Efficient Deployment of Long Context Large Language Models

Colin Dablain
David Liedtka

Overview

- Why Long Contexts?
- Transformer Inference Arithmetic
- Single Inference Request Memory Allocation Analysis
- Multiple Inference Request Analysis
- Positional Encoding Methods
- Unsolved Questions

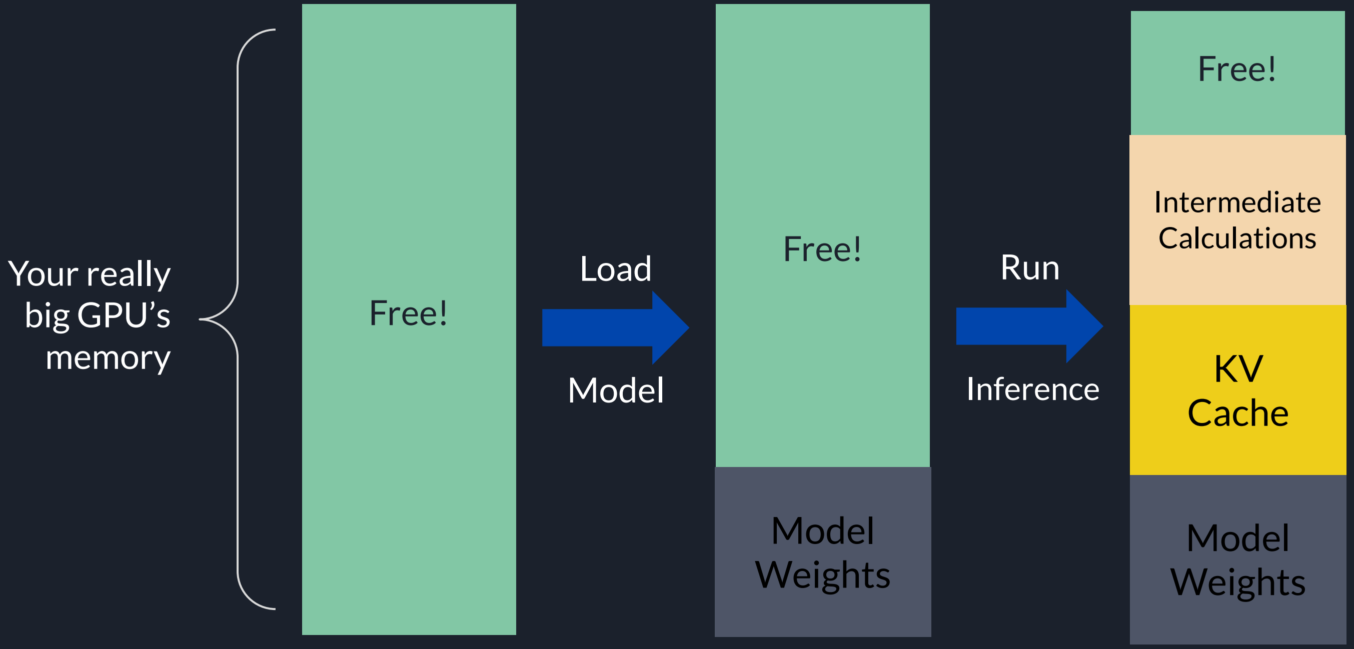
Why Long Contexts?

- LLMs with longer context windows can capture deeper contextual dependencies
- Enhanced capability to comprehend and summarize longer documents (i.e. RAG)
- Long chains of function calling and tool use
- Agents

Where does the GPU memory go?

```
RuntimeError: cuda runtime error : out of memory
```

Where does the GPU memory go?



Inference Math (Llama 2 7b on RTX3090)

```python
HIDDEN_SIZE = 4096
NUM_HEADS = 32
NUM_HIDDEN_LAYERS = 32

# set this to < NUM_HEADS for grouped query attention
NUM_KV_HEADS = 32
HEAD_DIM = HIDDEN_SIZE / NUM_HEADS

# 1 for 8 bit, 2 for 16 bit
# 8 bit is not widely supported in inference frameworks
KV_ENTRY_BYTES = 2

# 2x because you need one key and one value
bytes_per_kv_token = 2 * KV_ENTRY_BYTES * NUM_KV_HEADS * HEAD_DIM
print(f"{bytes_per_kv_token / 1000:.1f} KB per KV token")
```

✓ 0.0s

```
16.4 KB per KV token
```

```python
BYTES_PER_BF16_PARAMETER = 2

model_bytes = 7e9 * BYTES_PER_BF16_PARAMETER
print(f"{model_bytes / 1e9} GB required to load model")
```

✓ 0.0s

```
14.0 GB required to load model
```

```python
RTX_3090_BYTES = 24e9

bytes_for_kv_cache = RTX_3090_BYTES - model_bytes
print(f"{bytes_for_kv_cache / 1e9} GB left for KV cache")
```

✓ 0.0s

```
10.0 GB left for KV cache
```

```python
naive_token_estimate = bytes_for_kv_cache / bytes_per_kv_token
print(f"{naive_token_estimate:,.0f} token KV cache capacity")
```

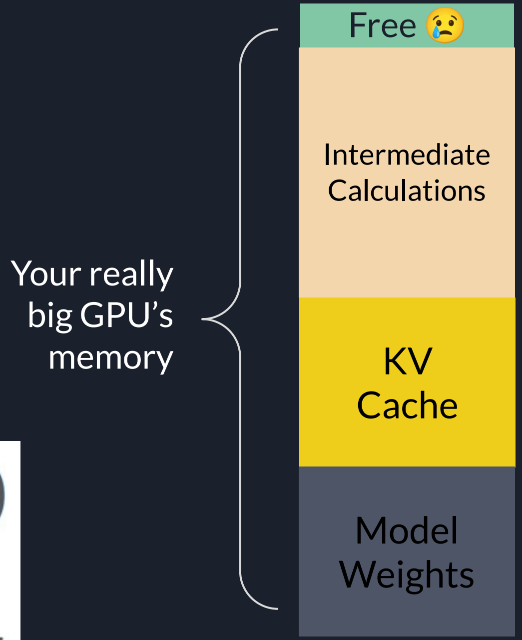
✓ 0.0s

```
610,352 token KV cache capacity
```

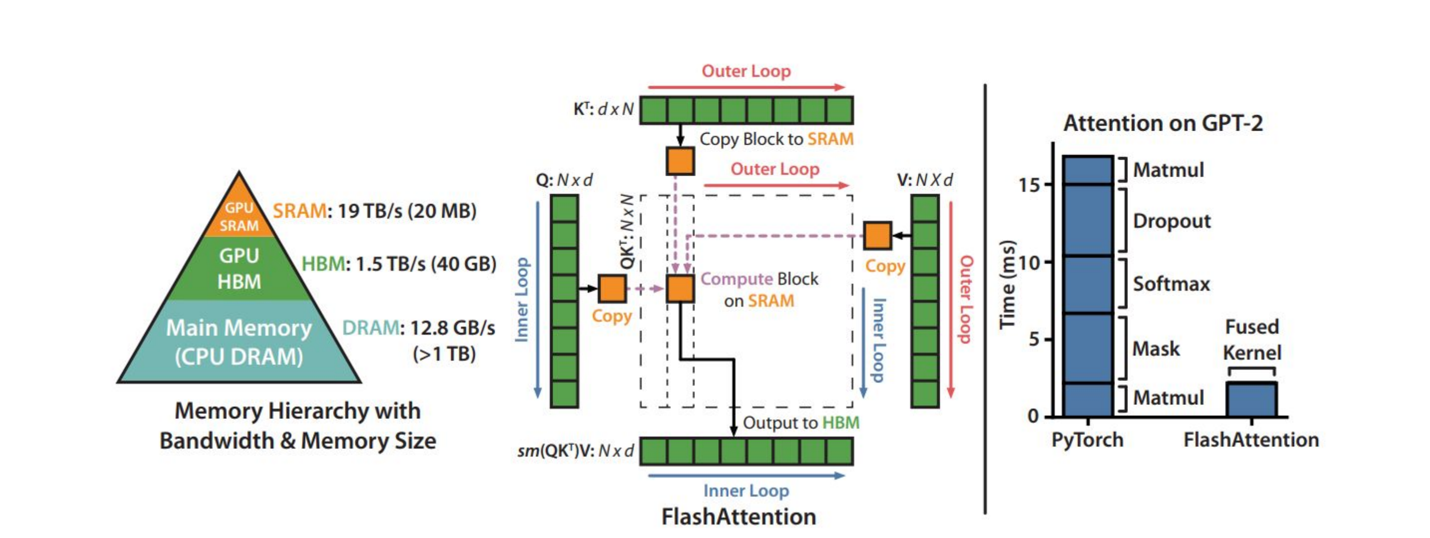
So What's the Problem?

- Intermediate computations also require (significant) GPU memory
- As the sequence length ***n*** increases, naive attention memory complexity goes as $O(n^2)$

$$\text{attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$



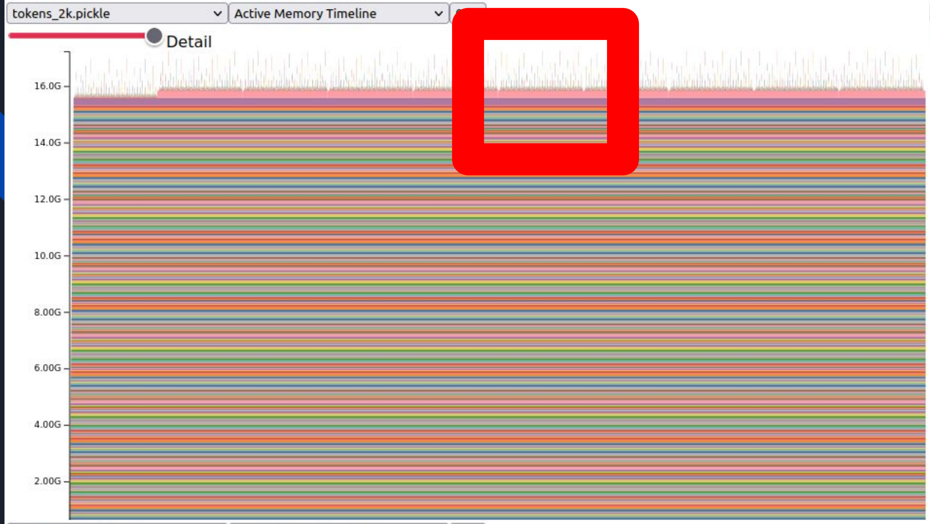
An O(n) Solution: Flash Attention

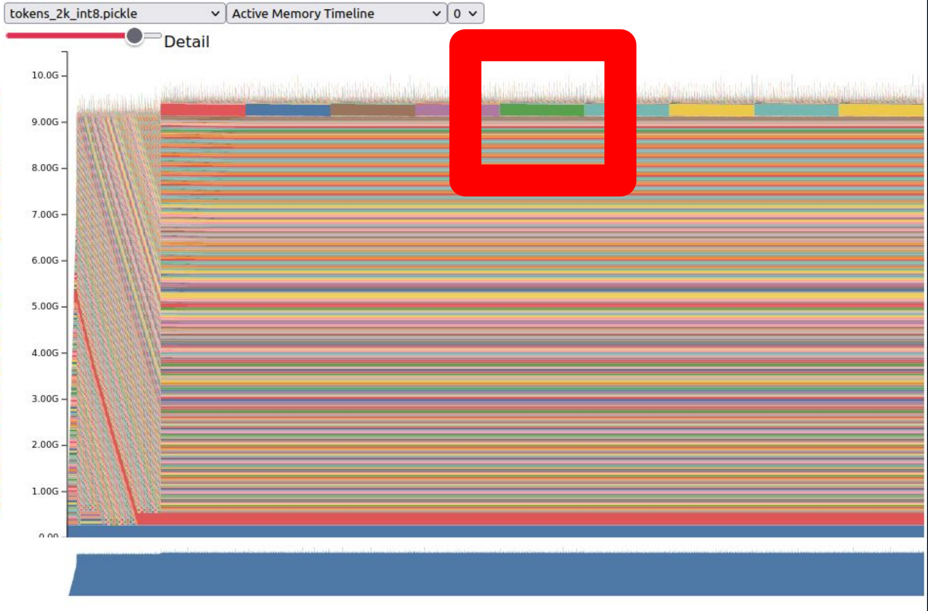


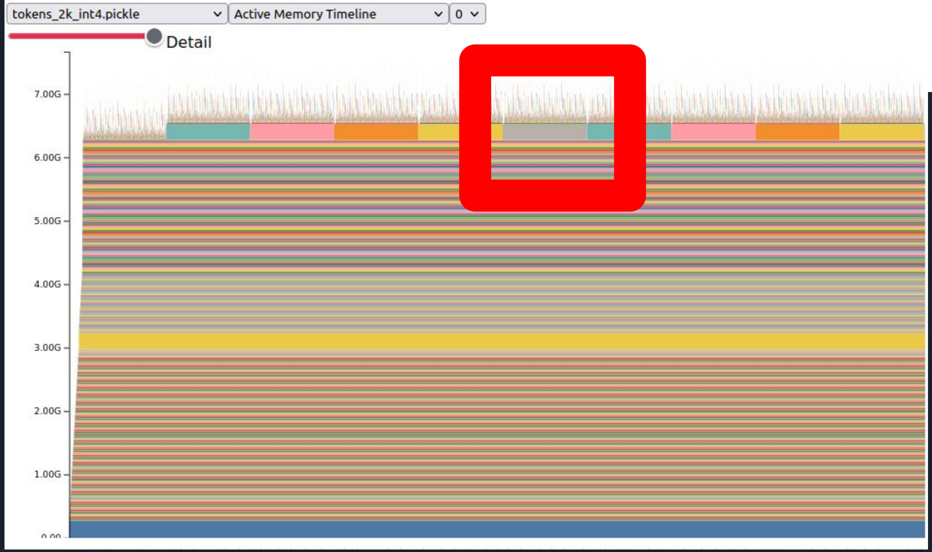
Reproduced from: Dao, Tri, et al. "Flashattention: Fast and memory-efficient exact attention with io-awareness." Advances in Neural Information Processing Systems 35 (2022): 16344-16359.

Single Inference Request Memory Allocation Analysis with

```
torch.cuda.memory._record_memory_history()
```



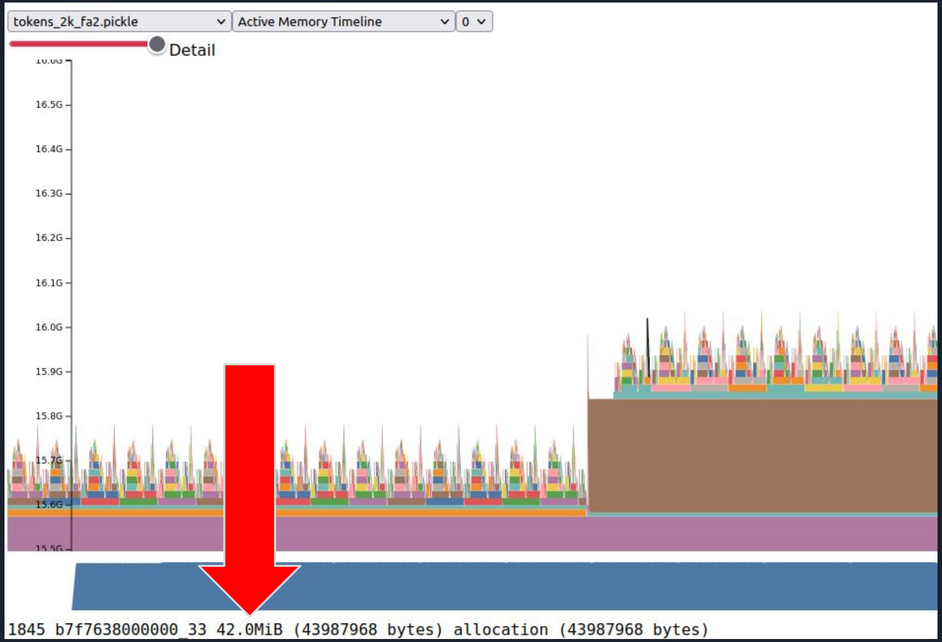


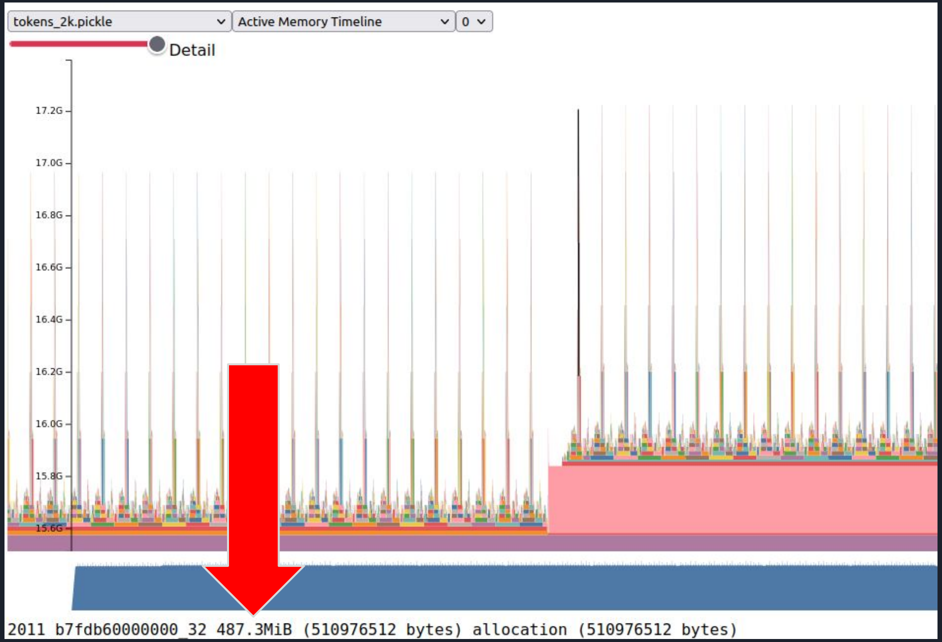


- Across precisions, KV entries are the same size (16 bit)
- Each large rectangle represents the repeated computation of KV entries at each generation step (10 tokens, no KV cache used here)
- Though it’s not visually clear for short sequence, each block is getting slightly larger
  - Because we’re adding a new KV pair at each generation timestep

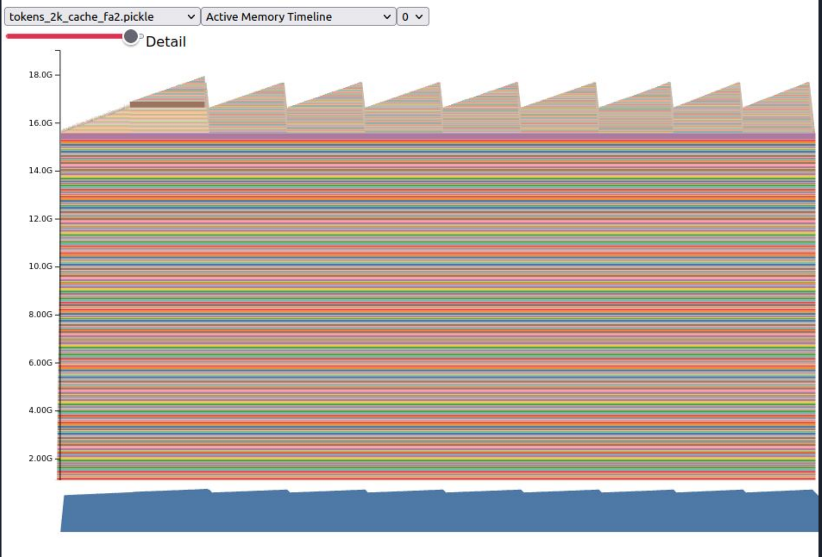
Flash Attention O(n) vs Pytorch Eager Attention O(n²)

- Even at the 2k sequence length, the difference in intermediate memory usage is an order of magnitude
  - 42 MiB vs 487 MiB

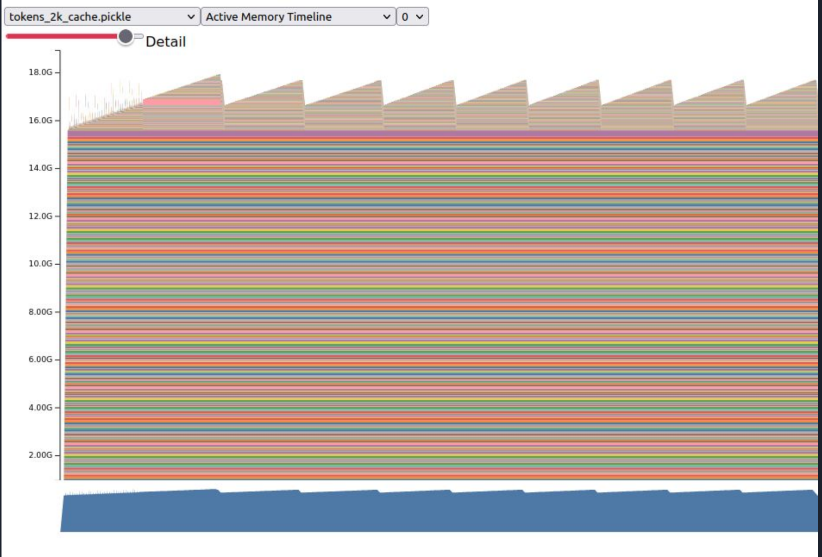




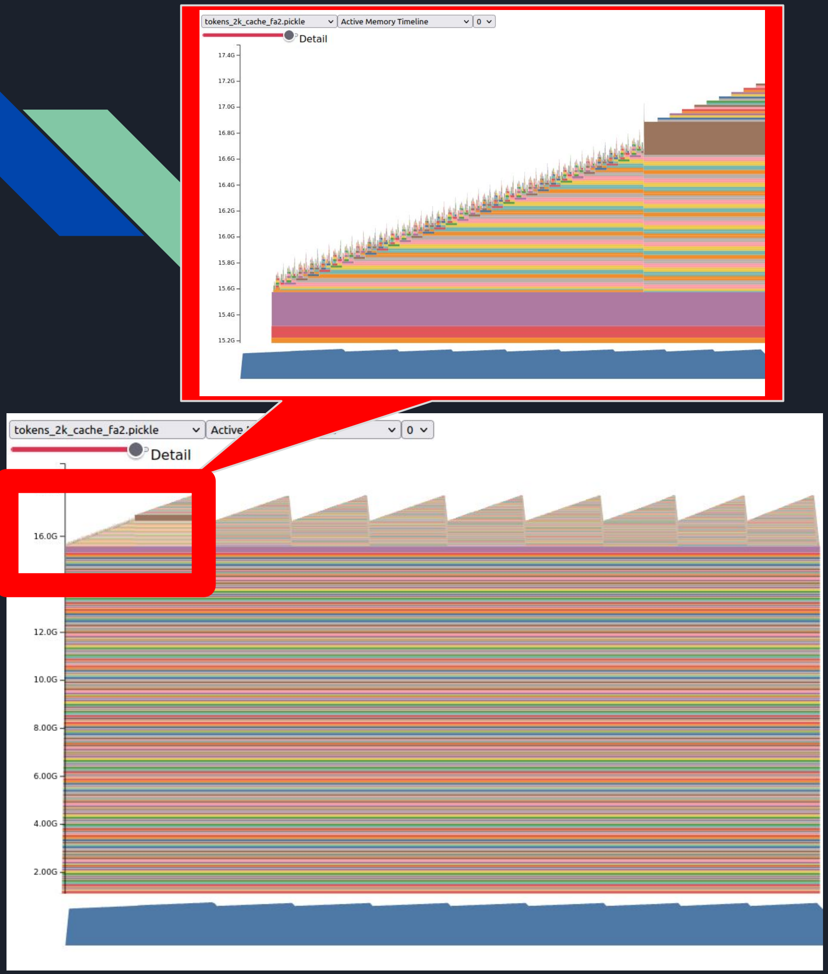
Flash Attention vs Pytorch Eager (with KV cache)



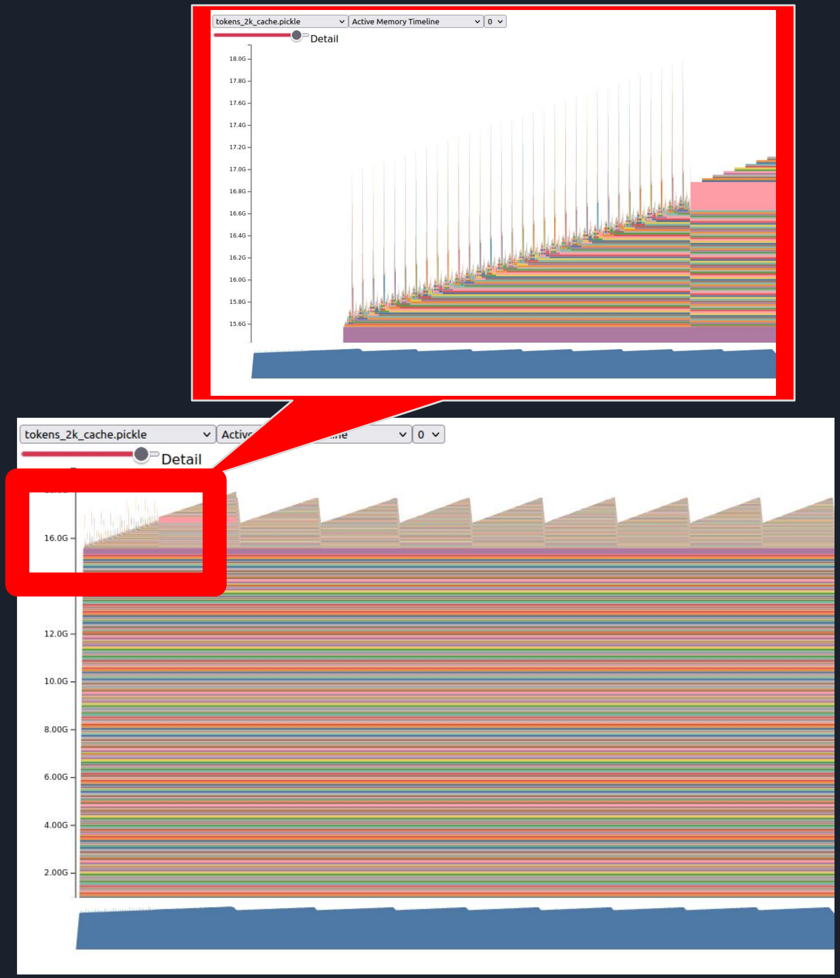
Flash Attention 2



Pytorch Eager Attention



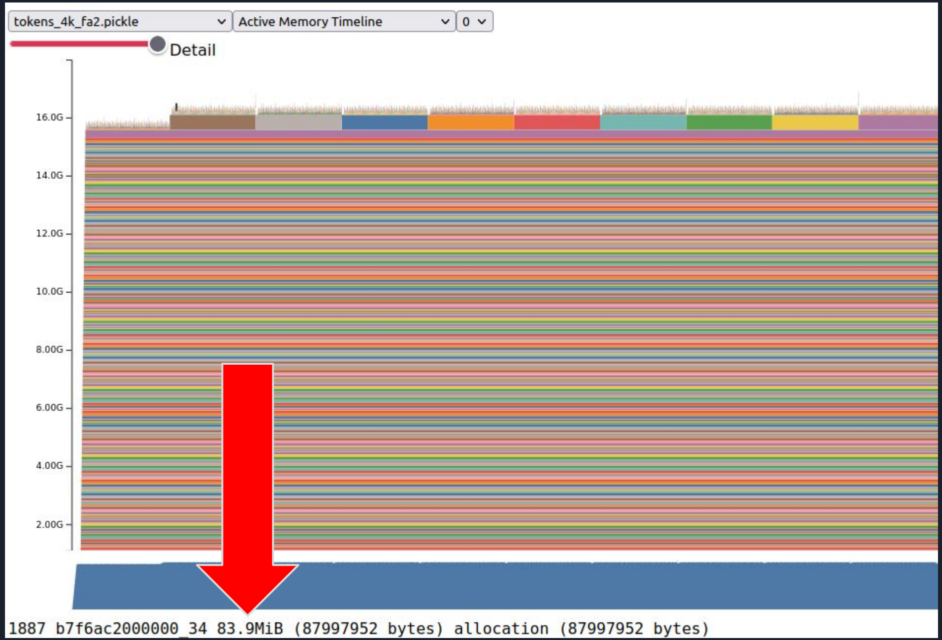
Flash Attention 2

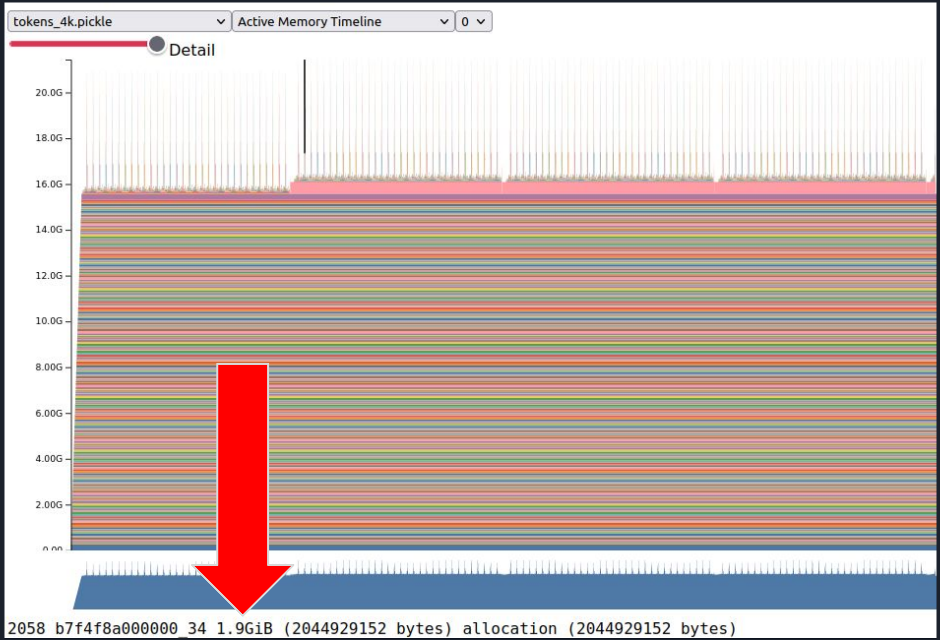


Pytorch Eager Attention

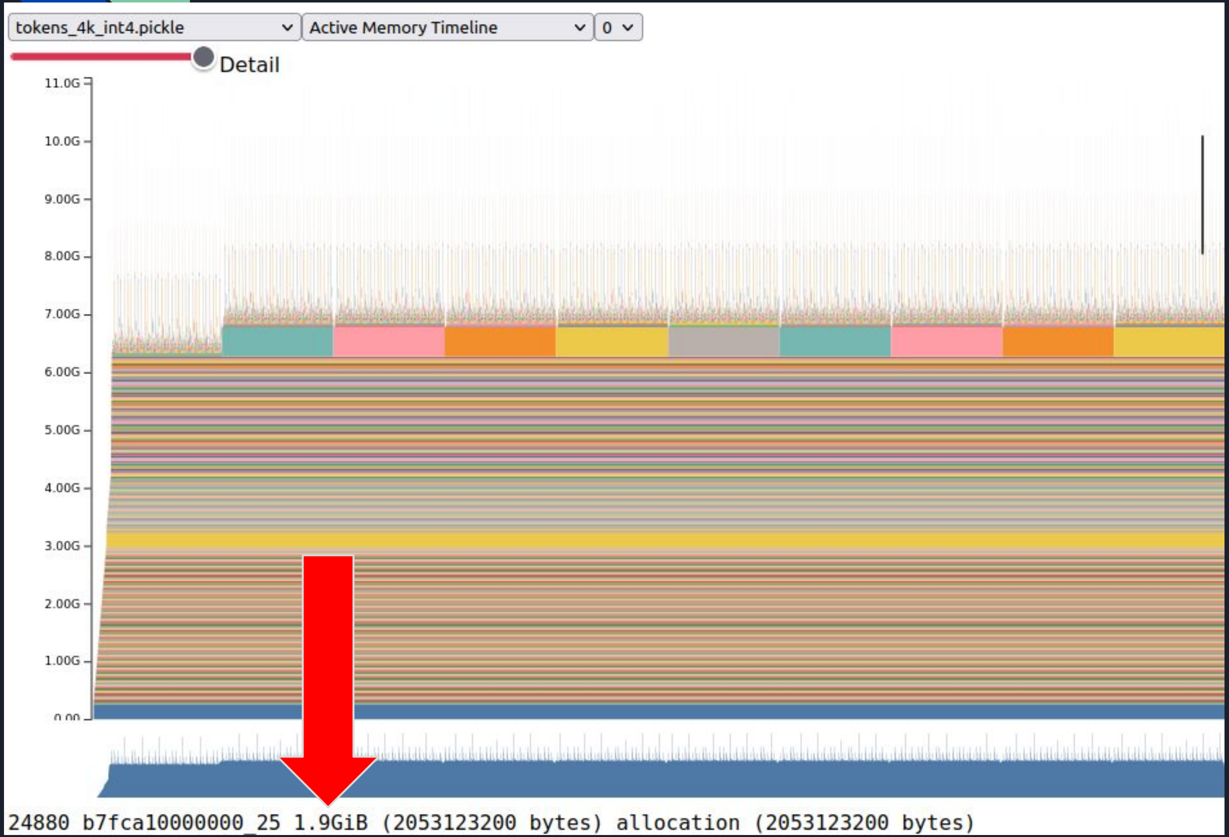
Flash Attention vs Pytorch Eager Attention (4k)

- At 4k sequence length, we OOM while generating token 5. Quadratic growth of intermediate memory use
  - 83.9 MiB (2x 42 MiB!!) vs 1.9 GiB



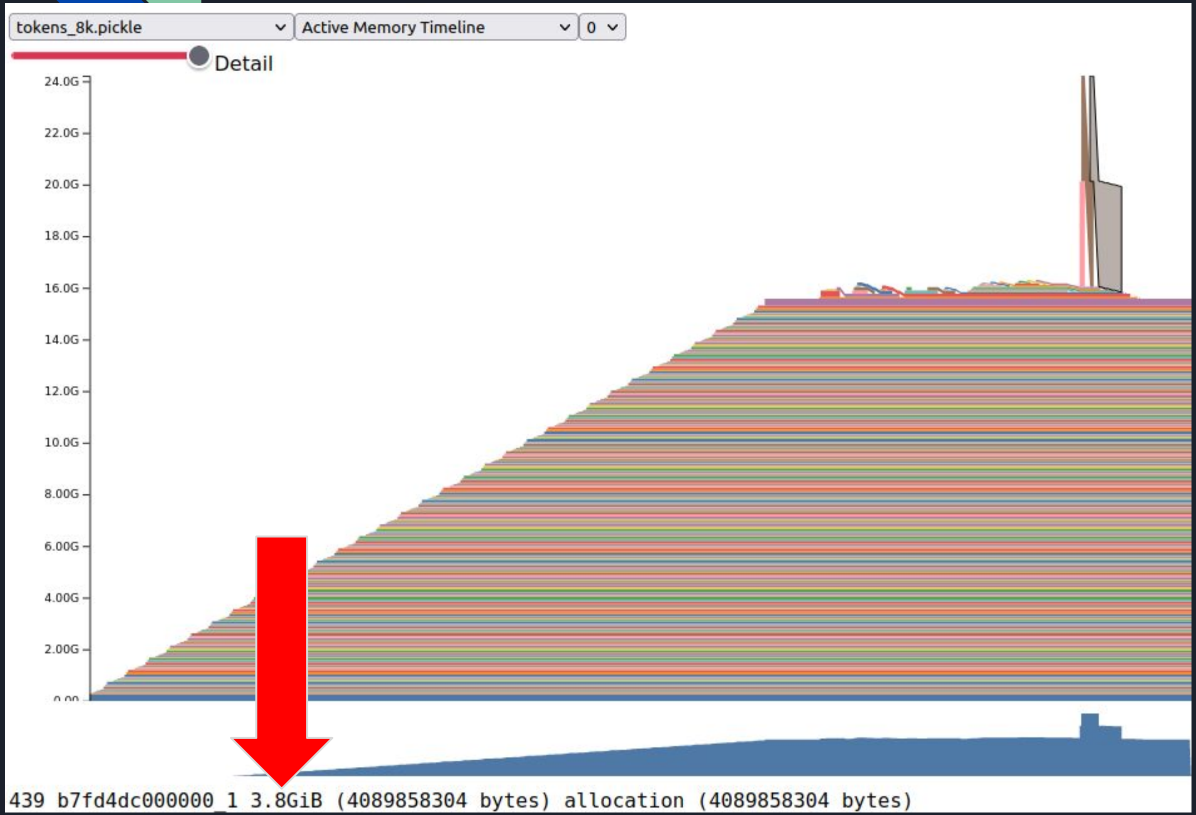


Quantization to the Rescue (INT4, 4k)



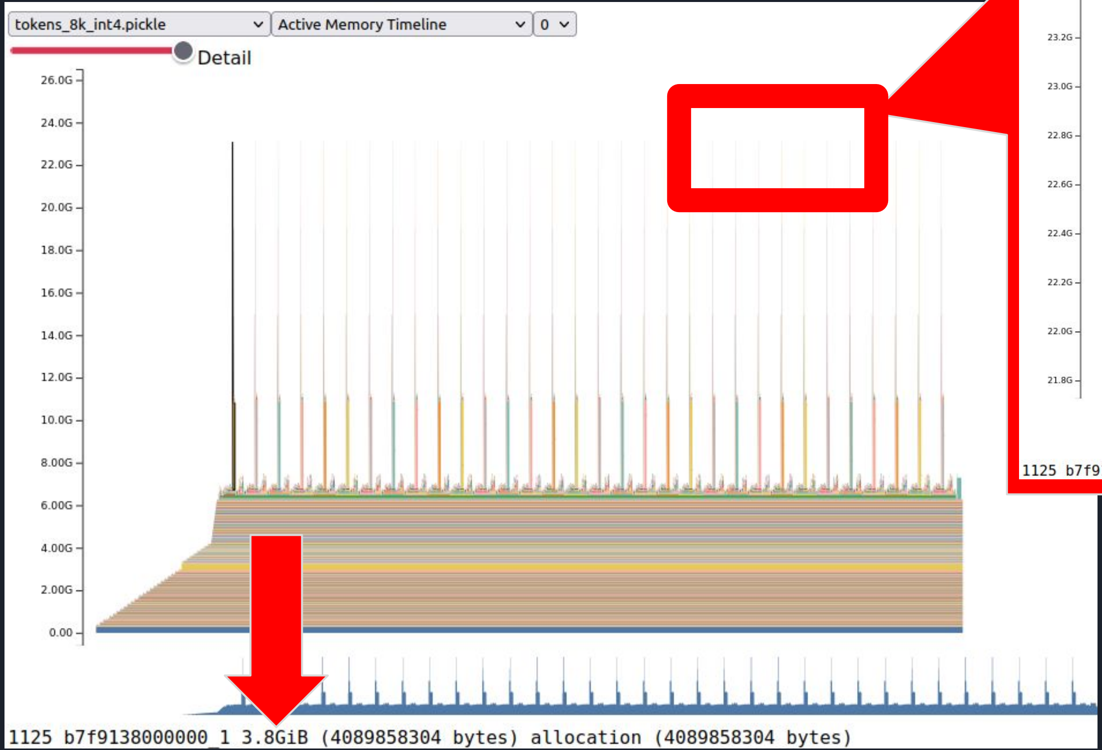
- We can successfully generate our 10 tokens by quantizing the model weights
- Intermediate memory requirements still 1.9 GiB
- KV entries still 16 bit
  - 487.8 MiB (~2x 243.8 MiB)

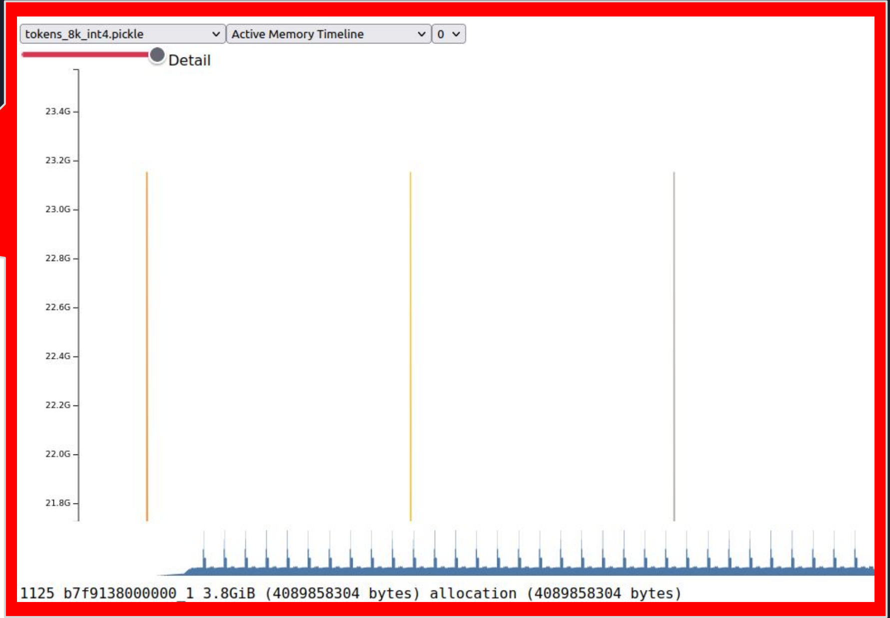
PyTorch Eager Attention at 8k tokens (BF16)



- Immediately OOMs while attempting to generate first token with 3.8 GiB allocation

The Limits of Quantization (Eager Attn, 8k tokens, INT4)



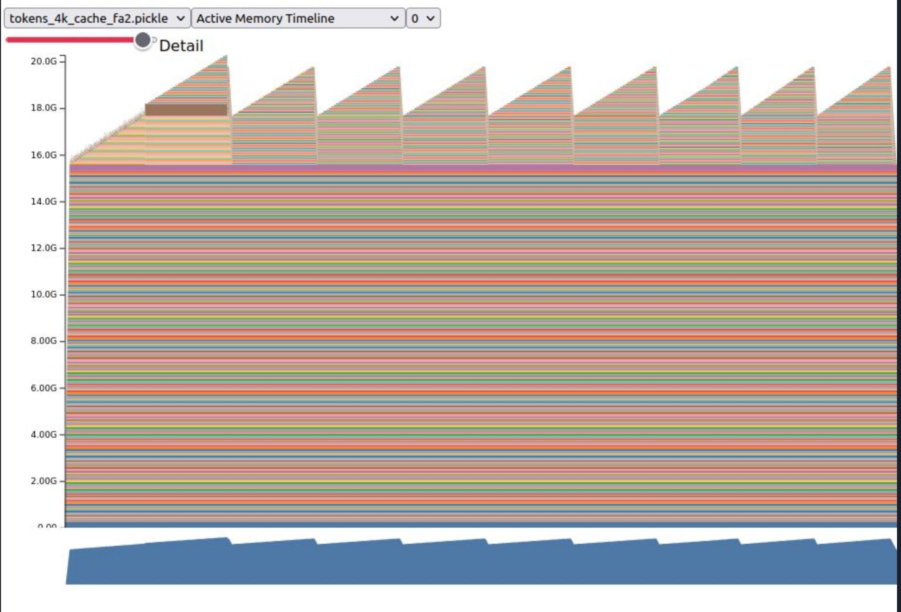


- 16.4 GiB intermediate allocation
  - (23.1 GiB - 6.7 GiB)
  - PyTorch mem_viz reads 3.8 GiB?
- Nearly OOM

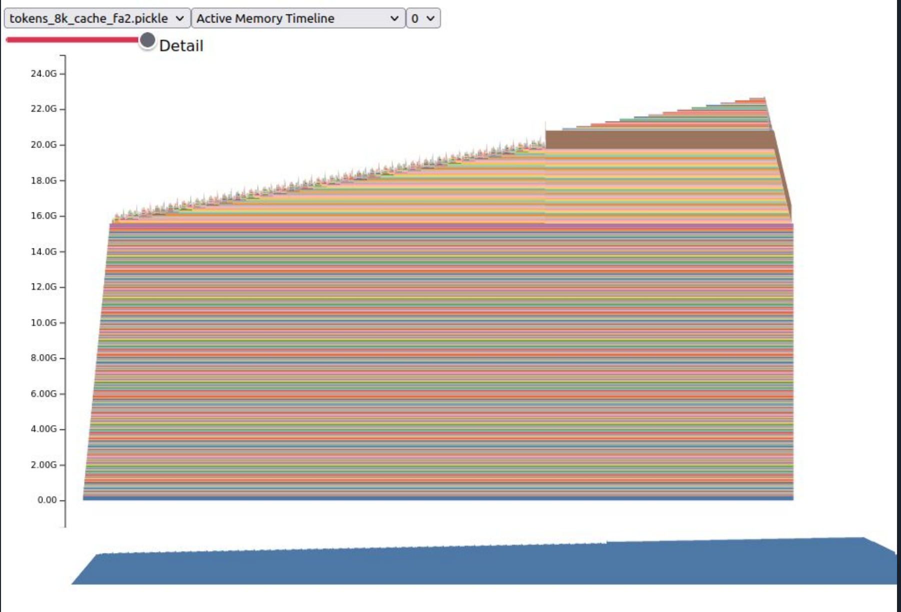
Flash Attention v2 at 8k tokens (BF16)

- 167.9 MiB intermediate allocation
  - 2x 83.9 MiB (4x 42 MiB!!)
- 975.8 MiB KV blocks
  - 2x 487.8 MiB, 4x 243.8 MiB

HuggingFace KV Cache



Successful 10 token generation; FA2, 4k seq len, BF16

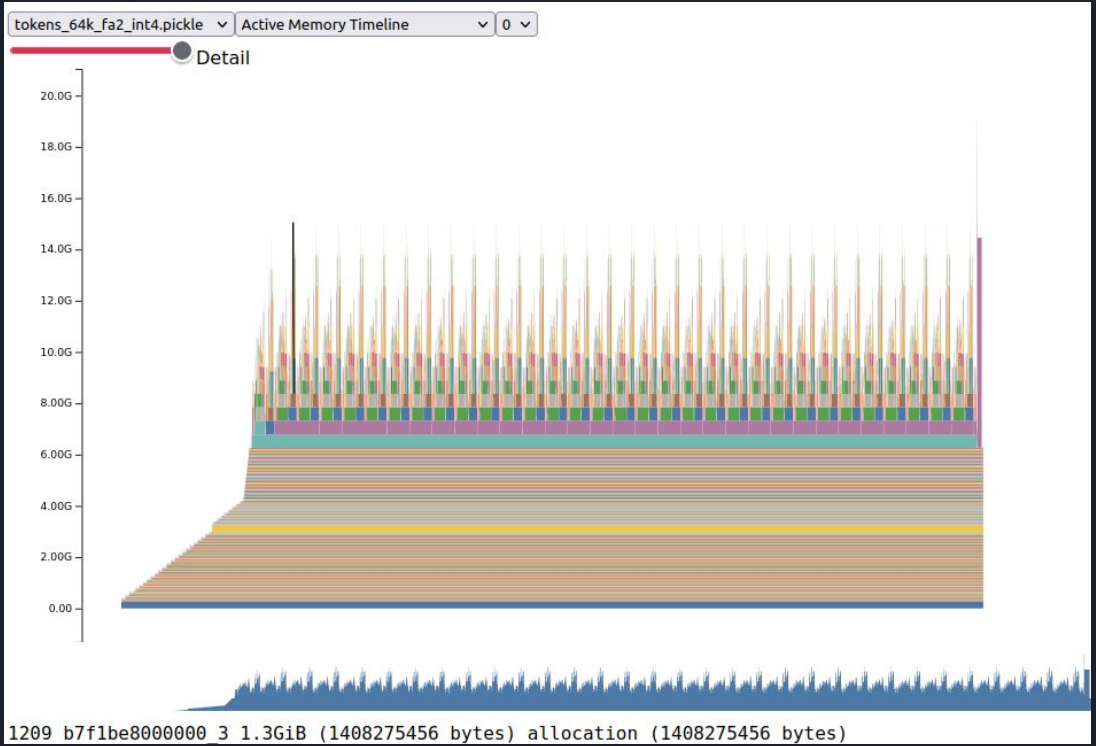


OOM during token 1 generation; FA2, 8k seq len, BF16

VLLM KV Cache

- Able to generate with up to a 15k token KV cache with a 3090
  - 16 bit weights, 16 bit KV cache
- Allocates up to `gpu_memory_utilization` during startup
- Does CUDA graph tracing as part of startup
  - Without deeper magic you can't probe the inner allocations

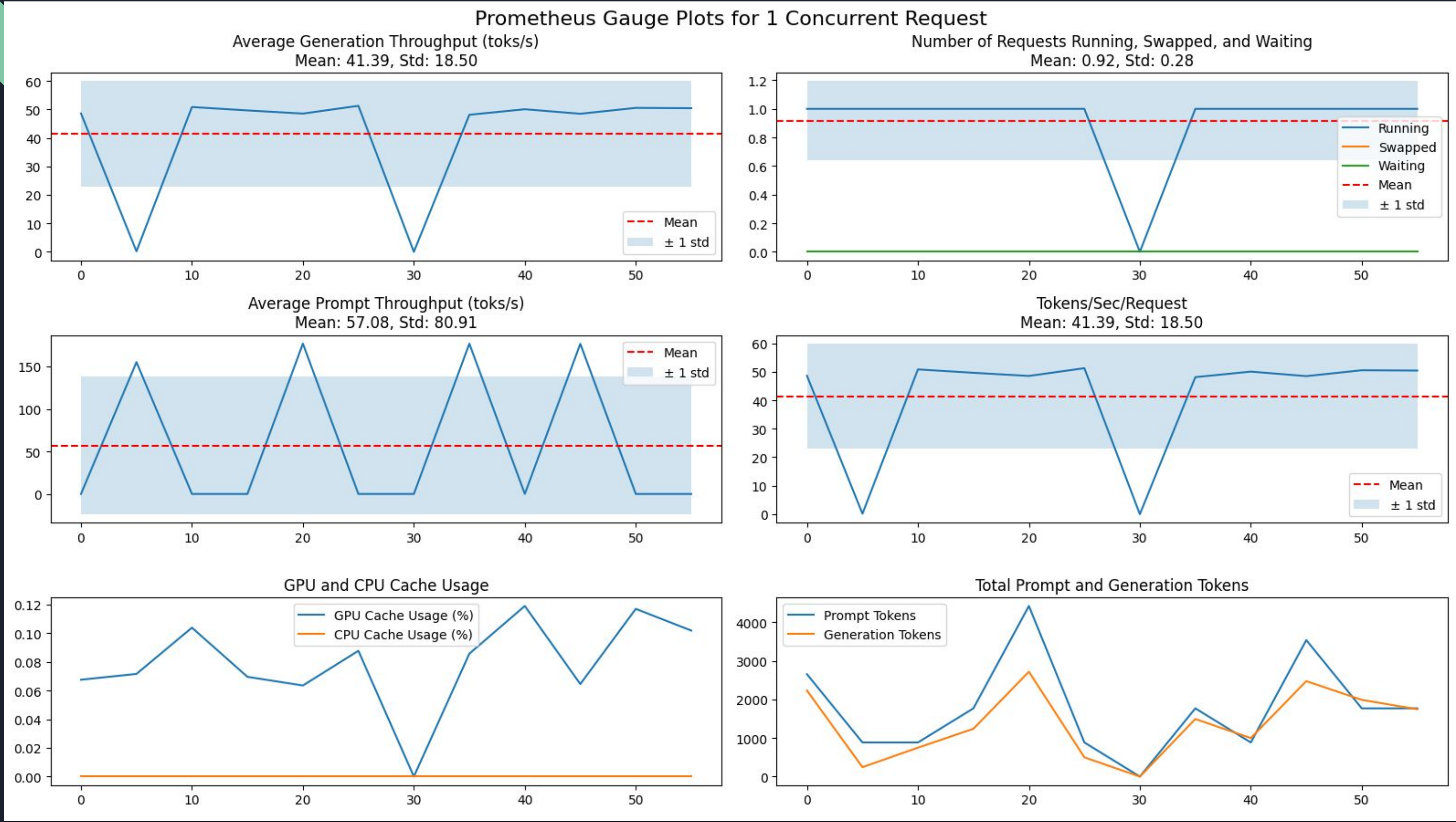
How far can you take Flash Attention?



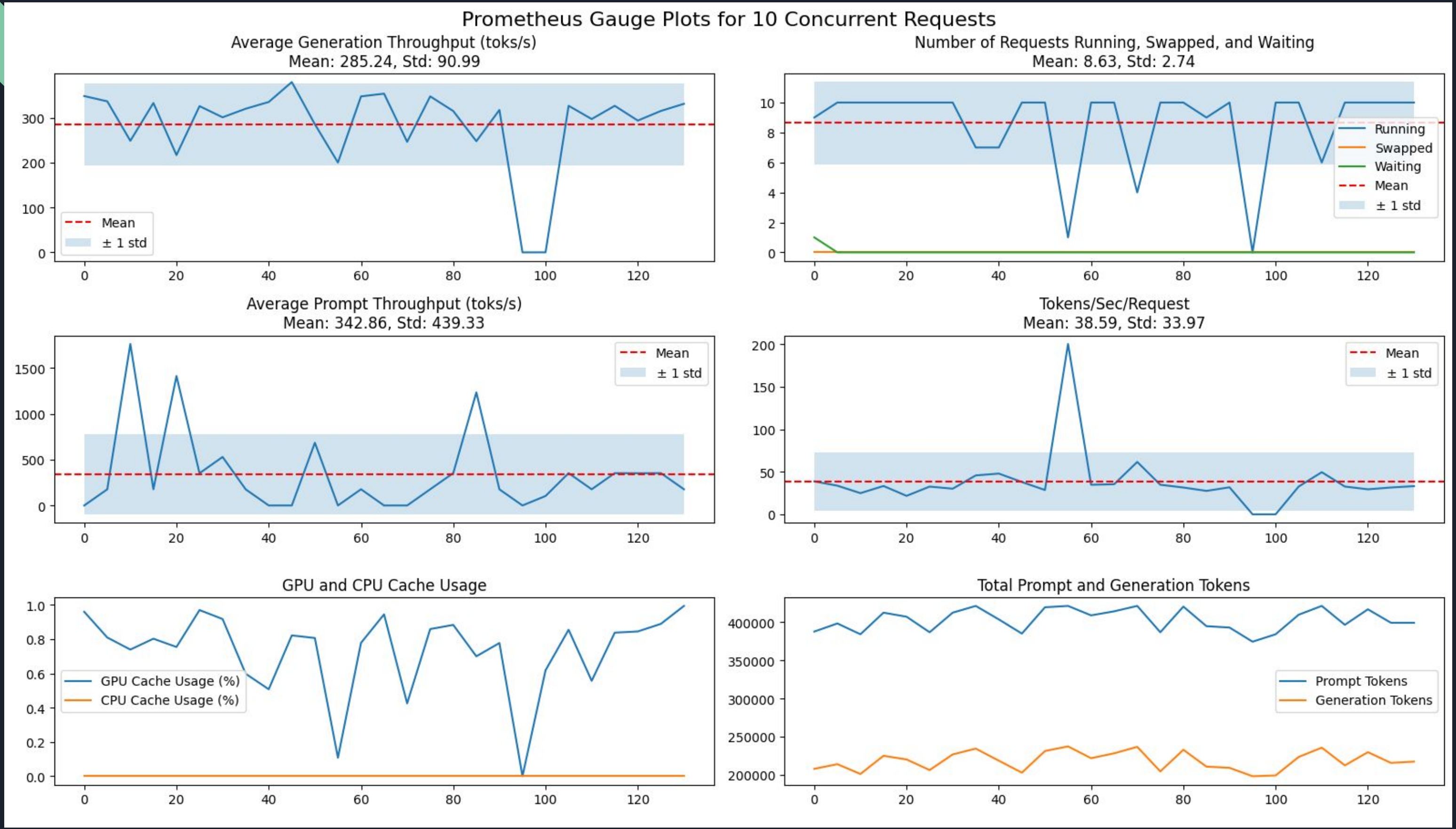
- At 64k tokens w/ INT4 weights, 1.3 GiB intermediate memory usage
  - 32 x 42 MiB
  - Linear scaling!
- In INT4 on a 3090…between 64k and 128k tokens

Multiple Inference Request Memory Allocation Analysis (VLLM and DCGM)

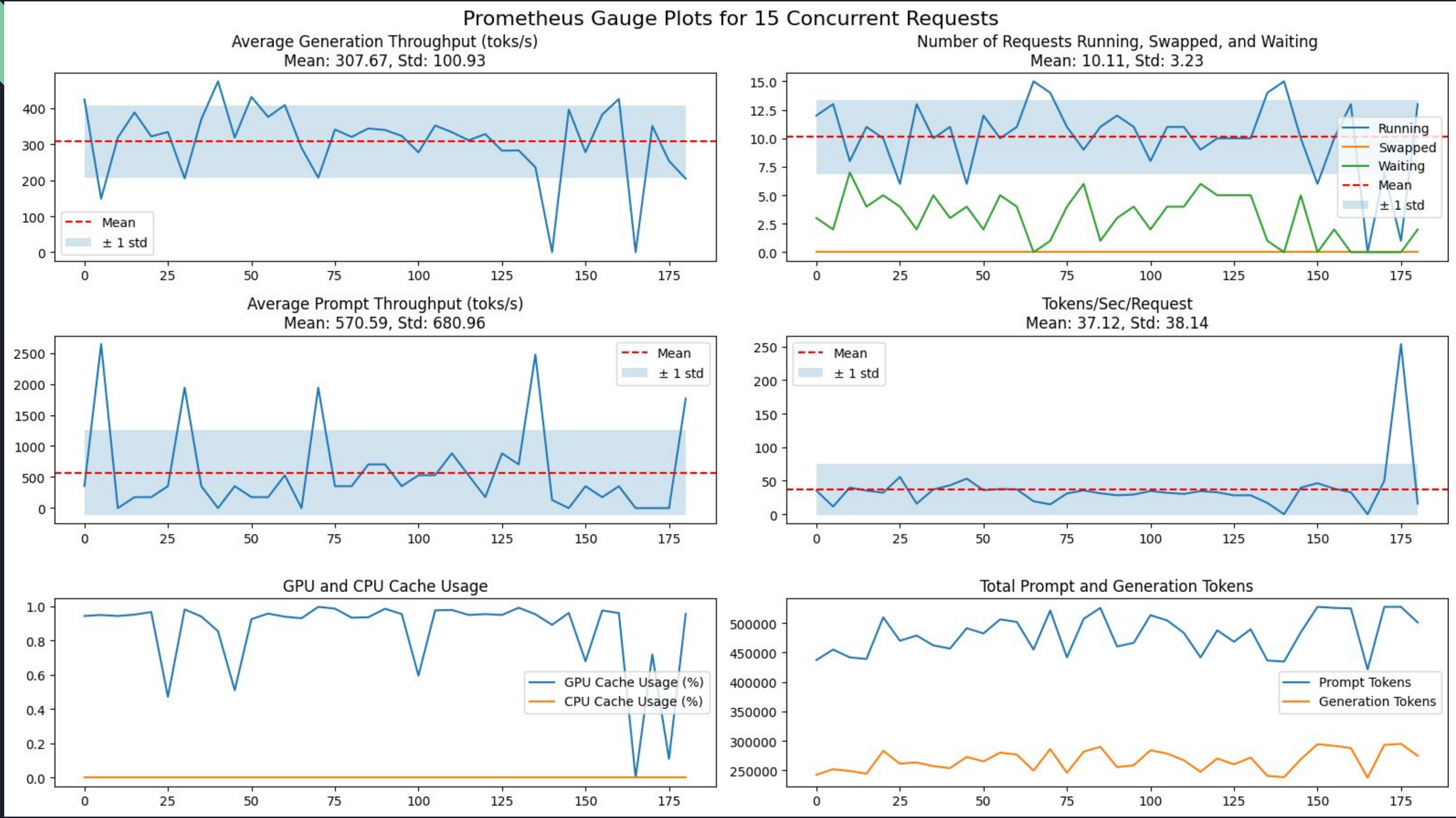
VLLM: 1 Concurrent Request



VLLM: 10 Concurrent Requests



VLLM: 15 Concurrent Requests



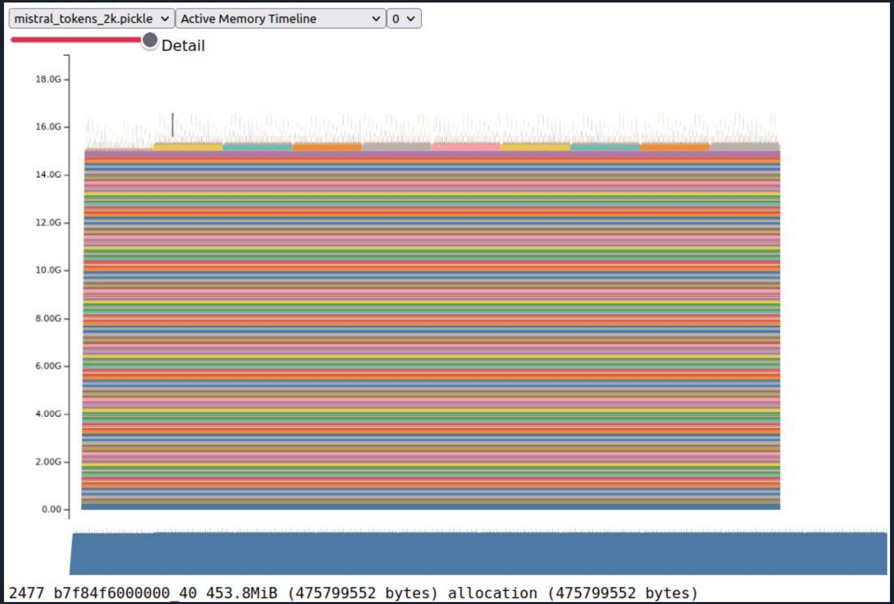
VLLM: 50 Concurrent Requests

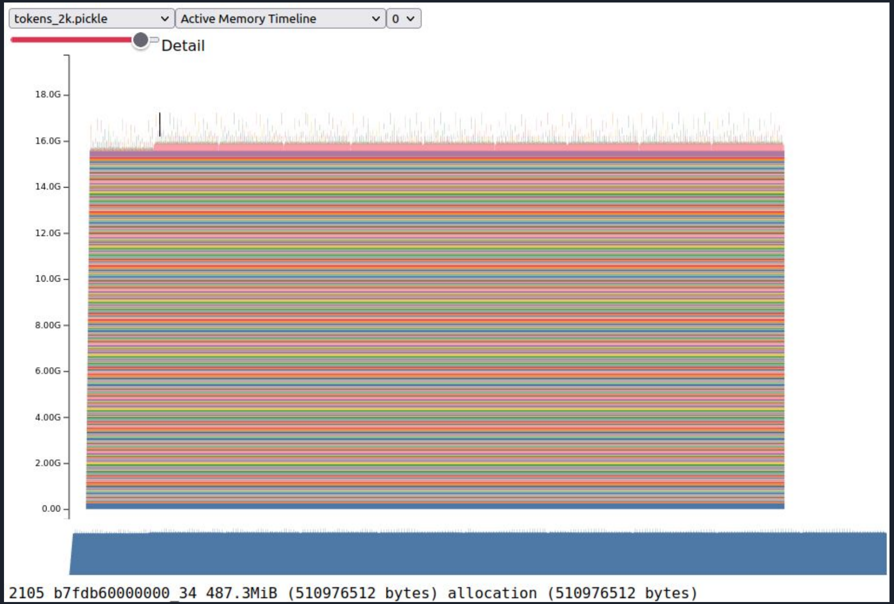
Additional Memory Improvements?

- KV Cache quantization
  - One option for decreasing KV cache size is using half precision for the KV cache
  - This is not supported widely in open source…VLLM only supports 16 bit and FP8 8 bit (Hopper GPUs only)
- Sparse Attention Implementations
  - I.e. Lineformer, BigBird, Mistral Sliding Window, etc
  - Require modifications to underlying models
  - Not well supported by continuous batching/PagedAttention implementations
- Speculative Decoding?
  - Improves per-token latency, not memory usage

What about Mistral?

- Still requires computing attention across 4096 tokens of “sliding window”





Challenge 2: Positional Encoding

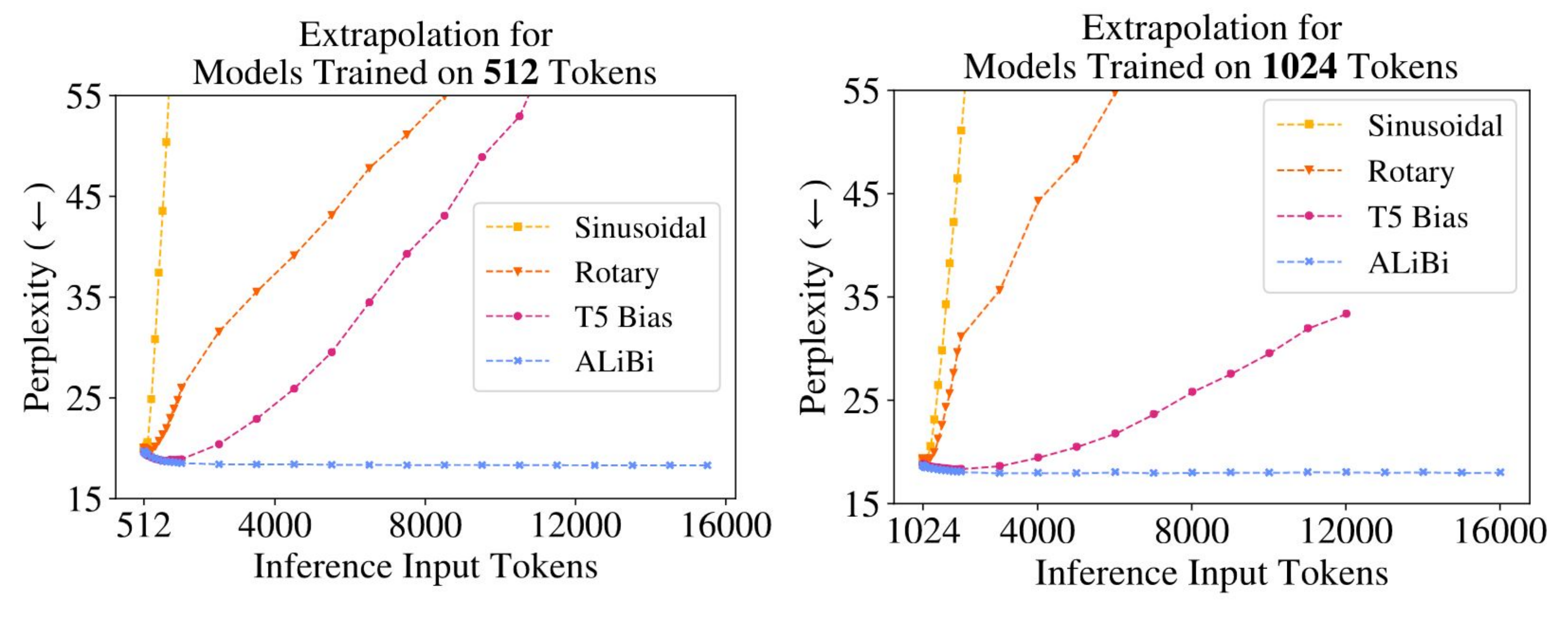
Positional Encodings?

- GPU memory isn't the whole story
- Transformers need a way to encode the relative position of each token in a sequence
    - Otherwise the inputs are effectively a bag of words (tokens)
- INT4 Llama 2 7b + Flash attention on a 3090 can do a >64k token forward pass
    - But it will generate nonsense
    - Why??

Training vs Inference Positional Encodings

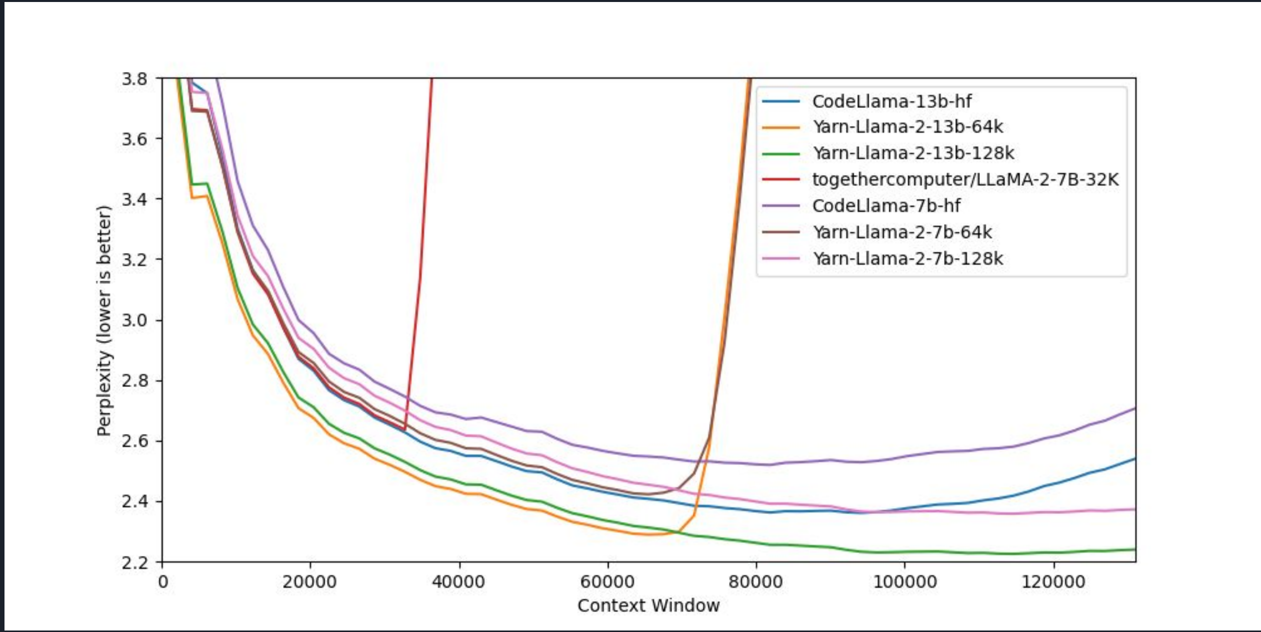
- We have 2 competing demands on sequence length:
  - During pretraining, we want short(er) sequences to reduce intermediate memory usage and increase batch size
  - During inference, we want long(er) sequences to maximize utility
- Many approaches have been proposed to rectify this

Attention with Linear Biases: ALiBi



Reproduced from: Press, Ofir, Noah A. Smith, and Mike Lewis. "Train short, test long: Attention with linear biases enables input length extrapolation." arXiv preprint arXiv:2108.12409 (2021).

YaRN: Yet another RoPE extensioN method



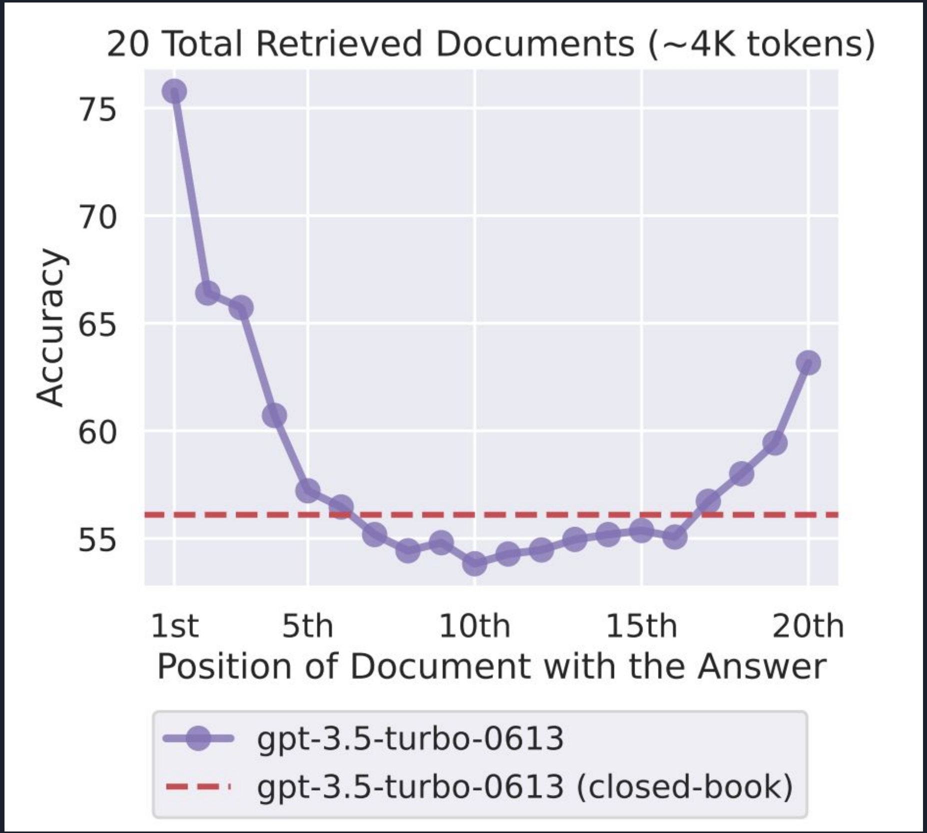
Reproduced from: Peng, Bowen, et al. "Yarn: Efficient context window extension of large language models." *arXiv preprint arXiv:2309.00071* (2023).

Solutions to the Positional Encoding Problem

- Pretraining-based approaches (train short, infer long)
  - ALiBi (Attention with Linear Biases)
- Fine-tuning based approaches (“context length extension fine-tuning”)
  - NTK-aware: CodeLlama
  - YaRN
- Dynamic interpolation approaches (“just interpolate at inference time”)
  - Linear Scaling
  - Dynamic NTK Scaling
  - Dynamic YaRN

Future Directions

- The next issue becomes how well does the model *actually utilize* the provided context window



Reproduced from: Liu, Nelson F., et al. "Lost in the middle: How language models use long contexts." arXiv preprint arXiv:2307.03172 (2023).

Questions?