[Tutorial] Inference Optimization for Foundation Models on AI Accelerators

Youngsuk Park, Kailash Budhathoki, Jonas Kübler, Yida Wang

AWS AI

08/25 2pm-5pm

KDD 2024, Barcelona, Spain

This has been a team effort!

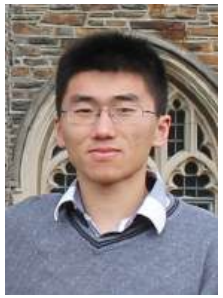
Youngsuk Park
Senior Scientist, AWS AI

Kailash Budhathoki
Senior Scientist, AWS AI

Jonas Kübler
Applied Scientist, AWS AI

Yida Wang
Principal Scientist, AWS AI

Liangfu Chen
Senior SDE, AWS AI



Jiaji Huang
Senior Scientist, AWS AI

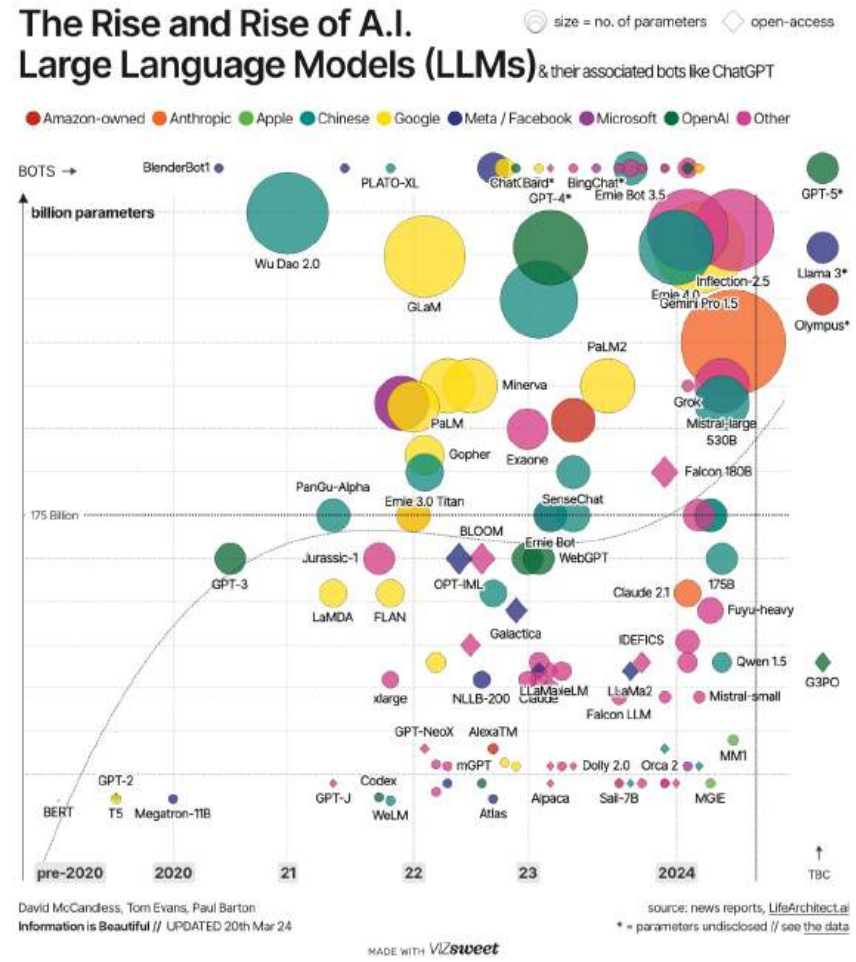
Matthäus Kleindessner
Senior Scientist, AWS AI

Luke Huan
Principal Scientist, AWS AI

Volkan Cevher
Amazon scholar, AWS AI

George Karypis
Senior Principal Scientist, AWS AI

Rise of Large Language Models and Cost



The rise and rise of AI-based Large Language Models (LLMs) like GPT4, LaMDA, LLaMa, PaLM and Jurassic-2.

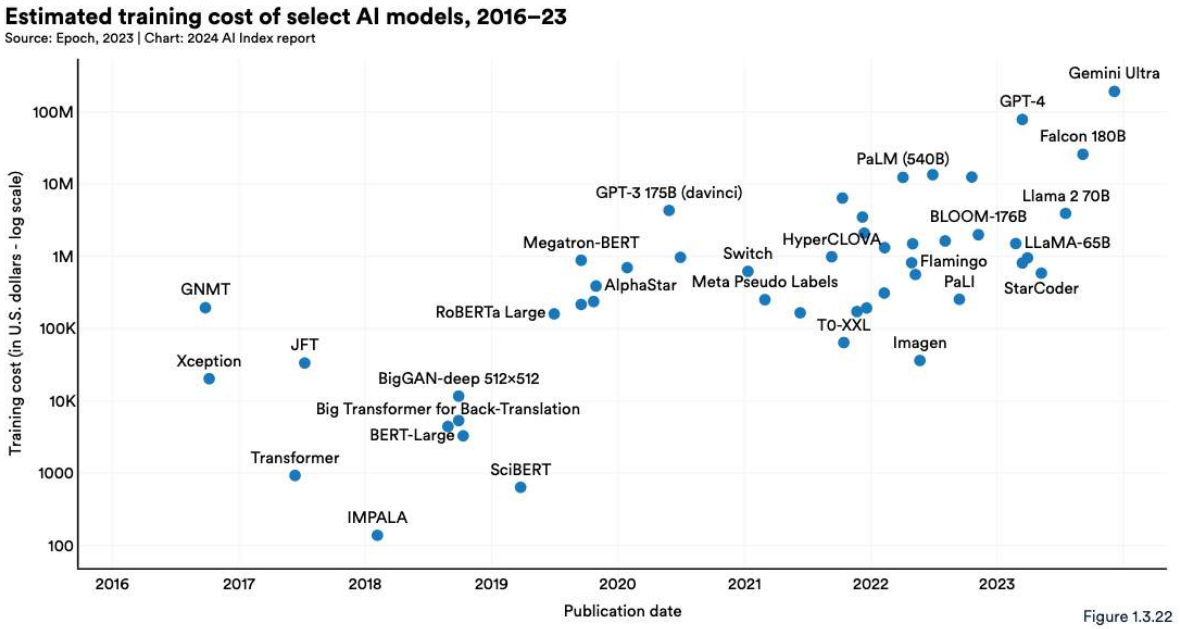


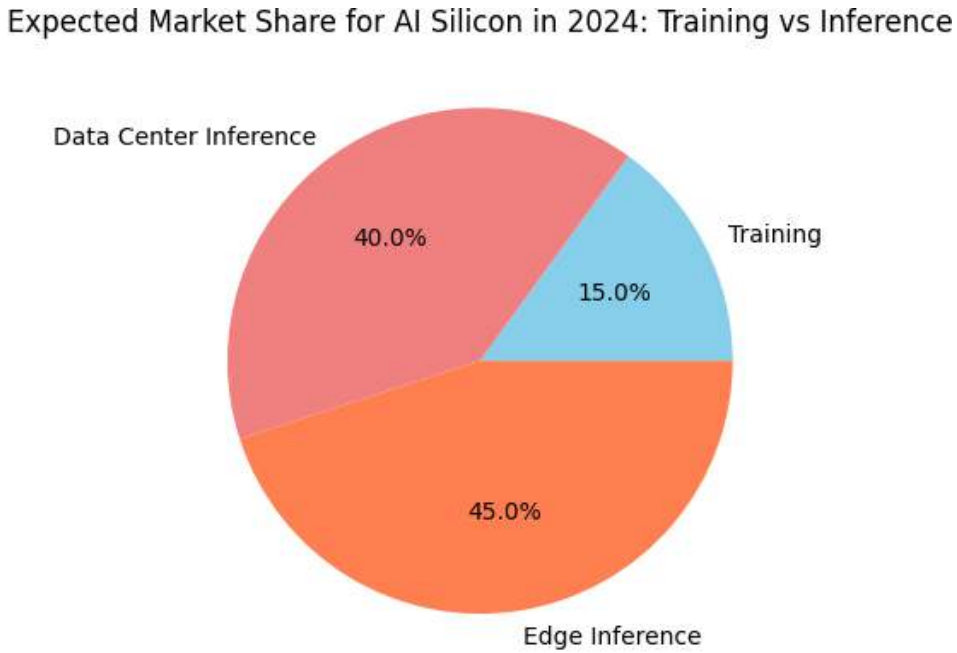
Figure 1.3.22

[Image credit:
(left) informationisbeautiful.net, LifeArchitect.ai
(right) Stanford AI Report, 2024]

Why inference optimization?

- Projected market size of inference vs training
  - LLM pre-training: one time, or update every month
  - LLM inference: far more frequent for serving millions of users with low latency.

- Inference optimization is essential to meet serving criteria of latency and energy cost
  - E.g., GPT3 (175 billion parameters), inference optimization can reduce from a few second to millisecond or lower to serve ChatGPT.



[Source: “The AI chip market landscape”, TECHSPOT, 2023]

Outline of tutorials

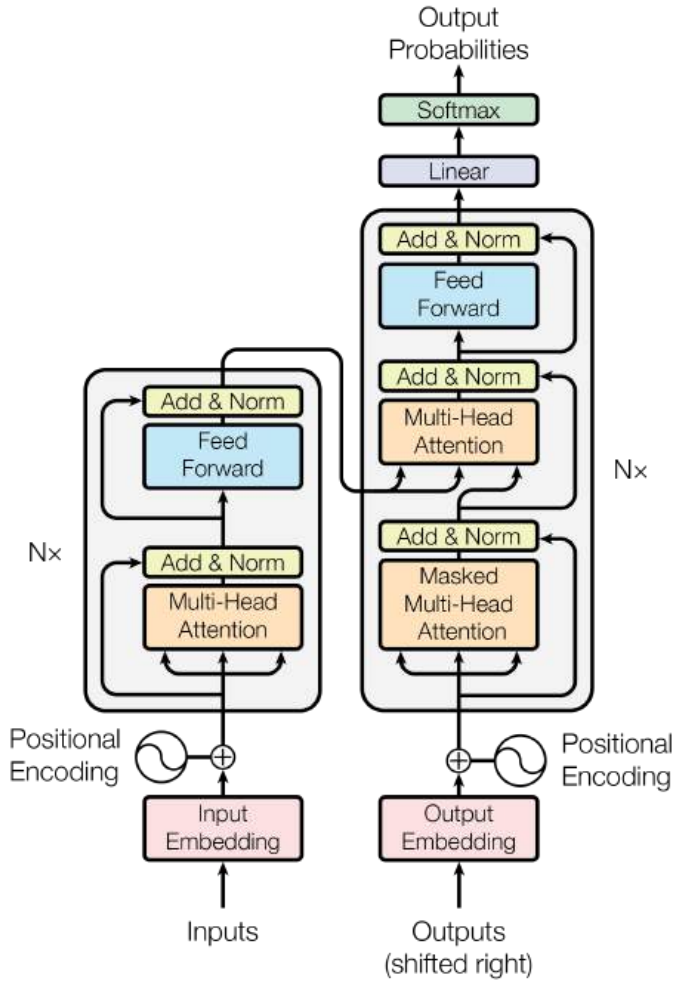
- Primer on LLM inference
- Primer on deep learning hardware accelerators
- Overview of the space of inference optimization opportunities and solutions
- Primer on model architecture choices
- Primer on system-level optimizations
- Primer on model compression – quantization & pruning & model distillation
- Primer on speculative decoding
- Case study using AWS Trn

Primer on LLM inference

Transformer models, pre-filling, auto-regressive decoding, key-value caches, compute complexity, etc.

Overview of the Transformer architecture

- Transformer models with attention module is most popular in LLM
- Consists of following components
  - Positional encoding
  - Transformer layer
    - Residual network
    - Multi-headed attention
      - Self-attention
      - Cross-attention
    - LayerNorm
    - Feed Forward (MLP)
  - Softmax

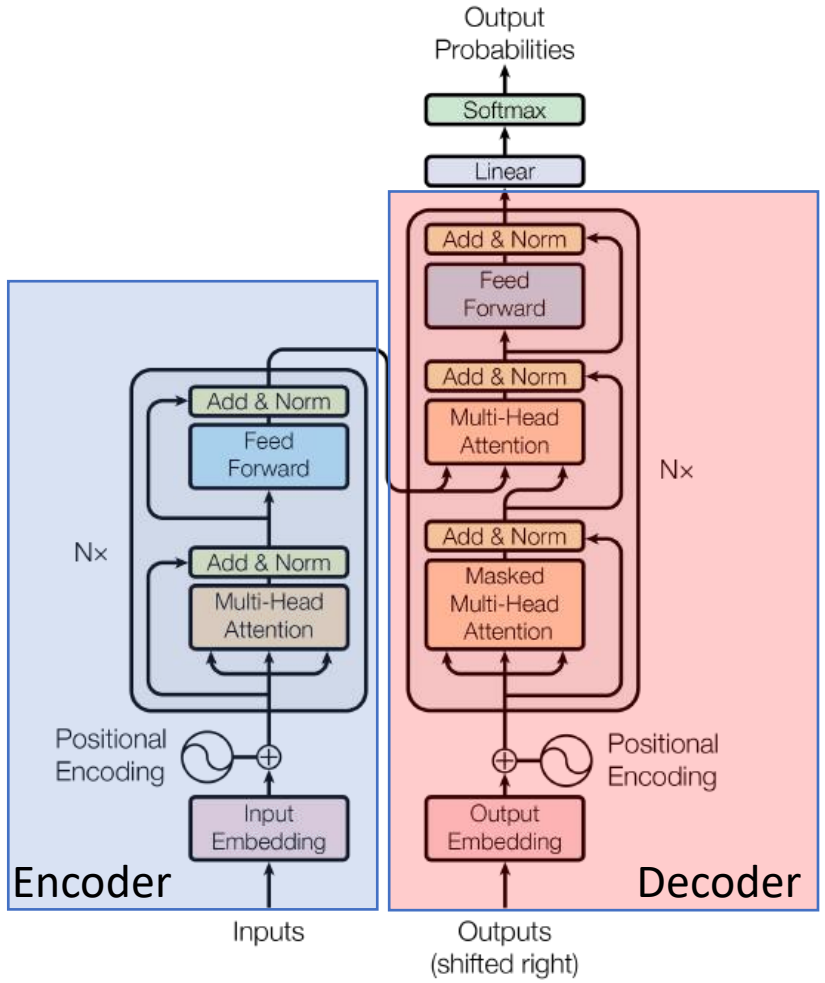


[Vaswani et al., 2017] the Transformer architecture

Vaswani et al., 2017. Attention is All You Need. NeurIPS 2017

Overview of the Transformer architecture

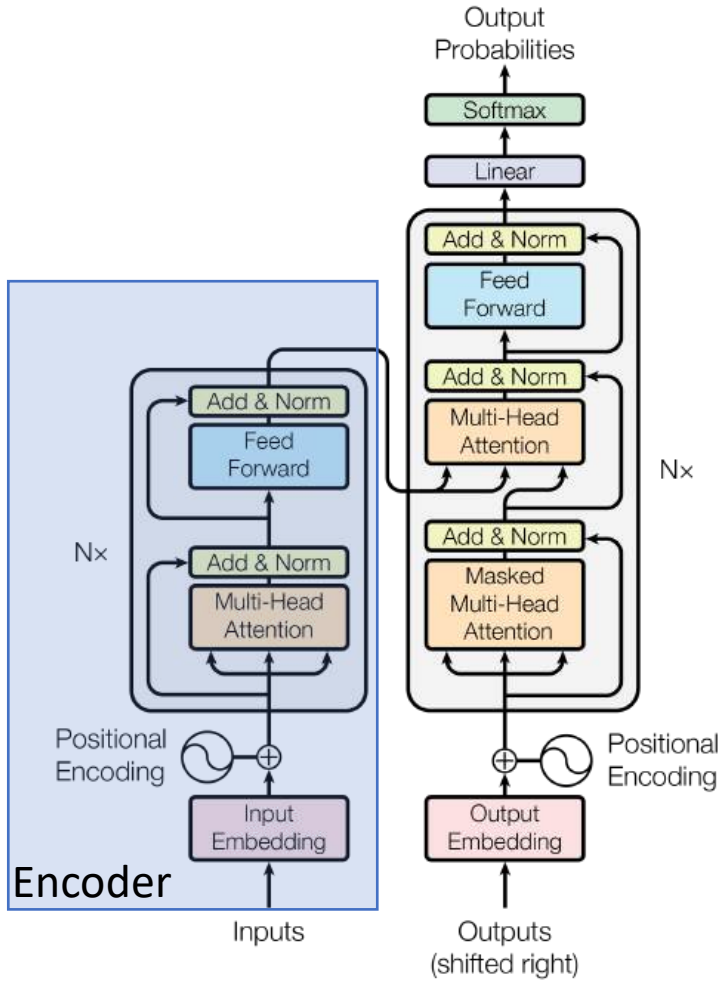
- LLM types are categorized based on encoder and decoder components



[Vaswani et al., 2017] the Transformer architecture

Overview of the Transformer architecture

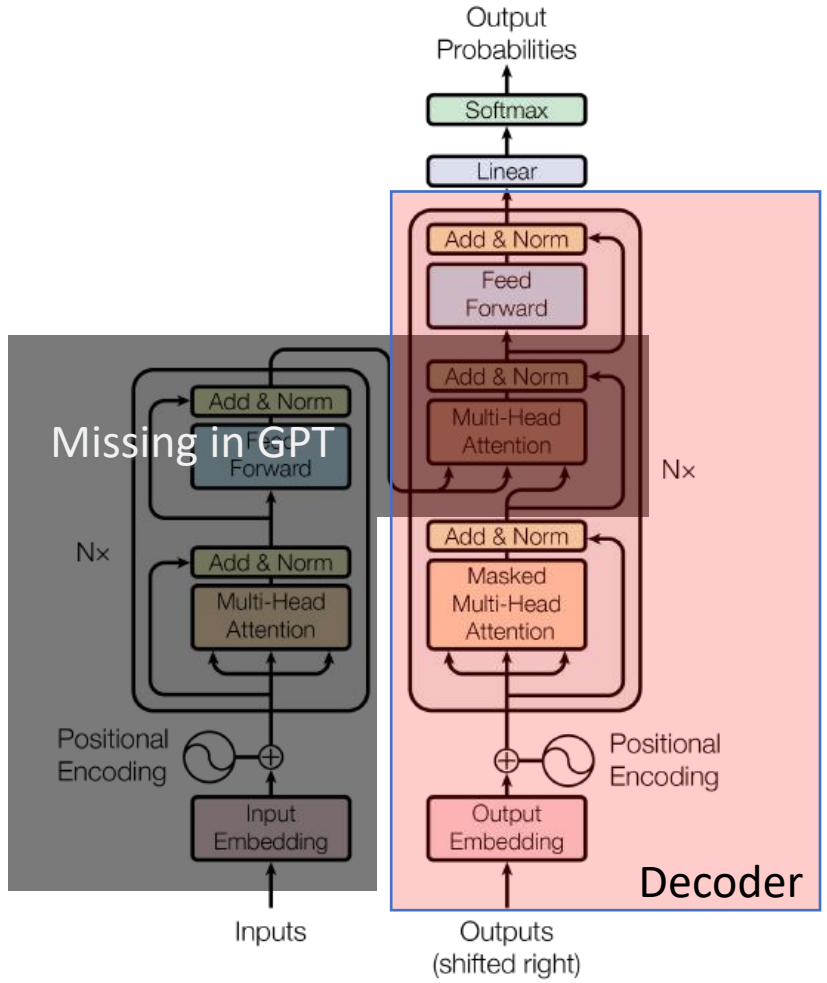
- LLM types are categorized based on encoder and decoder
  - Encoder only models
    - Input seq -> embeddings
    - E.g., BERT [Devlin et al., 2019]



[Vaswani et al., 2017] the Transformer architecture

Overview of the Transformer architecture

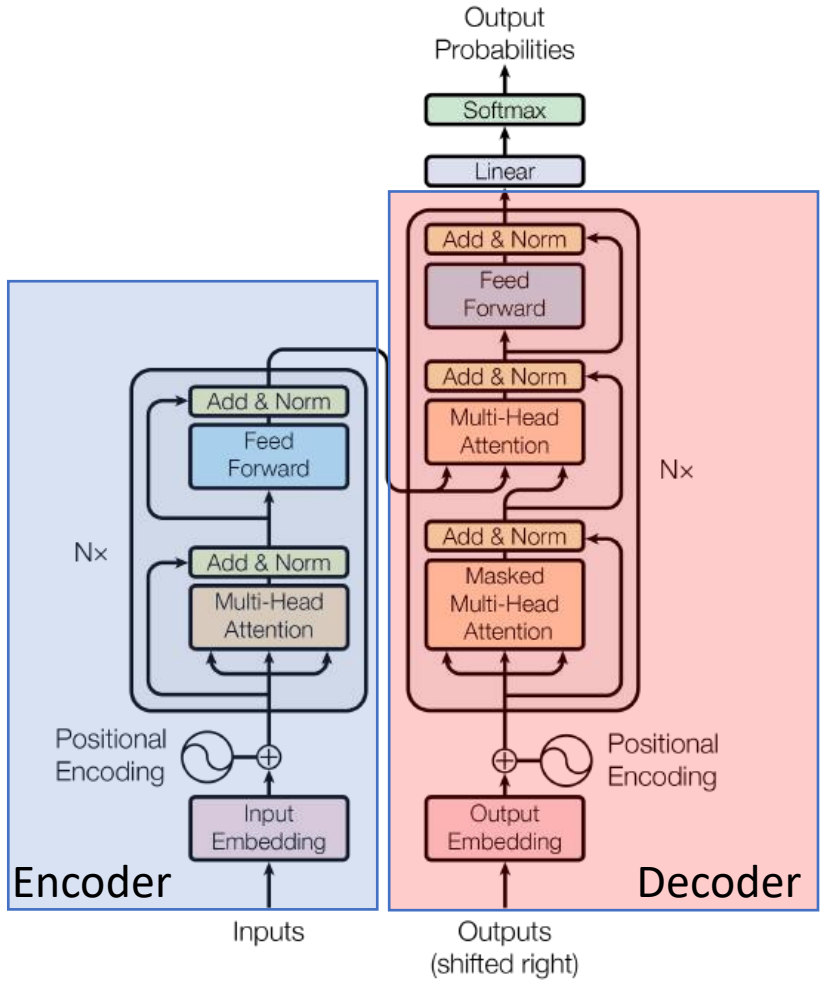
- LLM types are categorized based on encoder and decoder
  - Encoder only models
    - Input seq -> embeddings
    - E.g., BERT [Devlin et al., 2019]
  - Decoding only models
    - Input seq -> next output prob
    - E.g., GPT [Brown et al., 2020], LLaMa [Touvron et al., 2023]



[Vaswani et al., 2017] the Transformer architecture

Overview of the Transformer architecture

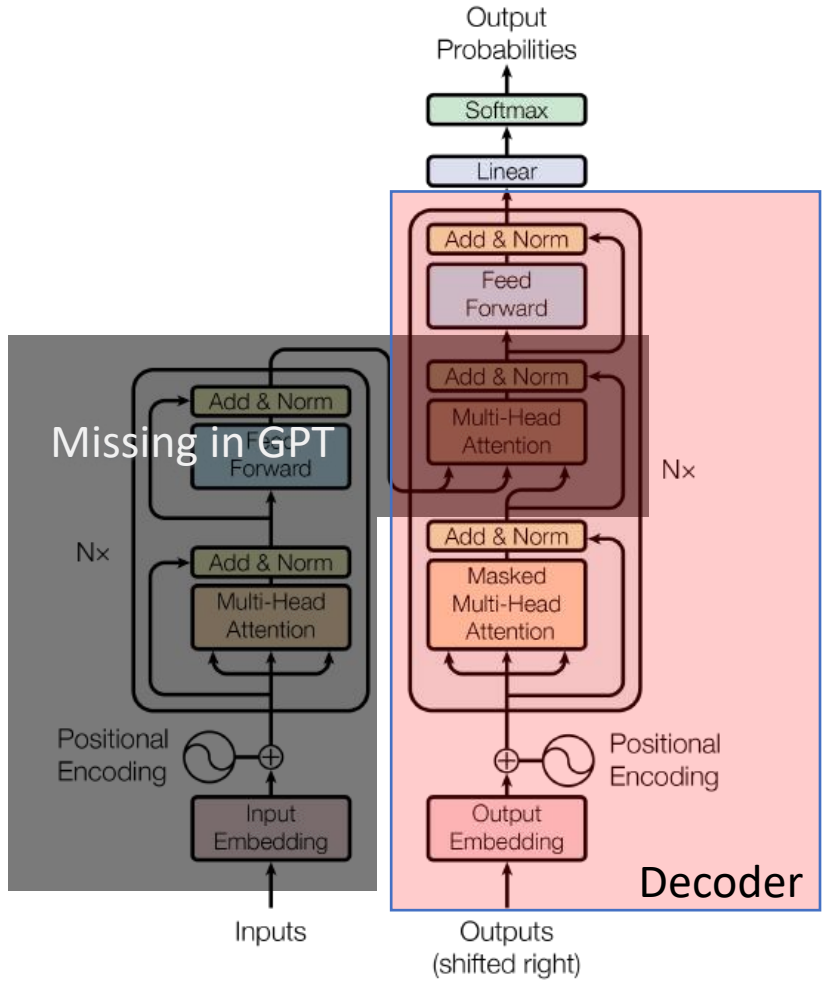
- LLM types are categorized based on encoder and decoder
  - Encoder only models
    - Input seq -> embeddings
    - E.g., BERT [Devlin et al., 2019]
  - Decoding only models
    - Input seq -> next output prob
    - E.g., GPT [Brown et al., 2020], Llama [Touvron et al., 2023]
  - Encoder-Decoder models
    - Input seq -> embeddings -> next output prob
    - E.g., T5 [Colin Raffel et al., 2019]



[Vaswani et al., 2017] the Transformer architecture

Overview of the Transformer architecture

- LLM types are categorized based on encoder and decoder
  - Encoder only models
    - Input seq -> embeddings
    - E.g., BERT [Devlin et al., 2019]
  - Decoding only models
    - Input seq -> next output prob
    - E.g., GPT [Brown et al., 2020], Llama [Touvron et al., 2023]
  - Encoder-Decoder models
    - Input seq -> embeddings -> next output prob
    - E.g., T5 [Colin Raffel et al., 2019]
- Focus on decoding-only for the rest of tutorial



[Vaswani et al., 2017] the Transformer architecture

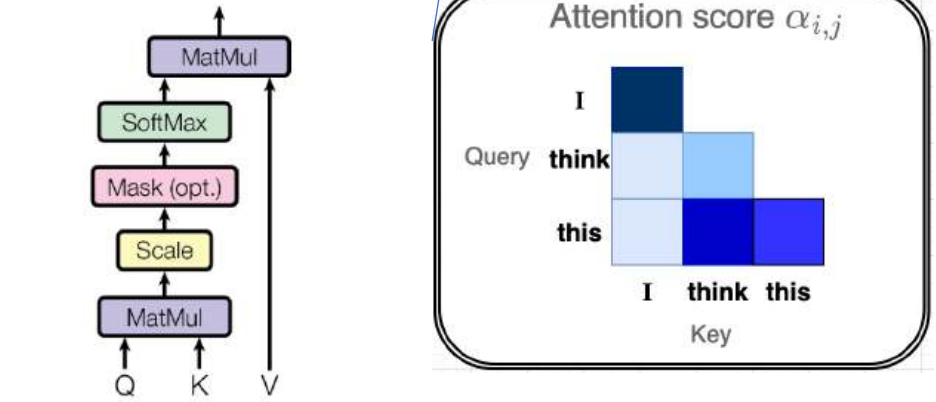
Overview of attention mechanism

- Self-attention captures relationship within input seq $X = [x_1, \dots, x_L]$
  - attention scores identify importance on where to attend more across different input parts

[Vaswani et al., 2017] the Transformer architecture

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

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



Overview of attention mechanism

- Self-attention captures relationship within input seq $X = [x_1, \dots, x_L]$
- Attention procedures
  - Step 1: for each input, compute query (q_j), key (k_j), and value (v_j) via linear projection.
  - Step 2: compute attention scores $\{\alpha_{ij}\}$ across inputs w.r.t. current input query x_i.
  - Step 3: retrieve output (z_j) by weighting values $\{v_j\}$ with scores $\{\alpha_{ij}\}$.

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

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

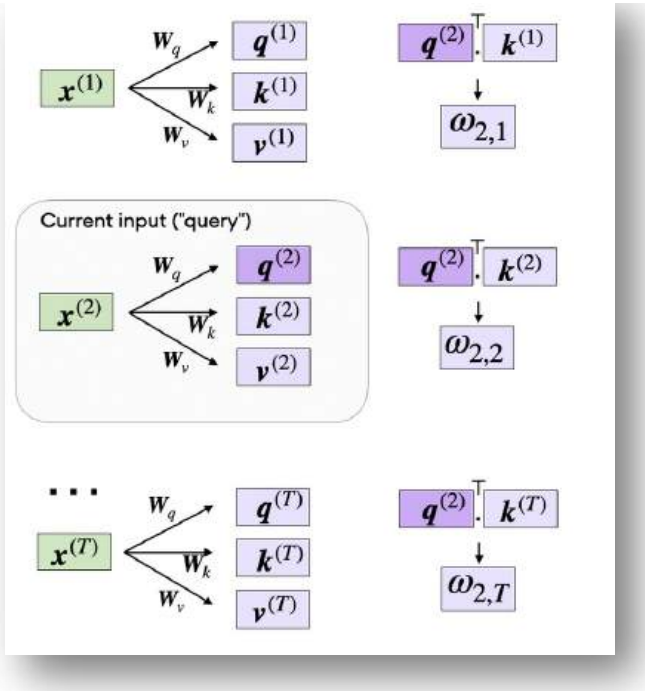


Image source: [Sebastian Raschka, /sebastianraschka.com/blog/2023]

Overview of attention mechanism

- Self-attention captures relationship within input seq $X = [x_1, \ldots, x_L]$
- Attention procedures
  - Step 1: compute query (q_j), key (k_j), and value (v_j) for each input via linear projection.
  - Step 2: compute attention scores $\{\alpha_{ij}\}$ across inputs w.r.t. current input query q_i.
  - Step 3: retrieve output (z_j) by weighting values $\{v_j\}$ with scores $\{\alpha_{ij}\}$.

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

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

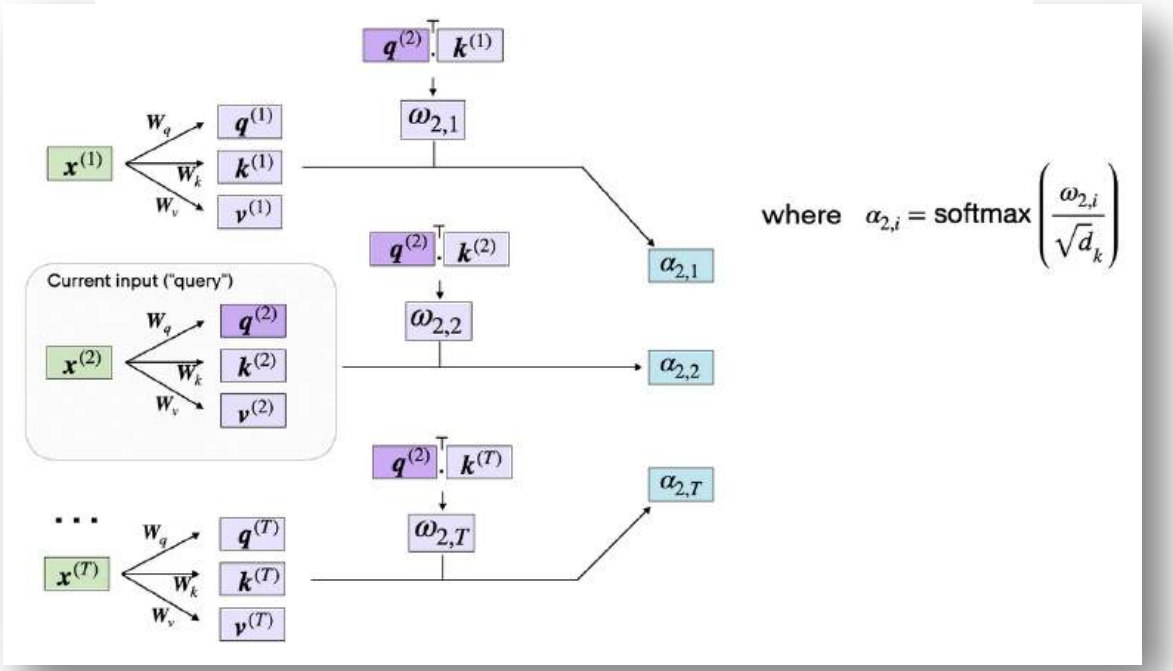


Image source: [Sebastian Raschka, /sebastianraschka.com/blog/2023]

Overview of attention mechanism

- Self-attention captures relationship within input seq $X = [x_1, \ldots, x_L]$
- Attention procedures
  - Step 1: compute query (q_j), key (k_j), and value (v_j) for each input via linear projection.
  - Step 2: compute attention scores $\{\alpha_{ij}\}$ across inputs w.r.t. current input query x_i.
  - Step 3: retrieve output z_i by combining values $\{v_j\}$ with attention scores $\{\alpha_{ij}\}$.

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

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

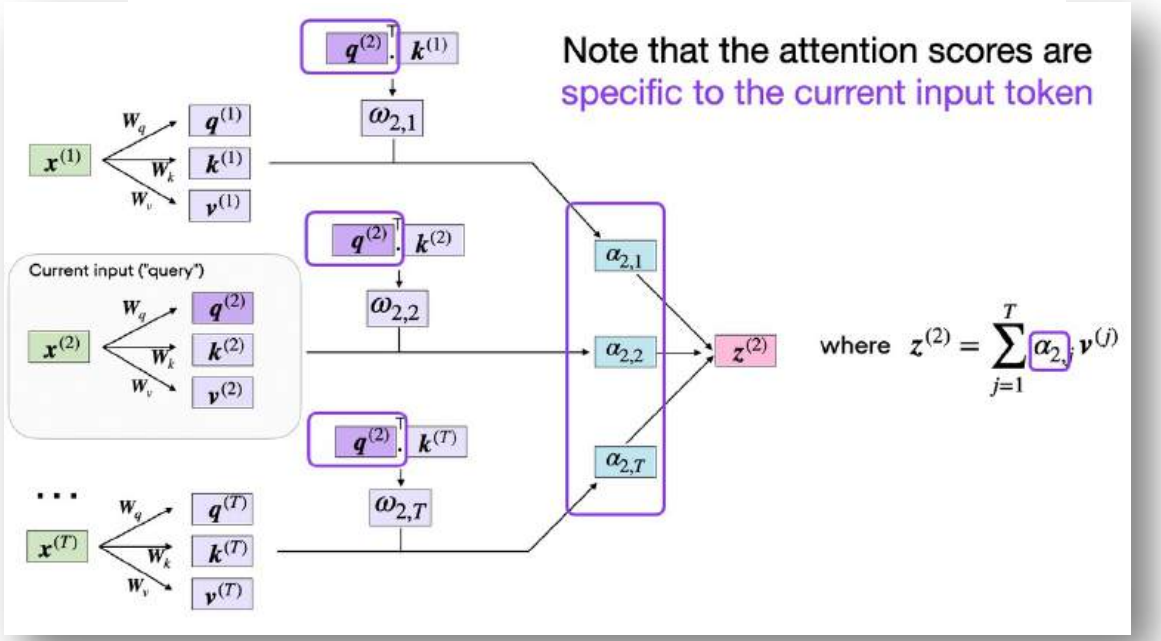


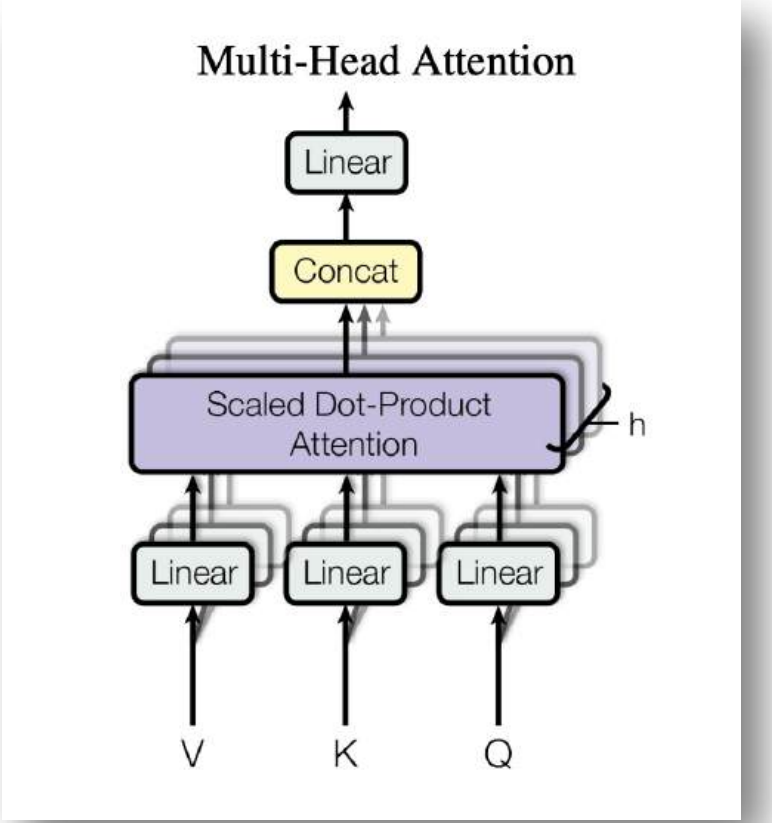
Image source: [Sebastian Raschka, /sebastianraschka.com/blog/2023]

Multi-head attention (MHA) block

- MHA has these h (self-)attentions
  - Hidden dimension (D) for queries, keys, and values is evenly split into multiple parts—each corresponding to a different *attention head*.

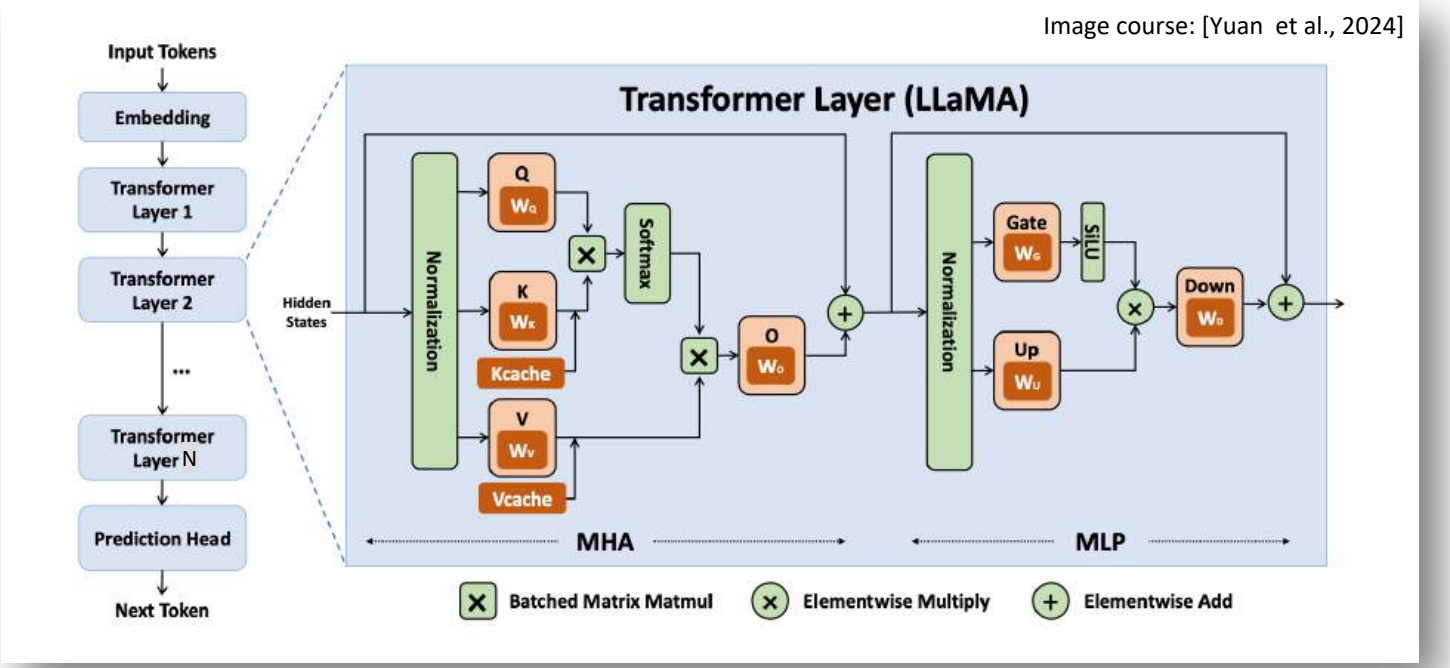
$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

  - Mult-heads, instead of single head, allows to capture distinct patterns within sequence



[Vaswani et al., 2017] the Transformer architecture

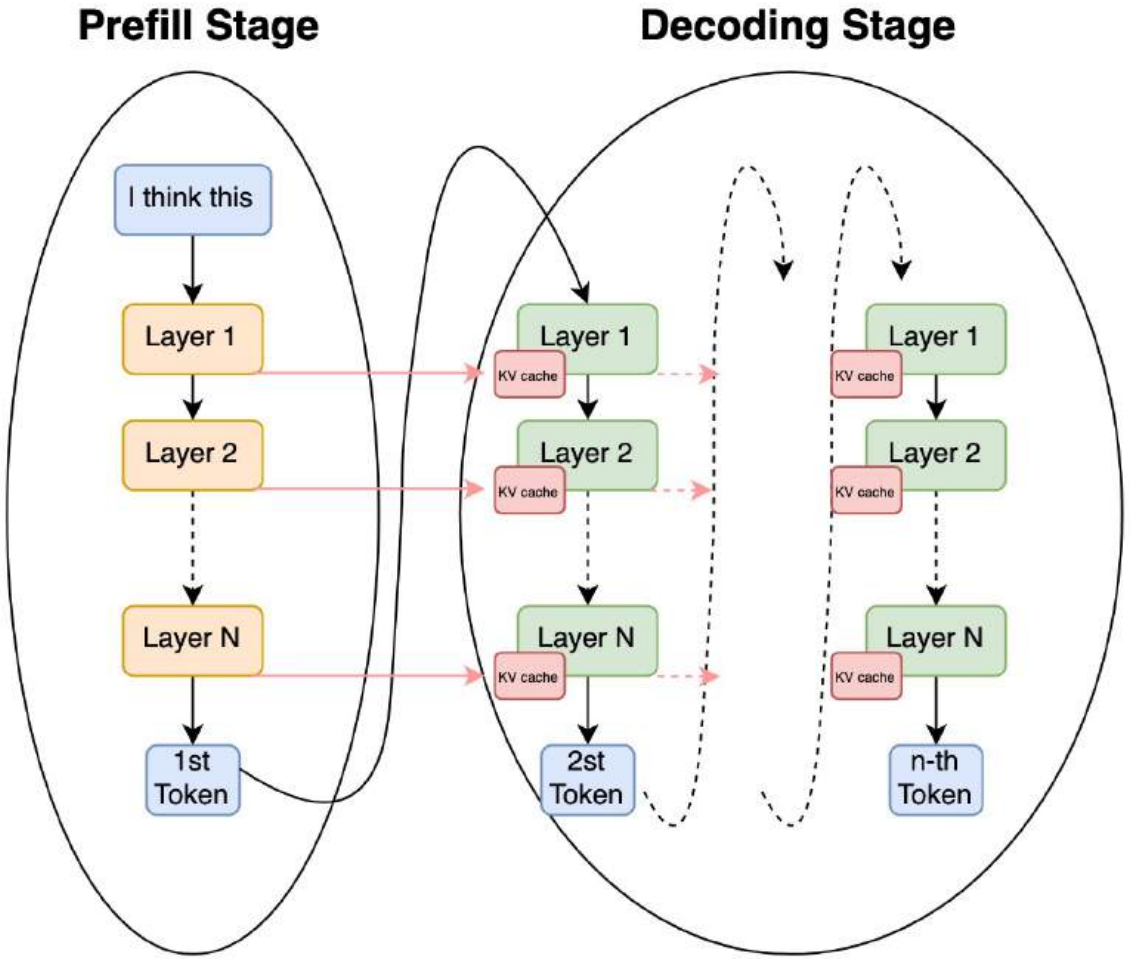
Overview with LLaMa model



- Transformer layer : MHA block + MLP block (projection up/down)
- Other blocks: token embeddings, prediction heads, etc.
- LLaMa2 7B: 32 layers, 4k hidden_dim, 12k intermediate size, 32 heads, 32k vocab

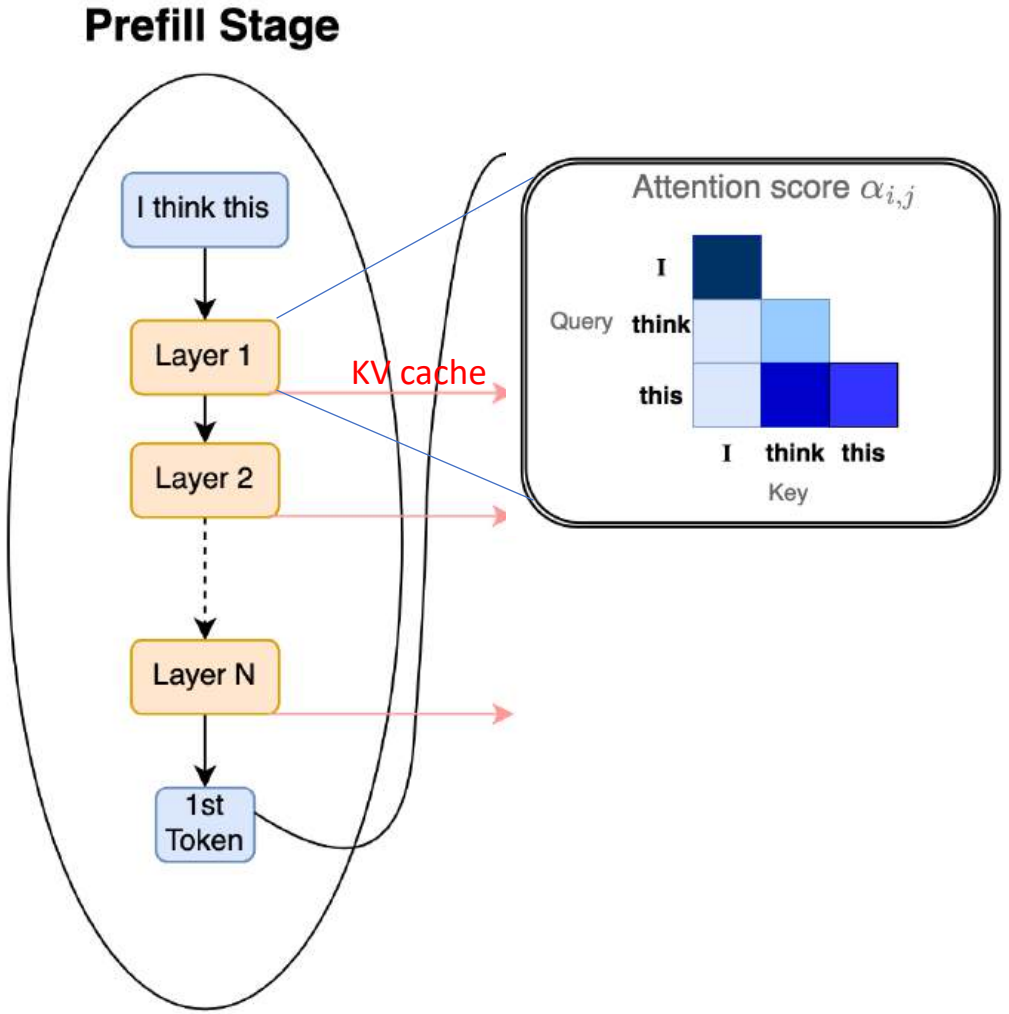
Overview of inference computations

- Inference task
  - Given an input sequence X, use LLMs to generate next n-continuation Y.
- Two-step solution
  - Step 1. Prefill stage
  - Step 2. Decoding stage



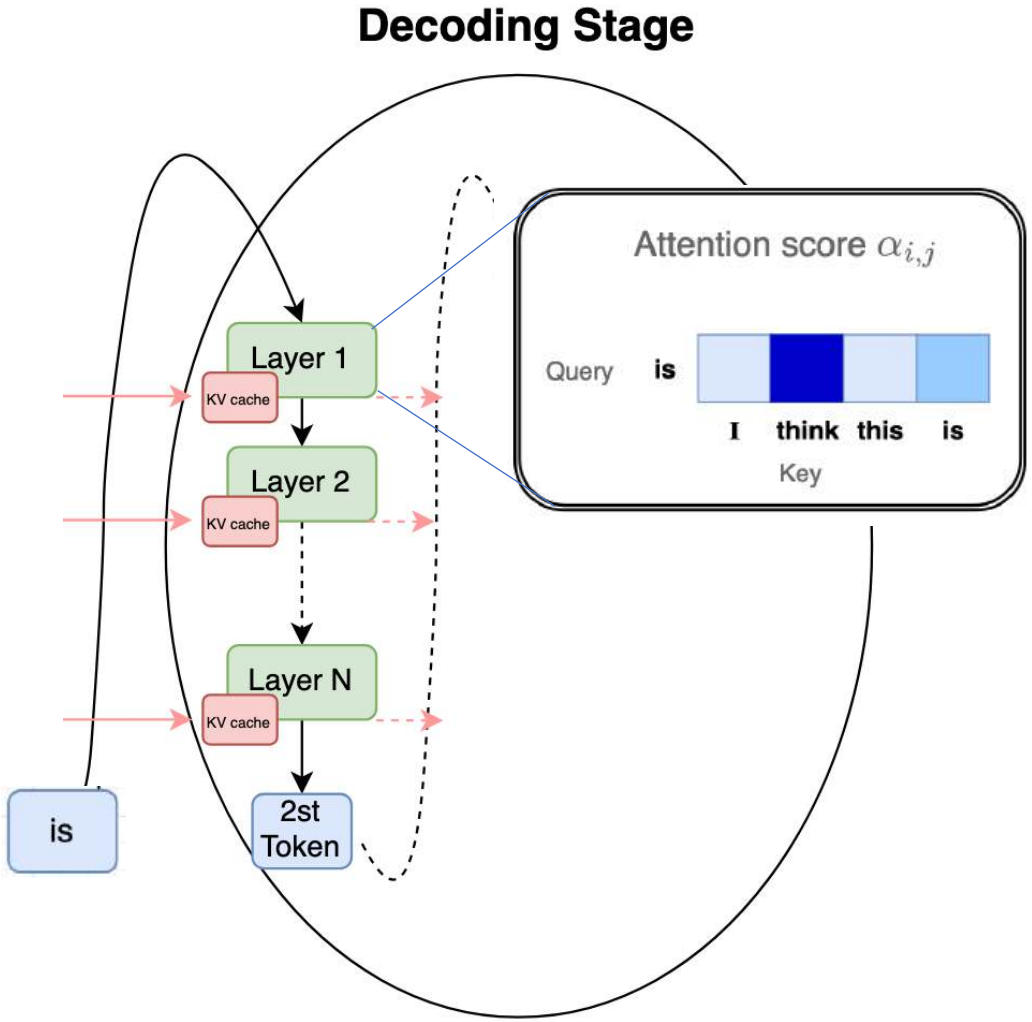
Overview of inference computations

- Inference task
  - Given an input sequence X, use LLMs to generate next n-continuation Y.
- Two-step solution
  - Step 1. Prefill stage:
    - Perform a forward pass on X and save key and value in each layer for input tokens (KV cache).
    - KV cache can be used for the decoding stage
    - Representations of all tokens in each layer is computed in parallel.



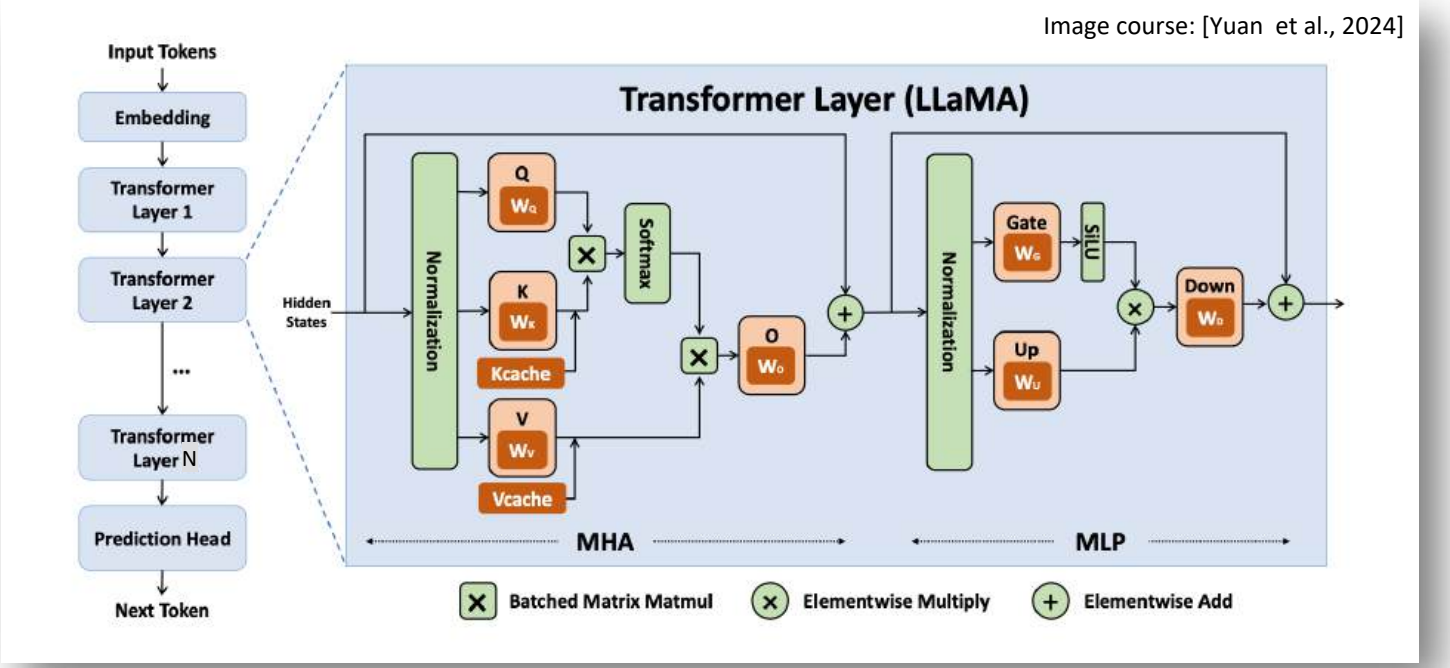
Overview of inference computations

- Inference task
  - Given an input sequence X, use LLMs to generate next n-continuation Y.
- Two-step solution
  - Step 1. Prefill stage
  - Step 2. Decoding stage
    - Generate tokens of Y (and its representation) in an autoregressive manner i.e., sequentially one-at-a-time
    - Can reuse some of previous representation, KV cache
    - Use a token sampling method, e.g., greedy, beam search, etc



Complexity of inference computations

Image course: [Yuan et al., 2024]



Per-layer compute cost of encoding a prefix of length L.

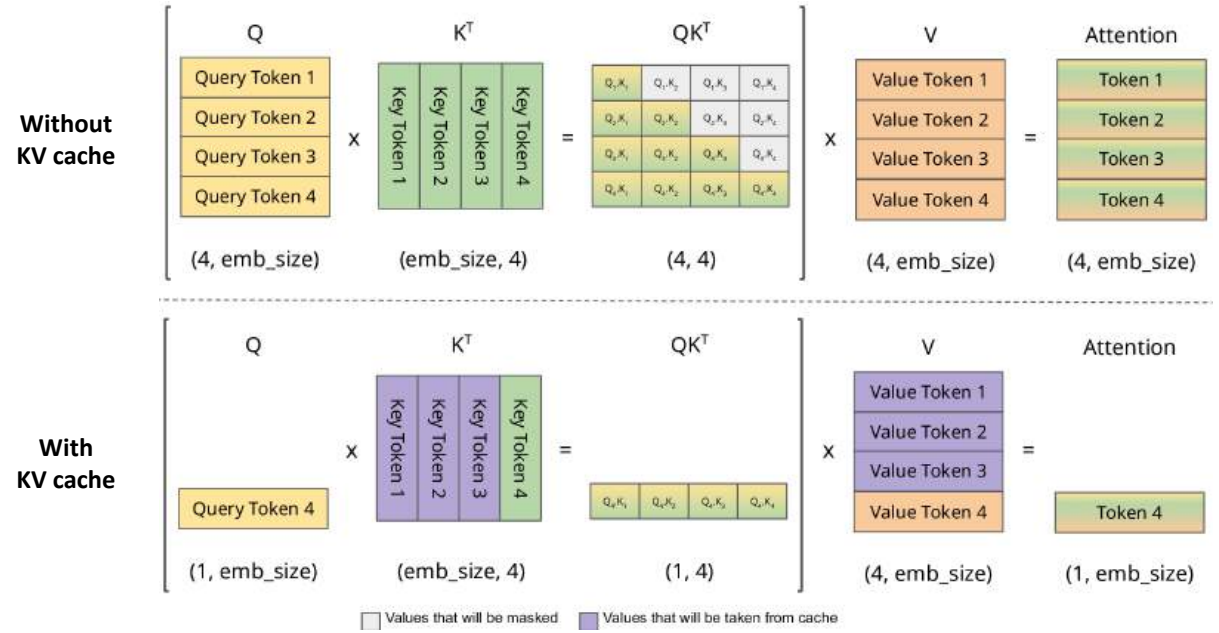
$$4Ld^2 + L^2d + 8Ld^2 = 12Ld^2 + L^2d$$

Per-layer compute cost of generating the $(L + 1)st$ token.

$$4d^2 + Ld + 8d^2 = 12d^2 + Ld$$

Importance of KV-cache

- Essential for reducing compute complexity of generating successive token → linear from quadratic.
- However, inference with KV-cache consumes more memory.
  - Memory for KV-cache for LLaMa2 7B:
    - 2K input → 1GB
    - 8K input → 8GB
    - Batch of 4x8K sequences → 32GB. ← larger than the model's parameters (14GB) !
  - Can reduce device utilization



For n[th] query token, KV- cache [pope eta al, 202] stores and reuses these past key-value pairs, eliminating the need of recalculation for every new token.
Image course: [João Lages, https://medium.com/@joaolages/kv-caching-explained]

Memory requirement

- Model parameters: #parameters x #precision (bytes)
- KV-cache per token: 2 hidden_dim (D) x #layers (N) x #precision (bytes)
- Others: intermediate tensors, CUDA reserved memory, fragmentation
- Example with LLaMa2-7B on NVIDIA RTX-4090 24GB
  - Parameters: 7B x 2 (float16) = 14 GBs
  - KV-cache per token: 2 x 4096 x 32 x 2 (float16) = 0.5 MBs/token
  - For 1GB of cache memory, can host Llama2-7B on one RTX-4090 with at most 1k tokens
  - 10k tokens (for remaining 10GB) can only serve 3 requests for 4k seq

References

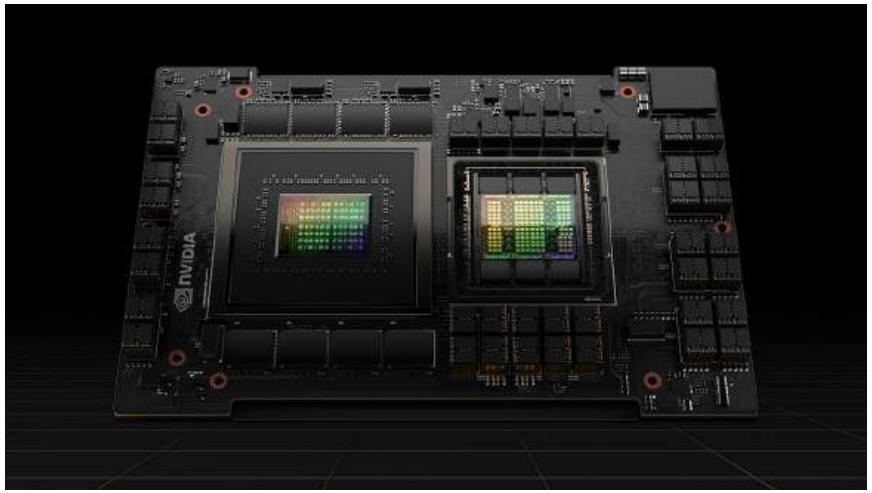
- Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL]
- Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. Advances in neural information processing systems 33 (2020), 1877–1901.
- Raffel, Colin, et al. "Exploring the limits of transfer learning with a unified text-to-text transformer." *Journal of machine learning research* 21.140 (2020): 1-67.
- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. Advances in neural information processing systems 30 (2017).

Primer on deep learning hardware accelerators

Hardware architecture, tensor core, low precision numerics, memory, network

Inference hardware

Deep learning accelerators are used for inference of today’s LLMs.



Inference hardware

Trainium 2 – 4x speed, 3x memory,
4x memory bandwith,
8x network bandwith

Technical Specifications	
	H100 SXM
FP64	34 teraFLOPS
FP64 Tensor Core	67 teraFLOPS
FP32	67 teraFLOPS
TF32 Tensor Core	989 teraFLOPS[2]
BFLOAT16 Tensor Core	1,979 teraFLOPS[2]
FP16 Tensor Core	1,979 teraFLOPS[2]
FP8 Tensor Core	3,958 teraFLOPS[2]
INT8 Tensor Core	3,958 TOPS[2]
GPU memory	80GB
GPU memory bandwidth	3.35TB/s
Decoders	7 NVDEC 7 JPEG
Max thermal design power (TDP)	Up to 700W (configurable)
Multi-instance GPUs	Up to 7 MIGs @ 10GB each
Form factor	SXM
Interconnect	NVLink: › 900GB/s PCIe › Gen5: 128GB/s

Each Trainium device consists of:

Compute	Two NeuronCore-v2 delivering 380 INT8 TOPS, 190 FP16/BF16/cFP8/TF32 TFLOPS, and 47.5 FP32 TFLOP.
Device Memory	32 GiB of device memory (for storing model state), with 820 GiB/sec of bandwidth.
Data Movement	1 TB/sec of DMA bandwidth, with inline memory compression/decompression.
NeuronLink	NeuronLink-v2 for device-to-device interconnect enables efficient scale-out training, as well as memory pooling between the different Trainium devices.
Programmability	Trainium supports dynamic shapes and control flow, via ISA extensions of NeuronCore-v2. In addition, Trainium also allows for user-programmable rounding mode (Round Nearest Even Stochastic Rounding), and custom operators via the deeply embedded GPSIMD engines.

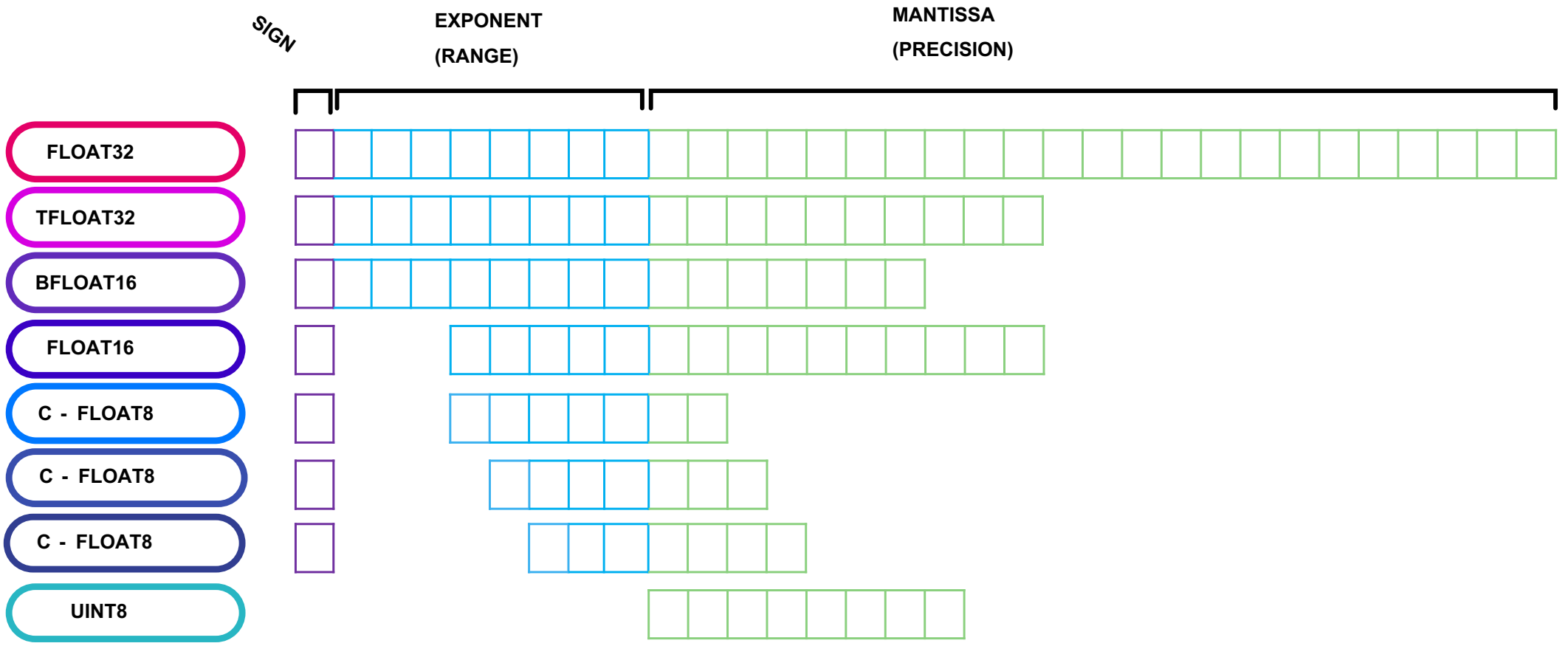
Cerebras Wafer-Scale Engine (WSE-2)

The Largest Chip in the World

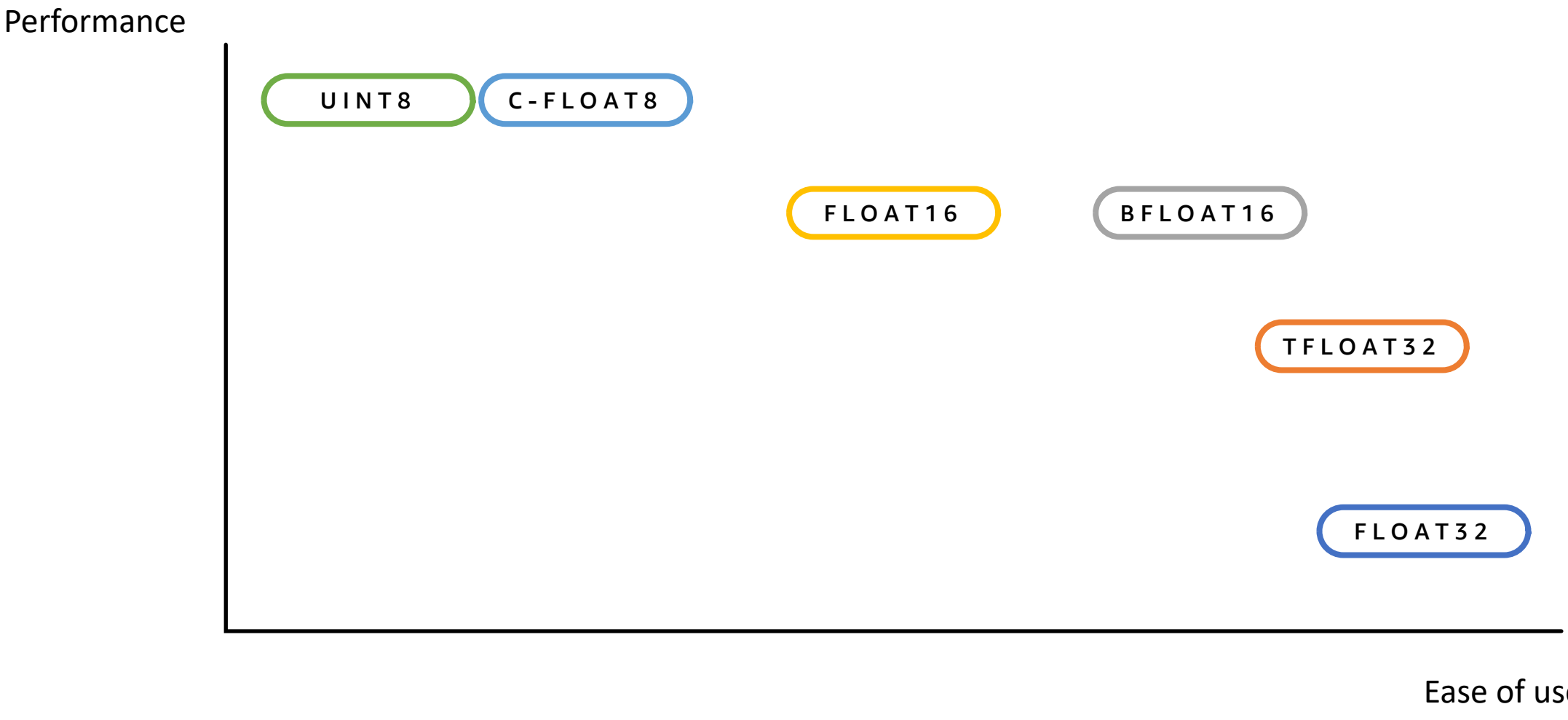
850,000 cores optimized for sparse linear algebra
46,225 mm² silicon
2.6 trillion transistors
40 gigabytes of on-chip memory
20 PByte/s memory bandwidth
220 Pbit/s fabric bandwidth
7nm process technology

Key elements:
- Matrix-matrix multiply units (tensor cores)
- Support for low precision numerics
- High memory bandwidth
- Large per-device fast/fastish memory
- High intra-/inter-connection bandwidth

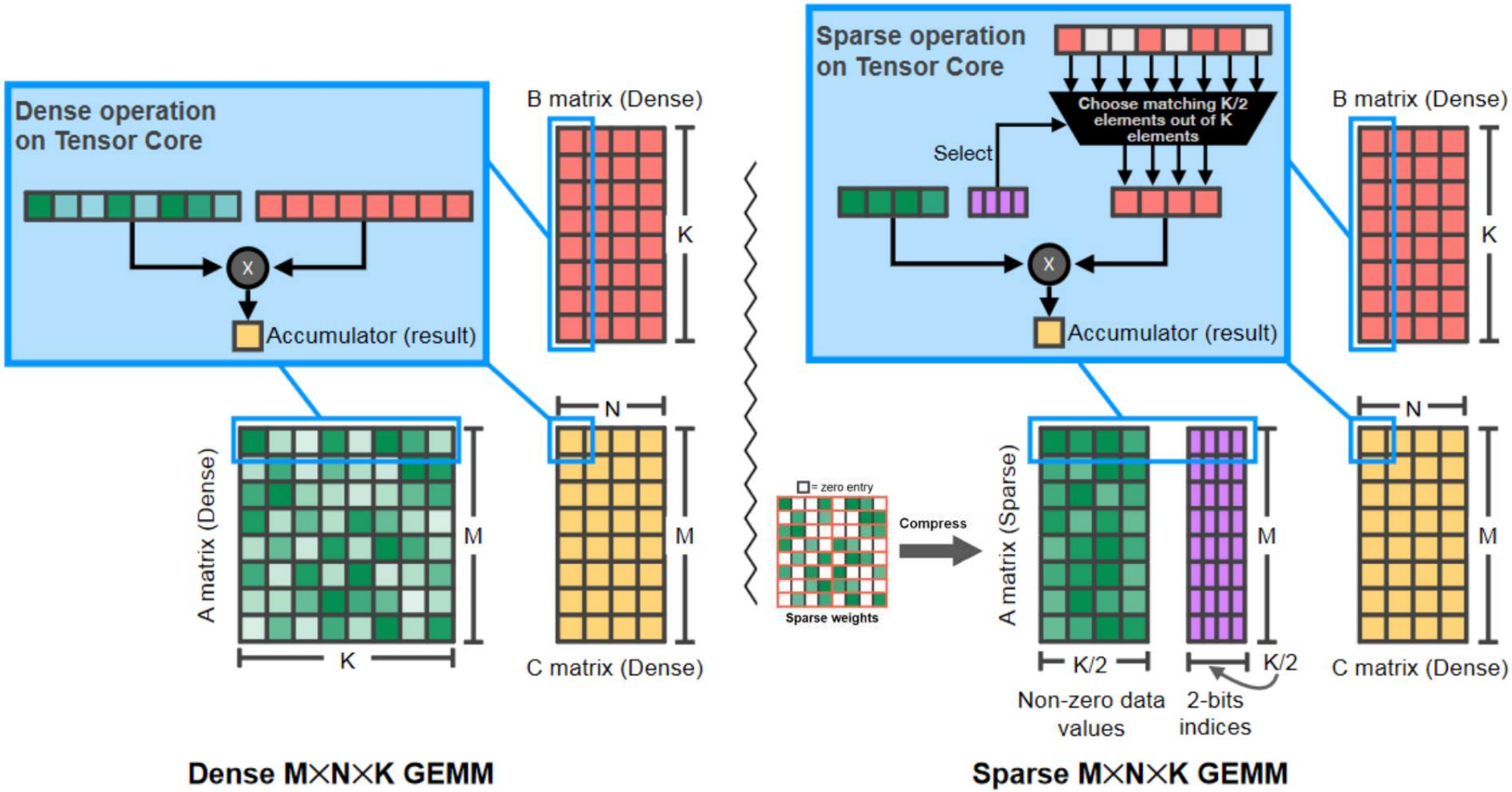
Rich data type selection



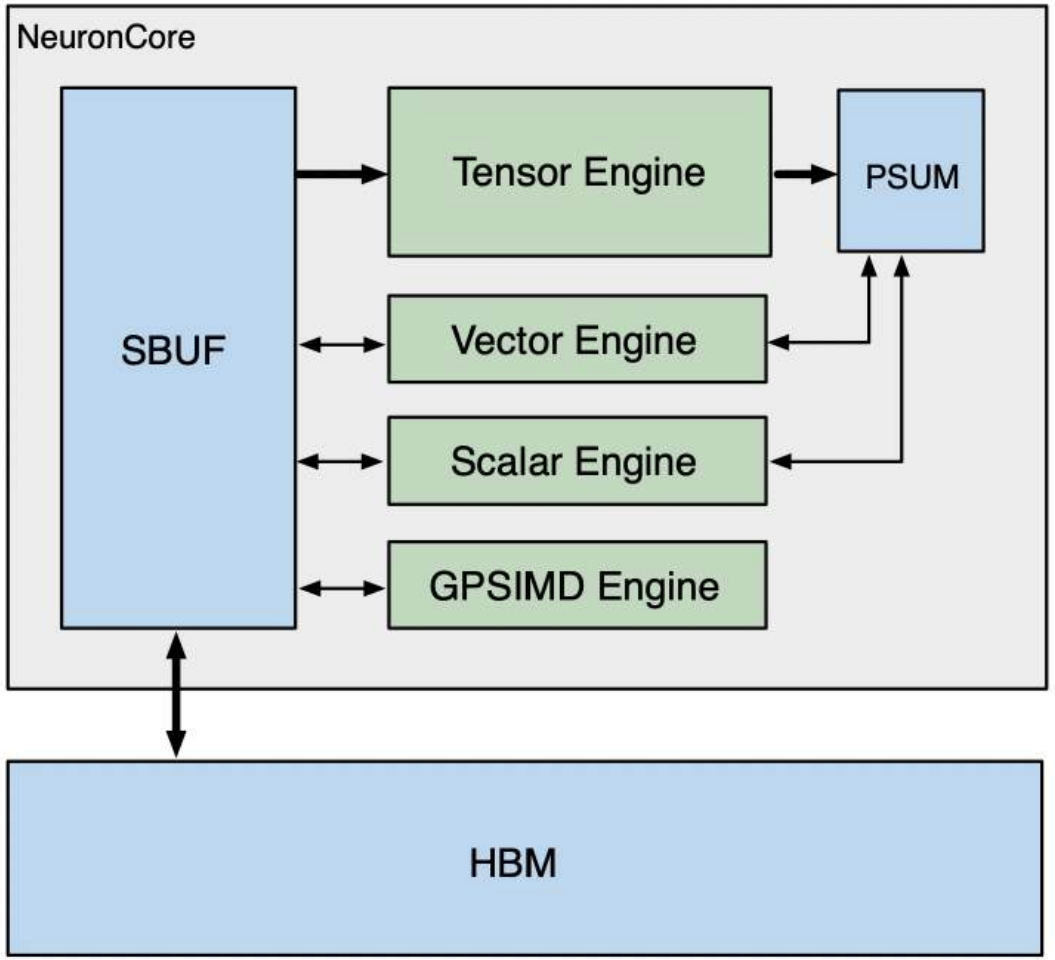
Performance benefits of low-precision types



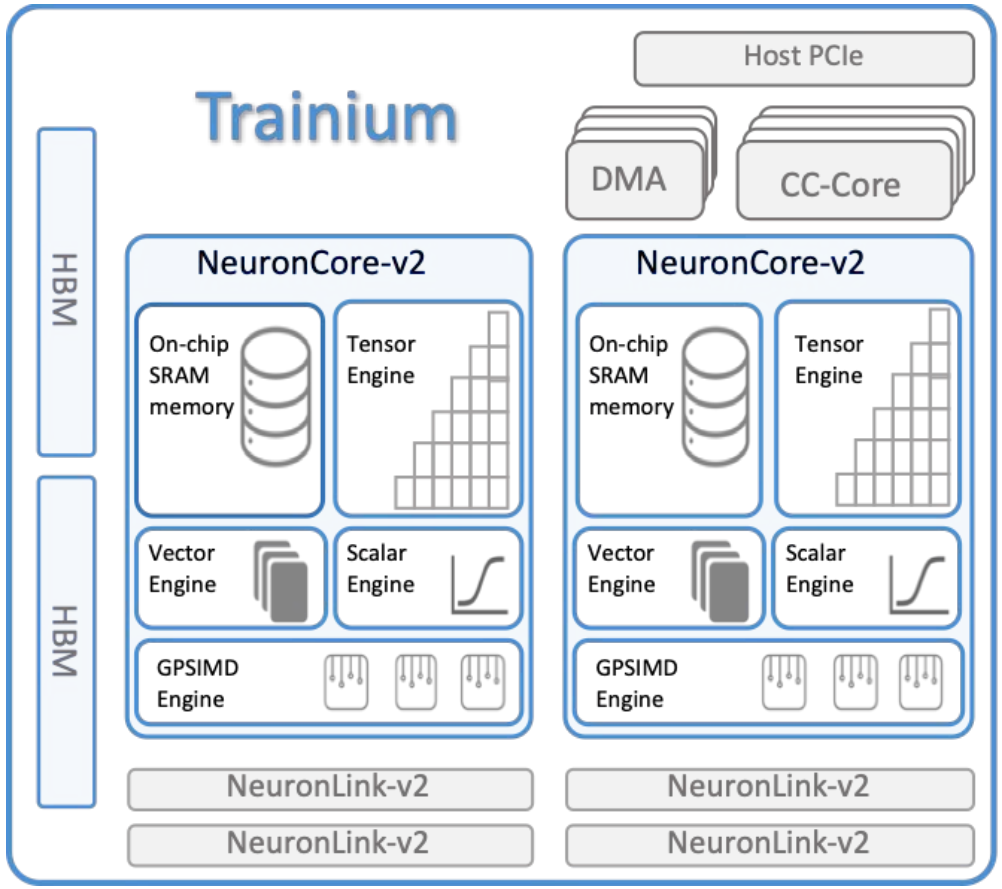
Tensor cores for dense/sparse matrix multiplication

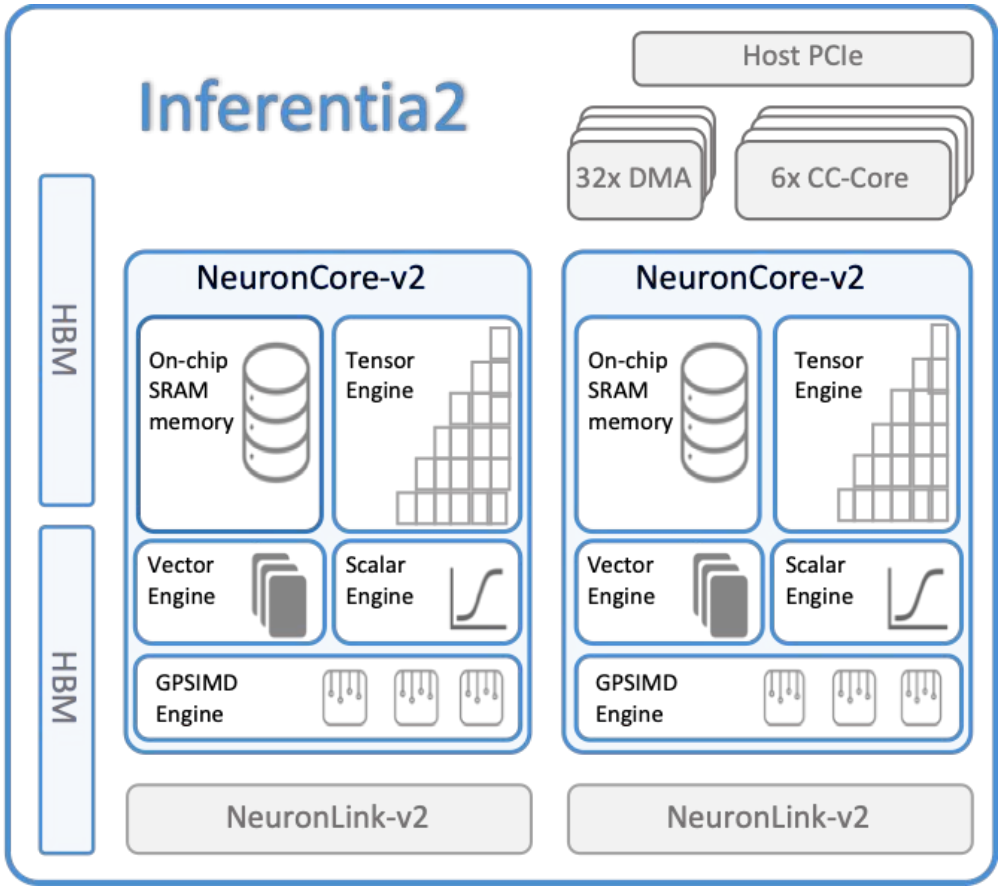


NeuronCore-v2 and its Local HBM



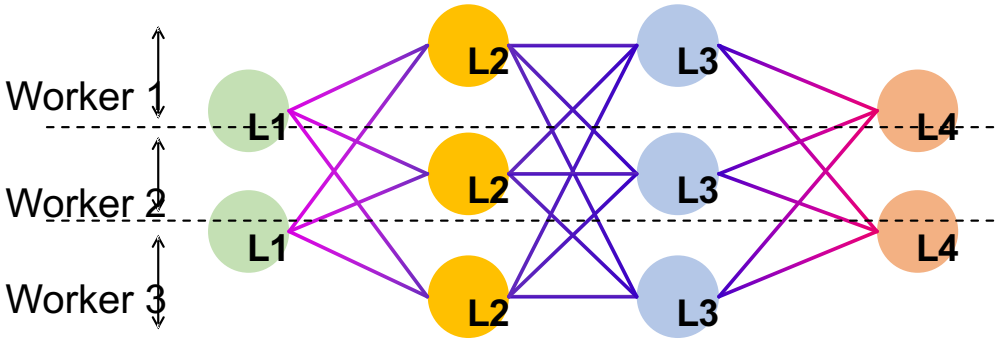
Trainium/Inferentia2 device diagrams



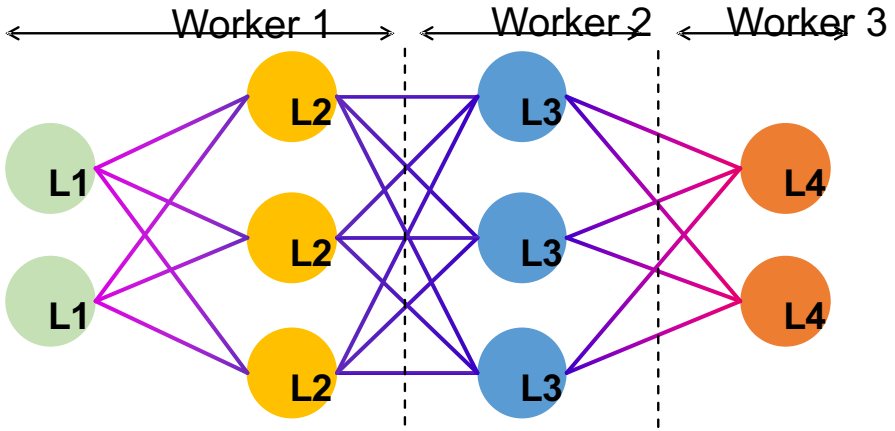


Distributed computation over multiple cores

Tensor Parallelism



Pipeline Parallelism



- Effective NeuronLink and PCI is supported and can be leveraged for distributed computation.

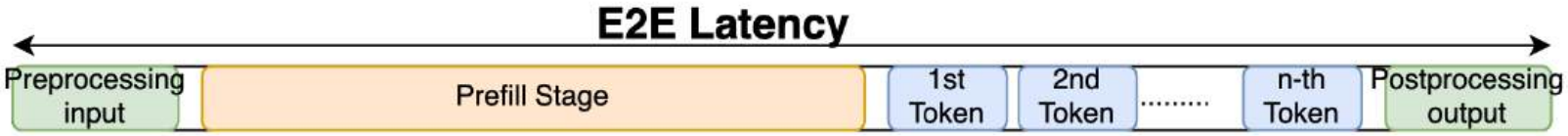
The many ways to optimize inference

Inference metrics, optimization space

Performance metrics

- **Prefill latency**
  - time-to-first-token latency (TTFT)
  - Critical waiting time under real-time LLM response to users (e.g., chatbot)
  - Affecting factors: model size, large input length, hardware
- **Decoding throughput (tokens / second)**
  - Inter-token latency (ITL) after the first token
  - Important under real time interaction (e.g. chatbot)
    - Faster is not always desirable. 6 English words/sec may be sufficient for chatbot
  - Affecting factors: batchsize, hardware

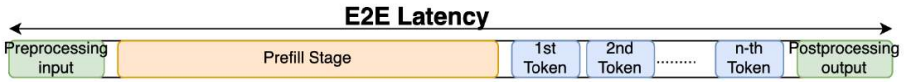
Performance metrics



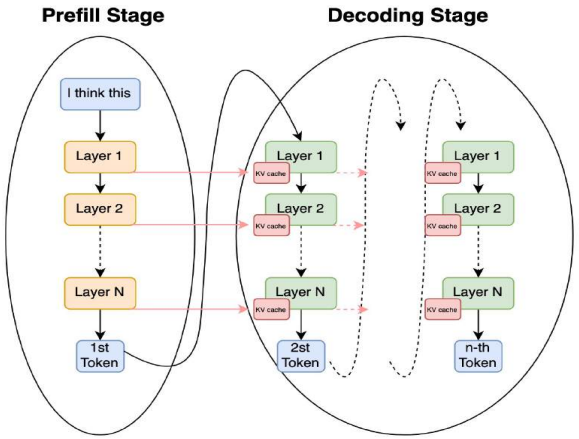
- **End-to-end latency**
  - Latency to complete inference process
  - crucial for assessing the overall user experience
    - often dependent on other components, e.g., processing JSON
  - Affecting factors: network latency, total tokens to generate, overall system, etc
- **Maximum request rate, a.k.a. QPS, (queries/second)**
  - measures how many concurrent queries can be handled per second
  - Important for serving models in production environments under multiple users
    - Example: Assume 20 QPS can be served (for P90 latency) via single GPU, then 300/20=15 GPUs is required to support QPS 300
  - Affecting factors: hardware resources, load balancing, etc.

The space of inference optimizations

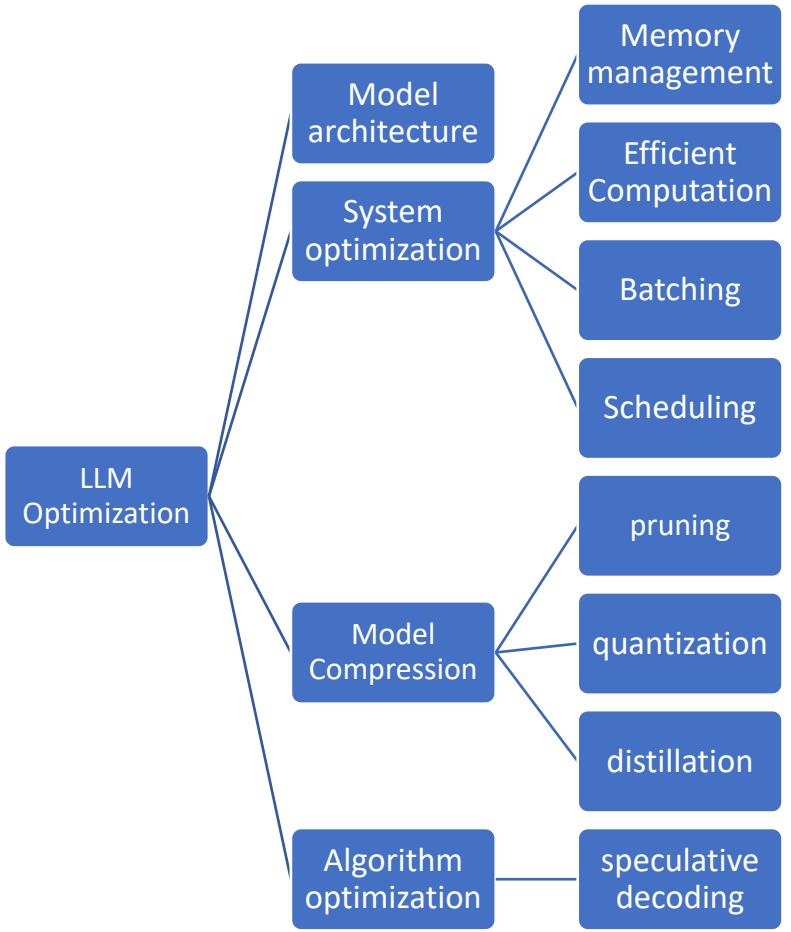
Metric:



Inference Stages:



- With these metrics, optimize prefill, decoding and other stages of LLM via
  - Model architecture choice
  - System optimizations
  - Model compressions
  - Algorithmic optimization



Primer on model architecture choice

Group Query Attention, Mixture of Experts

Architecture Choice for Fast Inference

- As scaling up model parameters to 70+B, encounter memory bound and compute bound.
  - For half precision Llama2 70B, multiple GPUs just to load 140G weights.
  - Quadratic computational $O(L^2d + Ld^2)$ during prefill
  - KV cache memory can consume large memory for long seq or large batch

Architecture Choice for Fast Inference

- As scaling up model parameters to 70+B, encounter memory bound and compute bound.
  - For half precision Llama2 70B, multiple GPUs just to load 140G weights.
  - Quadratic computational $O(L^2d + Ld^2)$ during prefill
  - KV cache memory can consume large memory for large input token

- Motivate to have more efficient Transformation architectures
  - Can we reduce K, V, Q computations and memory? -> Group Query Attention
  - Can we reduce projection computations? -> Mixture of Experts

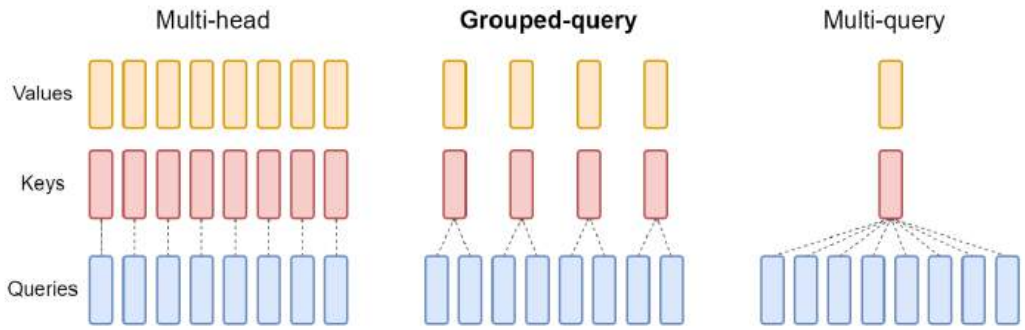
Architecture Choice for Fast Inference

- As scaling up model parameters to 70+B, encounter memory bound and compute bound.
  - For half precision Llama2 70B, multiple GPUs just to load 140G weights.
  - Quadratic computational $O(L^2d + Ld^2)$ during prefill
  - KV cache memory can consume large memory for long seq or large batch

- Motivate to have more efficient Transformation architectures
  - Can we reduce K, V, Q computations and memory? -> Group Query Attention
  - Can we reduce projection computations? -> Mixture of Experts

Group Query Attention

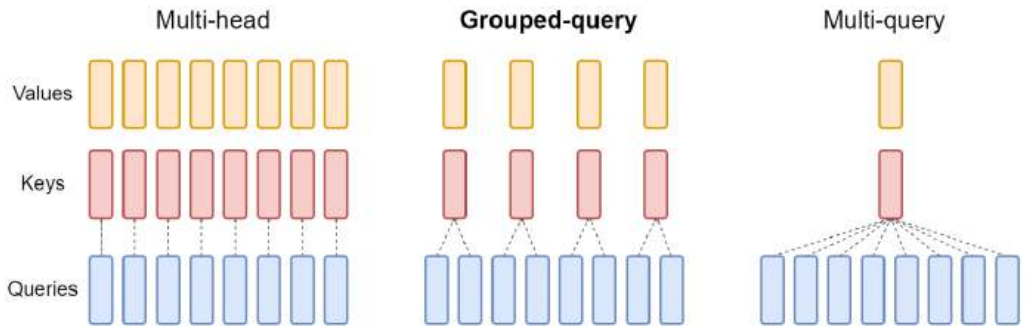
- Multi-headed attention (MHA): distinct query (Q), key (K), and value (V) for each head
  - K and V heads increase linearly to Q heads
- Multi-query attention (MQA): single K and V head shared across all query heads
  - Leverages repeated K and V, reduces inference latency
  - but sacrifices prediction quality
- Grouped-query attention (GQA): single K and V head shared within each query group
  - Balances between MHA and MQA, reducing memory/latency but preserving accuracy



[Ainselie et al. 2023] Overview of MHA (left), MQA (middle), and GQA (right)

Group Query Attention

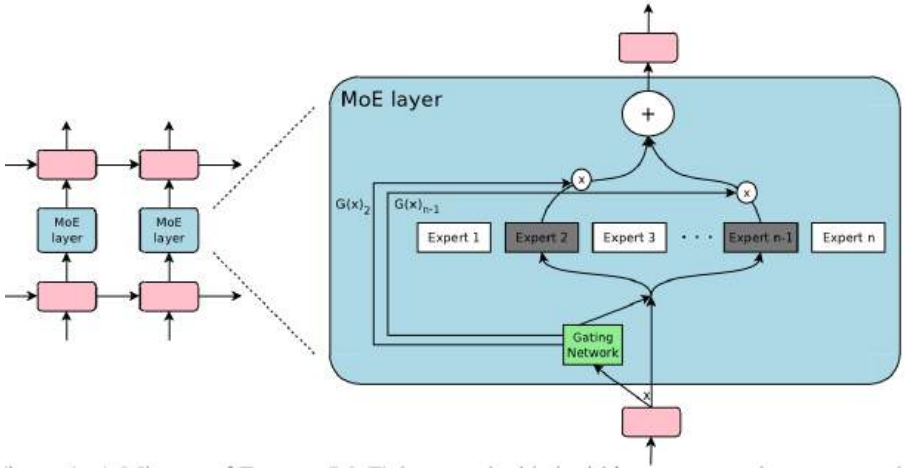
- Under distributed setting, evenly distribute KV heads to devices
  - KV heads/caches can be reused for multiple query heads (within each devices)
  - Llama2 70B has 64 Q heads, which are grouped with 8 KV heads,
  - i.e., one KV head per core in case of 8xA100



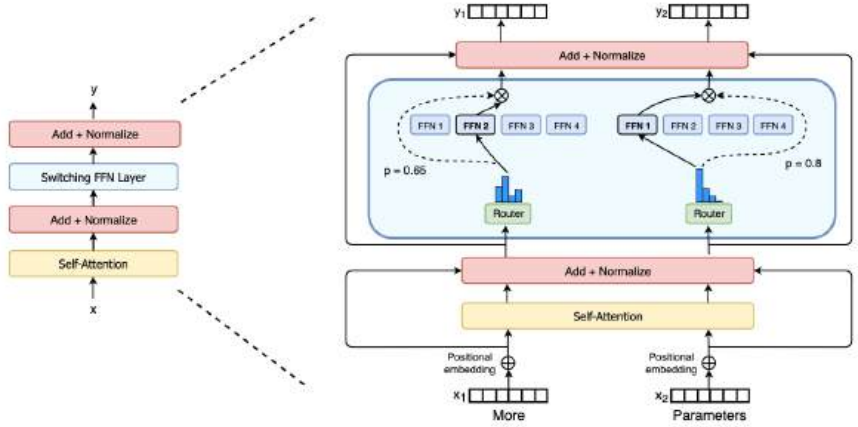
[Ainselie et al. 2023] Overview of MHA (left), MQA (middle), and GQA (right)

- GQA requires less memory of KV cache than MHA due to smaller # KV heads
  - E.g., KV cache of LLaMa2 70B (with GQA) is smaller than one of LLaMa2 7B.

Mixture of Experts for Transformer



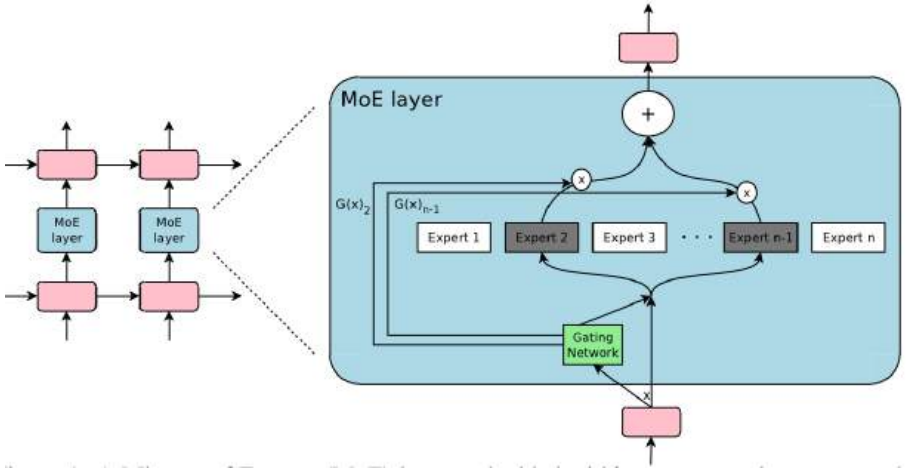
[Shazeer et al . 2017] MoE to LSTM



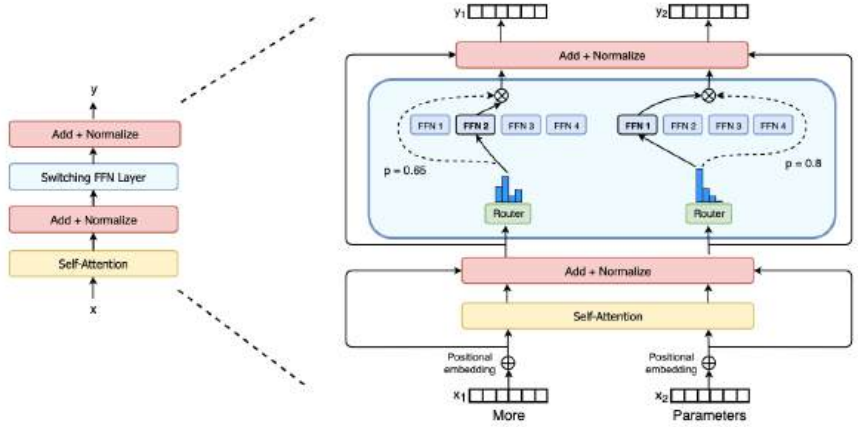
[Fedus et al . 2022] MoE (a sparse Switch FFN layer) to Transformer layer

- Mixture of Experts (MoE) is one way to scale up without increasing inference latency
  - Model size of hundreds of billion parameters
  - E.g., Mixtral 8 x 7B [Jiang et al, 2023] uses 8 experts
- Replaces dense Feed Forward Network (FFN) layer with sparse MoE FFN.
  - Few e experts are selected and activated to a given input (token or segment)
  - Effective (activated) number of parameters is the same as experts increase

Mixture of Experts for Transformer



[Shazeer et al . 2017] MoE to LSTM



[Fedus et al . 2022] MoE (a sparse Switch FFN layer) to Transformer layer

- Routing mechanism (or gate network)
  - Assigns few experts to a given input based attention weights, expert domain, etc
  - Critical in increasing inference throughput via load balancing over experts and expert parallelism over devices
- Expert Parallelism (EP) is effective under distributed setting
  - keeping each expert within a small group of devices
  - alleviating collective communications and dynamic loading, leading to fast inference
  - E.g., Mitral 8 x 7B uses 8 experts with choosing 2 of experts group for each token

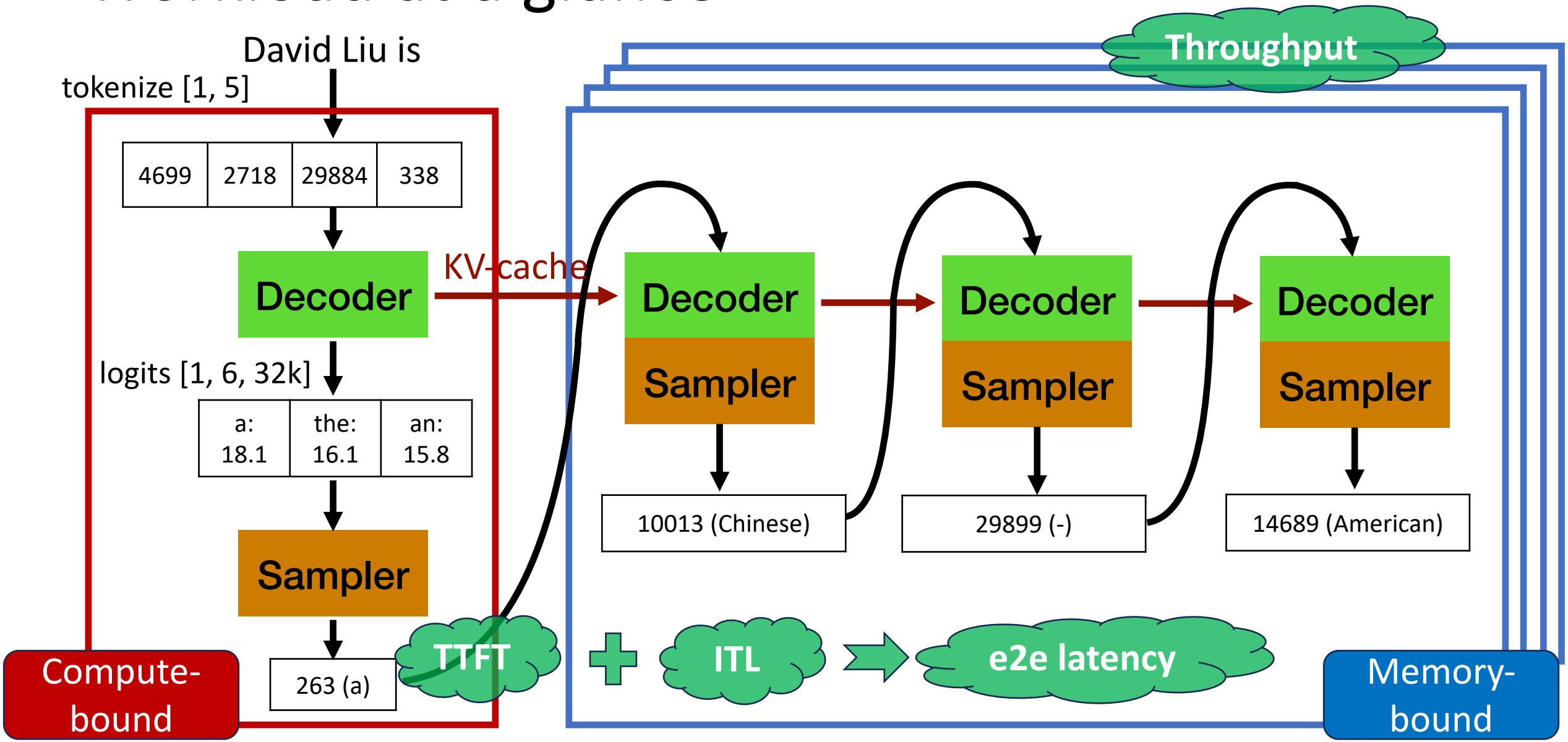
References

- Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. Gqa: Training generalized multi-query trans- former models from multi-head checkpoints. arXiv preprint arXiv:2305.13245 (2023).
- Hugo Touvron et al., 2023. Llama 2:Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288
- William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. Journal of Machine Learning Research 23, 120 (2022), 1–39.
- Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, De- vendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. arXiv:2310.06825 [cs.CL]
- Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. arXiv preprint arXiv:1701.06538 (2017).

System-level optimizations

1. Semantic-preserving
2. Adjust for the targeting compute platform

Workload at a glance

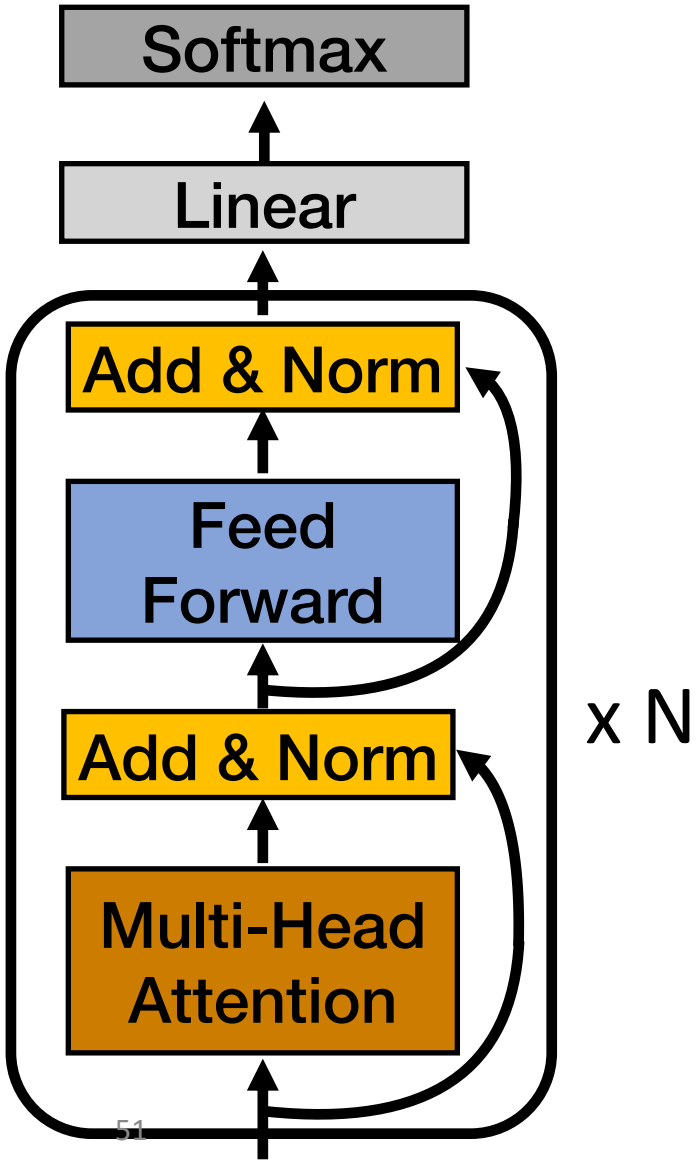


amazon | science

Outline

- Optimize the forward computation
- Make the device compute resource busier
- Leverage the LLM inference characteristics

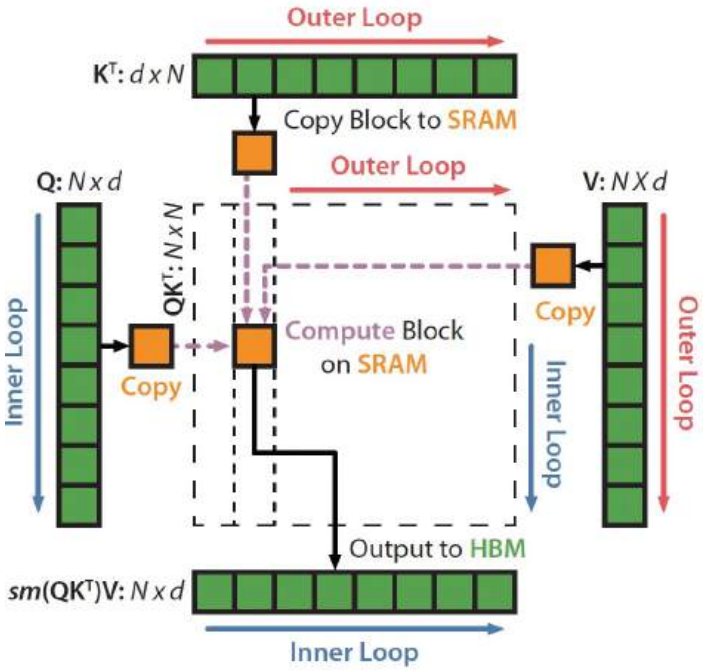
Optimize the forward computation



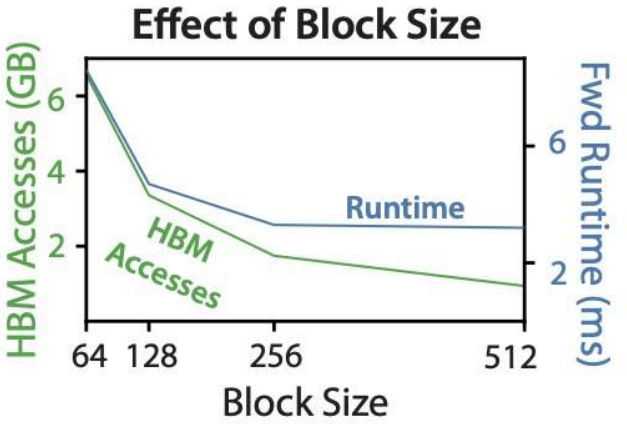
- Goal: Finish one forward computation as fast as possible
  - FlashAttention for prefill
  - FlashDecoding for decoding
  - Distributed inference

FlashAttention

- Standard self-attention: $softmax(\frac{QK^T}{\sqrt{(d_k)}})V$
  - $S = QK^T$, read QK ($\mathbb{R}^{Nd}$), write S ($\mathbb{R}^{N^2}$)
  - $P = softmax(S)$, read S ($\mathbb{R}^{N^2}$), write P ($\mathbb{R}^{N^2}$)
  - $O = PV$, read PV ($\mathbb{R}^{N^2}$, $\mathbb{R}^{Nd}$), write O ($\mathbb{R}^{Nd}$)



- FlashAttention: attention kernel with reduced ***IO*** (i.e., data transfer between HBM and SRAM on accelerators)
  - Decompose (tiling) softmax by scaling
  - Fuse the S, P, O computation tile-by-tile
  - Tile size is dependent to the SRAM size
    - GPU: 40 MB L2 shared, Trainium: 24 MB per core



Dao, Tri, et al. "FlashAttention: Fast and memory-efficient exact attention with IO-Awareness." *Advances in Neural Information Processing Systems* 35 (2022)

FlashAttention (conti.)

- Calculate P_i of softmax ($P_i = \frac{e^{S_i}}{\sum_j e^{S_j}}$) requires a complete matrix of S
- FlashAttention decomposes softmax to remove this constraint, so that P can be computed tile-by-tile

For each block b

$$A = Q_b K_b^T$$

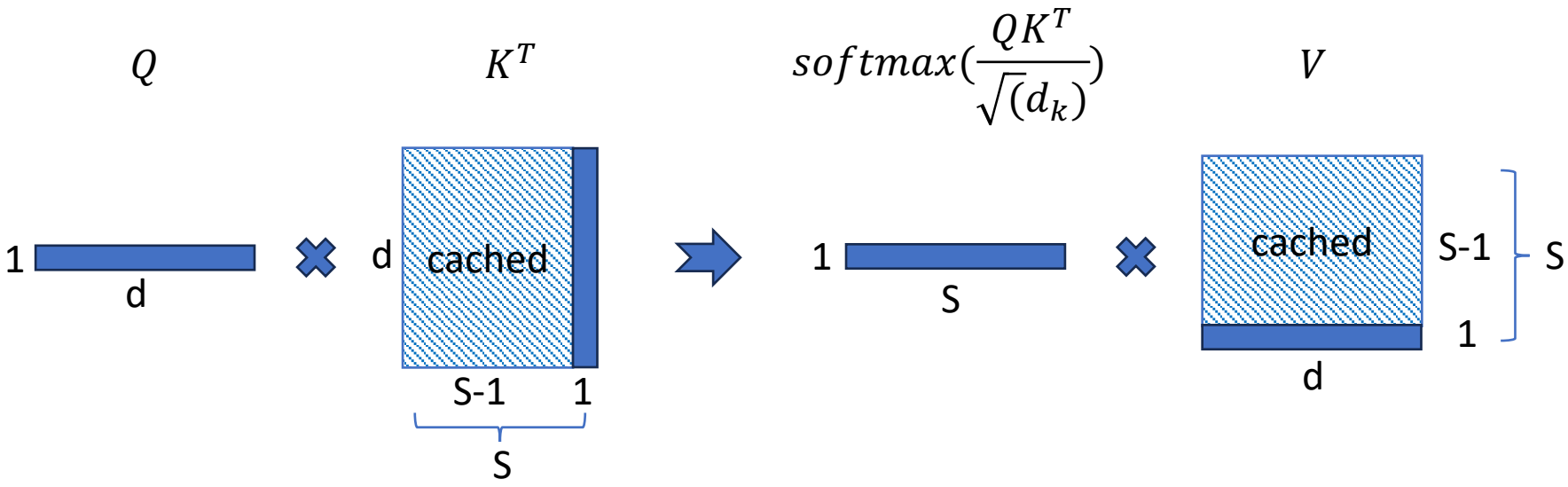
$$O = \frac{S \times O + e^A \times V_b}{S + A}$$

$$S = S + A$$

Dao, Tri, et al. "FlashAttention: Fast and memory-efficient exact attention with IO-Awareness." *Advances in Neural Information Processing Systems* 35 (2022)

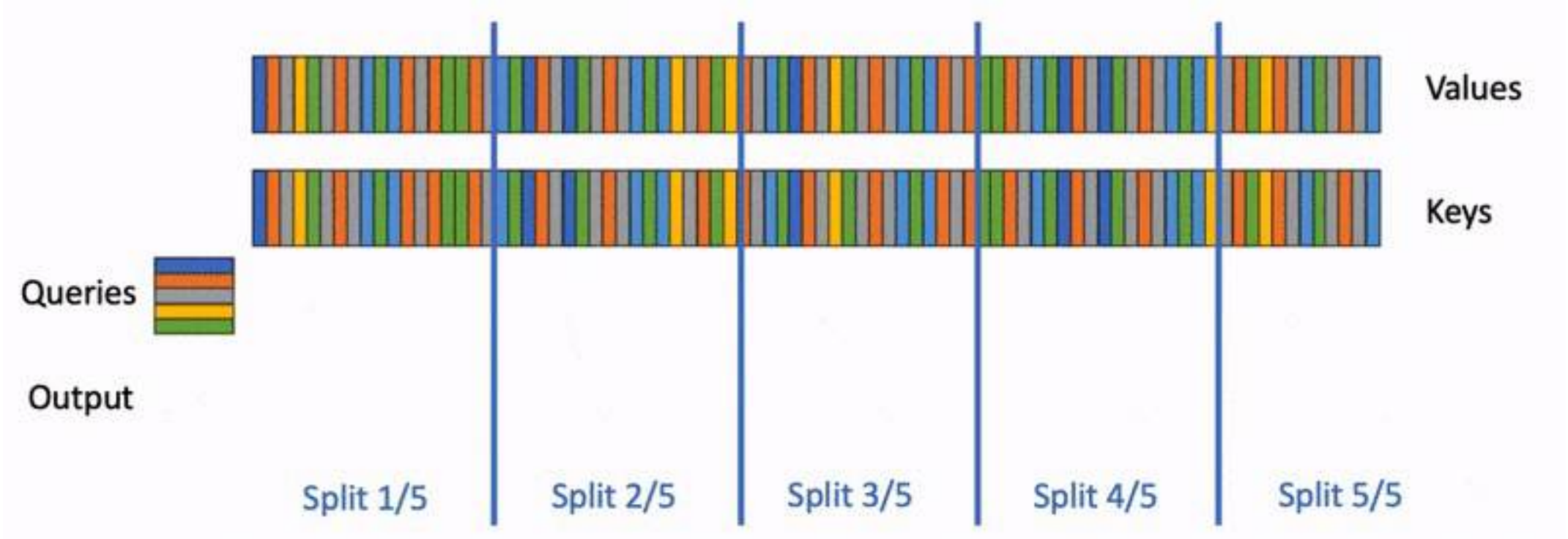
Decoding with KV-cache

- Standard self-attention: $softmax(\frac{QK^T}{\sqrt{(d_k)}})V$
  - Q becomes a vector
  - QK^T and $softmax(\frac{QK^T}{\sqrt{(d_k)}})V$ becomes matrix-vector multiplication (GEMV)



FlashDecoding

- One more parallelization degree: Sequence length

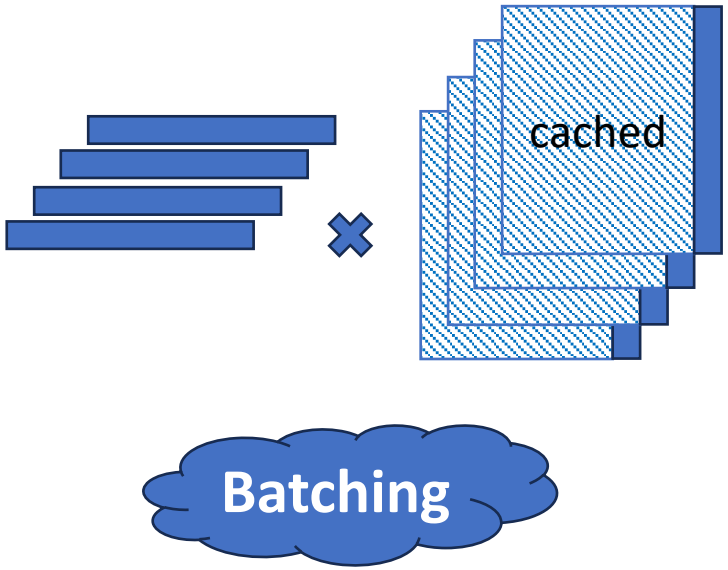


Dao, Tri, et al. https://crfm.stanford.edu/2023/10/12/flashdecoding.html

Distributed inference

- Tensor Parallelism (TP) on top of multi-head
  - Group query attention, #kv heads=8
  - What if #devices per node > 8?
    - Duplication
    - Partition
- Pipeline Parallelism (PP) across multiple nodes
  - Enable inference on very large models
  - Larger latency due to lower bandwidth between nodes
- Prefill-Decode Disaggregation
  - Run Prefill and decode on **separate** machines with **different** parallelization and hardware configurations

Make the device compute resource busier

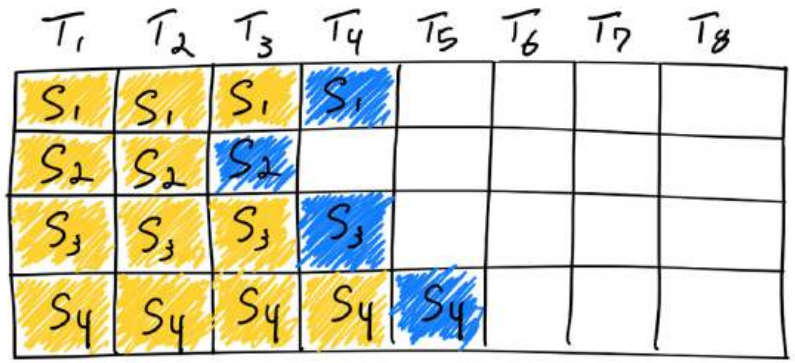


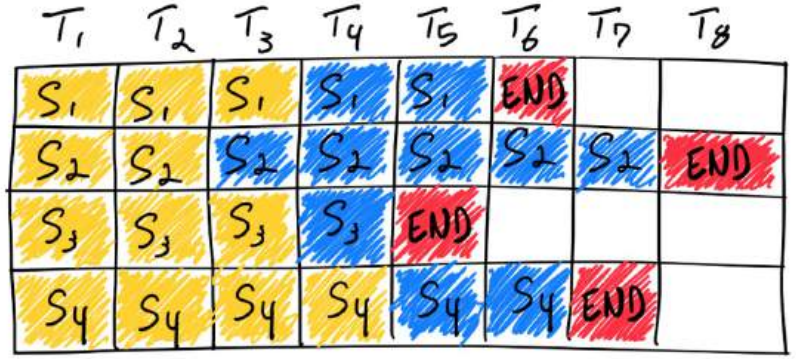
- Goal: Give more computation to occupy the compute resource
  - Continuous batching
  - PagedAttention
  - Chunked prefill

Continuous Batching

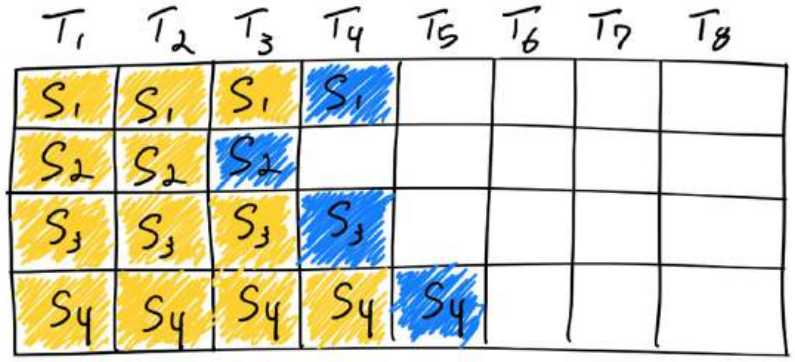
Idea: Do not wait until a request finishes. Let new requests join the current decoding batch immediately.

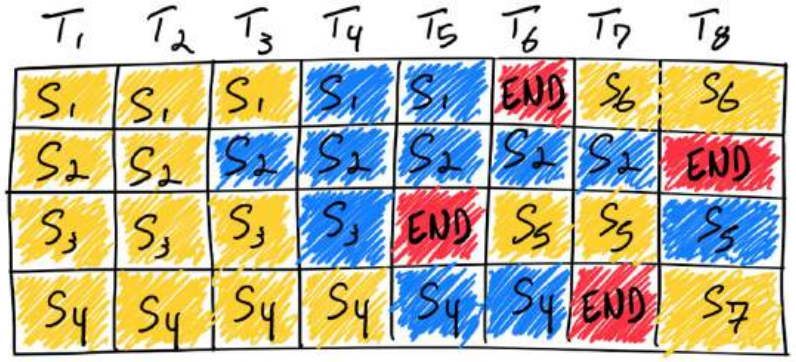
Static batching





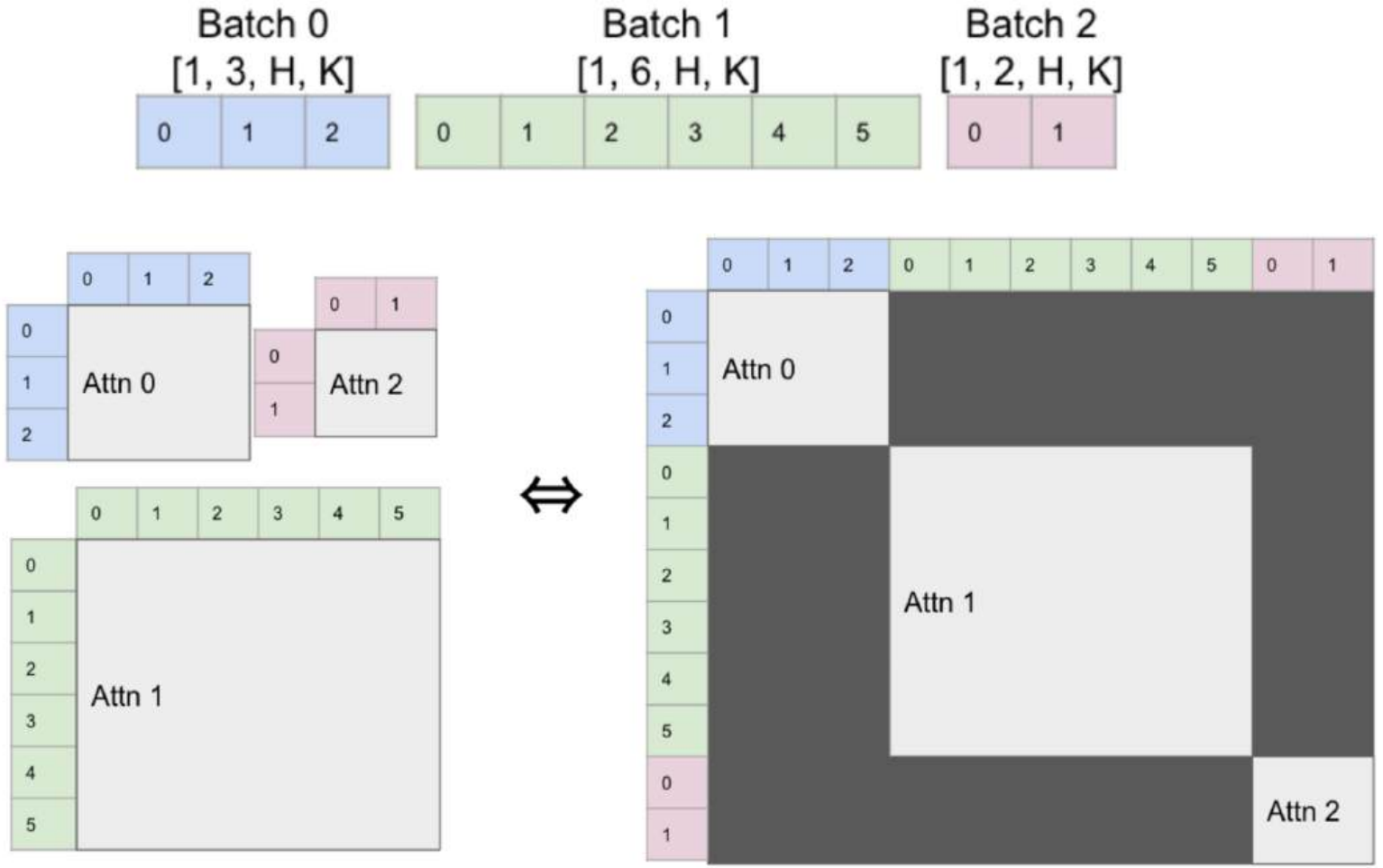
Continuous Batching





Reference: https://www.usenix.org/conference/osdi22/presentation/yu, https://www.anyscale.com/blog/continuous-batching-llm-inference

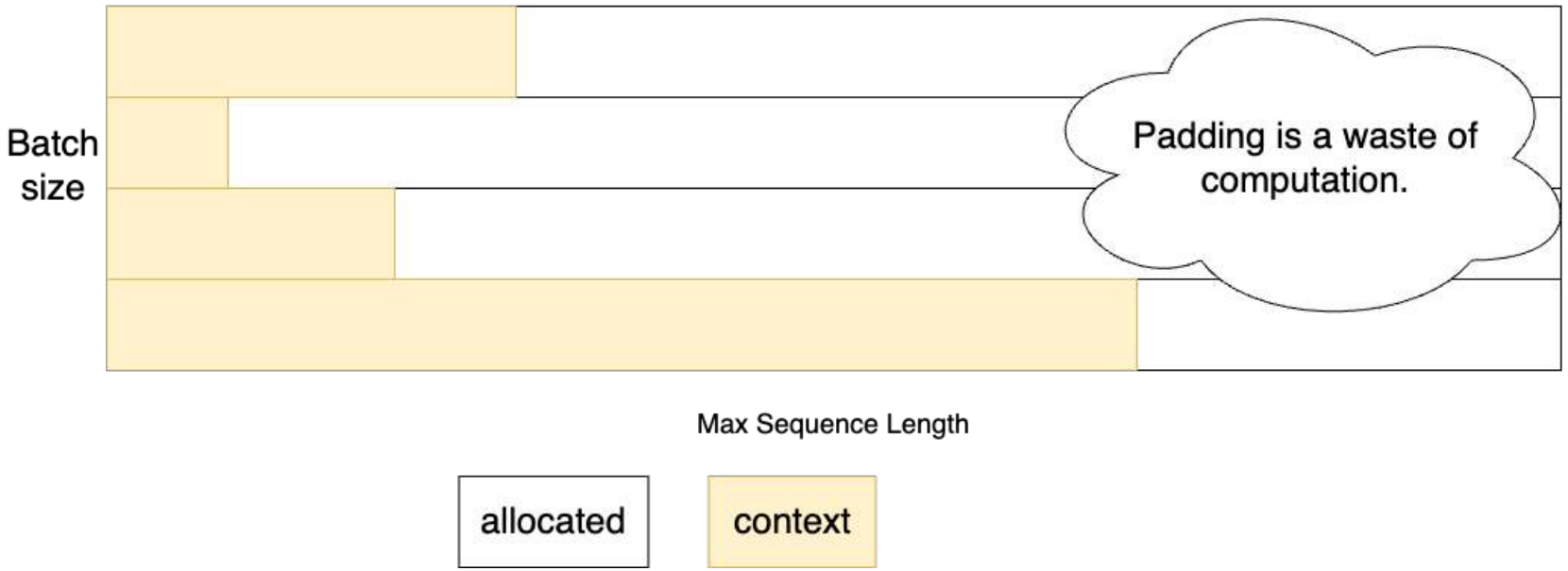
Block diagonal causal attention mask



Source: https://facebookresearch.github.io/xformers/components/ops.html

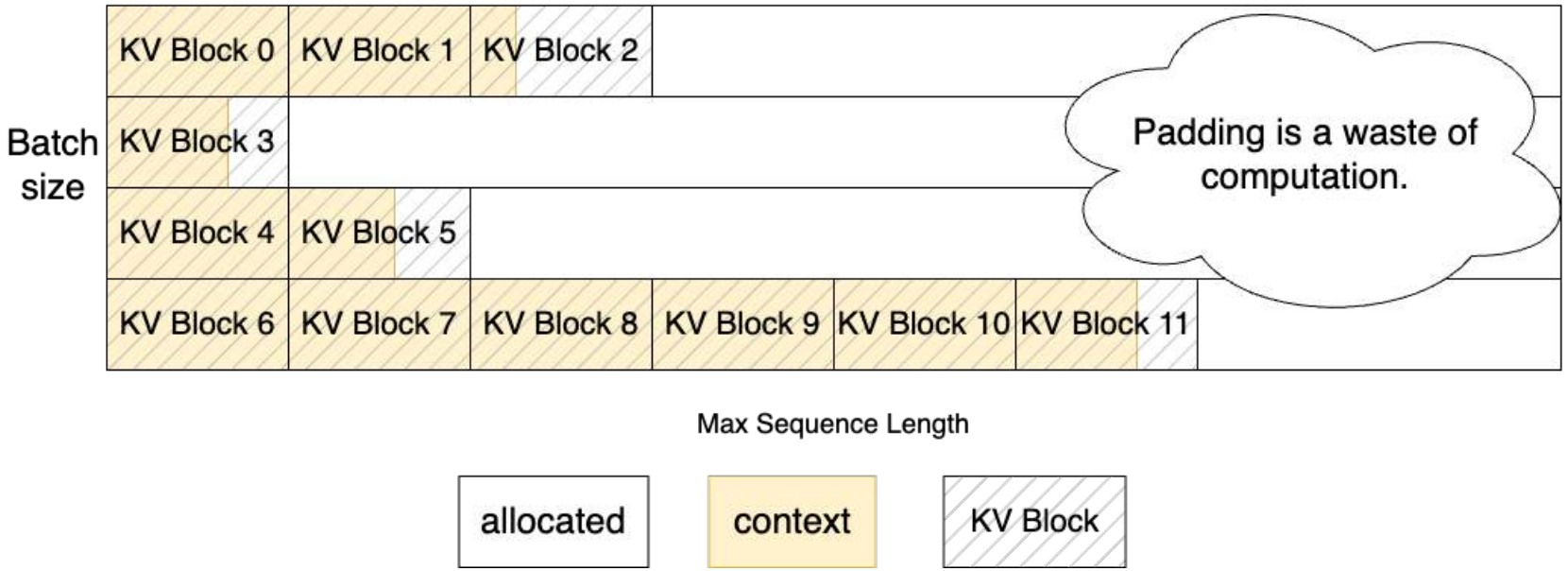
KV-Cache Management

- Pre-allocate contiguous space of memory to the request's maximum length — Serious memory fragmentation



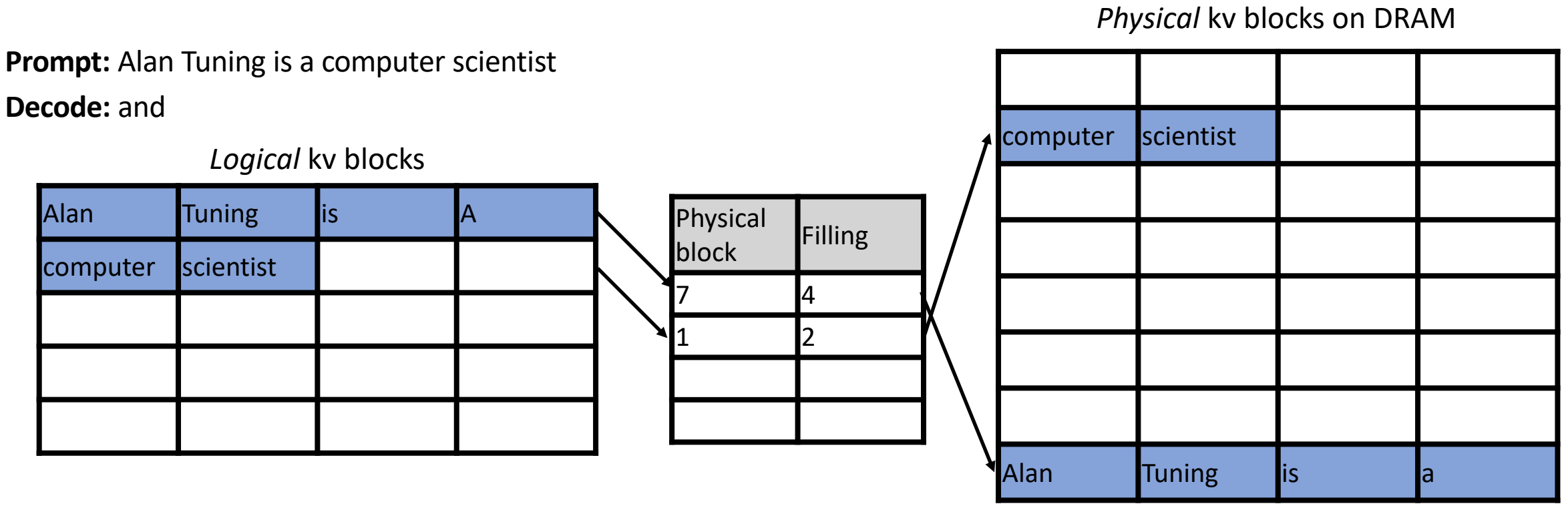
KV-Cache Management

- Pre-allocate contiguous space of memory to the request's maximum length — Serious memory fragmentation



KV-Cache Management with Paged Attention

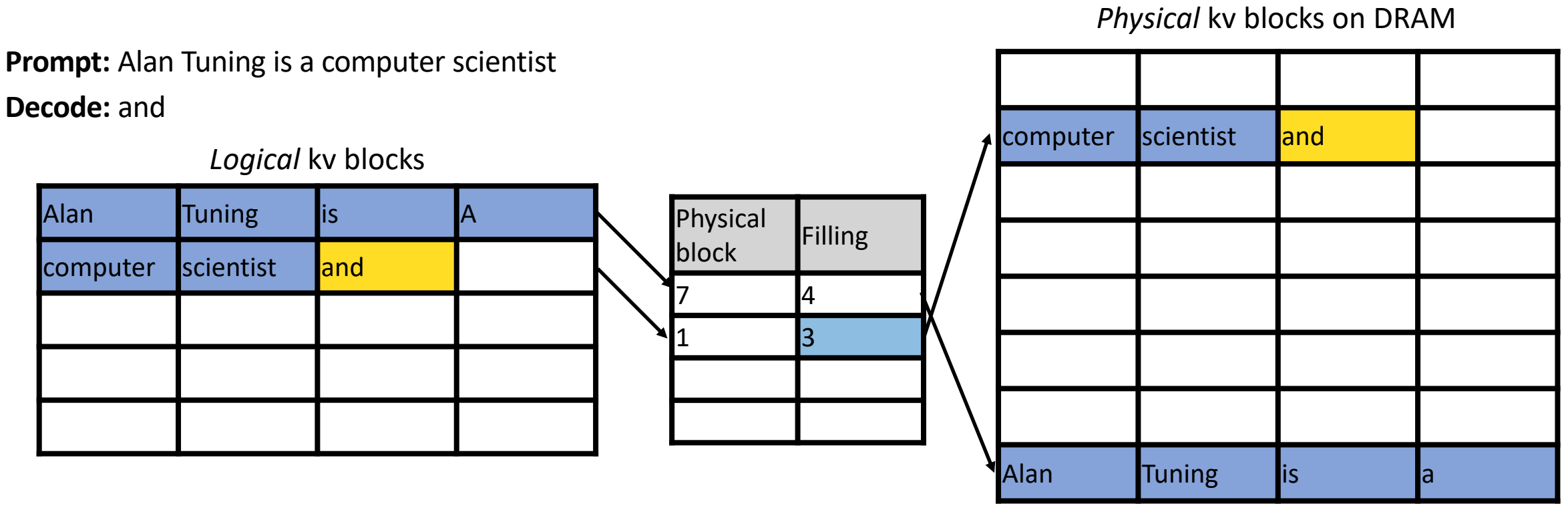
- Application-level memory paging and virtualization for attention KV Cache
- Leverage the concept of virtual memory in operating systems



Kwon, Woosuk, et al. "Efficient memory management for large language model serving with pagedattention." *SOSP* 2023.

KV-Cache Management with Paged Attention

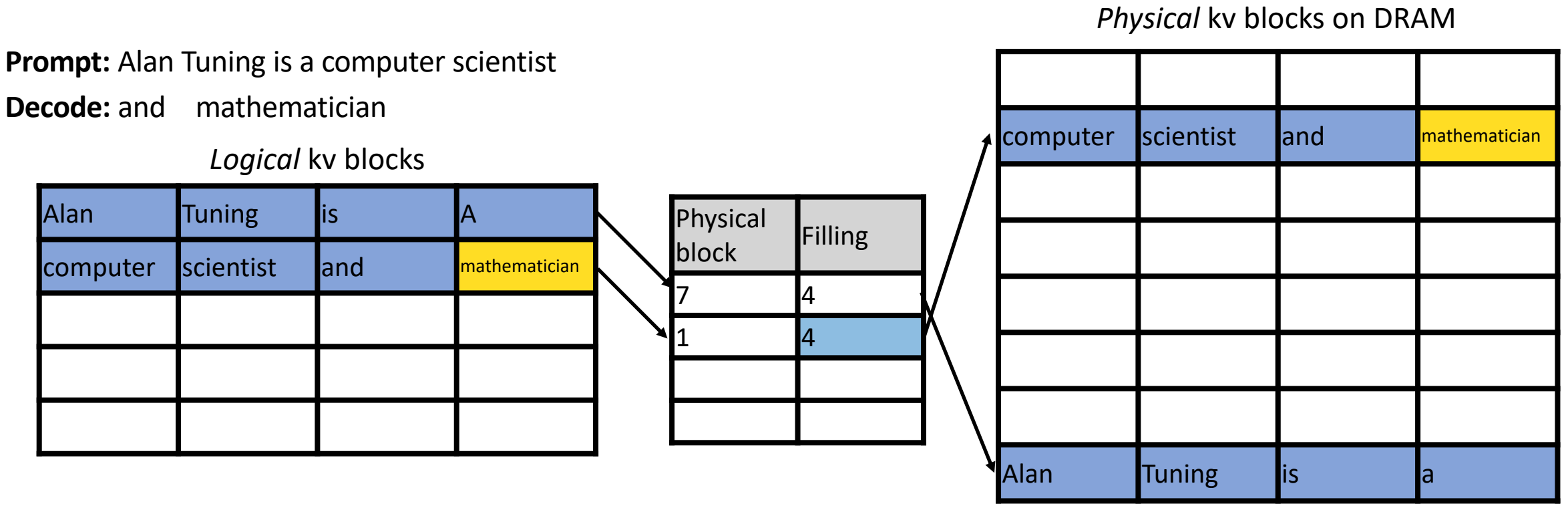
- Application-level memory paging and virtualization for attention KV Cache
- Leverage the concept of virtual memory in operating systems



Kwon, Woosuk, et al. "Efficient memory management for large language model serving with pagedattention." *SOSP* 2023.

KV-Cache Management with Paged Attention

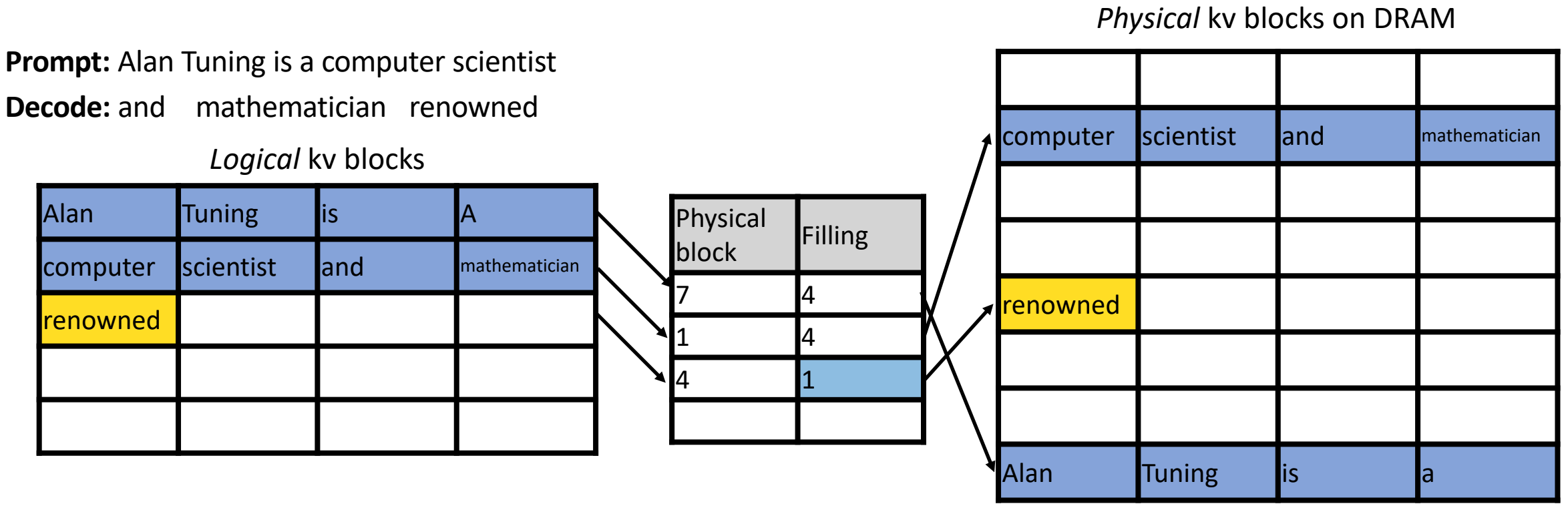
- Application-level memory paging and virtualization for attention KV Cache
- Leverage the concept of virtual memory in operating systems



Kwon, Woosuk, et al. "Efficient memory management for large language model serving with pagedattention." *SOSP* 2023.

KV-Cache Management with Paged Attention

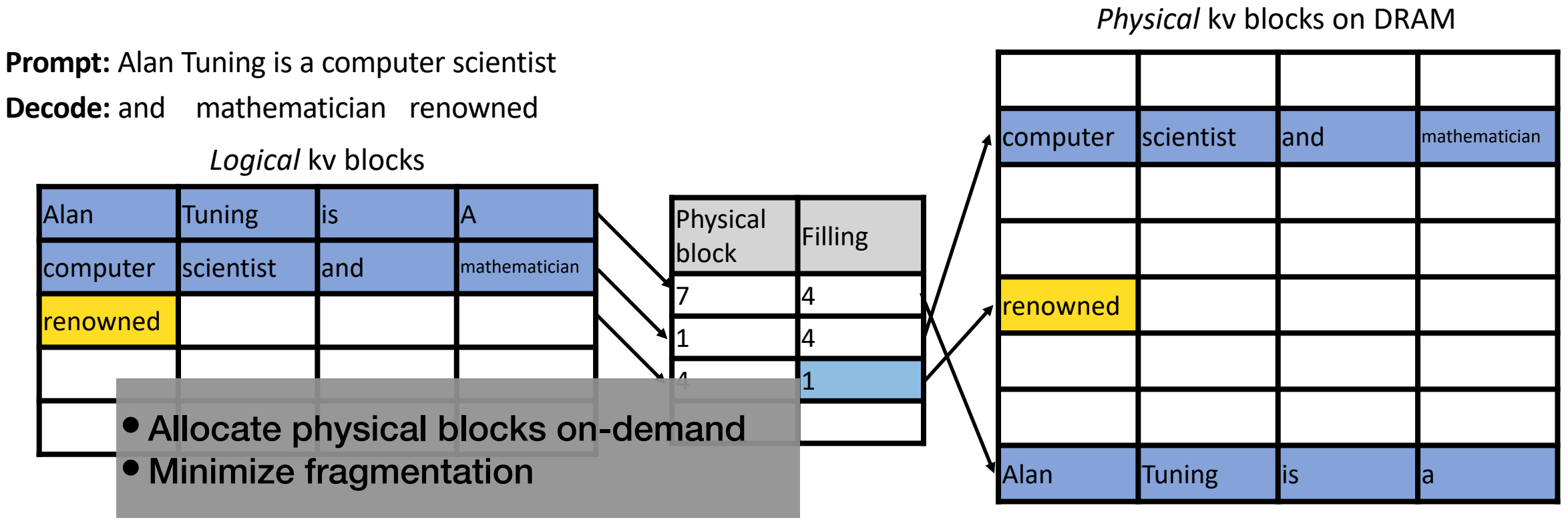
- Application-level memory paging and virtualization for attention KV Cache
- Leverage the concept of virtual memory in operating systems



Kwon, Woosuk, et al. "Efficient memory management for large language model serving with pagedattention." *SOSP* 2023.

KV-Cache Management with Paged Attention

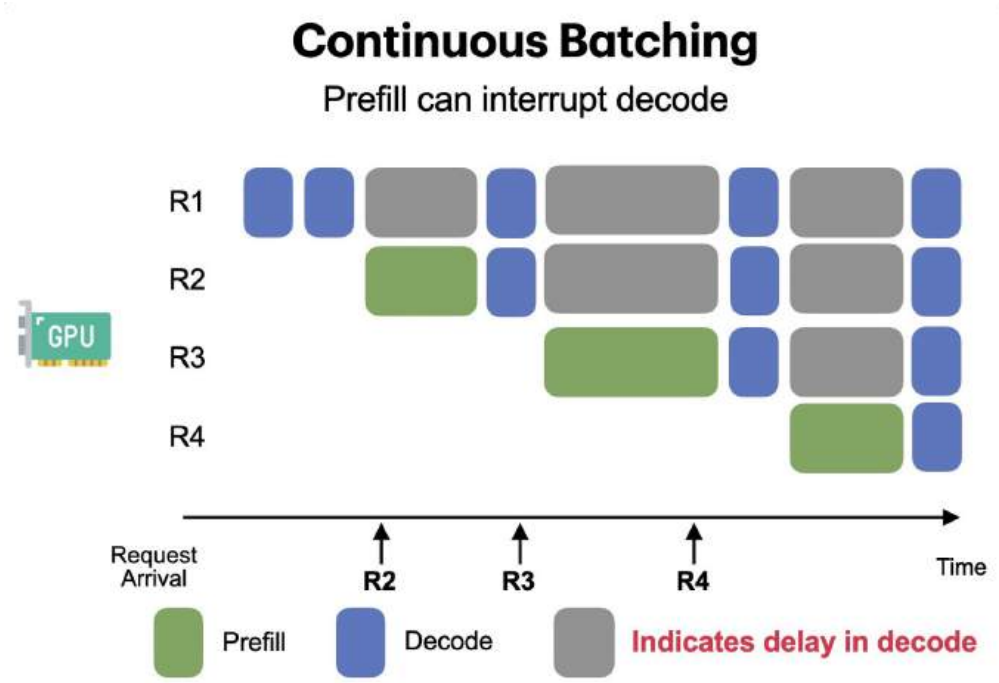
- Application-level memory paging and virtualization for attention KV Cache
- Leverage the concept of virtual memory in operating systems

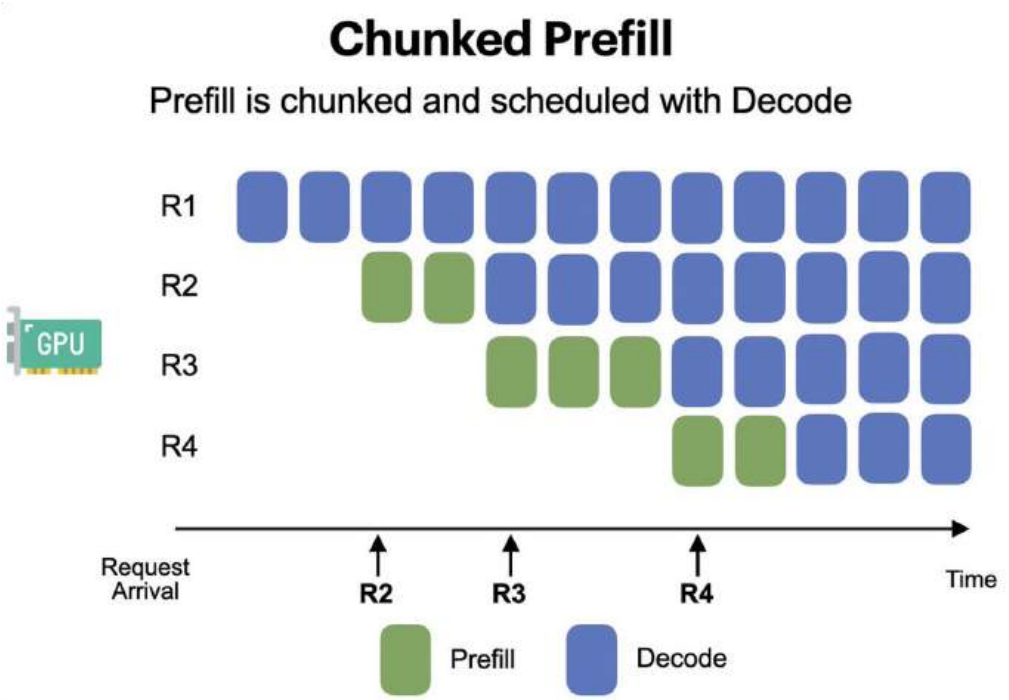


Kwon, Woosuk, et al. "Efficient memory management for large language model serving with pagedattention." *SOSP* 2023.

Chunked Prefill

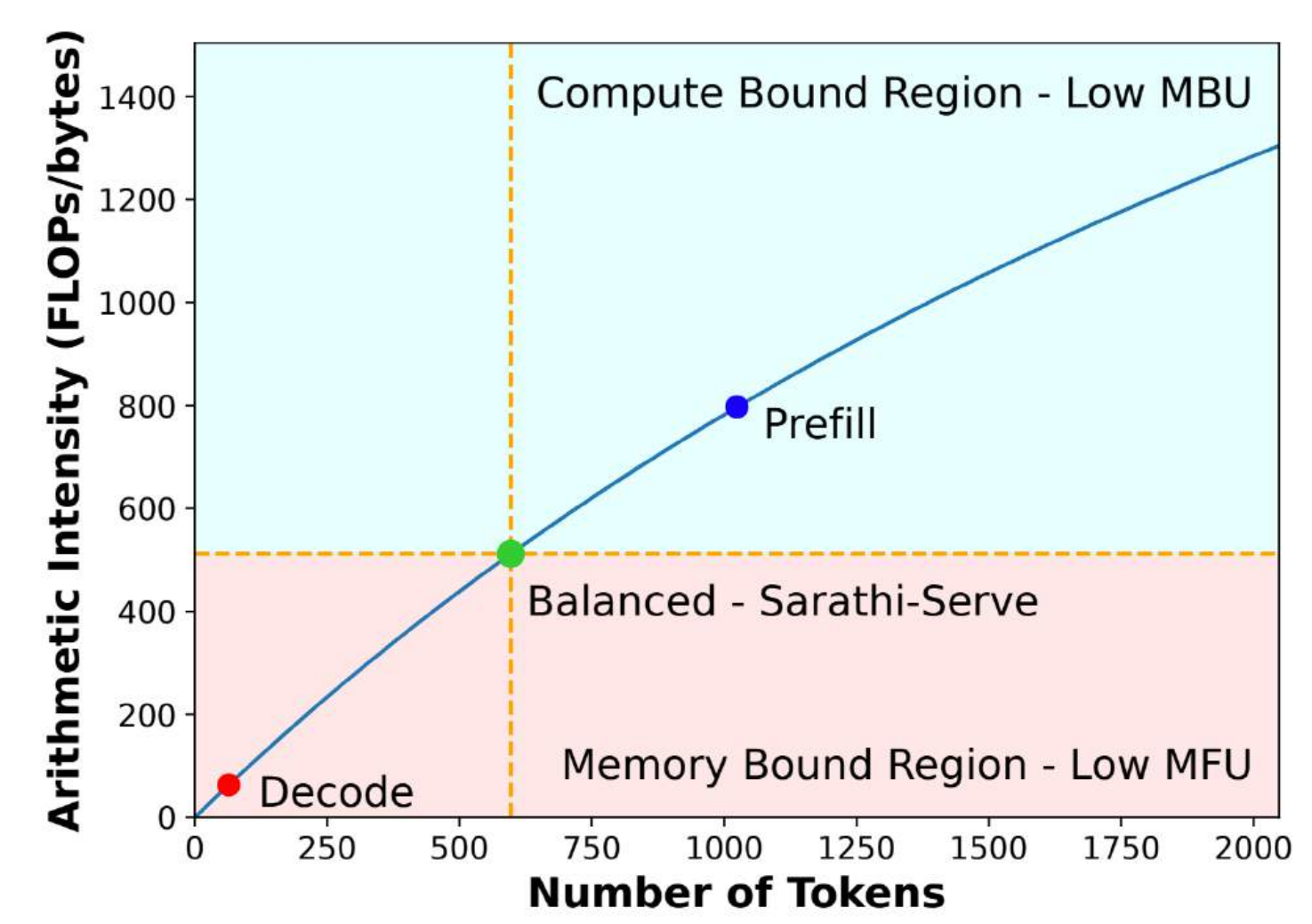
Idea: Do not run a prefill for a long sequence at once as it will block the decoding. Run them chunk by chunk.





Agrawal, Amey, et al. "Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve." *OSDI* 2024.

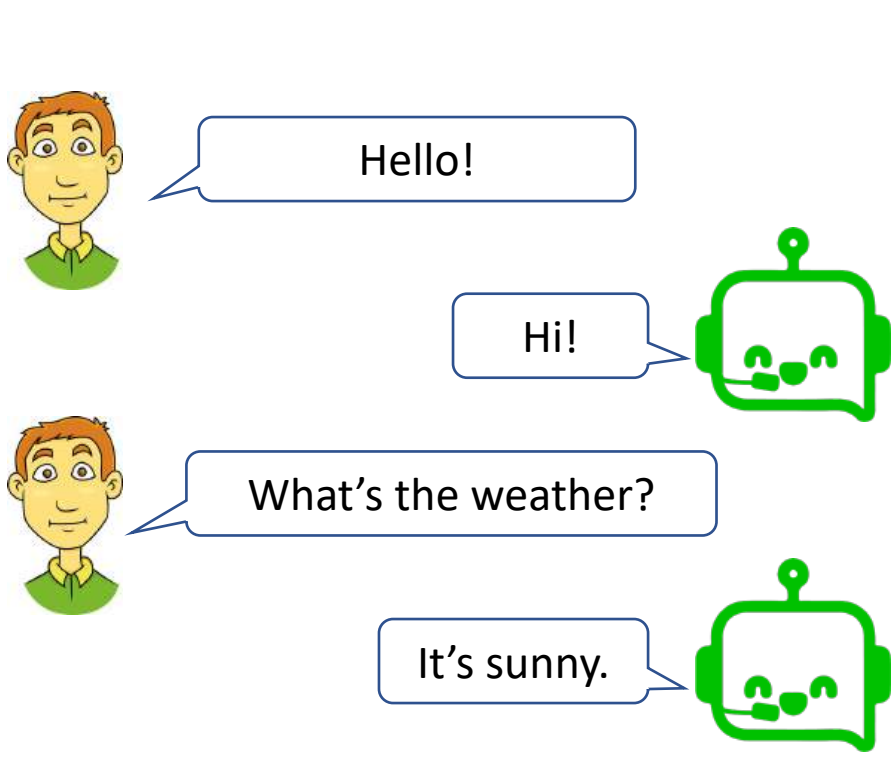
Chunked Prefill

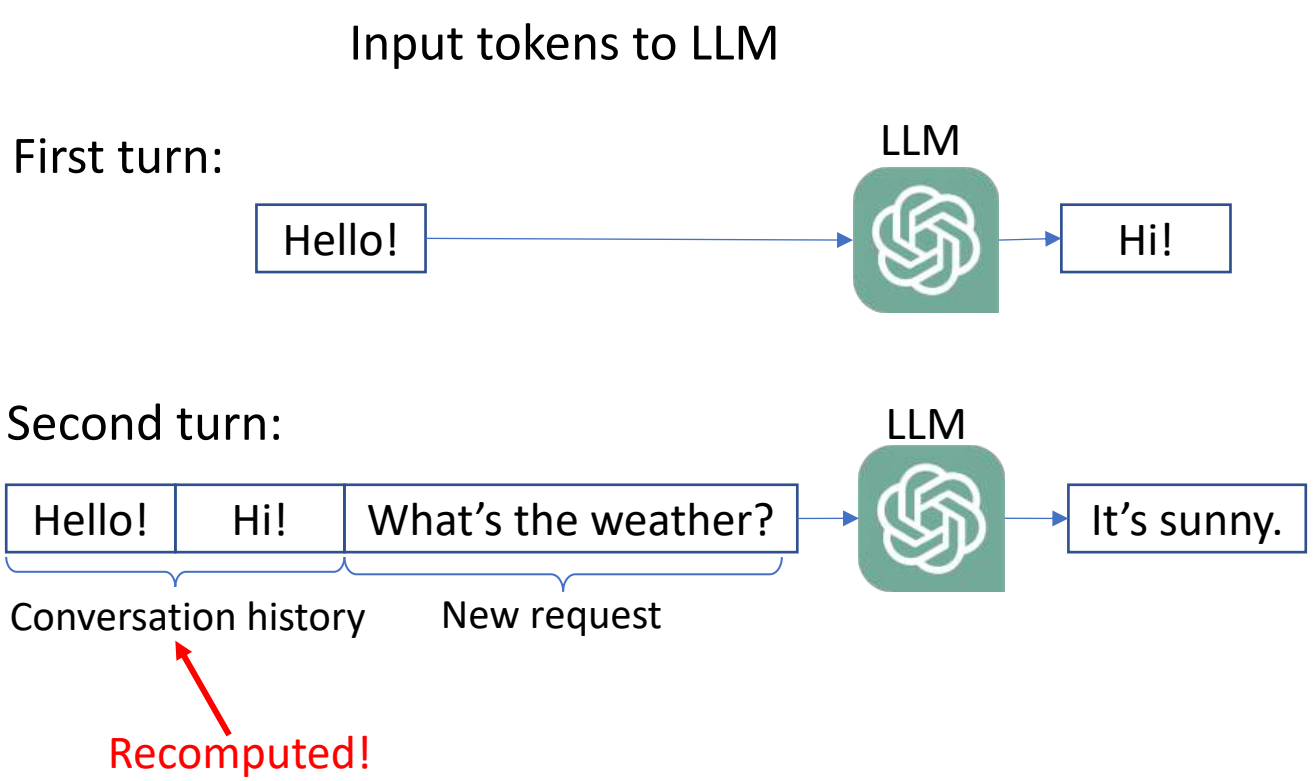


Agrawal, Amey, et al. "Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve." *OSDI* 2024.

Leverage the LLM inference characteristics

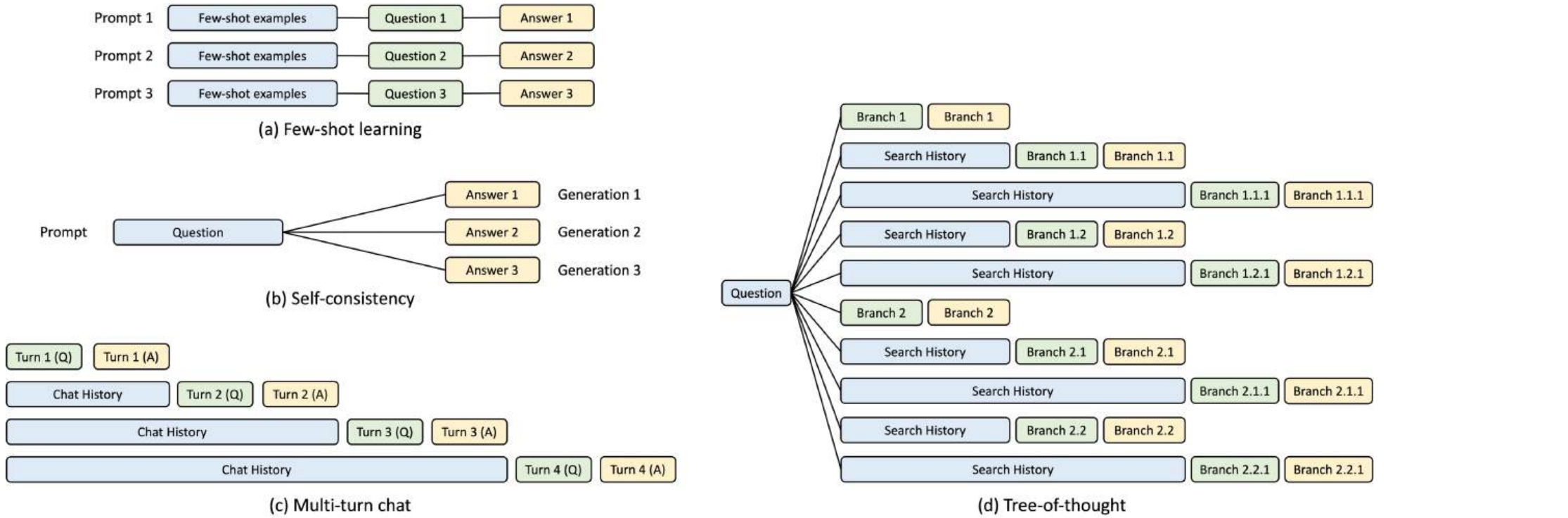
- LLM serving could be stateful





courtesy: Lingyun Yu, Pensieve talk

More examples

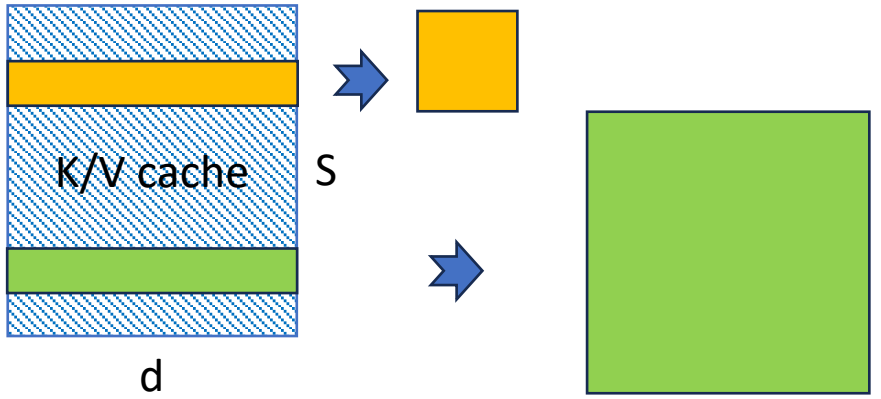


Different requests may have the same prefix of prompts, so reuse kv-cache across requests becomes crucial for LLM serving -> **Prefix Caching**

Zheng, Lianmin, et al. "Efficiently programming large language models using sglang." *arXiv preprint arXiv:2312.07104* (2023).

Pensieve

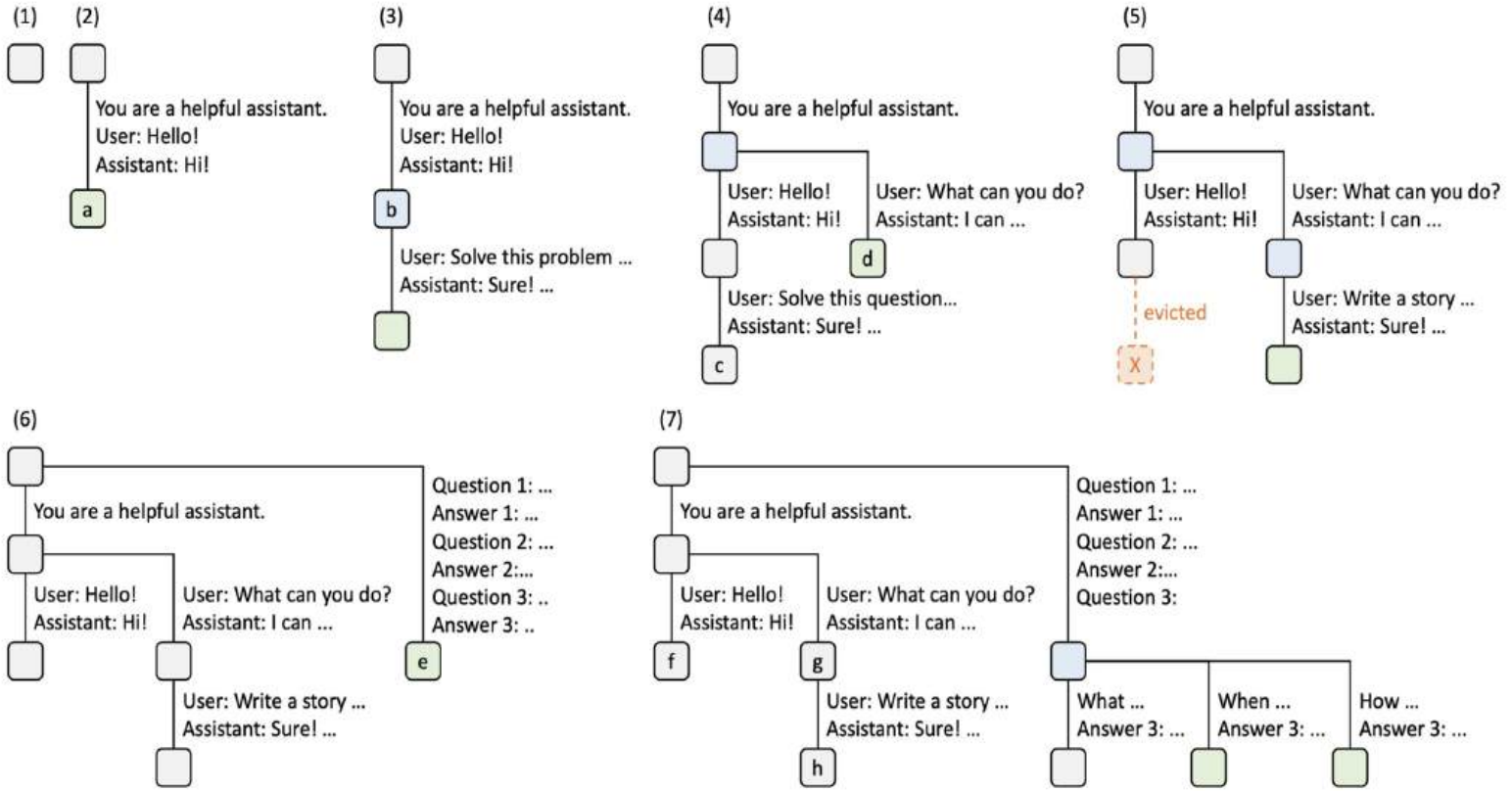
- CPU-accelerator two-layer cache
  - Meticulous engineering work
- Drops leading tokens when OOM
  - Leading tokens generates less computation since earlier tokens generates smaller attention matrices



Yu and Li. "Stateful Large Language Model Serving with Pensieve." *arXiv preprint arXiv:2312.05516* (2024).

RadixAttention

- Retains kv-cache in a radix tree for automatic reusing
- Leverages LRU eviction policy to maintain the tree size



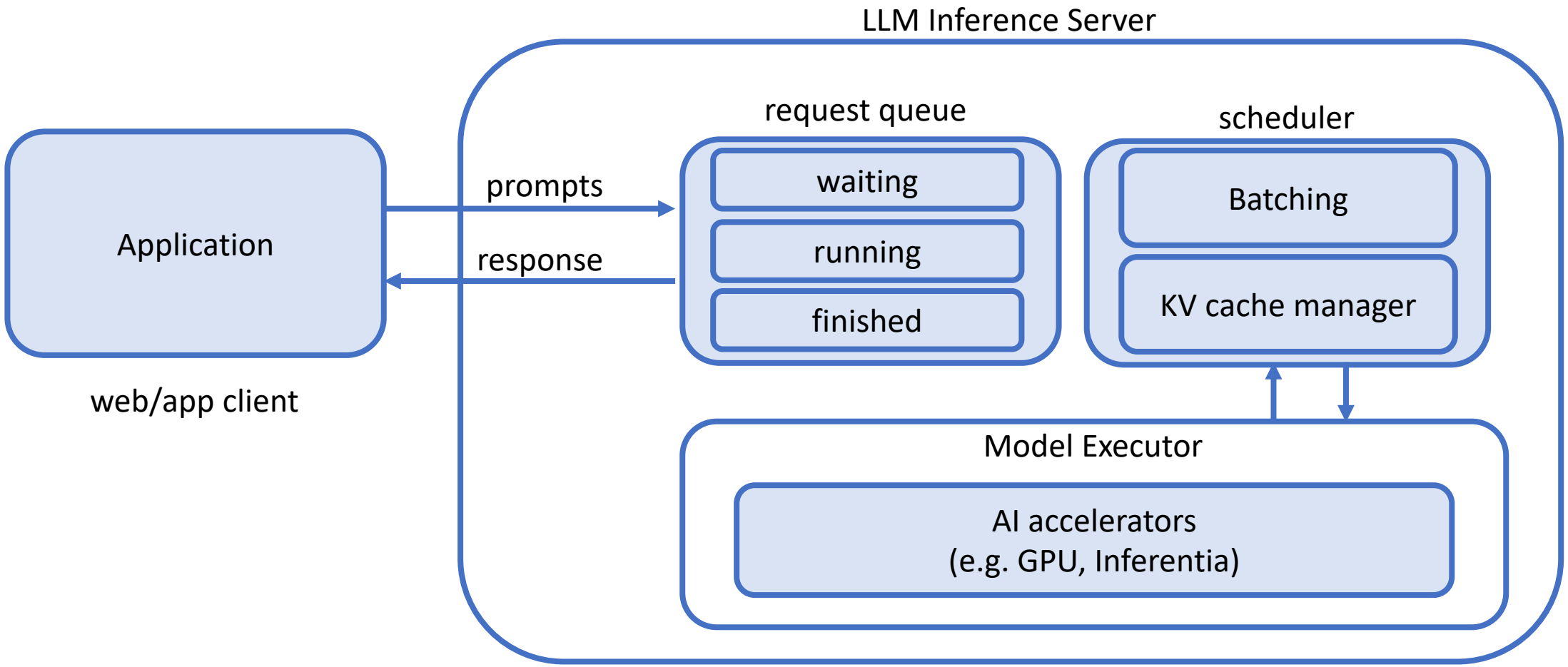
Zheng, Lianmin, et al. "Efficiently programming large language models using sglang." *arXiv preprint arXiv:2312.07104* (2023).

Putting together

Technique/Feature	Major target	By
FlashAttention	TTFT	Reduce memory traffic
FlashDecoding	ITL	Increase computation utilization
Distributed inference	Throughput	Increase computation power
Continuous batching	Throughput	Increase computation utilization
Paged KV Cache	Throughput	Increase memory utilization
Chunked prefill	ITL	Increase computation utilization
Prefix caching	Latency	Reduce duplicated computation

- Example of Inference cost on Amazon EC2 trn1 [Link]

Inference server



Existing solutions

	Pros	Cons
HuggingFace TGI	• Low HTTP overhead • HuggingFace ecosystem	• Flexibility to add new features
TensorRT-LLM	• Highly optimized Kernels	• Closed source, hard to add features • Nvidia GPU only
vLLM (PagedAttention)	• Large community • Up-to-date features	• Large overhead
SGLang (RadixAttention)	• High throughput • Low overhead	• High latency

Primer on model compression

1. Motivation
2. Quantization
3. Distillation
4. Pruning

Motivation

Current Trend: Larger models tend to do better on benchmarks.

Category	Benchmark	Llama 3 8B	Gemma 2 9B	Mistral 7B	Llama 3 70B	Mixtral 8x22B	GPT 3.5 Turbo	Llama 3 405B	Nemotron 4 340B	GPT-4 (0125)	GPT-4o	Claude 3.5 Sonnet
General	MMLU (5-shot)	69.4	**72.3**	61.1	**83.6**	76.9	70.7	87.3	82.6	85.1	89.1	**89.9**
	MMLU (0-shot, CoT)	**73.0**	72.3△	60.5	**86.0**	79.9	69.8	88.6	78.7◁	85.4	**88.7**	88.3
	MMLU-Pro (5-shot, CoT)	**48.3**	–	36.9	**66.4**	56.3	49.2	73.3	62.7	64.8	74.0	**77.0**
	IFEval	**80.4**	73.6	57.6	**87.5**	72.7	69.9	**88.6**	85.1	84.3	85.6	88.0
Code	HumanEval (0-shot)	**72.6**	54.3	40.2	**80.5**	75.6	68.0	89.0	73.2	86.6	90.2	**92.0**
	MBPP EvalPlus (0-shot)	**72.8**	71.7	49.5	**86.0**	78.6	82.0	88.6	72.8	83.6	87.8	**90.5**
Math	GSM8K (8-shot, CoT)	**84.5**	76.7	53.2	**95.1**	88.2	81.6	**96.8**	92.3♢	94.2	96.1	96.4♢
	MATH (0-shot, CoT)	**51.9**	44.3	13.0	**68.0**	54.1	43.1	73.8	41.1	64.5	**76.6**	71.1
Reasoning	ARC Challenge (0-shot)	83.4	**87.6**	74.2	**94.8**	88.7	83.7	**96.9**	94.6	96.4	96.7	96.7
	GPQA (0-shot, CoT)	32.8	–	28.8	**46.7**	33.3	30.8	51.1	–	41.4	53.6	**59.4**
Tool use	BFCL	**76.1**	–	60.4	84.8	–	**85.9**	88.5	86.5	88.3	80.5	**90.2**
	Nexus	**38.5**	30.0	24.7	**56.7**	48.5	37.2	**58.7**	–	50.3	56.1	45.7
Long context	ZeroSCROLLS/QuALITY	81.0	–	–	90.5	–	–	**95.2**	–	**95.2**	90.5	90.5
	InfiniteBench/En.MC	65.1	–	–	78.2	–	–	**83.4**	–	72.1	82.5	–
	NIH/Multi-needle	98.8	–	–	97.5	–	–	98.1	–	**100.0**	**100.0**	90.8
Multilingual	MGSM (0-shot, CoT)	**68.9**	53.2	29.9	**86.9**	71.1	51.4	**91.6**	–	85.9	90.5	**91.6**

Image Source: The Llama 3 Herd of Models, https://llama.meta.com/

Motivation

Challenge: Inference is *expensive* and *slow* with *large* models

- **Need multiple AI accelerator chips**

  Llama-3.1 405B
  - ≈ 810 GB in native precision (BF16)
  - 512 GB off-chip shared memory (16 chips) in an Amazon EC2 trn1n.32xlarge instance
  - Need at least 2 trn1n.32xlarge to load *weights only* in BF16
    USD 434, 505 in annual on-demand cost (at USD 24.78 / hr)

AWS EC2 Trainium Source: https://aws.amazon.com/ec2/instance-types/trn1/ (date: 8/18/2024)

Motivation

Challenge: Inference is *expensive* and *slow* with *large* models

- **Slower decoding**
  Large memory I/O from off-chip to on-chip memory for loading the weights and intermediate states

  For example, KV cache size per token is 320 KB for Llama-3.1 70B versus 504 KB for Llama-3.1 405B in BF16.

Motivation

Challenge: Inference is *expensive* and *slow* with *large* models

- **Slower prefill**
  Number of flops increases as the model size increases

  Forward pass through the linear layers (ignoring attention) of a decoder-only Transformer model with P parameters requires $\approx 2 \cdot P$ flops, and $\approx P.b$ bytes of memory access (b bytes per parameter) from accelerator's off-chip memory.

Napkin Math Source: https://kipp.ly/transformer-inference-arithmetic/#flops-counting

Key Idea

- Compress a large model into a smaller model
  - Use low-precision data types
  - Use fewer parameters
- Trade-off between accuracy and inference improvement

Benefits

- Reduces the number of flops
  - Applies to *some* model compression approaches
    (e.g., pruning but not quantization)
  - Speeds up prefill
    Removing 50% of the parameters of the model reduces the flops by a half

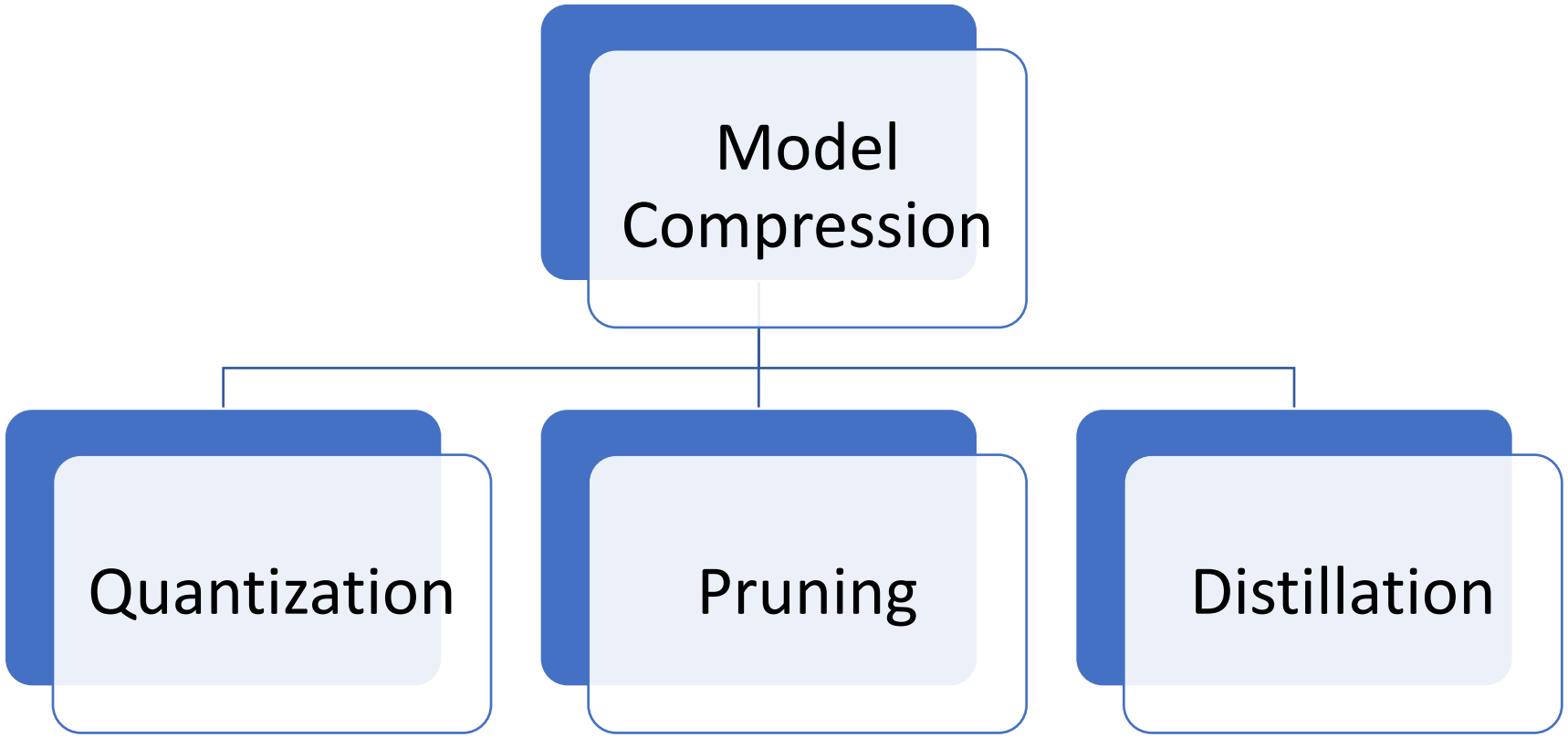
- Reduces the memory footprint of LLMs
  - Applies to *all* model compression methods
  - Number of AI accelerators to serve the model goes down
  - Speeds up decoding

Model compression measurement

$$\text{Compression Ratio } (r) = \frac{Uncompressed\ Model\ Size}{Compressed\ Model\ Size}$$

$r > 1 \Rightarrow$ effective compression

Model compression methods



Model quantization

Quantization

- Computes and stores *tensors* at lower bit-widths

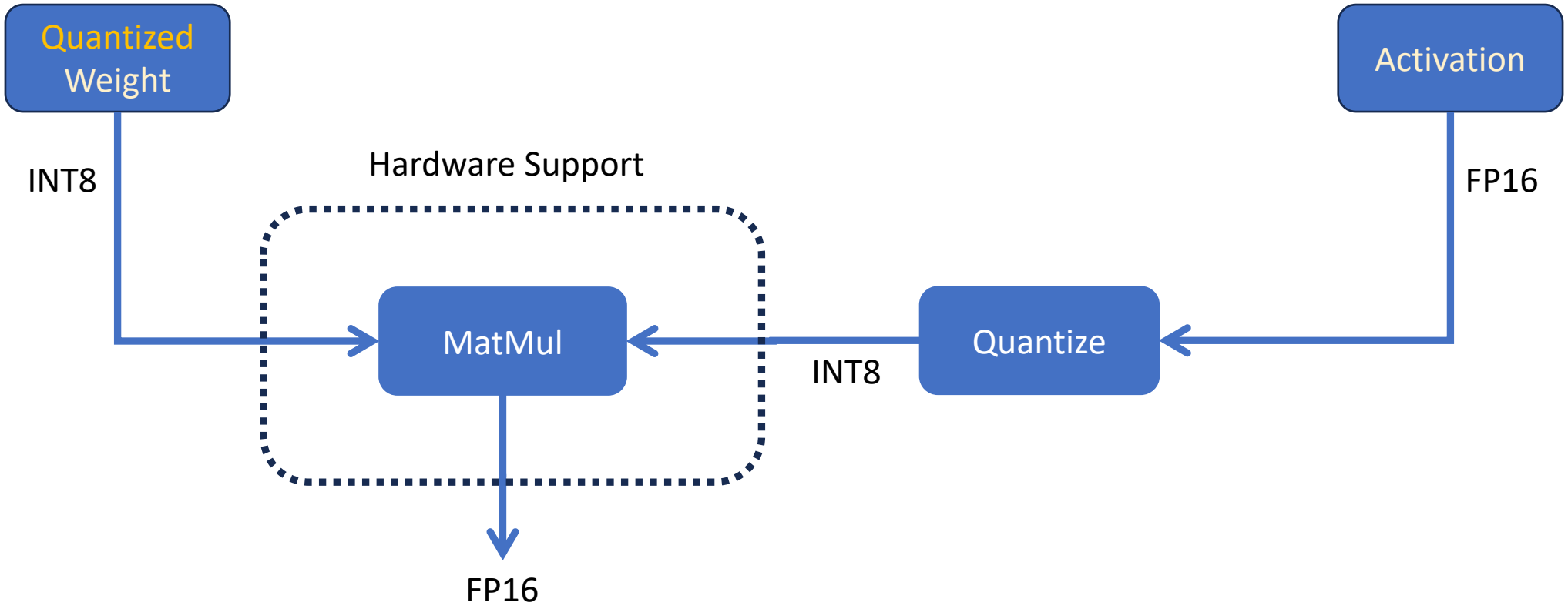
16 bit (e.g., BF16) → 8 bit (e.g., INT8) ⇒ $r = 2$

16 bit (e.g., BF16) → 4 bit (e.g., INT4) ⇒ $r = 4$

- Supported by popular AI accelerators

INT8	FP8
Amazon EC2 trn1/inf2 Neuron Chips NVIDIA Tensor Cores	Amazon EC2 trn2/inf3 Neuron Chips (upcoming!) NVIDIA Hopper Tensor Cores

Example Operation

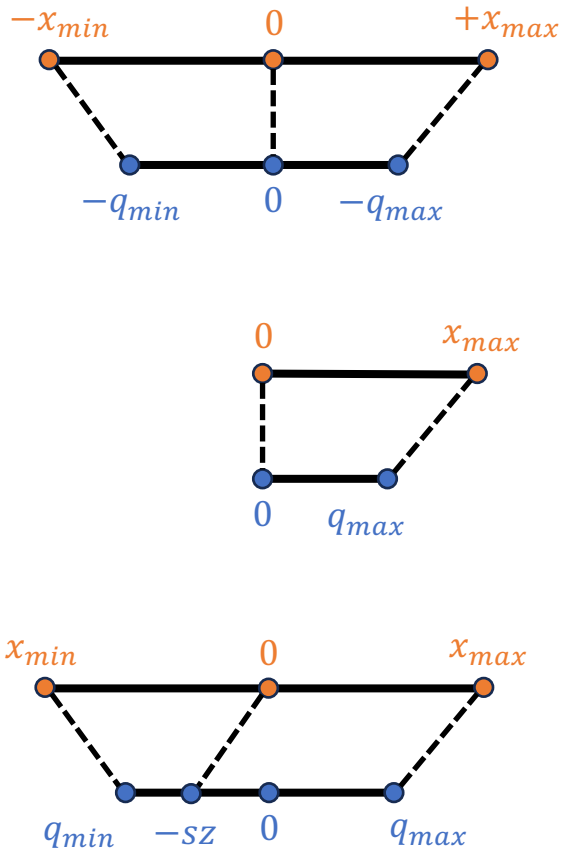


Quantization

- No reduction in #flops
  - Same #params as in non-quantized model
  - Dequantization step adds additional flops
  - Does not speedup prefill by a lot
- Reduces the #accelerators to host the model
- Reduces the memory I/O
  - Speeds up decoding

Symmetric vs Asymmetric quantization

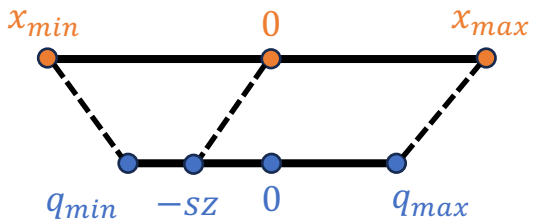
- **Symmetric:**
  Zero of higher bit-width coincides with zero of lower bit-width
  - **Signed** lower bit-width grid for symmetric distributions (e.g., Gaussian)
  - **Unsigned** lower bit-width grid for one-tailed distributions (e.g., ReLU activations)
- **Asymmetric:**
  Zero points do not coincide



Asymmetric quantization

Input: x, floating-point

Output: x_q in unsigned integer grid $\{0, \dots, 2^b - 1\}$



$$x_q = clamp\left(round\left(\frac{x}{s}\right) + z; 0, 2^b - 1\right)$$

z: where zero floating point maps to in UINT grid

s: is quantization step size (scale factor)

$round$: round-to-nearest integer operator

De-Quantization:

$$x \approx \hat{x} = s(x_q - z)$$

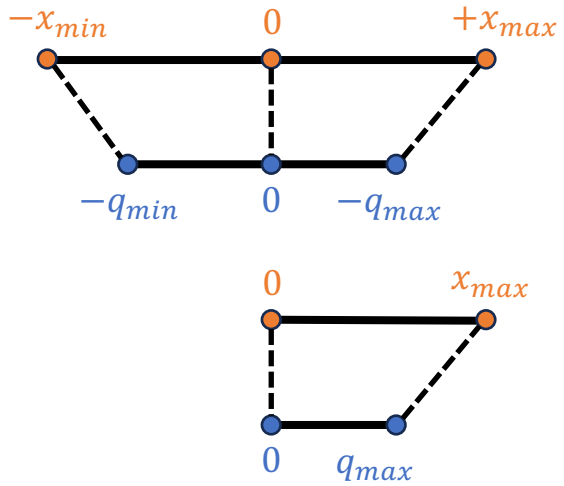
$$clamp\left(x; 0, 2^b - 1\right) = \begin{cases} 0, & x < 0 \\ x, & 0 \le x \le 2^b - 1 \\ 2^b - 1, & x > 2^b - 1 \end{cases}$$

Symmetric quantization

Input: x, floating-point vector

Output: x_{int} in unsigned integer grid $\{\mathbf{0}, \dots, \mathbf{2^b - 1}\}$ or signed integer grid $\{\mathbf{-2^{b-1}}, \dots, \mathbf{2^{b-1} - 1}\}$

$$x_q = clamp\left(round\left(\frac{x}{s}\right); \boldsymbol{a}, \boldsymbol{c}\right)$$



Zero floating point maps to zero in quantized grid (asymmetric special case with $z = 0$)

Less computational overhead (asymmetric case without $+z$ ops)

De-Quantization:

$$x \approx \hat{x} = s(x_q)$$

Fundamental problem with quantization

- **Clipping Error:** Any values of x that lie outside $\{q_{min}, q_{max}\}$ is clipped to the limits, where

$$q_{min} = s(\boldsymbol{a} - z)$$

$$q_{max} = s(\boldsymbol{c} - z)$$

  - Loss of information *only* for values outside the input range.
- **Rounding Error:** Quantized output does not map back to the original input $\hat{x} = x$
  - Loss of information for all values of input.
  - Bounded by $\{-\frac{1}{2}s, \frac{1}{2}s\}$ when the input x lies halfway between two quantization levels

Granularity of quantization parameters

Per-tensor basis:
Scale factor s and zero point z computed per tensor

Per-channel basis:
(s, z) computed per component along one dimension

Outlier values only impact their component instead of entire tensor
⇒ Better rounding error

Requires more memory!

Typically quantized components in LLMs

- Weights of linear layers in attention and feedforward layers
- Activations
- KV Cache

How to quantize different components?

Dynamic quantization

- Quantize weights offline
- Learn quantization parameters on-the-fly
- Read / write activations from / to memory in higher bit-width (e.g., FP32)
- Quantize activations on-the-fly
- Perform tensor operations in lower bit-width (e.g., INT8)

Ref: https://pytorch.org/docs/stable/quantization.html

How to quantize different components?

Static quantization

- Quantize weights offline
- Learn quantization parameters offline with calibration data
- Read / write activations from / to memory in lower bit-width (e.g., INT8)
- Quantize activations on-the-fly
- Perform tensor operations in lower bit-width (e.g., INT8)

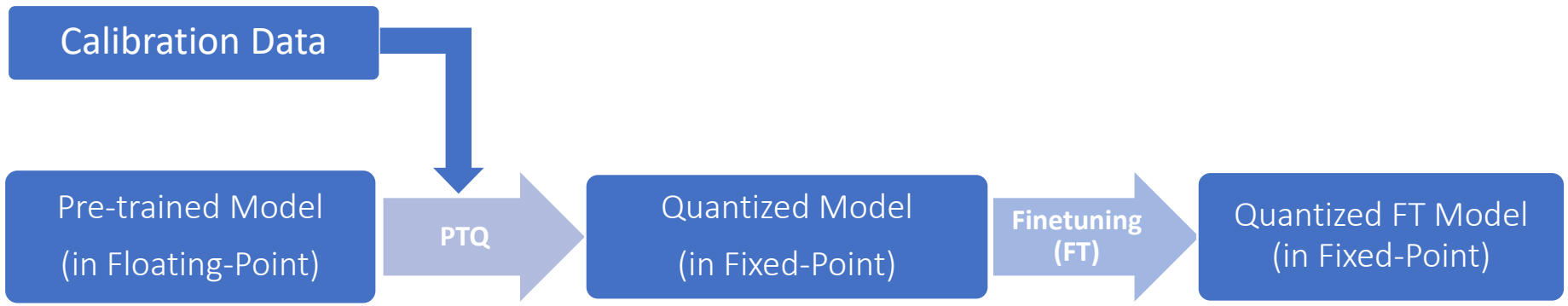
Ref: https://pytorch.org/docs/stable/quantization.html

When to quantize?

- Post-training quantization
- Quantization-aware training

Post-training quantization (PTQ)

- Quantize a pre-trained model (w/o access to the pre-training pipeline)
- Data-free or may require a small calibration dataset
- Cheap to run
- Significant accuracy degradation (typically) at less than 8 bit-width (finetuning needed to recover the accuracy drop)



Challenges in quantizing LLMs

- **Outlier:** hidden activation dimension with at least one value ≥ 6 in at least 25% of Transformer layers and at least 6% of all sequence dimensions
- Outliers in activations are ubiquitous as model size grows (beyond 6.7B), accordingly performance degrades rapidly.

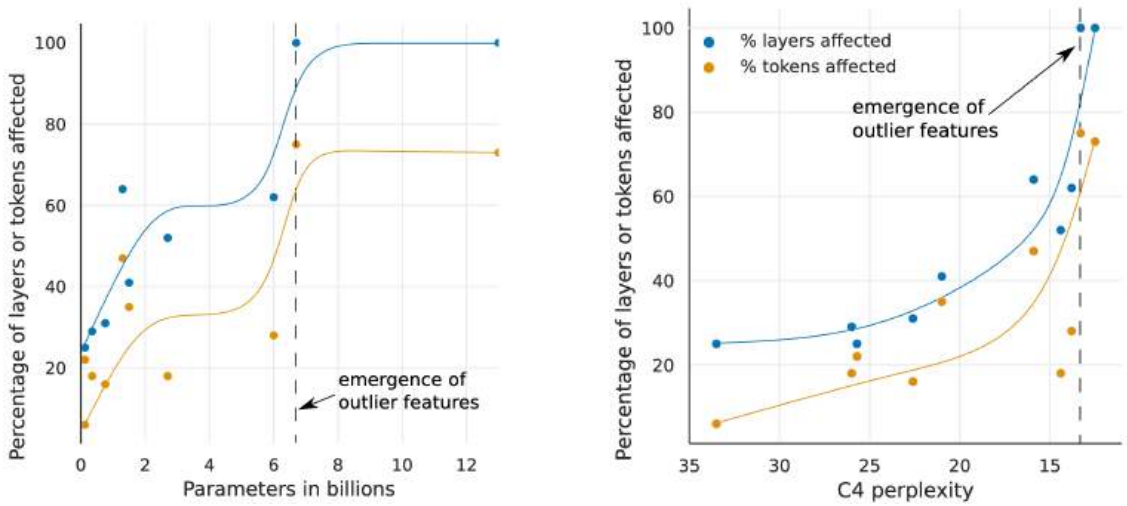


Image Source: Dettmers et al., LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale, NeurIPS 2022

PTQ methods

Methods quantizing *weights* only

- GPTQ (Frantar et al., ICLR 2023)
  - Quantizes weight matrix by minimizing squared error
  - Iteratively computes quantization error, and updates weights

- AWQ (Lin et al., MLSys 2024)
  - Small fractions of the weights are important (called salient weights)
  - Applies activation-aware scaling to the weights per-channel before quantization to protect the salient weights

- QuIP (Chee et al, NeurIPS 2023)
  - Transforms weight matrix to ~iid Gaussian
  - Minimizes a proxy error

Frantar et al., GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers, ICLR 2023
Lin et al., AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration, MLSys 2024
Chee et al., QuIP: 2-Bit Quantization of Large Language Models With Guarantees, NeurIPS 2023

PTQ methods

Methods quantizing *weights* and *activations*

- LLM.int8() (Dettmers et al., NeurIPS 2022)
  - Extracts outliers by column from weight and activation matrices
  - Performs matrix multiplication of outlier submatrices in FP16 and non-outliers in INT8
  - Efficient mixed-precision decomposition in AI accelerators is difficult

- SmoothQuant (Xiao et al., ICML 2023)
  - Scales activation matrix per channel to mitigate outliers, and scales weight in the reverse direction
  - Makes quantization overall easier

Dettmers et al., LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale, NeurIPS 2022
Xiao et al., SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models, ICML 2023

PTQ methods

Methods quantizing *KV cache*

- KVQuant (Hooper et al., arXiv 2024)
  - Long context length models typically demand a large KV cache memory
  - Quantizes Key activations before the rotary embedding to mitigate its impact on quantization
  - Derives per-layer sensitivity-weighted non-uniform datatypes that better represent the distributions
  - Isolates outliers separately for each vector to minimize skews in quantization ranges
  - Enables efficient long context length inference (e.g., 10 Million context length with Llama 7B on a 8 GPU machine)

Hooper et al., KVQuant: Towards 10 Million Context Length LLM Inference with KV Cache Quantization, arXiv 2024

SmoothQuant deep dive – motivation

Observations

- Weights are easy to quantize, but activations are hard due to **outliers**. But they persist in fixed channels.
- Small variance inside each channel (e.g., hidden dimension), but large across each token

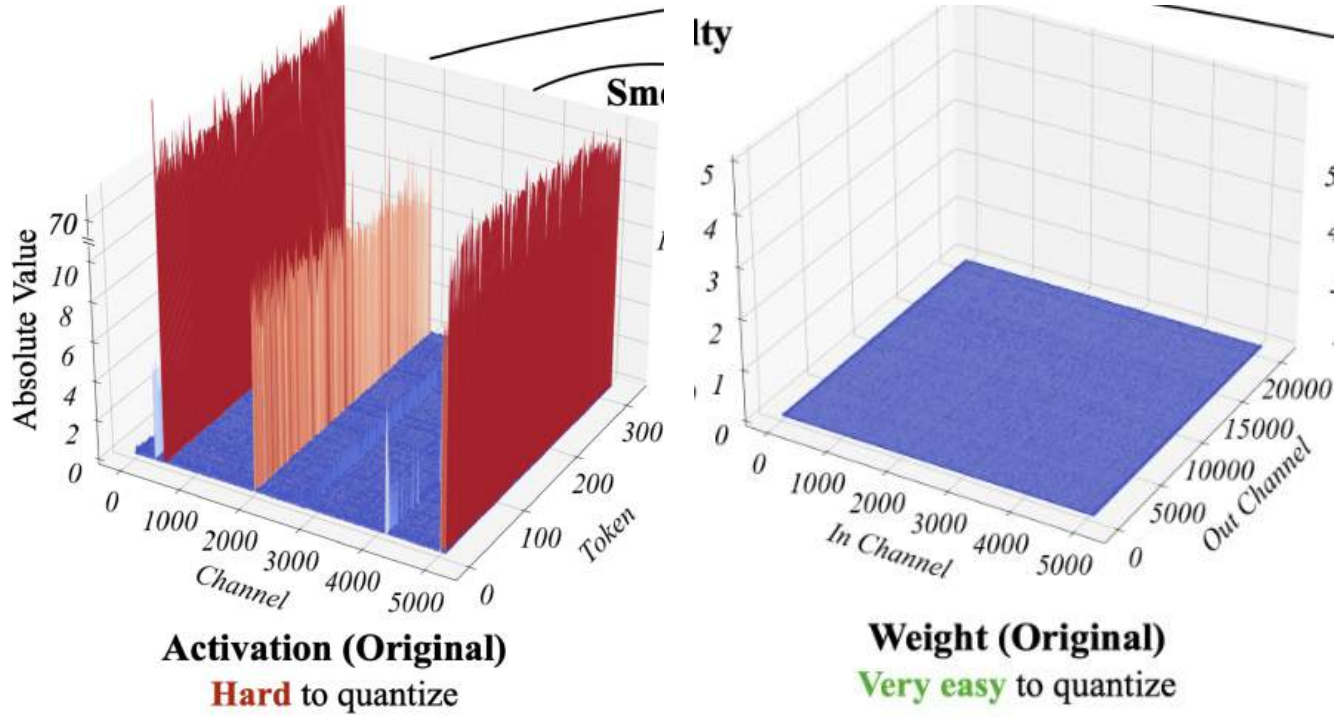
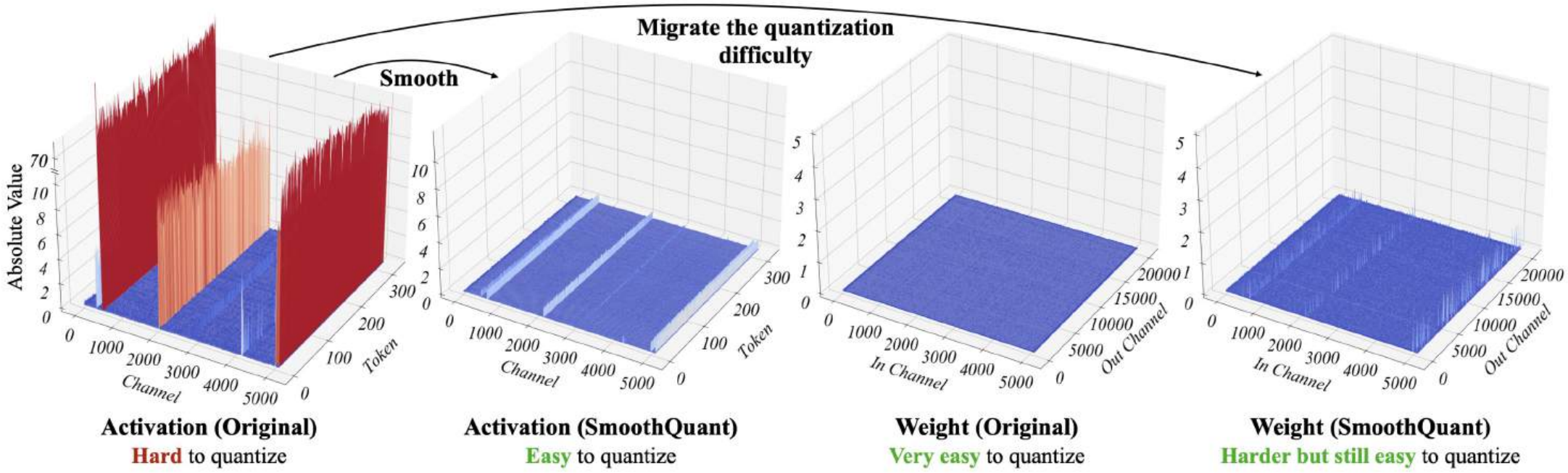


Image Source: Xiao et al., SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models, ICML 2023

SmoothQuant deep dive – key idea



- SmoothQuant migrates the quantization difficulty from activations to weights by scaling activation matrix per channel, and scaling weight in the reverse direction

Image Source: Xiao et al., SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models, ICML 2023

SmoothQuant deep dive – accuracy results

Method	#W	#A	#G	CommonSenseQA↑					
				PIQA	ARC-e	ARC-c	HellaSwag	Wino	Avg.
LLaMA3	16	16	-	79.9	80.1	50.4	60.2	72.8	68.6
RTN	4	16	128	76.6	70.1	45.0	56.8	71.0	63.9
	3	16	128	62.3	32.1	22.5	29.1	54.7	40.2
	2	16	128	53.1	24.8	22.1	26.9	53.1	36.0
	8	16	-	79.7	80.8	50.4	60.1	73.4	68.9
	4	16	-	75.0	68.2	39.4	56.0	69.0	61.5
	3	16	-	56.2	31.1	20.0	27.5	53.1	35.6
	2	16	-	53.1	24.7	21.9	25.6	51.1	35.3
GPTQ	4	16	128	78.4	78.8	47.7	59.0	72.6	67.3
	3	16	128	74.9	70.5	37.7	54.3	71.1	61.7
	2	16	128	53.9	28.8	19.9	27.7	50.5	36.2
	8	16	-	79.8	80.1	50.2	60.2	72.8	68.6
	4	16	-	76.8	74.3	42.4	57.4	72.8	64.8
	3	16	-	60.8	38.8	22.3	41.8	60.9	44.9
	2	16	-	52.8	25.0	20.5	26.6	49.6	34.9
AWQ	4	16	128	79.1	79.7	49.3	59.1	74.0	68.2
	3	16	128	77.7	74.0	43.2	55.1	72.1	64.4
	2	16	128	52.4	24.2	21.5	25.6	50.7	34.9
	8	16	-	79.6	80.3	50.5	60.2	72.8	68.7
	4	16	-	78.3	77.6	48.3	58.6	72.5	67.0
	3	16	-	71.9	66.7	35.1	50.7	64.7	57.8
	2	16	-	55.2	25.2	21.3	25.4	50.4	35.5
SliM-LLM	4	16	128	78.9	79.9	49.4	58.7	72.6	67.9
	3	16	128	77.8	73.7	42.9	55.5	72.8	64.5
	2	16	128	57.1	35.4	26.1	28.9	56.6	40.8
QuIP	4	16	-	78.2	78.2	47.4	58.6	73.2	67.1
	3	16	-	76.8	72.9	41.0	55.4	72.5	63.7
	2	16	-	52.9	29.0	21.3	29.2	51.7	36.8
DB-LLM	2	16	128	68.9	59.1	28.2	42.1	60.4	51.8
PB-LLM	2	16	128	57.0	37.8	17.2	29.8	52.5	38.8
	1.7	16	128	52.5	31.7	17.5	27.7	50.4	36.0
BiLLM	1.1	16	128	56.1	36.0	17.7	28.9	51.0	37.9
SmoothQuant	8	8	-	79.5	79.7	49.0	60.0	73.2	68.3
	6	6	-	76.8	75.5	45.0	56.9	69.0	64.6
	4	4	-	54.6	26.3	20.0	26.4	50.3	35.5

Llama-3 8B

Method	#W	#A	#G	CommonSenseQA↑					
				PIQA	ARC-e	ARC-c	HellaSwag	Wino	Avg.
LLaMA3	16	16	-	82.4	86.9	60.3	66.4	80.6	75.3
RTN	4	16	128	82.3	85.2	58.4	65.6	79.8	74.3
	3	16	128	64.2	48.9	25.1	41.1	60.5	48.0
	2	16	128	53.2	23.9	22.1	25.8	53.0	35.6
GPTQ	4	16	128	82.9	86.3	58.4	66.1	80.7	74.9
	3	16	128	80.6	79.6	52.1	63.5	77.1	70.6
	2	16	128	62.7	38.9	24.6	41.0	59.9	45.4
AWQ	4	16	128	82.7	86.3	59.0	65.7	80.9	74.9
	3	16	128	81.4	84.7	58.0	63.5	78.6	73.2
	2	16	128	52.2	25.5	23.1	25.6	52.3	35.7
SliM-LLM	4	16	128	82.9	86.5	59.0	66.2	80.7	75.1
	3	16	128	81.6	83.1	58.5	64.7	78.4	73.3
	2	16	128	76.2	66.3	45.7	55.4	63.7	61.5
QuIP	4	16	-	82.5	86.0	58.7	65.7	79.7	74.5
	3	16	-	82.3	83.3	54.9	63.9	78.4	72.5
	2	16	-	65.3	48.9	26.5	40.9	61.7	48.7
PB-LLM	2	16	128	65.2	40.6	25.1	42.7	56.4	46.0
	1.7	16	128	56.5	49.9	25.8	34.9	53.1	44.1
BiLLM	1.1	16	128	58.2	46.4	25.1	37.5	53.6	44.2
SmoothQuant	8	8	-	82.2	86.9	60.2	66.3	80.7	75.3
	6	6	-	82.4	87.0	59.9	66.1	80.6	75.2
	4	4	-	76.9	75.8	43.5	52.9	58.9	61.6

Llama-3 70B

Image Source: Huang et al., An Empirical Study of LLaMA3 Quantization: From LLMs to MLLMs, arXiv 2024

Quantization-aware training (QAT)

- Simulates inference-time quantization during training
  - Weights / activations are “fake quantized” in forward/backward pass
  - Compute / store at higher bit-width (e.g., FP32)
  - Parameters adjust to account for quantization later post-training
- Full training pipeline is needed
- Expensive to run but higher accuracy than PTQ

QAT methods

- BitNet (Wang & Ma et al., arXiv 2024)
  - Binarizes the weight to -1 or +1 (using signum)
  - Quantizes activations to 8 bits
  - Trains using Straight-Through Estimator (STE)
  - Gradient and Optimizers are stored in higher precision
- 1.5-bit LLM (Ma & Wang et al., arXiv 2024)
  - Quantizes the weight to -1, 0, +1
  - Quantizes activations to 8 bits
  - Trains similar to BitNet

Wang et al., BitNet: Scaling 1-bit Transformers for Large Language Models, arXiv 2024
Ma & Wang et al., The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits, arXiv 2024

QAT methods

- LLM-QAT (Liu et al., arXiv 2023)
  - Distillation on data generated by a pre-trained model
  - Quantizes weight, activation and KV cache up to 4 bits

Liu et al., LLM-QAT: Data-Free Quantization Aware Training for Large Language Models, arXiv 2023

1.5-bit LLM deep dive

BitNet Architecture: Transformer with nn.Linear replaced by BitLinear

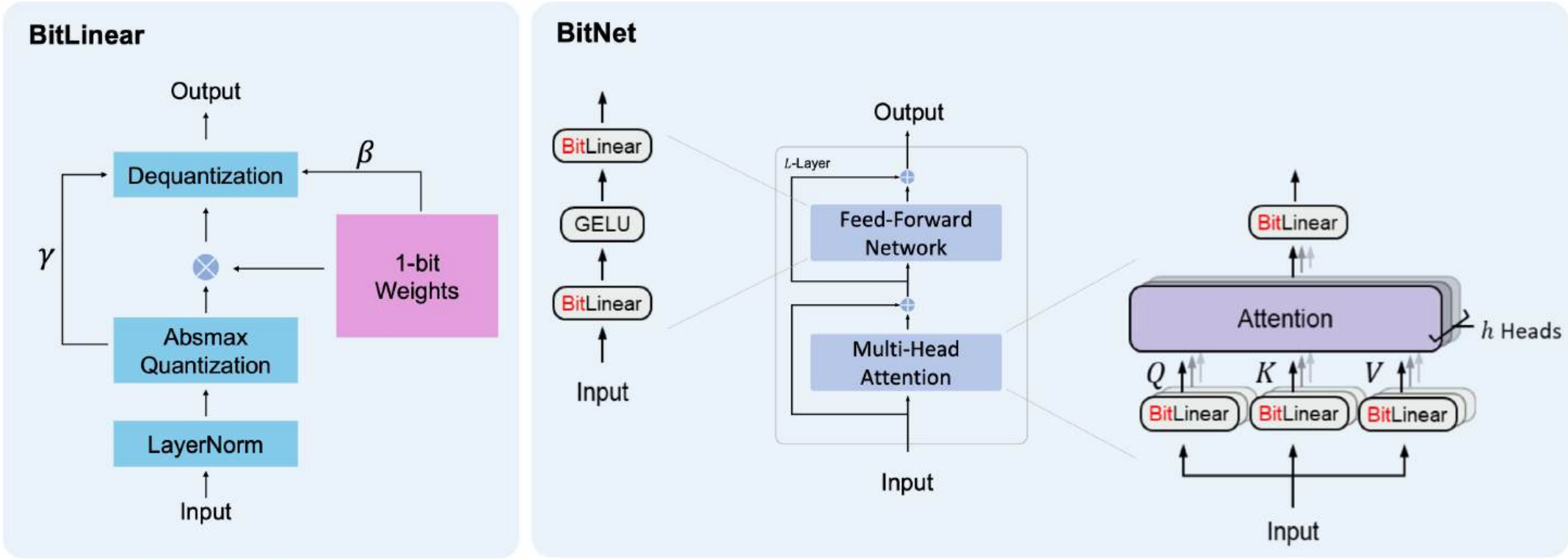
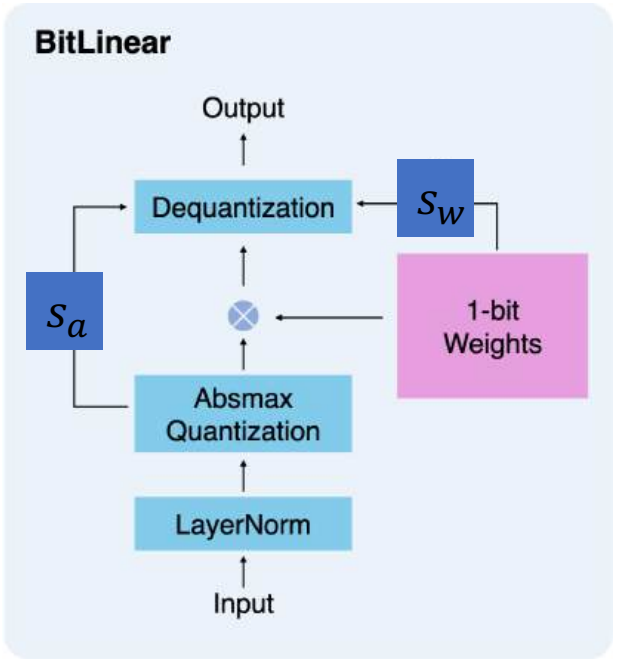


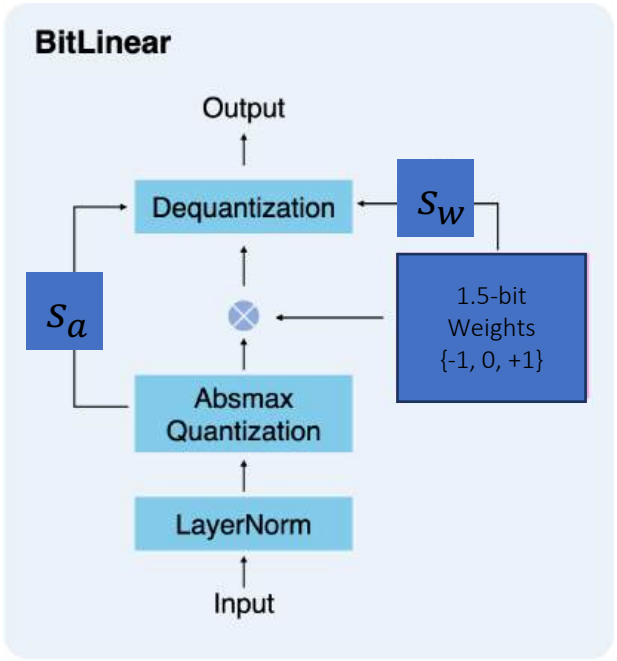
Image Source: Ma & Wang et al., The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits, arXiv 2024

1.5-bit LLM deep dive – architecture

1-bit BitLinear



1.5-bit BitLinear



- 1.5-bit LLM replaces 1-bit BitLinear with 1.5-bit BitLinear layer
- s_a and s_w are scaling parameters for activation and weights resp.

Image Source: Ma & Wang et al., The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits, arXiv 2024

1.5-bit LLM deep dive – accuracy results

- BitNet b1.58 can match the performance of the full precision baseline starting from a 3B size.

Models	Size	ARCe	ARCc	HS	BQ	OQ	PQ	WGe	Avg.
LLaMA LLM	700M	54.7	23.0	37.0	60.0	20.2	68.9	54.8	45.5
BitNet b1.58	700M	51.8	21.4	35.1	58.2	20.0	68.1	55.2	44.3
LLaMA LLM	1.3B	56.9	23.5	38.5	59.1	21.6	70.0	53.9	46.2
BitNet b1.58	1.3B	54.9	24.2	37.7	56.7	19.6	68.8	55.8	45.4
LLaMA LLM	3B	62.1	25.6	43.3	61.8	24.6	72.1	58.2	49.7
BitNet b1.58	3B	**61.4**	**28.3**	**42.9**	**61.5**	**26.6**	**71.5**	**59.3**	**50.2**
BitNet b1.58	3.9B	**64.2**	**28.7**	**44.2**	**63.5**	**24.2**	**73.2**	**60.5**	**51.2**

Image Source: Ma & Wang et al., The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits, arXiv 2024

Quantization Takeaways

- Quantization stipulates hardware support and efficient kernels
- Quantization does not reduce the number of floating point operations because the number of parameters is the same as the higher-precision model, instead adds flops due to dequantization step
- Minimal loss reported online for standard downstream tasks up to 4 bits with one-shot quantization. For example, see https://huggingface.co/collections/neuralmagic/llama-31-quantization-66a3f907f48d07feabb8f300. Lower bit-widths require quantization aware training or finetuning.

Model distillation

Model distillation history

Ensemble of models:

Performance(Ensemble of models) > Performance(single model)
Inference Cost(Ensemble of models) > Inference Cost(single model)

Model distillation:
Compress ensemble of models / Large model → smaller model
(Bucila et al., KDD 2006, Hinton et al., NeurIPS 2014)

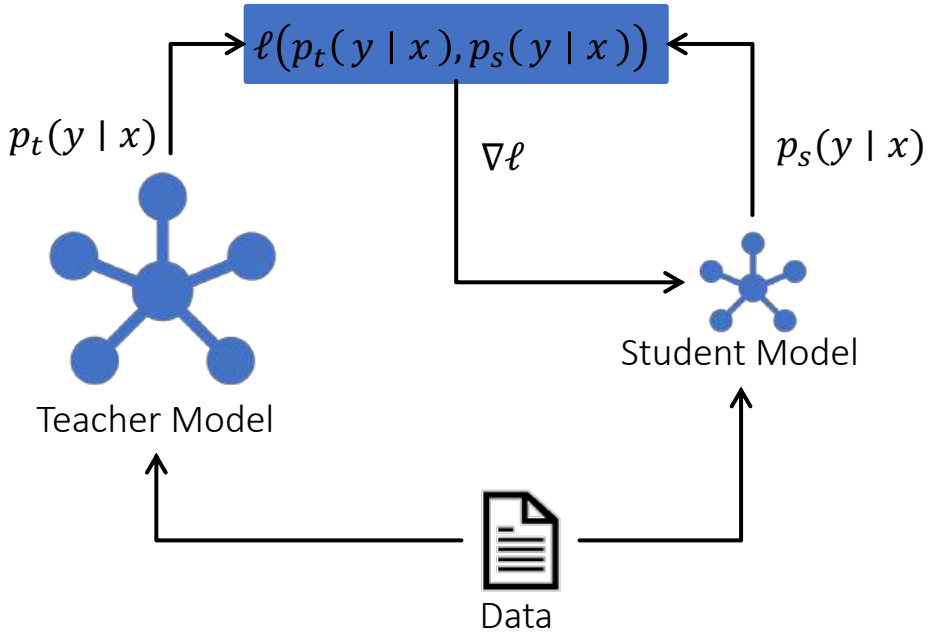
Similar scenario with LLMs with billions of parameters

Bucila et al., Model Compression, KDD 2006
Hinton et al., Distilling the Knowledge in a Neural Network, NeurIPS 2014

Key idea

Analogy from formal education:
Larger model (**teacher**) teaches the smaller model (**student**)

Train the student to match the teacher's perf. via a **distillation loss**



Benefits

- Train a smaller model (of same/different arch.) with fewer parameters
- Smaller memory footprint
  - Number of AI accelerators to host the model goes down
- Speeds up prefill
  - fewer flops during prefill
- Speeds up decoding
  - reduces the memory I/O for loading parameters and intermediate states

How to match a student to a teacher?

- Output logits (Hinton et al, NeurIPS DL Workshop 2014)
  - Compute probabilities $\left[p^{(1)}, \ldots, p^{(v)}\right]$ from logits $[z^{(1)}, \ldots, z^{(v)}]$ using softmax (for teacher & student) with v being the vocabulary size
  - Compute the cross-entropy loss
- Intermediate Weights (Romero et al., ICLR 2015)
  - Add an L_2 loss between teacher and student weights in addition to cross entropy loss
  - Apply linear transformation to match dimensionalities

Hinton et al., Distilling the Knowledge in a Neural Network, NeurIPS 2014
Romero et al., FitNets: Hints for Thin Deep Nets, ICLR 2015

How to match a student to a teacher?

- Intermediate Features (Huang and Wang, arXiv 2017)
  - Minimize the distance between intermediate activations of teacher/student
  - Maximum Mean Discrepancy, L_2 distance

Huang and Wang, Like What You Like: Knowledge Distill via Neuron Selectivity Transfer, arXiv 2017

How to match student to teacher?

- Intermediate Gradients (Zagoruyko & Komodakis, ICLR 2017)
  - L_2 distance between gradients $\nabla(Y)$ of intermediate activations Y_t and Y_s of teacher and student respectively
  - Compute probabilities $\left[p^{(1)}, \ldots, p^{(v)}\right]$ from logits $[z^{(1)}, \ldots, z^{(v)}]$ using Softmax (for teacher & student) with v being the vocabulary size
  - Compute the cross-entropy loss

- Sparsity Pattern, Relation between layers, …

Zagoruyko & Komadakis, Paying More Attention to Attention: Improving the Performance of Convolutional Neural Networks via Attention Transfer, ICLR 2017

Model distillation knowledge transfer set

Typical approach:

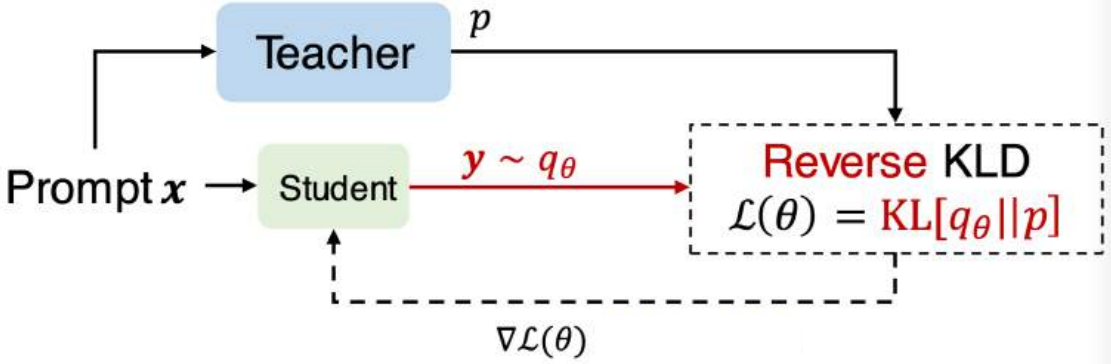
Use an external dataset also called transfer set

Another approach (Symbolic distillation):
Teacher generates synthetic data (West et al., ACL 2022)

West et al., Symbolic Knowledge Distillation: from General Language Models to Commonsense Models, ACL 2022

MiniLLM deep dive

- Minimizes reverse KL divergence between student distribution q_θ & teacher distribution p



$$\theta = \arg\min_\theta \mathcal{L}(\theta) = \arg\min_\theta \mathrm{KL}[q_\theta||p]$$
$$= \arg\min_\theta \left[- \mathop{\mathbb{E}}_{\boldsymbol{x}\sim p_{\boldsymbol{x}}, \boldsymbol{y}\sim q_\theta} \log \frac{p(\boldsymbol{y}|\boldsymbol{x})}{q_\theta(\boldsymbol{y}|\boldsymbol{x})} \right]$$

Image Source: Gu et al., MiniLLM: Knowledge Distillation of Large Language Models, ICLR 2024

Model distillation takeaways

- **Expensive.** Both models must be loaded into AI accelerators
- **Train small or distill?** Unclear how many training steps are needed during distillation compared to training the small model from scratch
- Performance of student hinges on the transfer set
- Difficulties in optimization (Stanton et al., NeurIPS 2021)
  - Fidelity of the student to its teacher vs generalization ability of the student in predicting unseen data

Stanton et al., Does Knowledge Distillation Really Work?, NeurIPS 2021

Model pruning

Motivation

- Models are large and have many components (at different granularities: transformer – layers / layer – blocks / block – matrices or heads / matrix – rows or entries).
- Pruning (i.e., removing) some components makes the model smaller and, ideally, faster to execute.
- Typically comes with a loss in model quality.

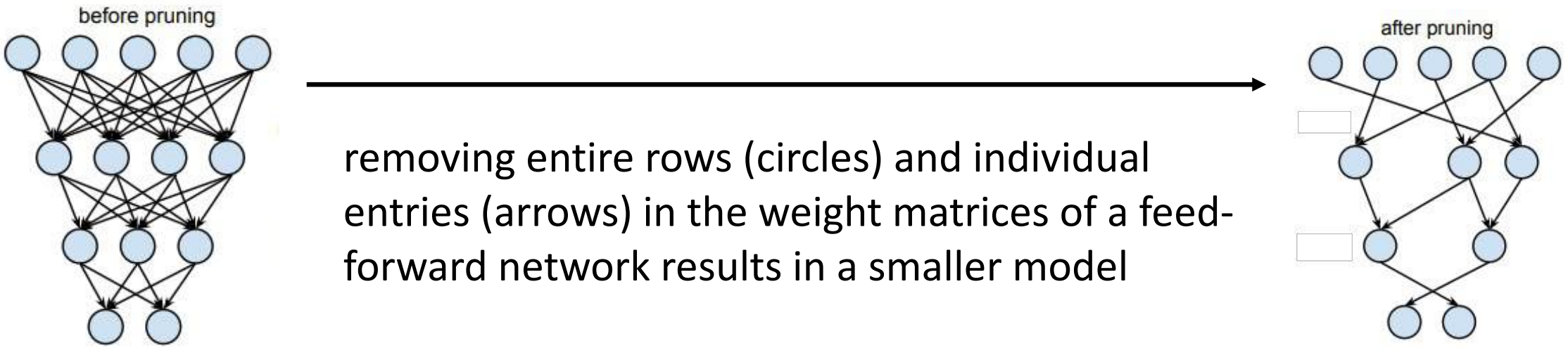


Image Source: *Learning both Weights and Connections for Efficient Neural Networks*, Han et al., Neurips 2015.

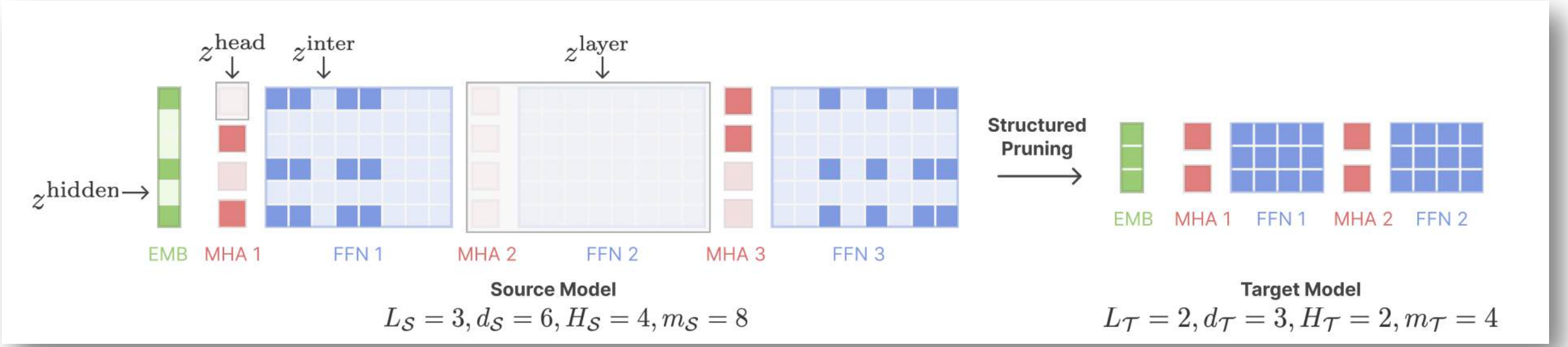
Structured Pruning vs Unstructured Pruning

- Structured Pruning
  - Removes whole components of the network.
  - When pruning larger structures like channels, blocks, or embedding dimensions, the speed-ups can easily be realized end-to-end.
  - SliceGPT (Ashkboos et al, ICLR 2024) / LLMPruner (Ma et al., NeurIPS 2023) / Sheared Llama (Xia et al., ICLR 2024) / LoRAPrune (Zhang & Yu et al., ACL 2024).

- Unstructured Pruning / Sparsity
  - Removes individual weights of the network.
  - Unstructured sparsity is mainly of academic interest since, so far, it does not lead to speed-up on hardware accelerators. But semi-structured sparsity does.
  - Magnitude Pruning (Janowsky, Physical Review1988) / Wanda (Sun et al., ICLR 2024)/ SparseGPT (Frantar & Alistarh, ICML 2023).

Sheared LLaMA: a method for structured pruning

- **GOAL:** remove entire components at different granularities (layers, hidden dimensions, attention heads, intermediate dimensions)
- **HOW:** specify target numbers for these (e.g., similar as in existing smaller model) and solve constrained optimization problem, minimizing the pre-training loss



Sheared LLaMA: Accelerating Language Model Pre-Training via Structured Pruning, Ma et al., Neurips 2023.

Sheared LLaMA: some details (continued)

- After pruning, pre-training is continued with the pruned model (0.4B tokens for pruning, 50B tokens for continued pre-training).

- Pre-training loss reduces at different rates across domains:
  - proposes *dynamic batch loading* where data points from different domains are sampled with different probabilities depending on the model's current loss on a domain (similar techniques have been proposed in the context of worst-class-error minimization or min-max fairness before).

Sheared LLaMA: results (downstream accuracy)

Model (#tokens for training)	Commonsense & Reading Comprehension					
	SciQ	PIQA	WinoGrande	ARC-E	ARC-C (25)	HellaSwag (10)
LLaMA2-7B (2T)†	93.7	78.1	69.3	76.4	53.0	78.6
OPT-1.3B (300B)†	84.3	71.7	**59.6**	57.0	29.7	54.5
Pythia-1.4B (300B)†	86.4	70.9	57.4	60.7	31.2	53.0
TinyLlama-1.1B (3T)†	**88.9**	73.3	58.8	55.3	30.1	60.3
Sheared-LLaMA-1.3B (50B)	87.3	**73.4**	57.9	**61.5**	**33.5**	**60.7**
OPT-2.7B (300B)†	85.8	73.7	60.8	60.8	34.0	61.5
Pythia-2.8B (300B)†	88.3	74.0	59.7	64.4	36.4	60.8
INCITE-Base-3B (800B)	90.7	74.6	63.5	**67.7**	40.2	64.8
Open-LLaMA-3B-v1 (1T)	91.3	73.7	61.5	67.6	39.6	62.6
Open-LLaMA-3B-v2 (1T)†	**91.8**	**76.2**	63.5	66.5	39.0	67.6
Sheared-LLaMA-2.7B (50B)	90.8	75.8	**64.2**	67.0	**41.2**	**70.8**

Model (#tokens for training)	Continued LM		World Knowledge			Average
	LogiQA	BoolQ (32)	LAMBADA	NQ (32)	MMLU (5)	
LLaMA2-7B (2T)†	30.7	82.1	28.8	73.9	46.6	64.6
OPT-1.3B (300B)†	26.9	57.5	58.0	6.9	24.7	48.2
Pythia-1.4B (300B)†	**27.3**	57.4	**61.6**	6.2	**25.7**	48.9
TinyLlama-1.1B (3T)†	26.3	60.9	58.8	**12.1**	25.5	50.0
Sheared-LLaMA-1.3B (50B)	26.9	**64.0**	61.0	9.6	**25.7**	**51.0**
OPT-2.7B (300B)†	26.0	63.4	63.6	10.1	25.9	51.4
Pythia-2.8B (300B)†	28.0	66.0	64.7	9.0	26.9	52.5
INCITE-Base-3B (800B)	27.7	65.9	65.3	14.9	**27.0**	54.7
Open-LLaMA-3B-v1 (1T)	28.4	70.0	65.4	**18.6**	**27.0**	55.1
Open-LLaMA-3B-v2 (1T)†	28.1	69.6	66.5	17.1	26.9	55.7
Sheared-LLaMA-2.7B (50B)	**28.9**	**73.7**	**68.4**	16.5	26.4	**56.7**

- Pruning ~60% of the parameters of LLaMA2-7B results in an average drop of ~8%.
- Pruned model outperforms smaller models of similar size, even when they have seen more tokens (3T for TinyLlama-1.1B vs 2T + 50B for Sheared-LLaMA-1.3B).

Sheared LLaMA: results (inference speed)

Model	Throughput	
	7B	
LLaMA-7B	37	
	1.3B	2.7B
LLM Pruner	41	40
Sheared-LLaMA	62	47
	50% sparsity	
Wanda (2:4)	-	42
Wanda (4:8)	-	42

- Pruning ~60% of the parameters of LLaMA2-7B results in a throughput increase of ~27%
- Wanda (to be discussed next) is reported to yield a throughput increase of ~14% at 50% pruning

Recent NVIDIA work also combines structured pruning and distillation
Compact Language Models via Pruning and Knowledge Distillation, Muralidharan et al., 2024

Sparsity: unstructured vs. structured

A matrix is *sparse* if most of its entries are zero.

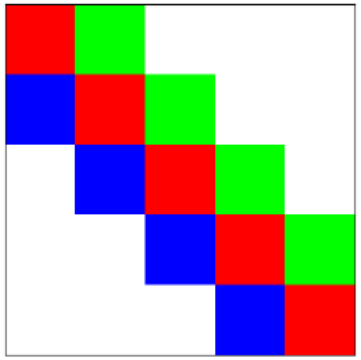
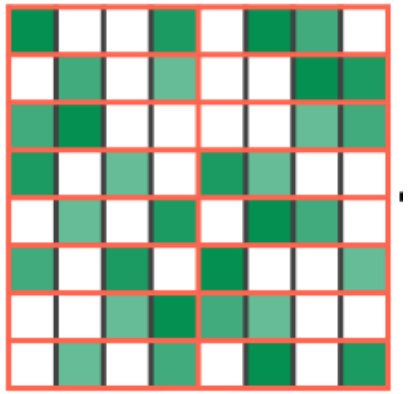
- Allows for reduced storage and should allow for faster operations (we don't need to store the zeros or use them when performing matmul etc.)
- Unstructured sparsity: zero entries can be located anywhere – speed-up is hard to realize on accelerators.
- Structured sparsity: zero entries must follow some pattern, e.g.,
  - band matrix 
  - n:m-sparsity 

Image Source: NVIDIA Blog

  - and many more; specialized kernels for fast matmul exist.

Semi-structured sparsity

- Keep *n* weights out of each block of size *m*.
- Can realize FLOPs reduction with specialized hardware: NVIDIA Ampere, upcoming TRN2/INF3.
- Example 2:4 at FP16 on NVIDIA Ampere:
  - Original: 4 weights each 16 bits ➔ 64 bits
  - Sparse: 2 weights each 16 bits and two indices of 2 bits ➔ 2*16+2*2 = 36 bits
  - Memory compression by factor 1.78x.
  - FLOPs reduction by 2x.

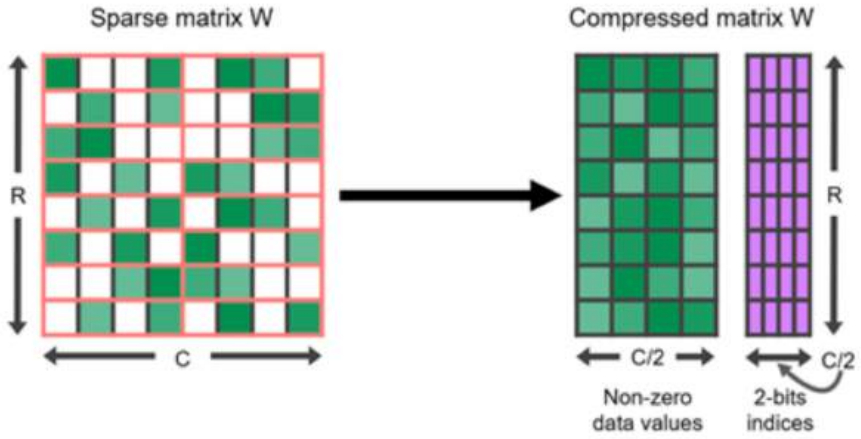
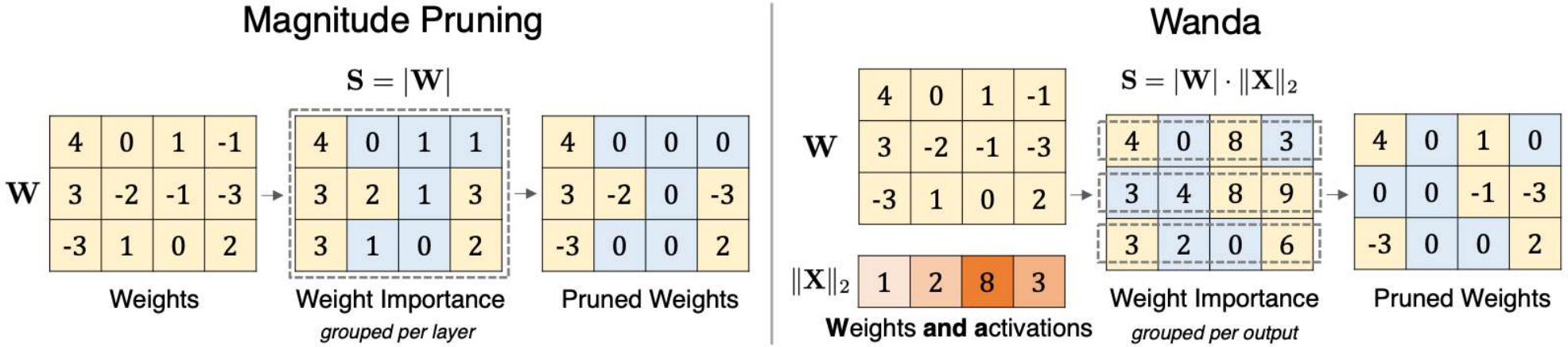


Image Source: NVIDIA Blog

Wanda: a method for both unstructured and structured sparsity

- Most simple pruning approach of magnitude pruning does not take activations into account, but a small subset of activation features can have very large magnitude (Dettmers et al., 2022).
- Wanda reweighs the weight magnitudes by the average squared activation magnitudes over a calibration dataset.



A Simple and Effective Pruning Approach for Large Language Models, Sun et al., ICLR 2024.

Wanda: some details

- Can be interpreted as a local variant of the Optimal Brain Damage method from LeCun et al. (1989)
- Straightforward to implement, only requires a single forward pass to compute norm of activations

Algorithm 1 PyTorch code for Wanda

```
# W: weight matrix (C_out, C_in);
# X: input matrix (N * L, C_in);
# s: desired sparsity, between 0 and 1;

def prune(W, X, s):
  metric = W.abs() * X.norm(p=2, dim=0)

  _, sorted_idx = torch.sort(metric, dim=1)
  pruned_idx = sorted_idx[:,:int(C_in * s)]
  W.scatter_(dim=1, index=pruned_idx, src=0)

  return W
```

- Can be readily extended to arbitrary *comparison groups*, e.g., n:m sparsity

Wanda: results

Mean zero-shot accuracies (%) of pruned LLaMA and LLaMA-2 models:

			LLaMA				LLaMA-2		
Method	Weight Update	Sparsity	7B	13B	30B	65B	7B	13B	70B
Dense	-	0%	59.99	62.59	65.38	66.97	59.71	63.03	67.08
Magnitude	✗	50%	46.94	47.61	53.83	62.74	51.14	52.85	60.93
SparseGPT	✓	50%	**54.94**	58.61	63.09	66.30	**56.24**	60.72	**67.28**
Wanda	✗	50%	54.21	**59.33**	**63.60**	**66.67**	**56.24**	**60.83**	67.03
Magnitude	✗	4:8	46.03	50.53	53.53	62.17	50.64	52.81	60.28
SparseGPT	✓	4:8	**52.80**	55.99	60.79	64.87	**53.80**	**59.15**	65.84
Wanda	✗	4:8	52.76	**56.09**	**61.00**	**64.97**	52.49	58.75	**66.06**
Magnitude	✗	2:4	44.73	48.00	53.16	61.28	45.58	49.89	59.95
SparseGPT	✓	2:4	**50.60**	**53.22**	58.91	62.57	**50.94**	54.86	63.89
Wanda	✗	2:4	48.53	52.30	**59.21**	**62.84**	48.75	**55.03**	**64.14**

Wanda: results with fine-tuning

Fine-tuning with pre-training objective can reduce the accuracy drop

on LLaMA-7B:

Evaluation	Dense	Fine-tuning	50%	4:8	2:4
Zero-Shot	59.99	✗	54.21	52.76	48.53
		LoRA	56.53	54.87	54.46
		Full	**58.15**	**56.65**	**56.19**
Perplexity	5.68	✗	7.26	8.57	11.53
		LoRA	6.84	7.29	8.24
		Full	**5.98**	**6.63**	**7.02**

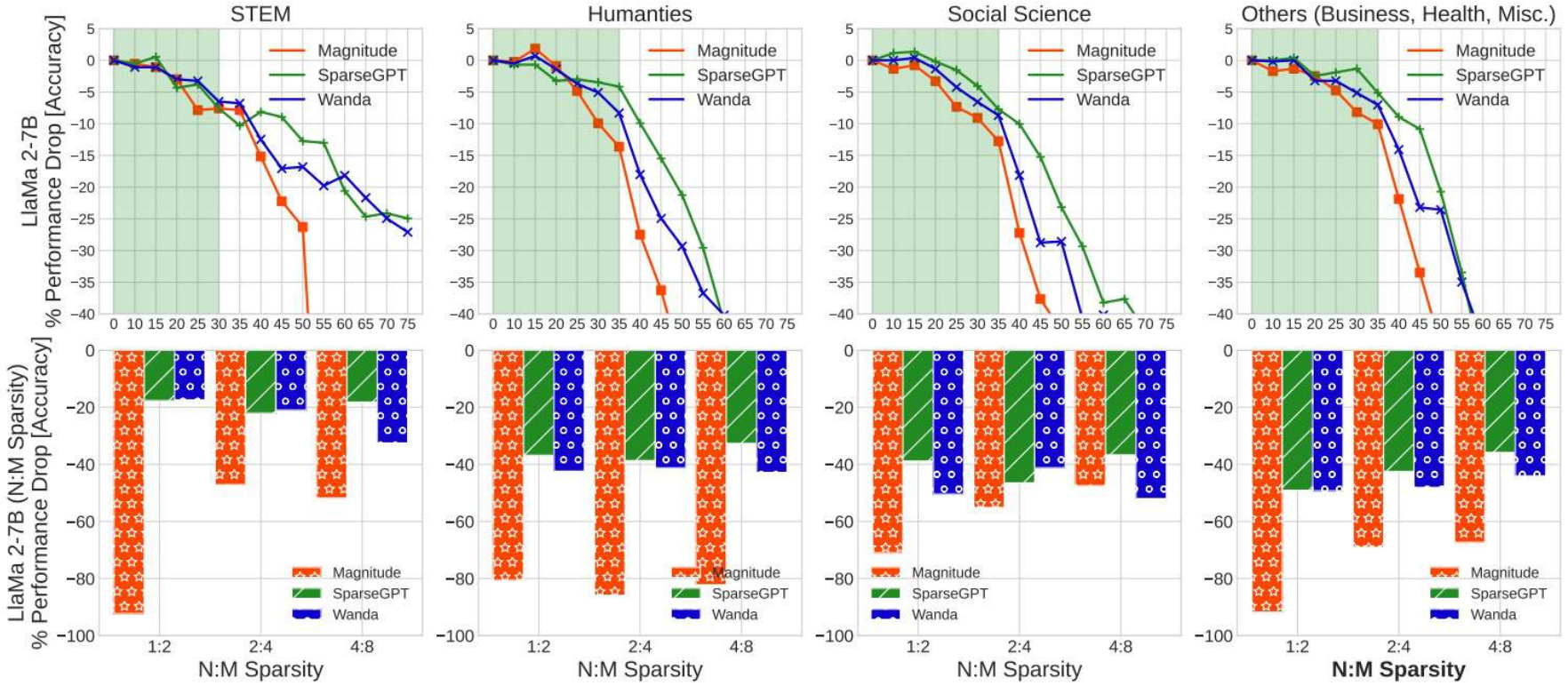
Wanda: speed-up

- While speed-up from structured pruning is usually straightforward to realize, realizing speed-up from sparsity is challenging and requires specialized tools.
- Structured 2:4-sparsity is supported in Pytorch>=2.1 on Nvidia GPUs with Compute Capability 8.0+.
- People report mixed results; Wanda-paper reports speed-up:
  - end-to-end latency speedup of 1.24× on LLaMA-7B.
  - speed-ups on matrix level.

LLaMA Layer	Dense	2:4	Speedup
q/k/v/o_proj	3.49	2.14	1.63×
up/gate_proj	9.82	6.10	1.61×
down_proj	9.92	6.45	1.54×

How well does pruning really work (w/o fine-tuning)?

- Results reported in Wanda paper (and others) look rather promising; e.g., <3.5% drop in average accuracy with 50% unstructured sparsity on LLaMA-2-7B
- Turns out the drop can be much higher on knowledge-intensive tasks



Compressing LLMS: The Truth is Rarely Pure and Never Simple, Jaiswal et al., ICLR 2024.

Open questions

- When does pruning work and what can we hope for (which properties should the model have, which downstream tasks should we care about, which pruning method should we use, etc.)?
- Is a pruned model (without fine-tuning) better or worse than a smaller dense model with similar parameter count? – literature answers this question inconsistently.
- How can we realize the theoretical speed-up of (unstructured) pruning on modern hardware?
- What is the interplay between pruning and other inference optimization techniques (e.g., quantization, speculative decoding)?

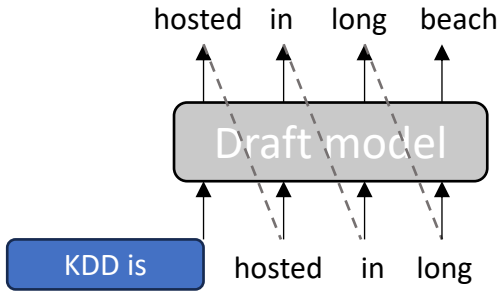
Model compression takeaways

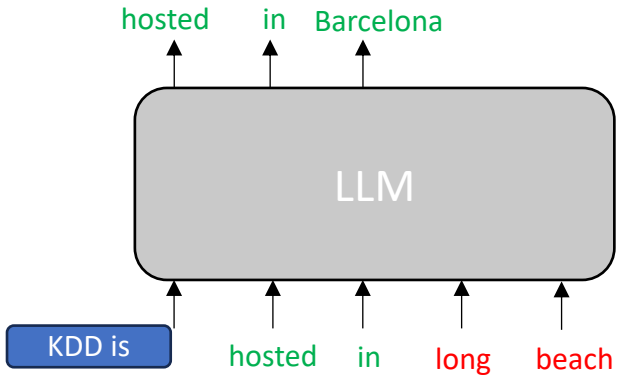
- Quantization stipulates hardware support and efficient kernels, but typically leads to lower accuracy drop compared to pruning (Kuzmin et al., NeurIPS 2023)
- Pruning may need efficient hardware support and kernels (e.g., N:M sparsity)
- Higher compression ratio leads to larger accuracy drop
- Finetuning helps to recover accuracy drop for pruning/quantization on specific tasks, but does not recover for all tasks
- Pruning and quantization can be combined in theory, but hasn't been explored in-depth for LLMs in the literature as much (Kuzmin et al., NeurIPS 2023)

Image Source: Ma & Wang et al., arXiv 2024

Speculative Decoding

- LLM Decoding is autoregressive, and memory bounded
- Speculative Decoding (Leviathan et. al, 2023) mitigates the memory bound
  - Draft tokens from a smaller model: $x \sim q(x)$
  - Verify (in parallel) the drafted tokens using the LLM: $p(x)$
  - Example
    - Draft 4 tokens, given the prompt
    - Input the prompt + draft tokens to LLM, reject when the LLM indicates a different generation





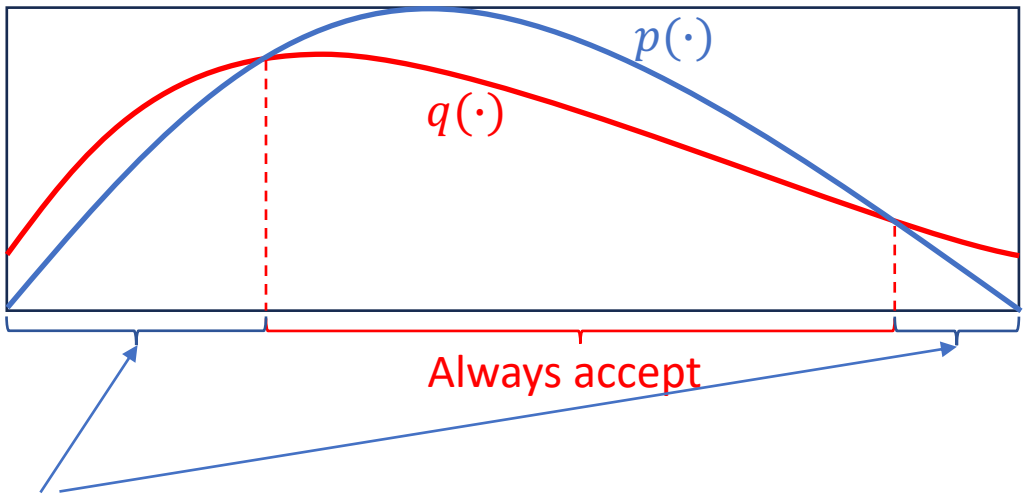
How to verify

- Greedy: Reject if $\arg\max p(x)$ does not equal the drafted
  - e.g., previous example: $\arg\max p(x) =$ “Barcelona” $\neq$ “long”
  - so, reject from “long” and afterwards
- More generally, rejection sampling based
  - draw $x \sim q(x)$
  - Accept x w.p. $\min\left\{\frac{p(x)}{q(x)}, 1\right\}$
    - So always accept when $q(x) < p(x)$
  - Reject x w.p. $1 - \min\left\{\frac{p(x)}{q(x)}, 1\right\}$ and redraw

$$x \sim \text{Normalize}(\max\{p(x) - q(x), 0\})$$

Rejection Sampling based verification

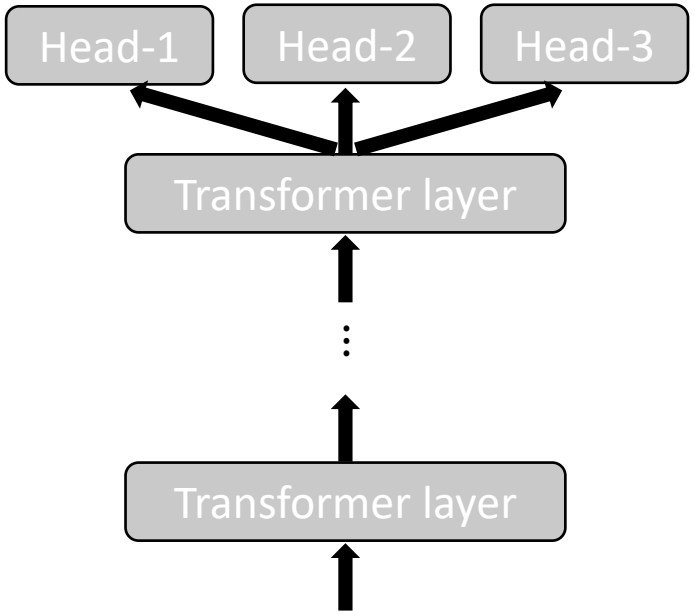
- An illustration



- Guaranteed to be equivalent to sampling from $p(\cdot)$!

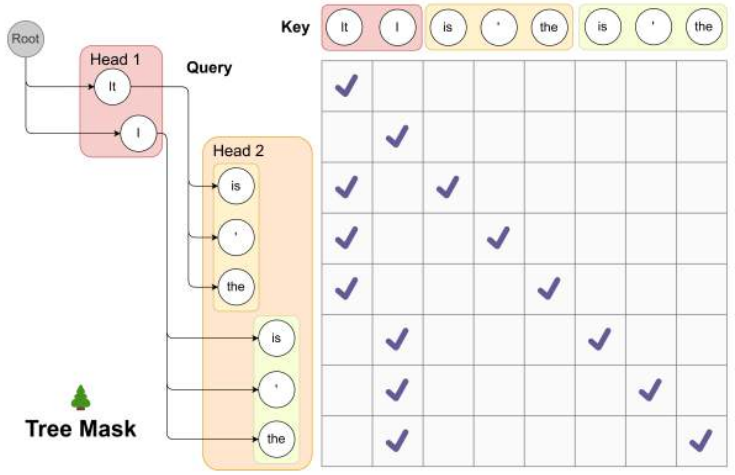
Choice of Draft Model

- Speed up depends on:
  - Closeness between $q(\cdot)$ and $p(\cdot)$
  - Speed ratio between the draft model and the LLM
- Up to 2-3x speedup reported
- How to choose the Draft Model?
  - A smaller model in the same family as the LLM
    (e.g., Tiny-Llama for Llama2-70B)
  - Share backbone of LLM,
    but use light-weight head(s) to predict multiple next tokens
    (e.g., MEDUSA, Cai et. al, 2023, illustrated right)



Improve Acceptance Rate

- Make $q(\cdot)$ and $p(\cdot)$ closer
  - "Align" the draft model to the LLM via Knowledge Distillation
  - Various distillation losses (e.g., DistillSpec, Zhou et. al, 2023)
- Tree-based drafting
  - Multiple draft token per time step
  - Proceed as long as one of them is accepted



Tree-based drafting adopted in Medusa

Image Source: *MEDUSA: Simple LLM Inference Acceleration Framework with Multiple Decoding Heads,* Cai et al. ICML'24

Optimal Draft Length

- Too short draft length may not fully exploit the power of the draft model
- Draft sequence can be longer if
  - $q(\cdot)$ and $p(\cdot)$ are close
  - Draft model is much faster than the LLM
- The closeness between $q(\cdot)$ and $p(\cdot)$ varies,
  - depending on the prefix
  - Adaptive draft length (see, right)

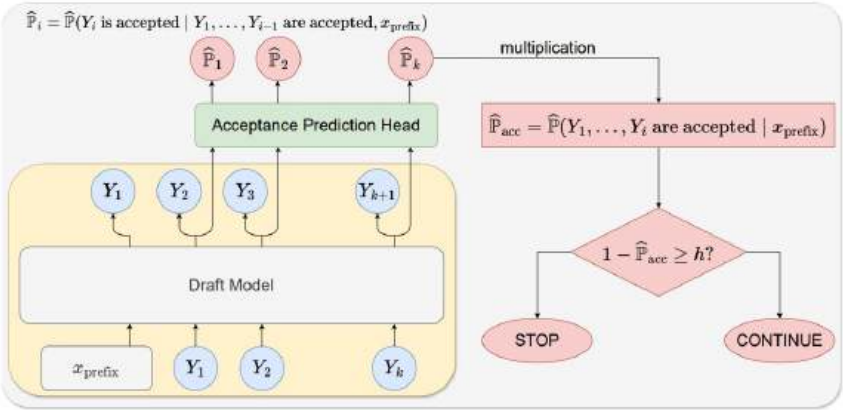


Image Source: *SpecDec++: Boosting Speculative Decoding via Adaptive Candidate Lengths,* Huang et al., 2024

Speculative Decoding Takeaways

- Speculative Decoding is lossless!
- Reduces # of target model forward calls and increases arithmetic int.
- Supported for example on Neuron devices (transformers-neuronx) or GPU (e.g. vllm).
- Various methods to generate drafts -- differ along:
  - Prerequisites: Needs to be trained or not.
  - Overhead: additional memory and latency for draft generation.
  - Quality: Alignment of the drafts with target model.
- Choice of method depends also on the model and application.

Case Studies

Case Study: Speculative Decoding on TRN1

Case Study: Speculative Decoding on TRN1

- Adapted from aws-neuron-samples (github link)
- We will use Llama-2 7B as a draft model for Llama-2 70B
- Steps:
  - Compile draft model for Neuron.
  - Compile target model with speculative decoding enabled.
  - Use SpeculativeGenerator wrapper around draft and target model.
  - Generate.

Speculative Decoding on TRN1 – Compilation

[1]:
```python
import time
import torch
from transformers_neuronx.llama.model import LlamaForSampling

SPECULATIVE_LENGTH = 5

print("\nStarting to compile Draft Model....")
# Load draft model
draft_neuron_model = LlamaForSampling.from_pretrained('./sd_models/draft_model', n_positions=128, batch_size=1, tp_degree=32, amp='bf16')
# compile to neuron
draft_neuron_model.to_neuron()
print("\nCompleted compilation of Draft Model")

print("\nStarting to compile Target Model....")
# Load target model
target_neuron_model = LlamaForSampling.from_pretrained('./sd_models/target_model', n_positions=128, batch_size=1, tp_degree=32, amp='bf16')
# Enable speculative decoder
target_neuron_model.enable_speculative_decoder(SPECULATIVE_LENGTH)
# compile to neuron
target_neuron_model.to_neuron()
print("\nCompleted compilation of Target Model")
```

Speculative Decoding on TRN1 Generation with SD

```
prompt = "I am at KDD conference in Barcelona. After the conference tutorial today I will"
input_ids = tokenizer(prompt, return_tensors="pt").input_ids

# create SpeculativeGenerator
spec_gen = SpeculativeGenerator(draft_neuron_model, target_neuron_model, k = SPECULATIVE_LENGTH)

start_spec_timer = time.time()
response = spec_gen.sample(input_ids=input_ids, sequence_length=50)
end_spec_timer = time.time()
print(f"\nSpeculative sampling response generation took {end_spec_timer - start_spec_timer} s")
generated_text = tokenizer.decode(response[0])
print(f"\nDecoded tokens: {generated_text}")
```

```
Speculative sampling response generation took 0.5017242431640625 s

Decoded tokens: <s> I am at KDD conference in Barcelona. After the conference tutorial today I will be giving a talk on my work on the topi
c of "Data Mining Patterns" in the Data Mining and Machine Learning track.
The talk is based
```

Speculative Decoding on TRN1
Generation without SD

```
start_auto_r_timer = time.time()
autor_response = target_neuron_model.sample(input_ids=input_ids, sequence_length=50)
end_auto_r_timer = time.time()
generated_text = tokenizer.decode(autor_response[0])
print(f"\nAutoregressive sampling response generation took {end_auto_r_timer - start_auto_r_timer} s")
print(f"\nDecoded tokens: {generated_text}")
```

```
Autoregressive sampling response generation took 0.7435760498046875 s

Decoded tokens: <s> I am at KDD conference in Barcelona. After the conference tutorial today I will attend the second meeting of the new re
search sub-committee on Knowledge Discovery from Aggregate Data. The first meeting was in Rome.
The meeting
```

Speculative Decoding on TRN1 Generation with k=50

[9]:
```
prompt = "I am at KDD conference in Barcelona. After the conference tutorial today I will"
input_ids = tokenizer(prompt, return_tensors="pt").input_ids

# create SpeculativeGenerator
spec_gen = SpeculativeGenerator(draft_neuron_model, target_neuron_model, k = SPECULATIVE_LENGTH)

start_spec_timer = time.time()
response = spec_gen.sample(input_ids=input_ids, sequence_length=50)
end_spec_timer = time.time()
print(f"\nSpeculative sampling response generation took {end_spec_timer - start_spec_timer} s")
generated_text = tokenizer.decode(response[0])
print(f"\nDecoded tokens: {generated_text}")
```

```
Speculative sampling response generation took 0.8887739181518555 s

Decoded tokens: <s> I am at KDD conference in Barcelona. After the conference tutorial today I will be giving a talk on "Mining Social Netw
orks for Recommendation" at 11:30am.
I will be talking about the
```

- Setting the draft speculation length too large leads to a slowdown!

Speculative Decoding on TRN1 -- Takeaways

- Speculative Decoding gives clear speedup when draft length is set well.
- The speedup varies by prompt and also per generation.
- Hosting the draft model consumes additional memory.
- Although the distribution is guaranteed to be the same, the individual sampled generations can be different.

- The generation is provably lossless!

Case Study: Int8 on TRN1

Case Study: Int8 on TRN1

- TRN1 supports INT8 weight-only quantization through transformers-neruonx.
- We will quantize Open-Llama-7B from BF16 to Int8.
- Steps:
  - Compile draft model with quantization and weight tiling for Neuron.
  - Load Model.
  - Generate.

Case Study: Int8 on TRN1 - BF16 Baseline

[1]:
```
import torch
from transformers import AutoTokenizer
from transformers_neuronx import LlamaForSampling, NeuronConfig, QuantizationConfig
```

[2]:
```
INT8 = False

if INT8:
    # Set the weight storage config use int8 quantization and bf16 dequantization
    neuron_config = NeuronConfig(
        quant=QuantizationConfig(quant_dtype='s8', dequant_dtype='bf16'),
        weight_tiling=True
    )
else:
    # Set the weight storage config use int8 quantization and bf16 dequantization
    neuron_config = NeuronConfig(weight_tiling=True)
```

[3]:
```
# Create the Neuron model
model = LlamaForSampling.from_pretrained(
    'openlm-research/open_llama_7b_v2',
    amp='bf16', # NOTE: When using quantization, amp type must match dequant type
    neuron_config=neuron_config
)
```

Case Study: Int8 on TRN1 - BF16 Baseline

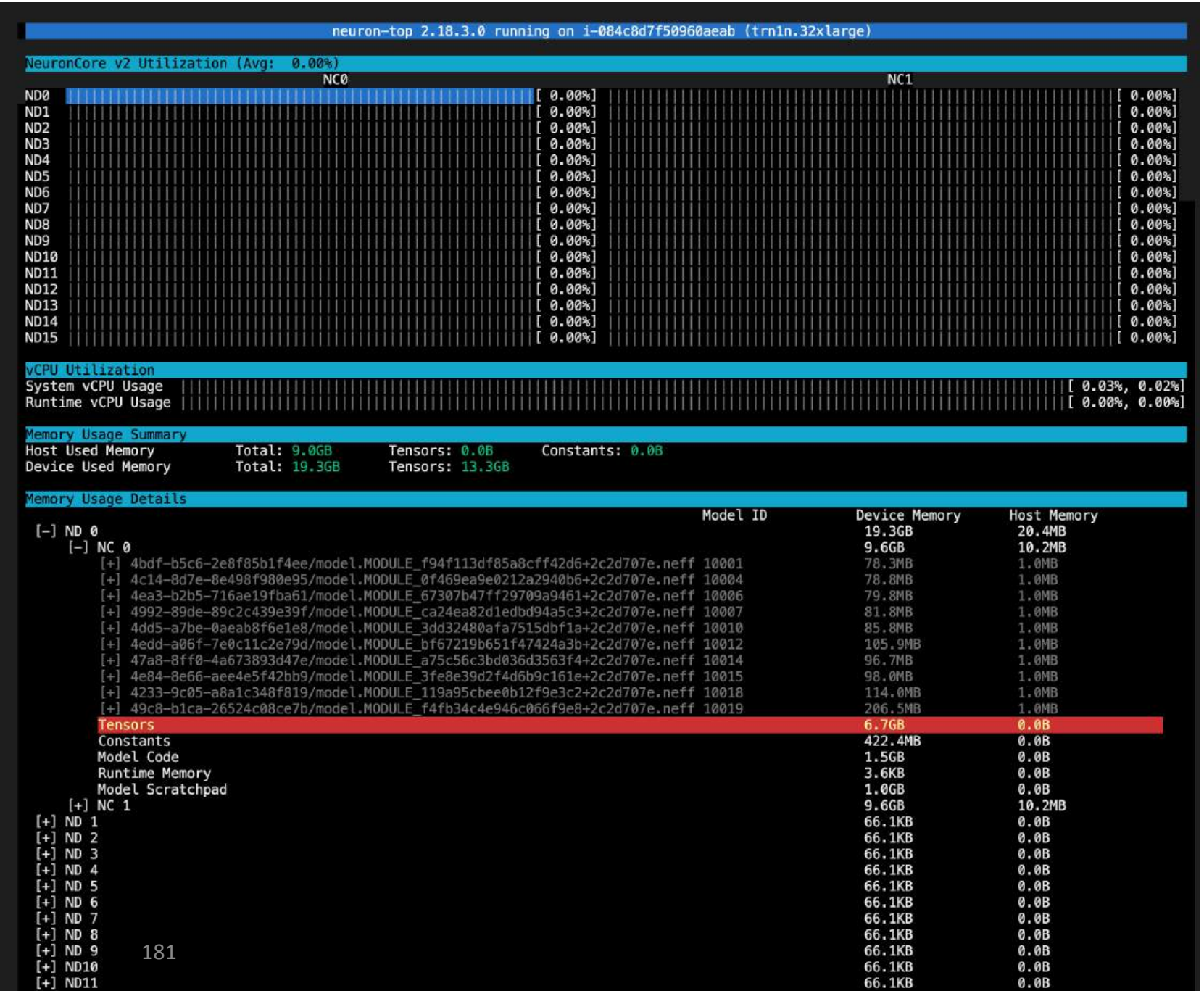
[4]:
```
#### when running for the first time we need to compile
# model.to_neuron()
# model.save('./compiled_models/open_llama_7b_v2_int8_tiled')
# model.save('./compiled_models/open_llama_7b_v2_bf16_tiled')


#### afterwards we can directly load the compiled checkpoints
if INT8:
    model.load('./compiled_models/open_llama_7b_v2_int8_tiled')
else:
    model.load('./compiled_models/open_llama_7b_v2_bf16_tiled')

model.to_neuron()
```

Case Study: Int8 on TRN1 - BF16 Baseline

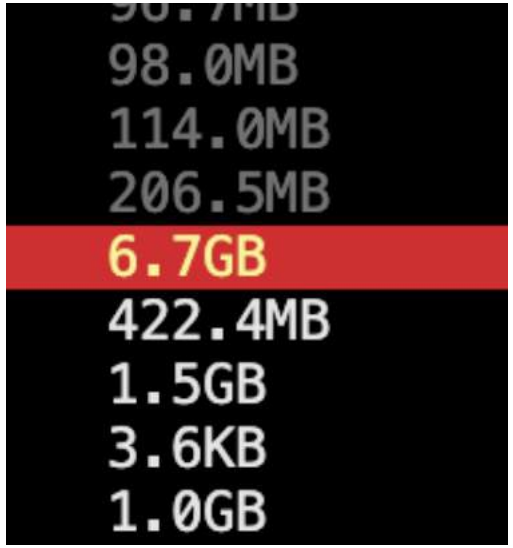
[7]:
```
# Get a tokenizer and example input
tokenizer = AutoTokenizer.from_pretrained('openlm-research/open_llama_7b_v2')
text = "I am at KDD conference in Barcelona. After the conference tutorial today I will"
encoded_input = tokenizer(text, return_tensors='pt')
```

[8]:
```
import time
# Run inference
start = time.time()
with torch.inference_mode():
    generated_sequence = model.sample(encoded_input.input_ids, sequence_length=128, start_ids=None, top_k=1)
    print(len(generated_sequence[0]))
print([tokenizer.decode(tok) for tok in generated_sequence])

print(f"\n \n Generation took {time.time()- start} seconds.")
```

```
128
['<s> I am at KDD conference in Barcelona. After the conference tutorial today I will be giving a talk on the topic of “Data Mining in the Cl
oud”.\nThe talk will be on the 11th of August at 16:00 in room 1.\nThe abstract is as follows:\nData mining is a process of discovering patte
rns in large data sets. The data sets are often very large and distributed over many machines. The data mining process is often computational
ly intensive and requires a lot of memory. In this talk we will discuss the challenges of data mining in the cloud. We will discuss the probl
ems of']


 Generation took 3.1939985752105713 seconds.
```



BF16 model uses 2x6.7GB of memory for the weight tensors



Case Study: Int8 on TRN1 – Int8

[1]:
```
import torch
from transformers import AutoTokenizer
from transformers_neuronx import LlamaForSampling, NeuronConfig, QuantizationConfig
```

[2]:
```
INT8 = True

if INT8:
    # Set the weight storage config use int8 quantization and bf16 dequantization
    neuron_config = NeuronConfig(
        quant=QuantizationConfig(quant_dtype='s8', dequant_dtype='bf16'),
        weight_tiling=True
    )
else:
    # Set the weight storage config use int8 quantization and bf16 dequantization
    neuron_config = NeuronConfig(weight_tiling=True)
```

[3]:
```
# Create the Neuron model
model = LlamaForSampling.from_pretrained(
    'openlm-research/open_llama_7b_v2',
    amp='bf16', # NOTE: When using quantization, amp type must match dequant type
    neuron_config=neuron_config
)
```

Case Study: Int8 on TRN1 – Int8

[4]:
```
#### when running for the first time we need to compile
# model.to_neuron()
# model.save('./compiled_models/open_llama_7b_v2_int8_tiled')
# model.save('./compiled_models/open_llama_7b_v2_bf16_tiled')


#### afterwards we can directly load the compiled checkpoints
if INT8:
    model.load('./compiled_models/open_llama_7b_v2_int8_tiled')
else:
    model.load('./compiled_models/open_llama_7b_v2_bf16_tiled')

model.to_neuron()
```

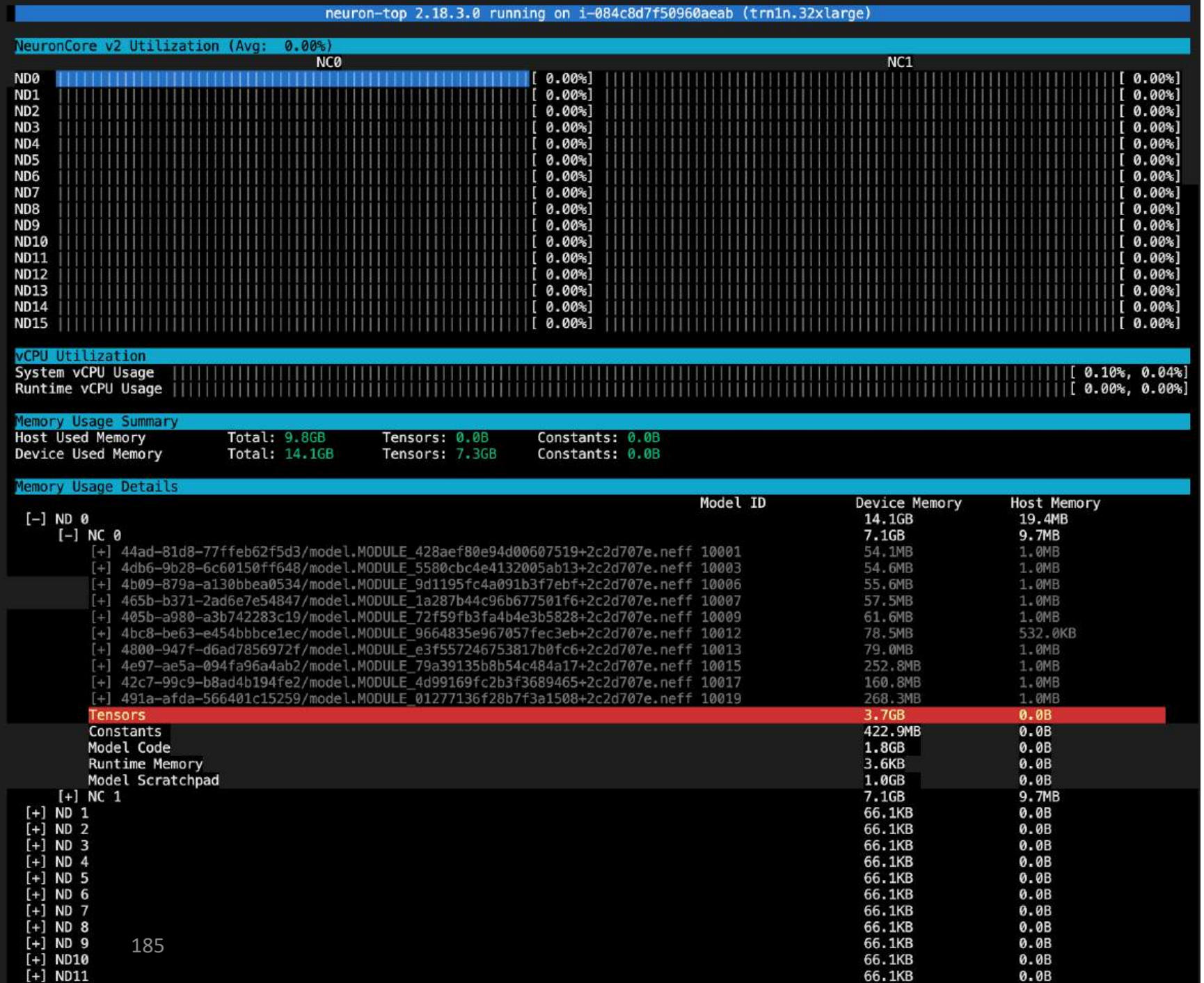
Case Study: Int8 on TRN1 – Int8

[5]:
```
# Get a tokenizer and example input
tokenizer = AutoTokenizer.from_pretrained('openlm-research/open_llama_7b_v2')
text = "I am at KDD conference in Barcelona. After the conference tutorial today I will"
encoded_input = tokenizer(text, return_tensors='pt')
```

```
You are using the default legacy behaviour of the <class 'transformers.models.llama.tokenization_llama.LlamaTokenizer'>. This is expected, and
```

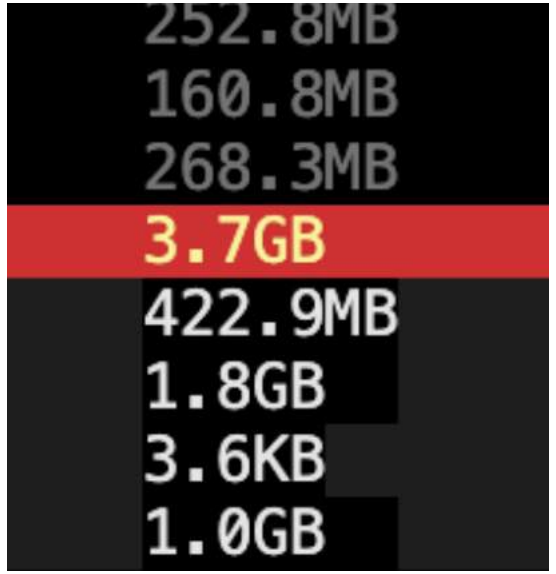
[7]:
```
import time
# Run inference
start = time.time()
with torch.inference_mode():
    generated_sequence = model.sample(encoded_input.input_ids, sequence_length=128, start_ids=None, top_k=1)
    print(len(generated_sequence[0]))
print([tokenizer.decode(tok) for tok in generated_sequence])

print(f"\n \n Generation took {time.time()- start} seconds.")
```

```
128
['<s> I am at KDD conference in Barcelona. After the conference tutorial today I will be giving a talk on the topic of “Data Mining for the W
eb of Data”.\nThe talk will be in the afternoon session of the conference.\nThe talk will be in the afternoon session of the conference.\nThe
talk will be in the afternoon session of the conference.\nThe talk will be in the afternoon session of the conference.\nThe talk will be in t
he afternoon session of the conference.\nThe talk will be in the afternoon session of the conference.\nThe talk will be in the afternoon sess
ion of the conference.\nThe talk']


 Generation took 2.094780683517456 seconds.
```



Int8 model uses 2x3.7GB of memory for the weight tensors

=> compression to 55%



amazon | science

Case Study: Int8 on TRN1 – Model output

Prompt: *I am at KDD conference in Barcelona. After the conference tutorial today I will*

- BF16:
  be giving a talk on the topic of "Data Mining in the Cloud".
  The talk will be on the 11th of August at 16:00 in room 1. […]
- Int8:
  be giving a talk on the topic of "Data Mining for the Web of Data".
  The talk will be in the afternoon session of the conference […]

- We observe a small deviation in the model output even with top_k=1.
  Deviation only happens for tokens where ambiguity is very large.

Case Study: Int8 on TRN1 -- Takeaways

- Weight compression of 1.8x.
- Quality of output is maintained.
- Generation speedup on example of 1.5x.
  - The weights are decompressed on the fly and the computation is done in BF16. Thus, we actually have *more* compute that in pure BF16.
  - When using weight tiling, however, we only need to load the compressed weights. Since decoding is memory-bound, this results in an overall speedup.

We’re hiring full-timers and interns!

- Foundation models, LLMs
- Efficient LLM training and inference, e.g., low-precision training, compression, pruning
- Distributed optimization
- System, High-performance Computing, Compiler

Locations: Santa Clara (US), Seattle (US), Tübingen (GER)

Reach out to Yida Wang (wangyida@amazon.com)