High Performance LLMs From First Principles (2024)

Session 8
https://github.com/rwitten/HighPerfLLMs2024

rwitten@

Goal: learn how to achieve **high performance** for LLMs

This week:

- Answer some questions
  - what is model flop utilization?
  - look at a good xprof
  - what is flash attention?
  - why is LLM inference so weird?
- And implement inefficient inference

Next week:

- Efficient inference
- Last questions!

Program (write code in Jax)

Predict (roofline on napkin or spreadsheet)

Profile (run code, compare to predictions)

My Asks

Please ask lots of questions! Just raise your hand or speak up!

If there are topics you're interested in, message me between sessions.

Join the discord! https://discord.gg/2AWcVatVAw

Do the exercises! Give feedback, ask questions!

Website: https://github.com/rwitten/HighPerfLLMs2024

Model Flop Utilization

- We kind of glossed over it but last week we did the following calculation:
  - (1) What are the number of FLOPs in a step (across all devices)?
    - TOTAL_MODEL_FLOPS ~= 6 * BATCH_IN_TOKENS * NUMBER_PARAMETERS
  - (2) How long did the step take? How many devices are there per step?
    - TOTAL_DEVICE_SECONDS = PER_STEP_TIME * jax.device_count()
  - (3) Model FLOPs/device/second = TOTAL_MODEL_FLOPS/TOTAL_DEVICE_SECONDS
  - (4) Model Flop Utilization (MFU) = Model FLOPs/device/second / Peak FLOPs/device/second
- When we see 189 TFLOP/s/device in (3), we know our Peak TFLOP/s/device is 275 so our MFU is 68.7%.
  - This is the fraction of the theoretically achievable performance that we're actually achieving.
- P.S. – critically we might find the FLOPs in the profiling (which is counting how much the hardware is working) doesn't match the model! We'll see an example with softmax.

Good Training Xprof

- (In this context the “FLOPs utilization” matches MFU which was also 69%).
- We’re spending the time all-gathering weights (makes sense since this is FSDP)
- And reduce-scattering weights (which we haven’t talked about but happens in the backwards pass during FSDP).
- [Xprof (for Googlers)]

Overall TPU FLOPS utilization is 68.79%

Memory bandwidth utilizations: Hbm 24.97% , On-chip Read 2.65% , On-chip Write 3.45%

Modifying your model's architecture, data dimensions, and improving the efficiency of CPU operations may help reach the TPU's FLOPS potential.

"IDLE" represents the portion of the total execution time on device that is idle.

Group by Category | Order by wasted time | Top 90% | Exclude Idle | Ops Limit: 10

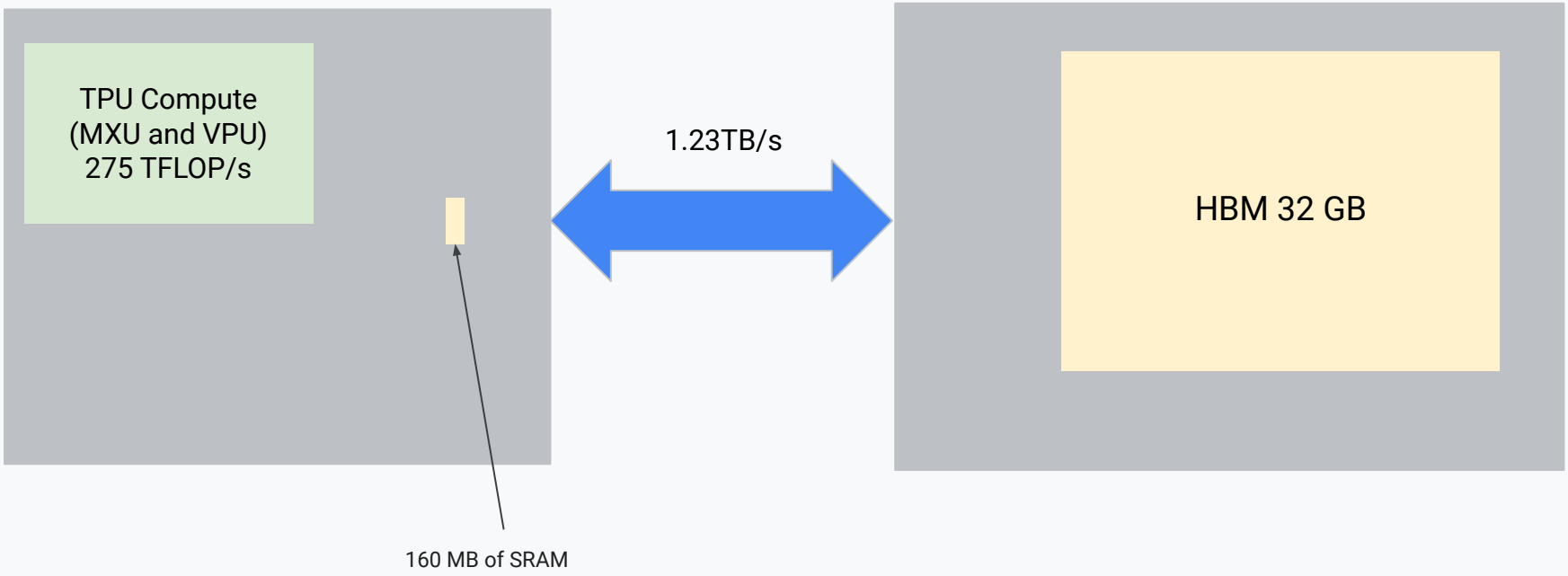
TIME %	WASTED %	HBM %	HLO OP NAME	FRAMEWORK OP TYPE	FLOPS	HBM
100.00%	31.21%	100.00%	▼ jit_step(2892941650699670884)	-	68.79%	HBM 24.97%
77.02%	8.23%	74.22%	▶ convolution fusion	-	89.31%	HBM 24.06%
9.17%	8.71%	1.81%	▶ all-gather	-	0.0%	HBM 4.94%
6.57%	6.17%	1.61%	▶ all-reduce-scatter fusion	-	0.01%	HBM 6.13%
1.40%	0.59%	3.25%	▶ loop fusion	-	0.23%	HBM 57.96%
0.42%	0.28%	0.56%	▶ custom fusion	-	0.02%	HBM 33.62%
0.34%	0.0%	4.95%	▶ copy-done	-	0.0%	HBM 366.47%
0.19%	0.19%	0.00%	▶ concatenate	-	0.0%	HBM 0.59%
0.15%	-	0.0%	▶ async-done	-	0.0%	HBM -
0.14%	0.12%	0.09%	▶ non-fusion elementwise	-	0.15%	HBM 16.40%
0.12%	0.05%	0.30%	▶ data formatting	-	0.0%	HBM 60.65%

8 categories or ops have been left out.

What Is Flash Attention? First Softmax

- Before we talk about flash attention, let's consider softmax (a core piece of flash attention) over a large input.
- Let's imagine we want to do a softmax(A) * B
  - Let's code it in Python!
- Key observation –
  - Naive solution needs to load A, write back softmax(A) and then load softmax(A) and B to output a single number!
    - 4 passes over N bytes
  - Fast solution loads A and B simultaneously and computes everything in one pass!
  - And we're very memory bound here – there are only a few flops per entry.
- P.S. – softmax is tricky! What happens when A is large?
  - This is an example of adding flops to the computation.

System Diagram For TPUs (v4)

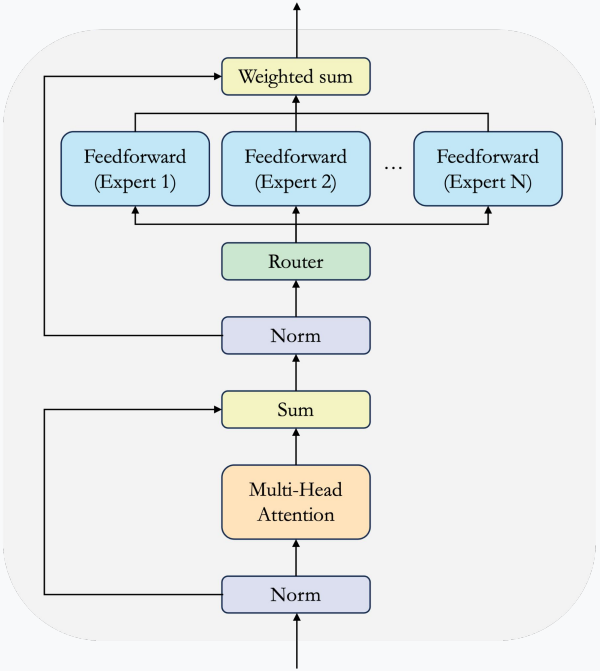


Flash Attention Summary

- In attention:
  - we form a W matrix by multiplying a [SEQUENCE_LENGTH, HEAD_DIM] Q matrix with a [HEAD_DIM, SEQUENCE_LENGTH] K^T matrix.
  - This matrix is [SEQUENCE_LENGTH, SEQUENCE_LENGTH] – which can be large.
  - Then we take softmax and multiply softmax(W) * V, where V is a [SEQUENCE_LENGTH, HEAD_DIM] matrix. Giving us an output in [SEQUENCE_LENGTH, HEAD_DIM].
- The idea of flash attention is that we don't form all of W!
- After we finish making a chunk of W, we immediately apply the rolling softmax trick to W and V – giving us the equivalent of partial_output and partial_denominator.
- And we renormalize at the end!
- Key observation from our little example, we replaced:
  - Original: Partial_output += $\text{softmax}(W)_i$ * V
  - With:
    - Partial_output += $\exp(W)_i$ * V
    - Denominator += $\exp(W)_i$

Mixture of Experts

- In a typical LLM, we have one feedforward.
- In a mixture of expert, we have N feedforward and each token goes through a subset K of them.
- With N=K, this is a dense LLM.
- Mixtral 8x22B is a typical MoE which has 141B total parameters but only 39B are active for a specific token. (2 out of 8 experts.) So the number of FLOPs per token is equivalent to a 39B model.
- The hope is that Mixtral 8x22B is “smarter” than a dense model with 39B parameters but (hopefully) has an inference cost close to a model with 39B parameters. We’d expect Mixtral to be “less smart” than a dense model with 141B total parameters.
- Tradeoff:
  - N > K requires more communication and memory pressure than N = K
  - Same number of flops
  - Empirical question in practice



Picture credit: https://blog.javid.io/p/mixtures-of-experts

Why Isn't LLM Training the Same As Inference?

- LLM Training is always of the form:
  - Input: ["[BEGIN]", "so", "long", "and", "thanks", "for"]
  - Output: "So", "long", "and", "thanks", "for"
- LLM Inference is very different:
  - Input ["[BEGIN]", "so", "long", "and"]
  - Output "thanks" "for"
  - But imagine the choice gets made to output "I", so now the prefix is ["[BEGIN]", "so", "long", "and", "I"]
    - Now the right completion is no longer "for" – maybe it is "thank"!
- What is happening?
  - The LLM answers the question: "given this prefix, what is the probability distribution on the next token".
  - The LLM doesn't actually choose that token – we need to specify it outside of the neural network.
- Review class_start.py showing the difference.
- Do class_pretend_inference_is_training_start.py.

Thanks!
Ping me (rwitten@google.com) with feedback, suggested topics, etc!