tasks to check if pretraining is healthy
- → ↑ batch size will help, but max memory is still 80GB

→ Online: latency optimization
- → When batch size is large, we become compute bound ⇒ latency increases
- → Latency should not be slower than human read speed.
- → But we want large batch size to improve efficiency.

Context length scaling

Suppose we want to model 100K context
- → Prefilling
  - → Takes significantly longer, since inputs are much longer
  - → Latency to first token is imp!
- → Decoding
  - → Large KV cache now, because context is large. Uses moar memory!!
- → Author says difficult to improve here. However, Multi-Head Latent attention (Deepseek-V2) addresses this & ↓ memory significantly.

2. MLsys: Flash Attention & vLLM

vLLM → GPU management
Flash Attention → Reduce I/O by keeps most ops in SMs, reducing memory access overhead.

Paged Attention

→ Construct mem management system similar to CPU mem mgmt, to ↑ utilization & ↓ I/O

Flash Attention

→ What is the Flop, time & memory complexity of Flash Attention?
- → Memorize this !!

Key idea
- → Instead of storing full attention matrix in HBM, do blockwise computation of dot product, such that all computation performed in L2 cache

Key advantage:
- → ↓↓↓ memory usage. Can fit 100K context
- → ↑↑↑ throughput, particularly for small models when large portion of flop is in dot-product operation

Flash Decoding

Key idea
- → Instead of using single query to scan the KV cache, duplicate the query such that different chunks of KV cache can be scanned in parallel.

3. Modelling: Architecture & Decoding Algorithm

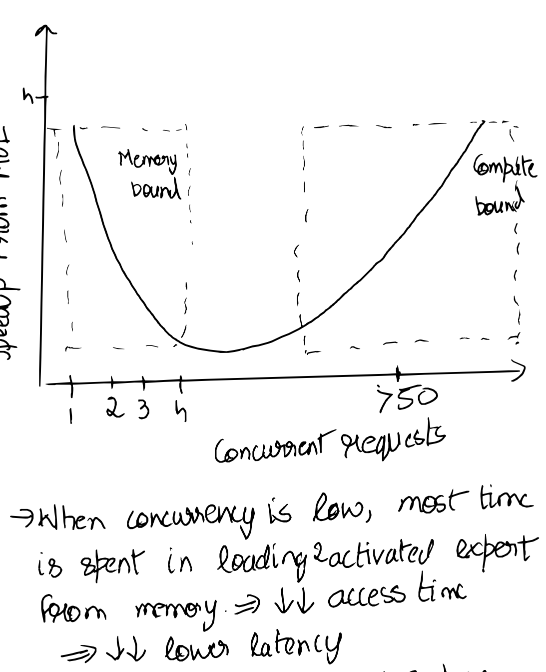
Distillation & sparse attention
- → Distillation
  - → Finetune a small model using larger model logits
- → Sparse attention
  - → Works well for small models
  - → only mistral sliding window attention works on large models (so far).

Quantization
- → Quantize weight to int8
- → Does not harm model performance.
- → Mandatory now to deploy large models

MQA & GQA
- → MQA speed up training & inference. ↓↓ memory & ↓↓↓ compute
- → For small models, MQA <<< full attention
  For large models, MQA ≅ Full attention
- → Author says SOTA models use MQA. GQA should be more efficient? Llama3 has GQA AFAIK!

Advanced Techniques

MoE
- → Say we have 7B activation, 34B params total
- → Can we get same:
  - → perf as 34B
  - → throughput better than 34B
  - → Latency similar to 7B
- → How much is speedup?



- → When concurrency is low, most time is spent in loading 2 activated expert from memory ⇒ ↓↓ access time ⇒ ↓↓ lower latency
  - → For single query, MoE has lower latency than dense because we need to read less params from memory.
- → When concurrency is high, we enter FLOP bound regime, but MoE has ↓↓ activation than dense ⇒ ↑↑ throughput
  - → Less FLOP per token ⇓ ↑↑ batch size ⇒ ↑↑ throughput
- → Can also convert dense model to MoE. Called MoEFication?

Early exit
- → For some tokens, do not need to compute all transformer layers. Just some will be enough, since they are easy tokens
- → Use gate to determine early exit or not.

Blockwise Decoding

speculative decoding
- → Decode multiple tokens at a time to fully use SMs.
- → Draft model to predict tokens. Discard token if probability does not match large model.
- → Problems:
  - → Weak & may have ↑ rejection rates
  - → Accuracy vs overhead tradeoff: Draft ↓↓ overhead but also has ↓↓ accuracy
  - → 2 models in GPU
    - → not necessarily true
    - → load small model in CPU using llama.cpp & multiplex requests.

We want to use large model for proposal. Hence use Medusa
- → Use multiple heads to decode multiple tokens at a time
- → Use larger model as draft model.

Other techniques
- → Lookahead decoding
- → Retrieval based speculative decoding
- → EAGLE
- → n-gram decoding.

Techniques to always consider

MLSys	Modelling
Model parallelism	MQA
Flash attention	Speculative decodin
vLLM/paged attention	
Quantization	