Algorithms and Systems for Efficient Inference in Generative AI

Rajarshi Saha, Aninda Manocha, Youngsuk Park

Neuron Science, AWS Annapurna Labs

Tuesday, **January 20, 2026** from **2:00pm to 6:00pm**

AAAI 2026, Singapore EXPO at Room: **Garnet 218**

Speakers

Youngsuk Park
Senior Applied Scientist
AWS Annapurna Labs

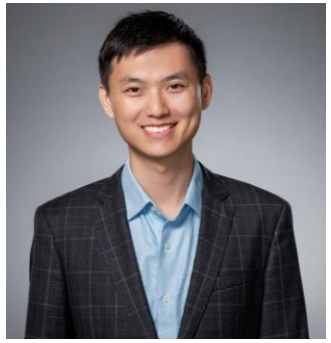
Rajarshi Saha
Applied Scientist
AWS Annapurna Labs

Aninda Manocha
Applied Scientist
AWS Annapurna Labs

This has been a team effort!

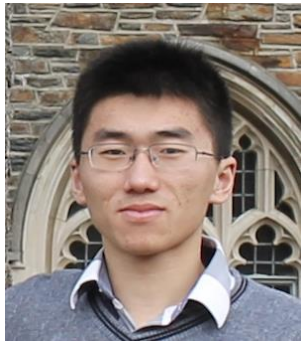
Lingfan Yu
Applied Scientist

Kaan Ozkara
Applied Scientist



Wei Tang
Applied Scientist

Tao Yu
Applied Scientist



Jiaji Huang
Senior Scientist

Liangfu Chen
Senior SDE

Jonas Kübler
Senior Scientist

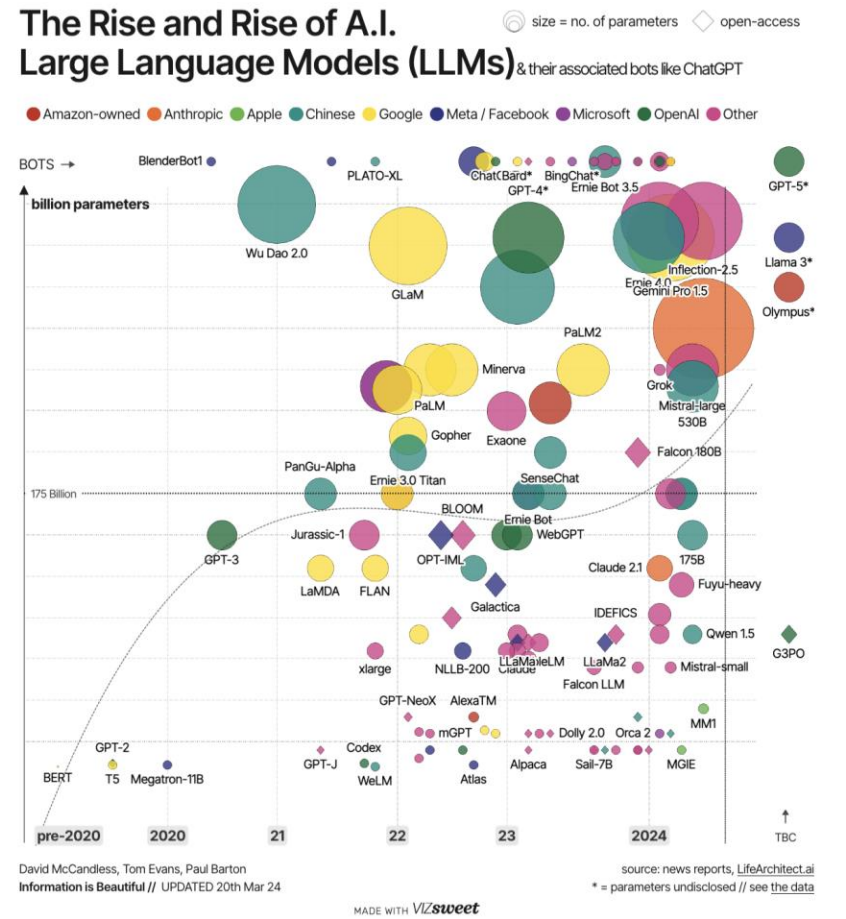
Yida Wang
Principal Scientist

George Karypis
Senior Principal Scientist

Tutorial Website

https://neuron-science.github.io/inference_optimization/

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.

informationisbeautiful.net, LifeArchitect.ai 2024

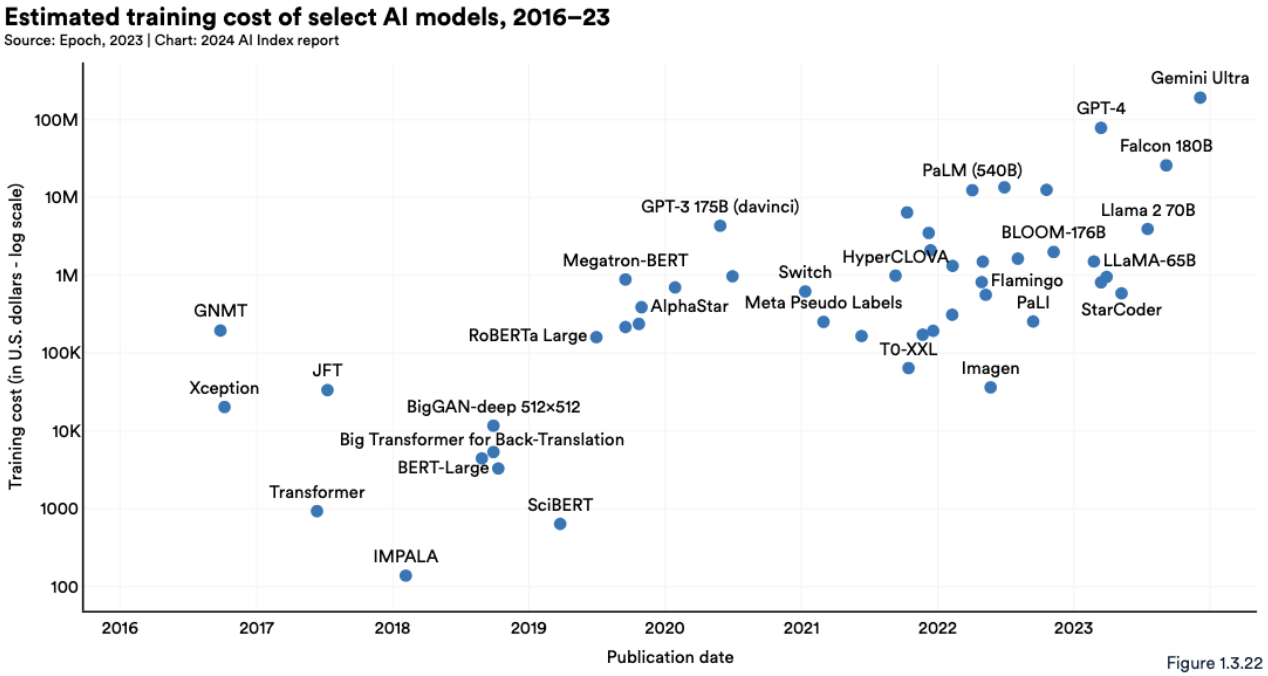


Figure 1.3.22

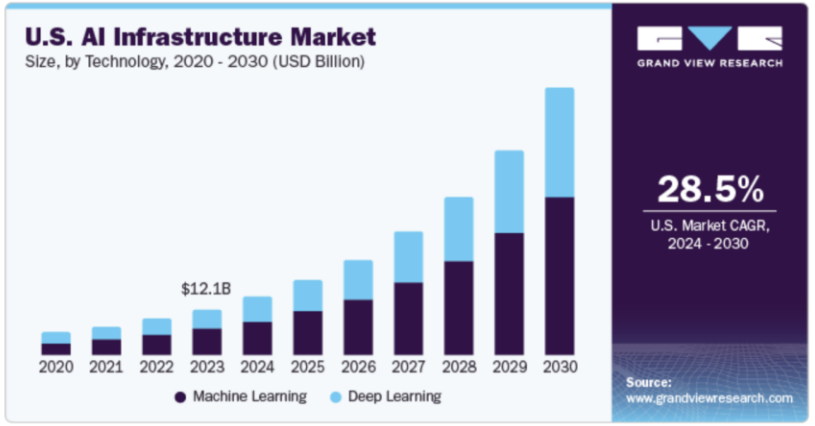
Stanford AI Report, 2024

Growing size and complexity of LLMs pose significant challenges for efficient inference

AI Hardware and Infra System

- LLM training and inference is a resource-intensive endeavor that demands robust hardware configurations
- AI is driving a $7 trillion race to scale data centers, reflecting massive investment in compute infrastructure to support LLM and AI demands.
- AI infrastructure market is projected to grow from USD 35.4 billion in 2023 to USD 223.5 billion by 2030, with a compound annual growth rate of 30.4%.

Hardware Requirements for LLM Training [Blog link]

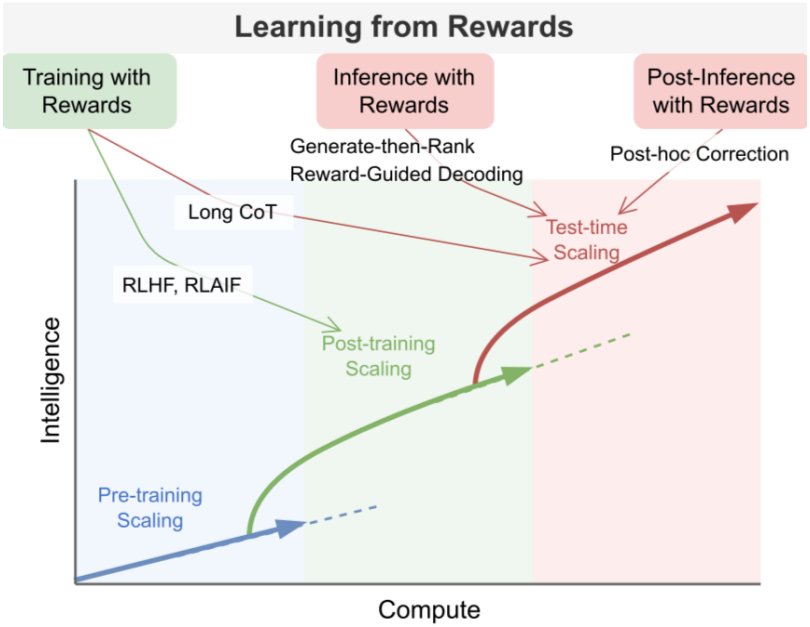


AI Infrastructure Market Size, Share & Trends Analysis Report

amazon | science

Why Inference Optimization?

- **Inference is ubiquitous**:
  - Training happens once (maybe once per month); inference happens millions to billions of times
  - Small inefficiencies compounds into massive cost and latency at scale (multiple concurrent requests)
- **Inference is the bottleneck in post-training**:
  - RLHF / RLAIF / preference optimization are inference-heavy
  - Each policy update requires large rollout and evaluation
  - Faster inference directly accelerates the post-training loop
- **Scaling laws are shifting.** Frontier gains come from:
  - Better decoding
  - Better post-training
  - Better inference-time compute allocation



Credits: Wu, Sailing AI by the stars, 2025

Inference optimization enables new capabilities: long-context models, real-time applications, on-device and edge deployment, and inference-time scaling methods

Tutorial Outline

- **Primer: Foundations of Generative Inference**
- Algorithmic and Modeling-Level Inference Optimizations
- Systems-Level Optimizations
- Open Source Frameworks and Tools

Primer: Foundations of Generative Inference

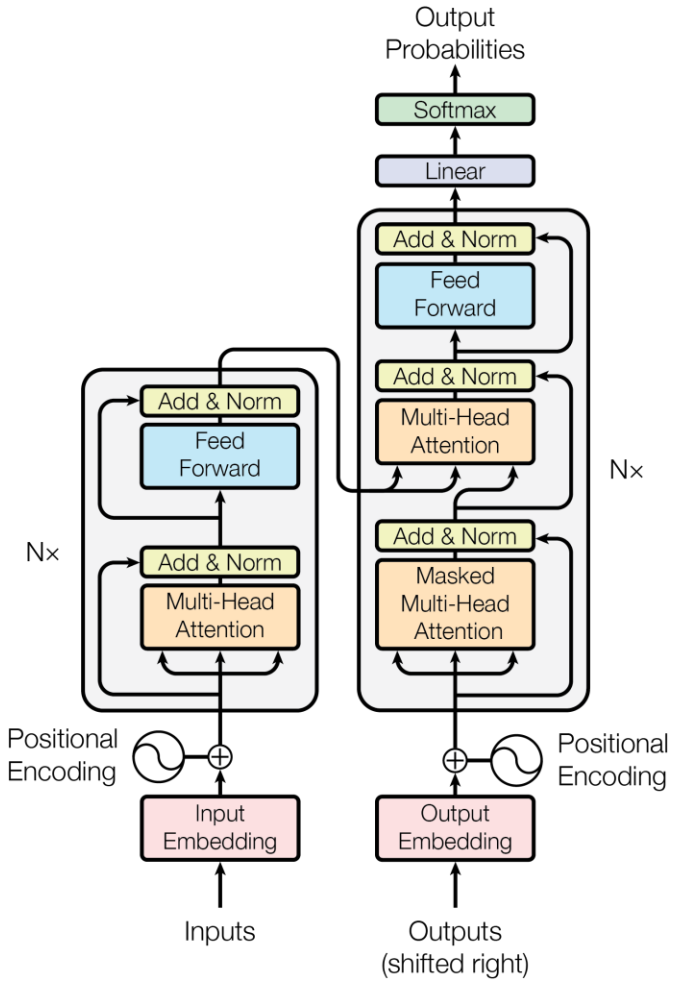
- Overview of LLM Architecture
- Overview of AI Hardware & System
- Overview of Inference Performance

Overview of LLM Architecture

- Decoder only models
- Self-attention, cross-attention
- MLP
- Vocabulary, tokenizers
- Token and Positional Embedding

Transformer Architecture

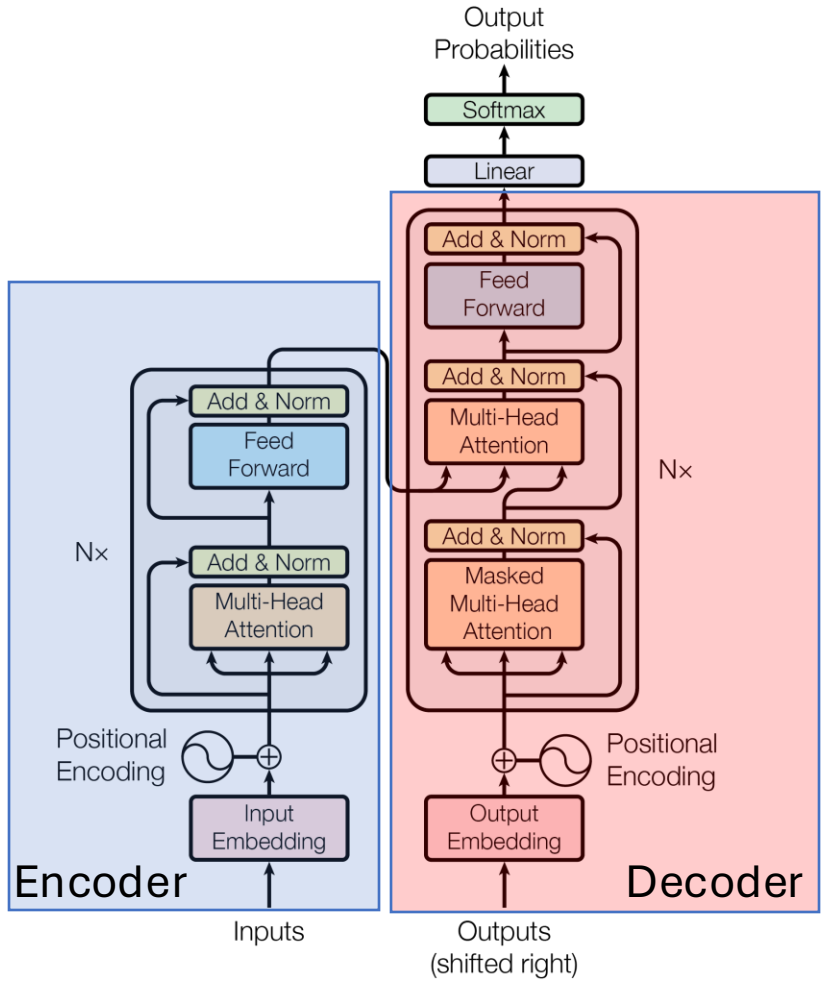
- Transformer models with attention layer is most popular in LLM
- Consists of model components
  - Positional encoding
  - Transformer layer
    - Residual network
    - Multi-headed attention
      - Self-attention
      - Cross-attention
    - LayerNorm
    - Feed Forward (MLP)
  - Softmax



[Vaswani et al., 2017] the Transformer architecture

Transformer Architecture

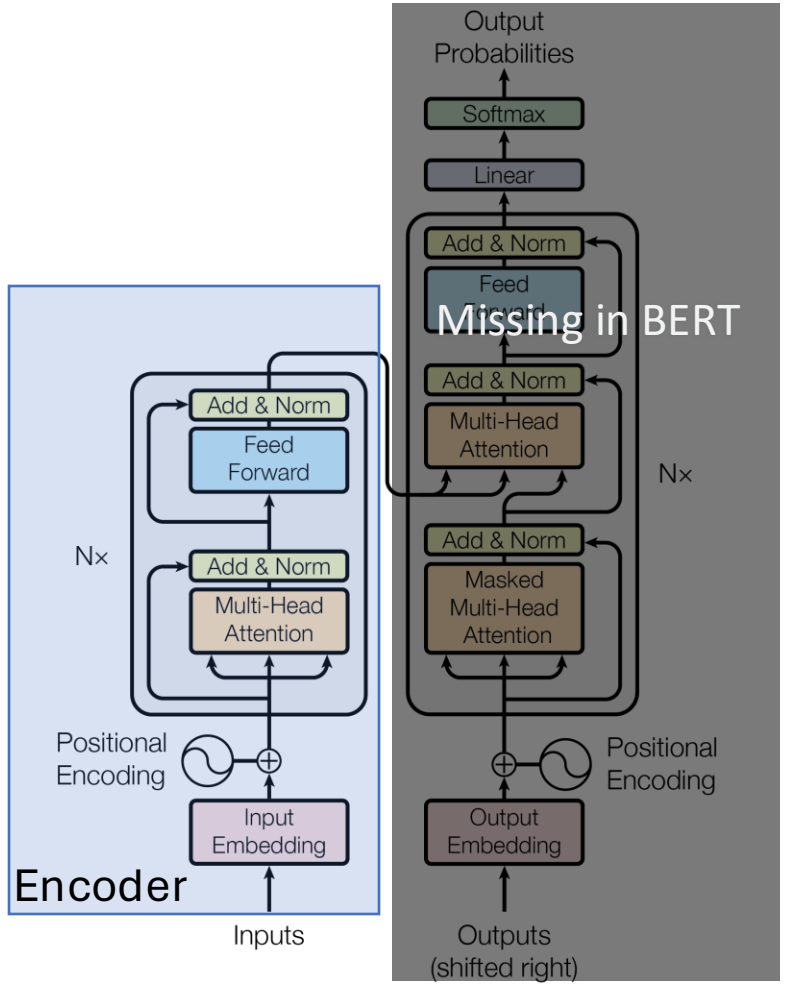
- LLM types are categorized based on encoder and decoder



[Vaswani et al., 2017] the Transformer architecture

Transformer Architecture

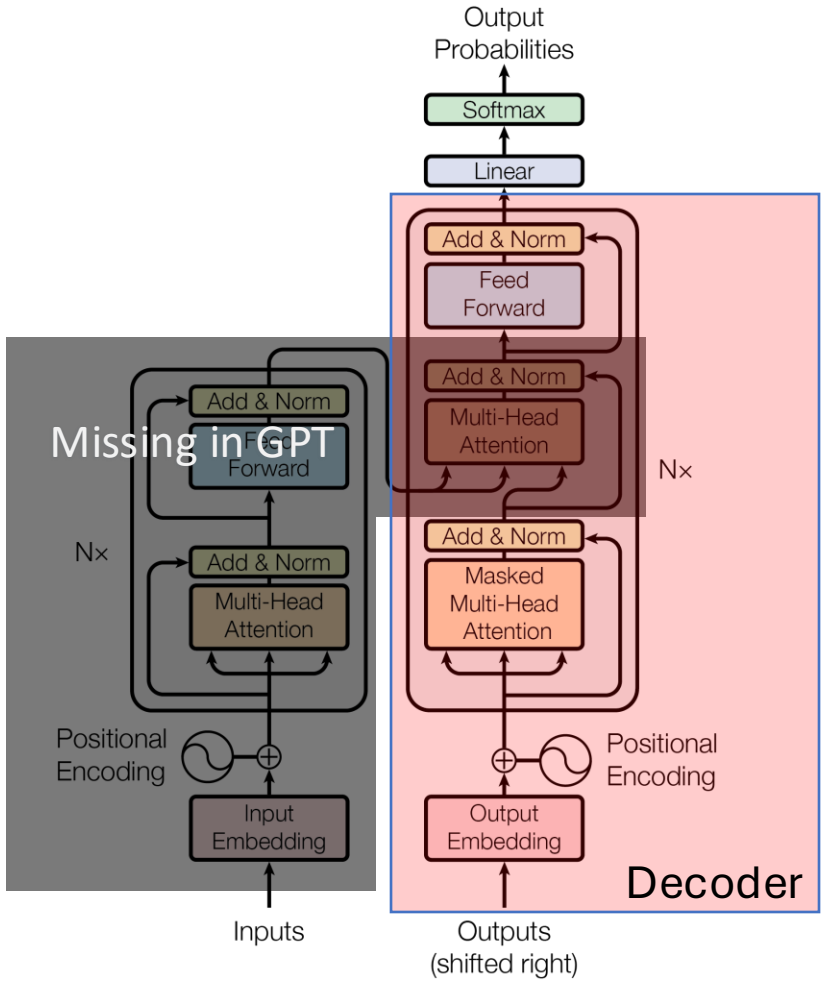
- LLM types are categorized based on encoder and decoder
  - Encoder only models
    - Input seq -> (contextualized hidden) embeddings
    - E.g., BERT [Devlin et al., 2019]



[Vaswani et al., 2017] the Transformer architecture

Transformer Architecture

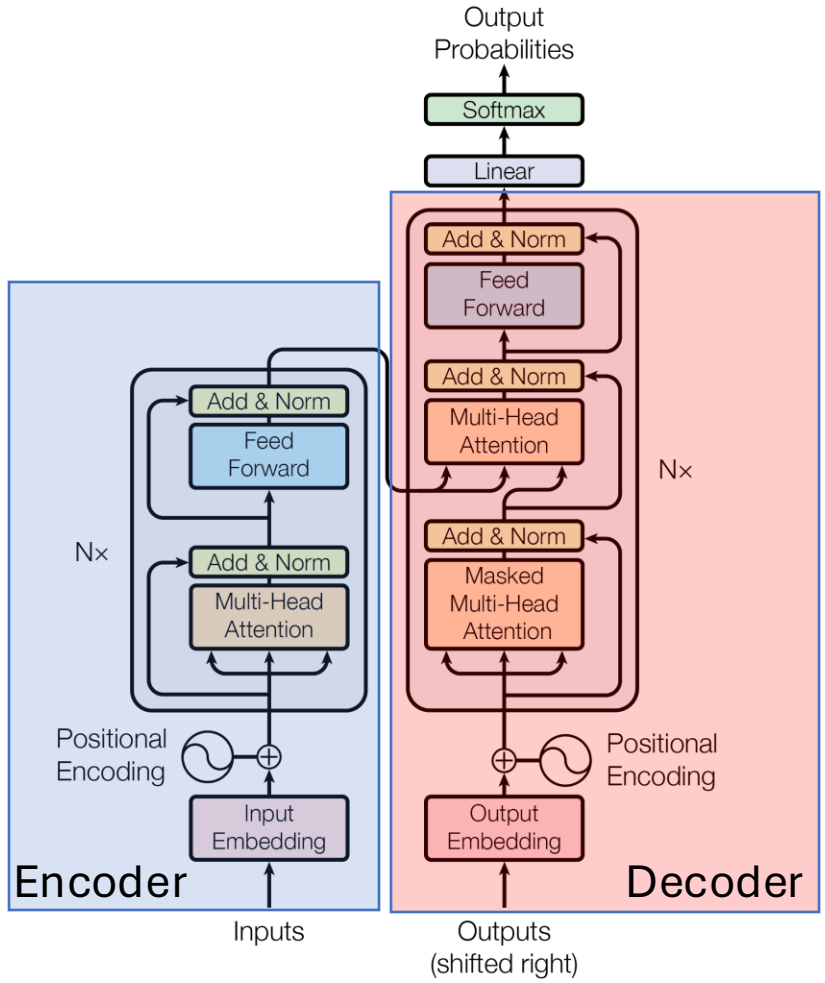
- LLM types are categorized based on encoder and decoder
  - Encoder only models
    - Input seq -> (contextualized hidden) embeddings
    - E.g., BERT [Devlin et al., 2019]
  - Decoding only models
    - Input seq -> next output prob
    - E.g., GPT [Brown et al., 2020], Llama [Touvron et al.,2023], Qwen3 [Yang et al, 2025]



[Vaswani et al., 2017] the Transformer architecture

Transformer Architecture

- LLM types are categorized based on encoder and decoder
  - Encoder only models
    - Input seq -> (contextualized hidden) 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], Qwen3 [Yang et al, 2025]
  - Encoder-Decoder models
    - Embeddings (context) + previous tokens -> next output prob
    - E.g., T5 [Colin Raffel et al., 2019]

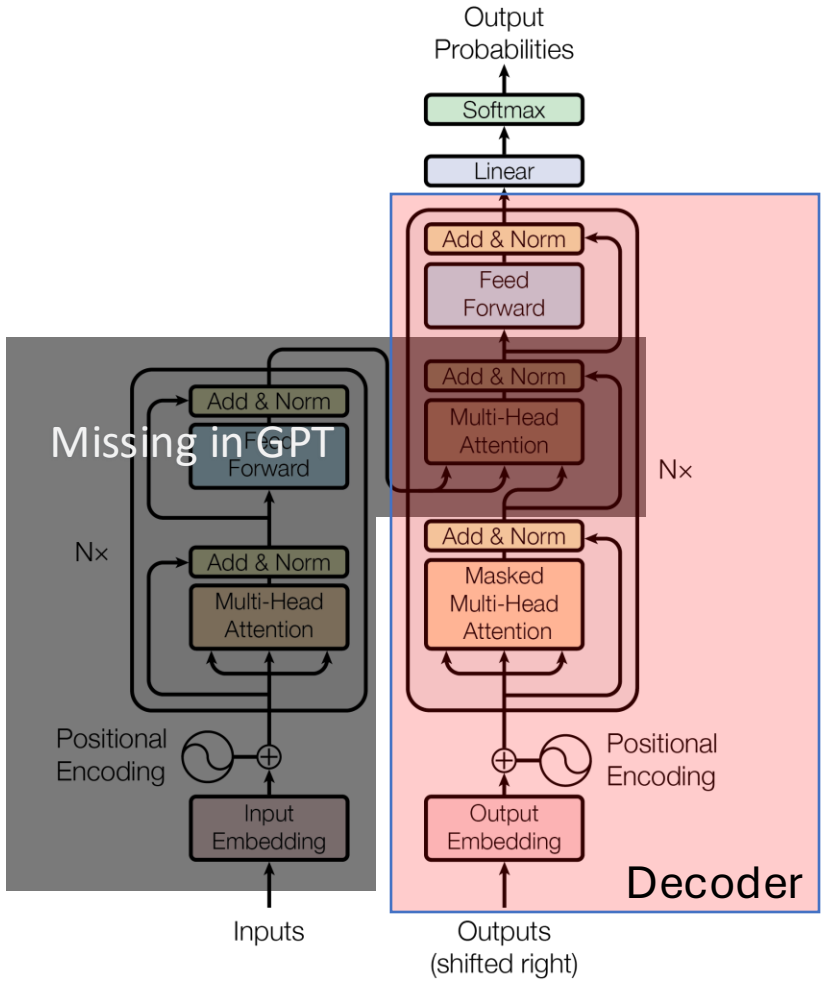


[Vaswani et al., 2017] the Transformer architecture

Transformer Architecture

- LLM types are categorized based on encoder and decoder
  - Encoder only models
    - Input seq -> (contextualized hidden) 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], Qwen3 [Yang et al, 2025]
  - Encoder-Decoder models
    - Embeddings (context) + previous tokens-> next output prob
    - E.g., T5 [Colin Raffel et al., 2019]

Tutorial mostly focused on Decoder-Only



[Vaswani et al., 2017] the Transformer architecture

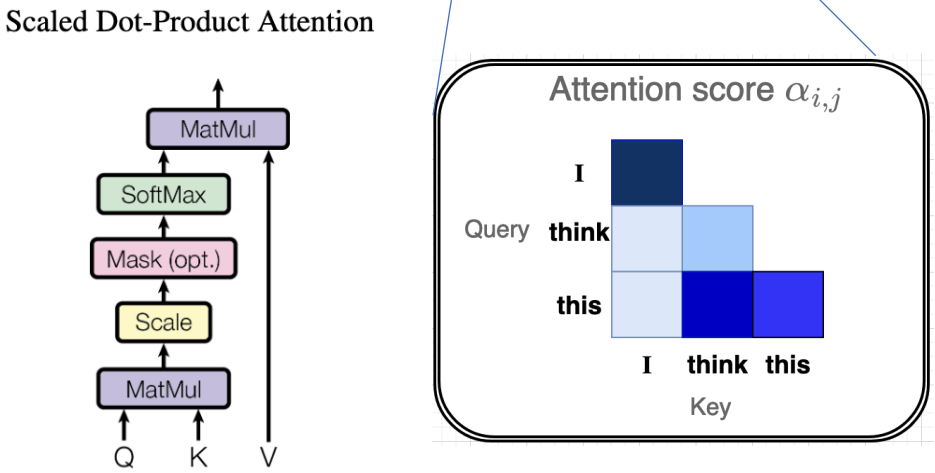
Self-Attention Mechanism

- Self-attention captures relationships within input seq $X = [x_1, \ldots, x_L] \in R^{L \times d}$ via computing attention score $\{\alpha_{i,j}\}$
  - that determine how much each token should weigh the others when forming contextual representations.

- In short, compute Q, K, V, and taking weighted sum of V from the attention score computed between Q and K

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

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



[Vaswani et al., 2017] the Transformer architecture

Self-Attention Mechanism

- Attention procedures for each input x_i
  - Step 1: Project each input token into query (q_i), key (k_i), and value (v_i) using learned linear transformations
  - Step 2: Compute causal attention scores $\{\alpha_{ij}\}_{j=1}^{i}$ where $w_{ij} = q_i^T k_j \in R, j \leq i$, ensuring the token attends only to previous (or current) positions.
  - Step 3: Generate the output representation z_i by taking a weighted sum of value $\{v_j\}$ using the attention scores $\{\alpha_{ij}\}$.

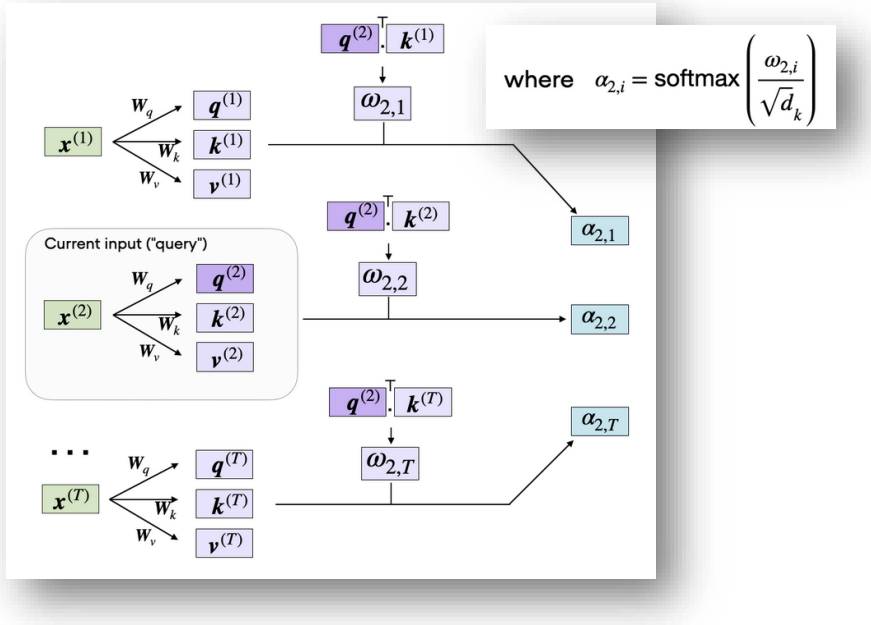


Image source: [sebastianraschka.com/blog/2023]

Self-Attention Mechanism

- Attention procedures for each input x_i
  - Step 1: Project each input token into query (q_i), key (k_i), and value (v_i) using learned linear transformations
  - Step 2: Compute causal attention scores $\{\alpha_{ij}\}_{j=1}^{i}$ where $w_{ij} = q_i^T k_j \in R, j \leq i$, ensuring the token attends only to previous (or current) positions.
  - Step 3: Generate the output representation z_i by taking a weighted sum of value $\{v_j\}$ using the attention scores $\{\alpha_{ij}\}$.

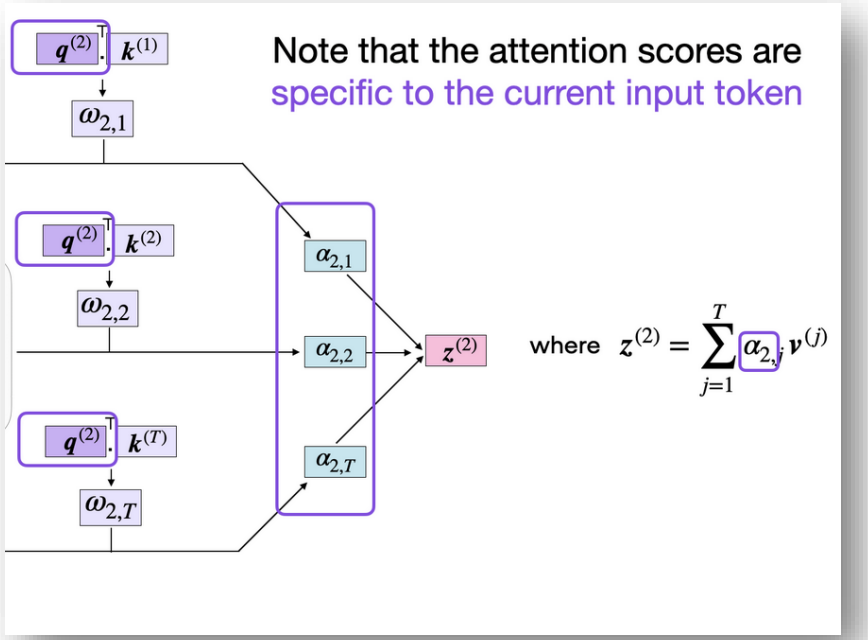


Image source: [sebastianraschka.com/blog/2023]

Cross-attention Mechanism

- In cross-attention, queries Q come from one modality A, and keys (K) and values (V) come from another modality B.

- Core Idea is to *inject* or *fuse* information from a second source B into the first A, i.e., tokens in A attend to tokens in B.

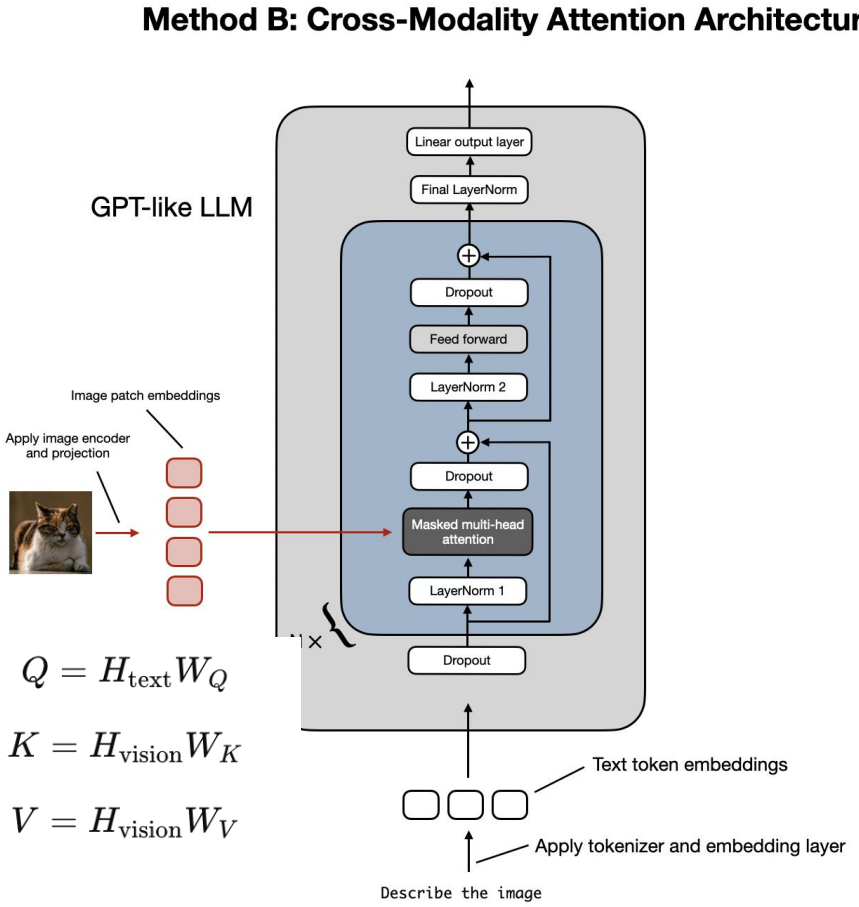
$$\text{CrossAttn}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

$Q \in \mathbb{R}^{L_a \times d}$ from source A

$K, V \in \mathbb{R}^{L_b \times d}$ from source B

Cross-attention Mechanism

- **Case: Text–Image Understanding**
  - Purpose: Inject visual grounding into text generation.
  - **Q**: text hidden states from the LLM
  - **K, V**: vision encoder outputs (image patch embeddings)
    - **Q from text**: “What does this word need to know about the image?”
    - **K from vision**: “Where in the image is the relevant information located?”
    - **V from vision**: “What is the actual visual content to inject?”
  - **e.g., LLaMA 3.2, 4 multimodal**



https://sebastianraschka.com/blog/2024/understanding-multimodal-llms.html

Multi-Head Attention (MHA) Block

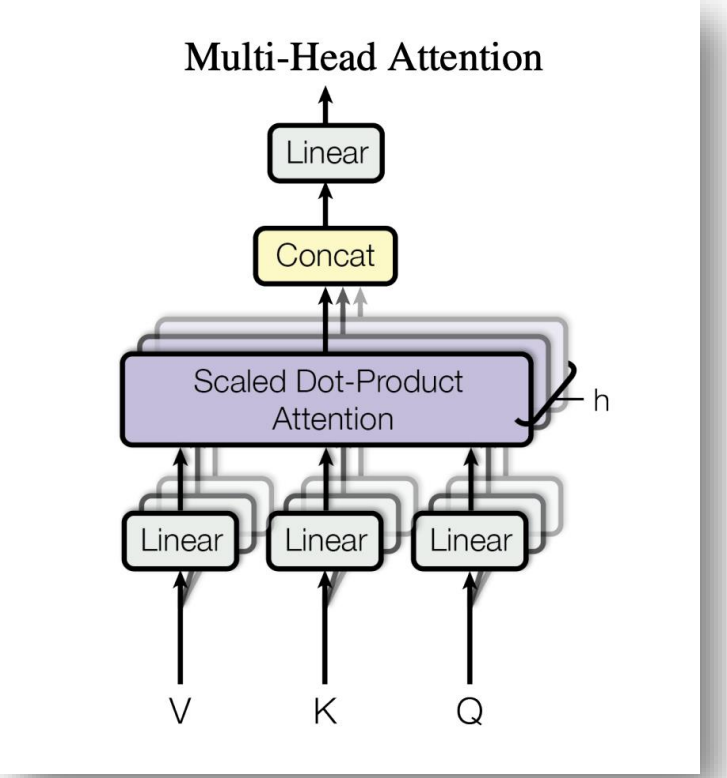
- **MHA** splits model hidden dimension *d_model* evenly across *n_h* parallel heads, i.e., *d_head* = *d_model//n_h*

- Q, K V are processed independently by individiual head

$$\text{head}_i = \text{Attention}(Q_i, K_i, V_i) \quad (Q_i, K_i, V_i) = (XW_i^Q, XW_i^K, XW_i^V)$$

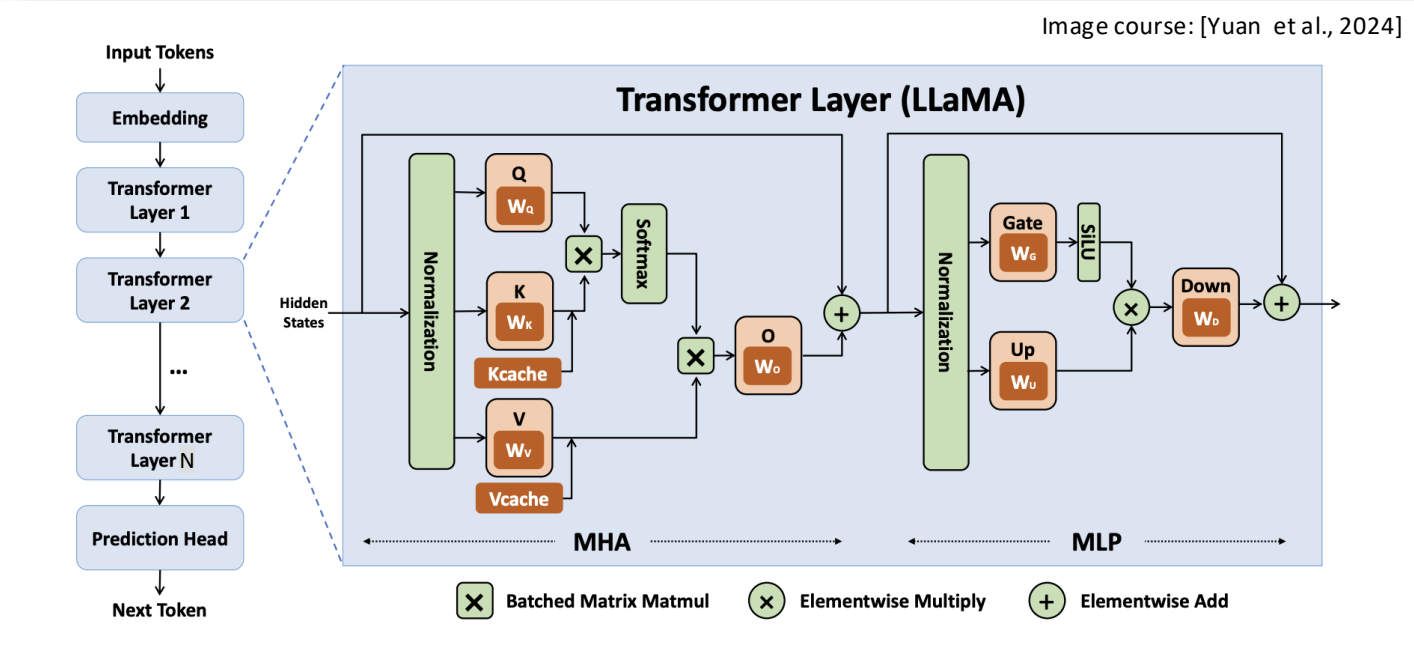
- Concetenate all the heads, project with output weight *W_o* to construct full *d_model*

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_{nh})W^O$$



[Vaswani et al., 2017] the Transformer architecture

Example: Llama-2 with Transformer Layer

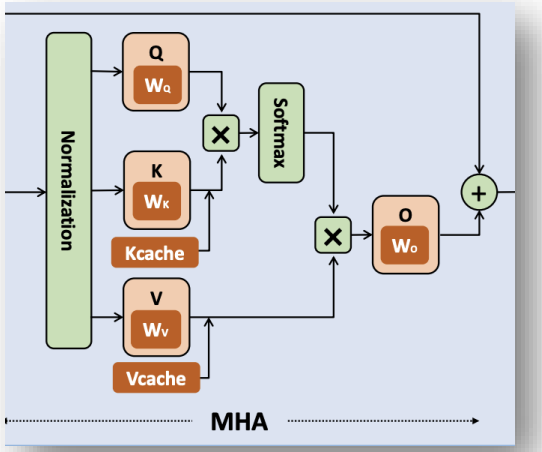


- L Transformer layer : MHA block + MLP block (projection up/down)
- MLP block uses SwiGLU (replacing Re/GeLU in GPT2)
- RMS norm (replacing Layernorm in GPT2)
- Other blocks: token embeddings, prediction heads, etc

Attention Computation

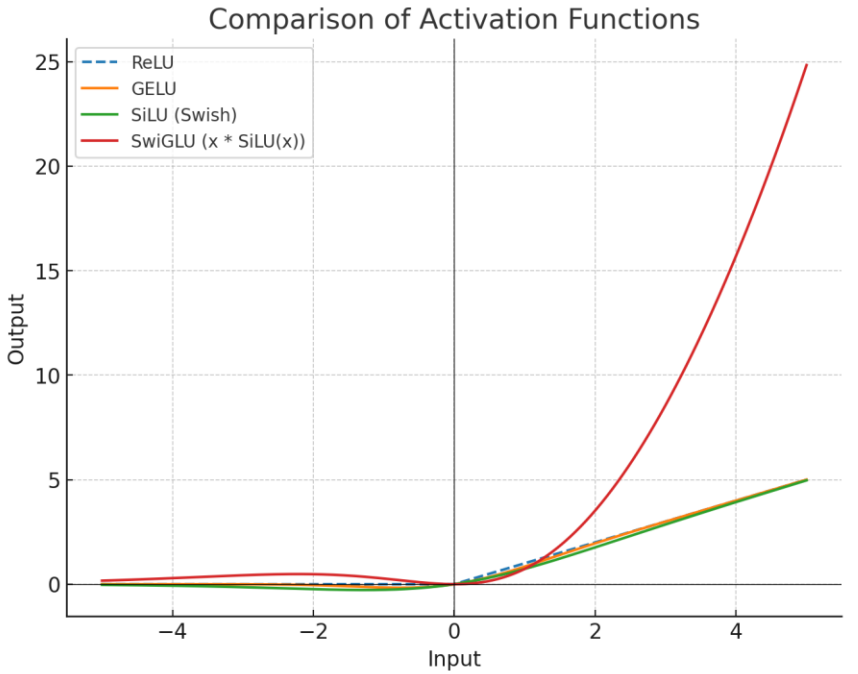
- four Q, K, V, O linear projection
  - $O(4BTd_{head}^2)$, quadratic on head dimension
- Attention dot product, softmax, V projection
  - $O(2BT^2d_{head})$, quadratic on seq. len. T
    - Mat-mul between $R^{Td_{head}}$ by$R^{d_{head}T}$ matrices on each sample
    - Mat-mul betweenR^{TT} by$R^{Td_{head}}$ matrices
- For MHA, multiply by n_h num. of heads

Image course: [Yuan et al., 2024]



Activation Functions for MLP

- **ReLU**: Simple, sparse, but non-smooth at 0.
- **GELU**: Smooth probabilistic gating, used in BERT and GPT.
- **SiLU (Swish)**: Smooth and self-gated, better gradient flow.
- **SwiGLU**: Adds an explicit gating mechanism making it more expressive and widely adopted in modern LLMs (PaLM, LLaMA).



MLP with SwiGLU

- SwiGLU (SiLU-activated Gated Linear Unit):
  - Activation

$$\sigma_{\text{SiLU}}(z) = z \cdot \sigma_{\text{sigmoid}}(z)$$

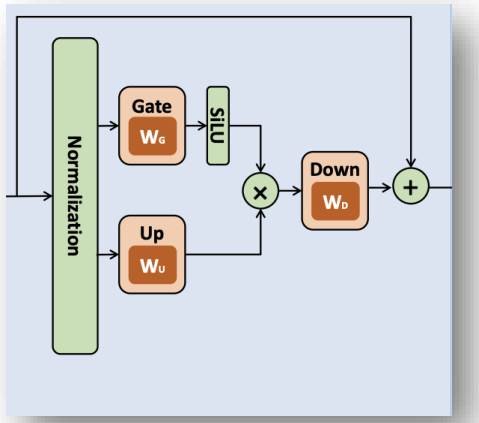
  - Gating controls elementwise pathway
  - Output combines the gated path with an additional linear projection

$$\text{SwiGLU}(X) = \big((XW_1) \odot \sigma_{\text{SiLU}}(XW_3)\big)W_2$$

- Three linear projections
  - $O(3eBTd^2)$, quadratic on hidden dim., expansion ratio e

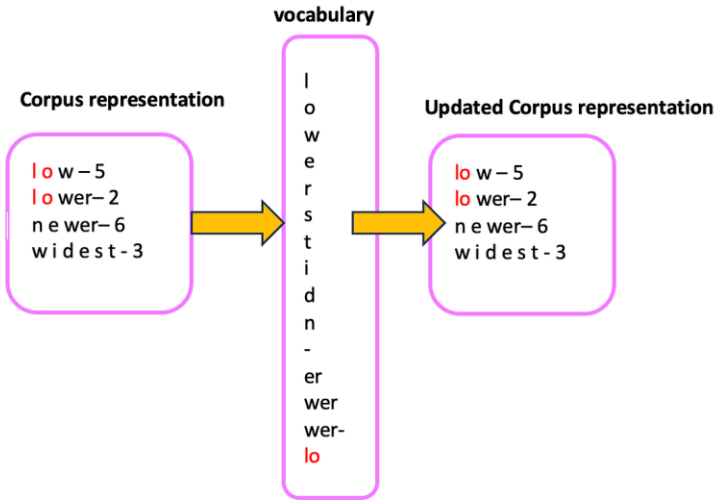
- (More advance MLP, e.g., Mixture of Experts, will be cover later)

Image course: [Yuan et al., 2024]



Vocabulary and Tokenizer

- The vocabulary size V is crucial:
  - too small leads to long token sequences
  - too large wastes parameters and memory
  - finding the right balance is key for model efficiency and performance.



- Byte Pair Encoding (BPE) tokenization builds V by repeatedly merging the most freq pairs of symbols (characters or subwords) into new tokens
- BPE is popular choice for Llama3, Qwen3, etc
- Example of Llama3:
  - 8B, 70, 405B variant of LLaMA 3.1, this dimension is 4k, 8k, 16k
  - base V plus additional tokens to better support multiple languages

Byte Pair Encoding (BPE) Tokenizers

- Byte Pair Encoding (BPE) Example
  - Dict={tug:9 mug:7 fun:11 hug:5 run:8}
- Initial vocab V starts with char:count:
  - V={t:9 u:40 g:21=9+7+5 m:7 f:11 n:19 h:5 r:8}
- Step 1 – Count pairs from current V and check freq from dict:
  - tu:9 ug:21=9+7+5 mu:7 fu:11 un:19=11+8 ru:8
  - ug:21=9+7+5 is most freq pair to add
- Step 2 – Update vocab:
  - Add: ug:21, remove overlapping counts: u→19=40-21 g→0=21-21
  - V= {t:9 u:19 m:7 f:11 n:19 h:5 r:8 ug:21}

Byte Pair Encoding (BPE) Tokenizers

- Byte Pair Encoding (BPE) Example
  - Dict={tug:9 mug:7 fun:11 hug:5 run:8}
- Initial vocab V starts with char:count:
  - V={t:9 u:40 g:21 m:7 f:11 n:19 h:5 r:8}
- Step 1 – Count pairs from current V and check freq from dict:
  - tu:9 ug:21=9+7+5 mu:7 fu:11 un:19=11+8 ru:8
  - ug:21=9+7+5 is most freq pair to add
- Step 2 – Update vocab:
  - Add: ug:21, remove overlapping counts: u→19=40-21 g→0=21-21
  - V= {t:9 u:19 m:7 f:11 n:19 h:5 r:8 ug:21}

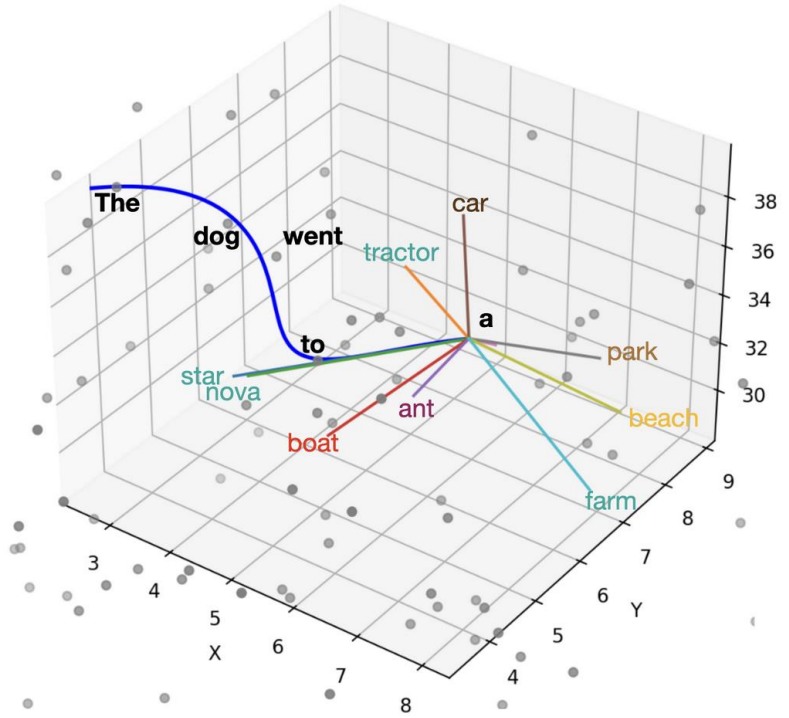
Byte Pair Encoding (BPE) Tokenizers

- Byte Pair Encoding (BPE) Example
  - Dict={tug:9 mug:7 fun:11 hug:5 run:8}
- Initial vocab V starts with char:count:
  - V={t:9 u:40 g:21 m:7 f:11 n:19 h:5 r:8}
- Step 1 – Count pairs from current V and check freq from dict:
  - tu:9 ug:21=9+7+5 mu:7 fu:11 un:19=11+8 ru:8
  - ug:21=9+7+5 is most freq pair to add
- Step 2 – Update vocab:
  - Add: ug:21, remove overlapping counts: u→19=40-21 g→0=21-21
  - New V= {t:9 u:19 m:7 f:11 n:19 h:5 r:8 ug:21}

Token Embedding

- Token embedding transforms token into embedding dimension via look-up table E
  - X = E[tokens]
- $O(BTd_{model})$ computational cost
  - No dependency on V
- Related vocabs (semantics, freq) are closer in embedding space
- Visualization
  - applying PCA to reduce them to 3D, clustering by token frequency or semantic usage

```
def forward(self, token_ids: int[Tensor, " ..."])
    -> Float[Tensor, " ... embedding_dim"]:
    return self.embedding_weight[token_ids]
```



Ron J's Projects

Position Embedding

- Transformers lack an inherent sense of order, so positional encoding provides tokens with position information

- RoPE (Rotary Position Embedding) encodes relative positions directly into the attention mechanism using rotations
  - preserving relative ordering compared to absolute position encodings.
  - more flexible for extrapolation to longer contexts, efficient to implement

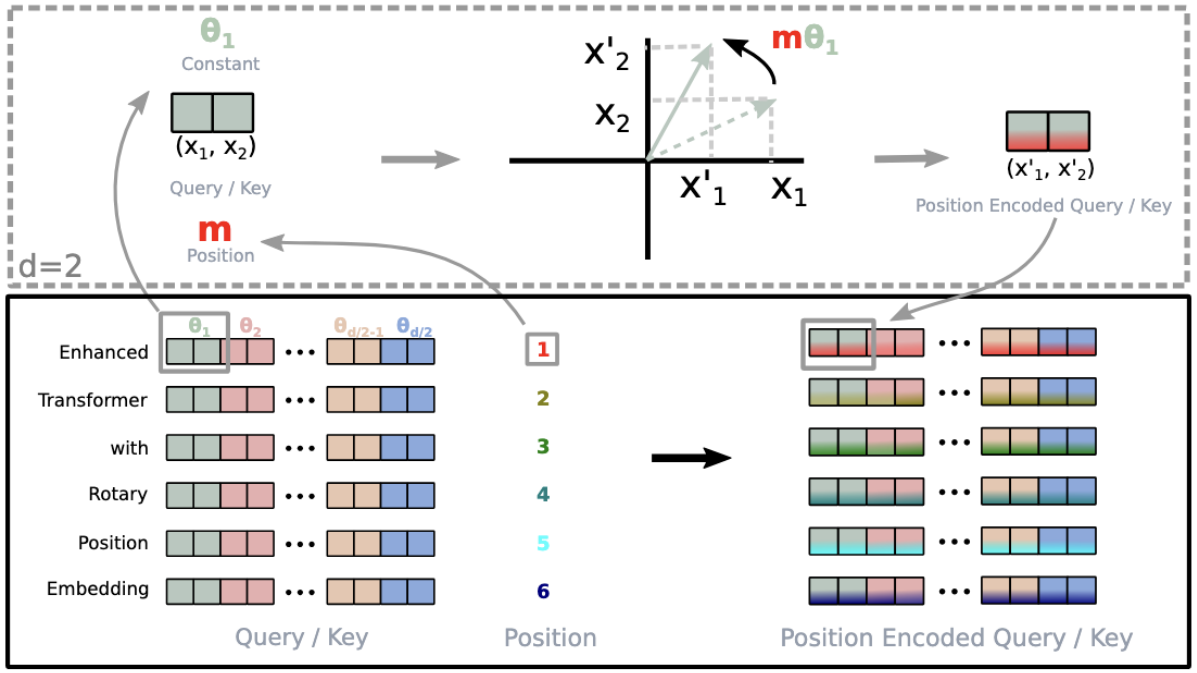


Figure 1: Implementation of Rotary Position Embedding(RoPE).

Rotary Position Embedding (RoPE)

- RoPE applies rotation matrix to the query and key, i.e., $q_i^{rope} = R(\Theta)q_i$
- The inner product after rotation depends only on the relative position $i - j$
- R(θ) consists of collections of 2 by 2 rotation matrix with $\theta_{i,k} = i/\Theta^{2k/d}$. The rotation angle $\theta_{i,k}$ gets smaller for higher index in hidden dimension and old time-stamp i.

$$(q_j^{\mathrm{RoPE}})^\top k_j^{\mathrm{RoPE}} = \left(R^d_{\Theta,i} q_i\right)^\top \left(R^d_{\Theta,j} k_j\right) = q_i^\top R^d_{\Theta,i-j} k_j$$

$$\left(R^d_{\Theta,i}\right)^\intercal R^d_{\Theta,j} = R^d_{\Theta,\,i-j}$$

$$R^d_{\Theta,i-j} = \mathrm{Diag}\left(R(\theta_{i-j,k}), \ldots, R(\theta_{i-j,d/2})\right) \in \mathbb{R}^{d\times d}$$

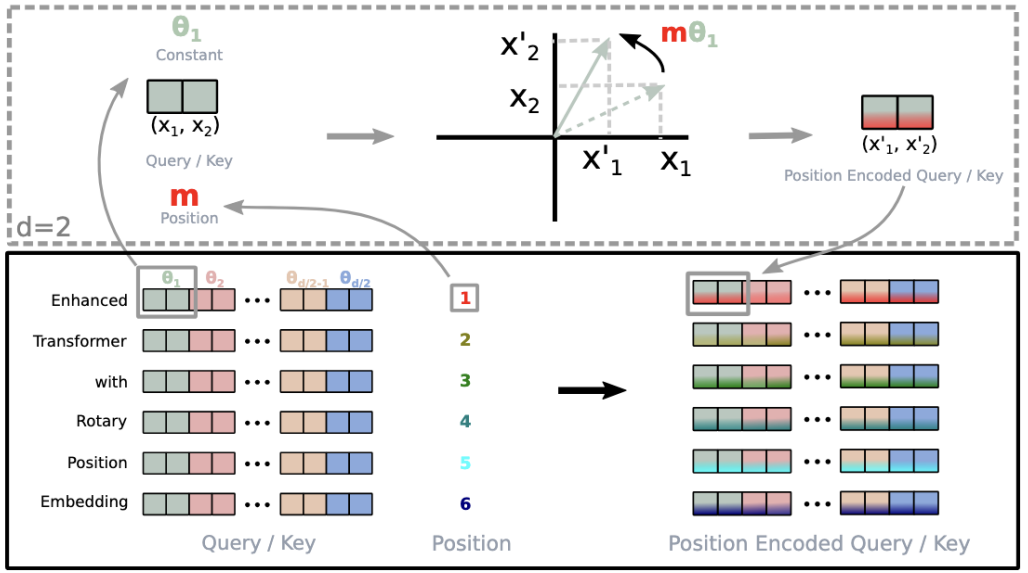
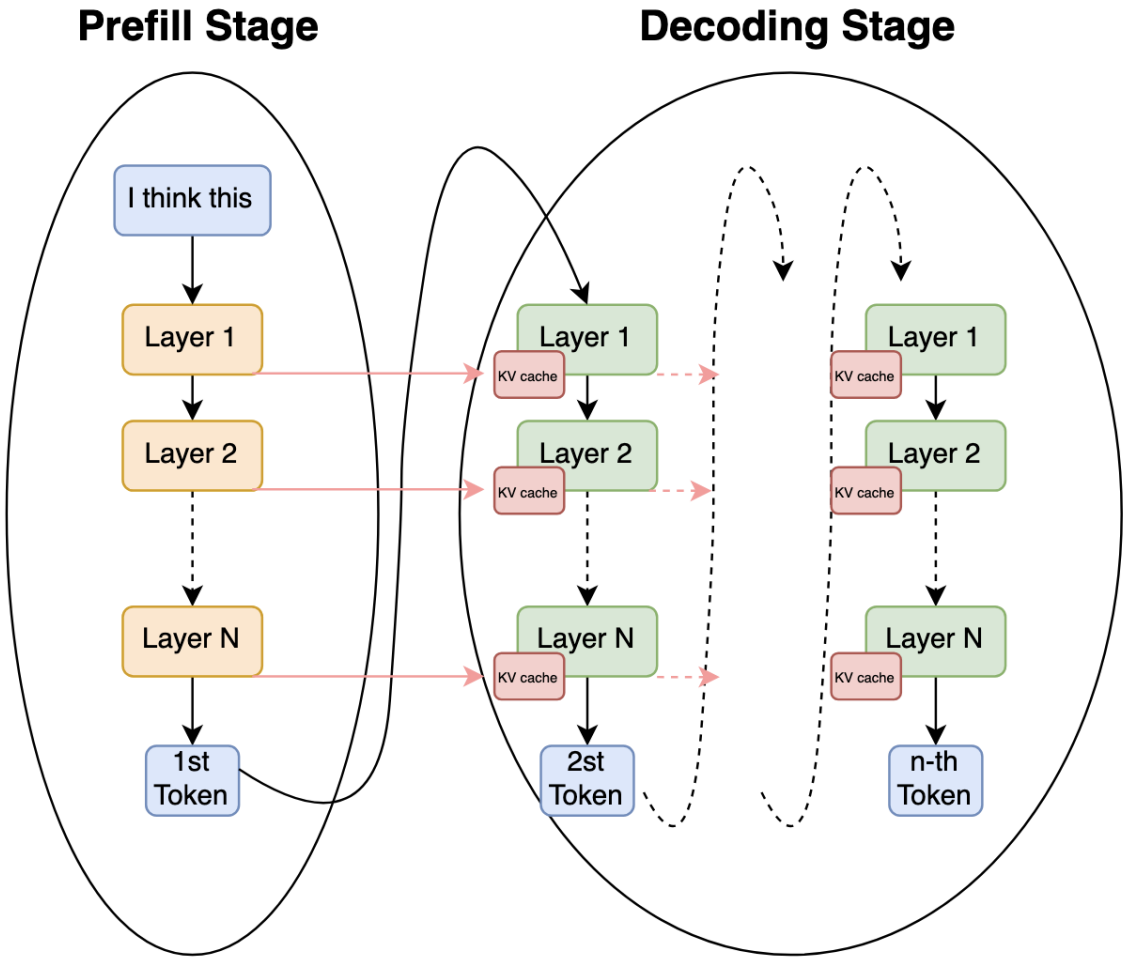


Figure 1: Implementation of Rotary Position Embedding(RoPE).

Inference Task & Computations

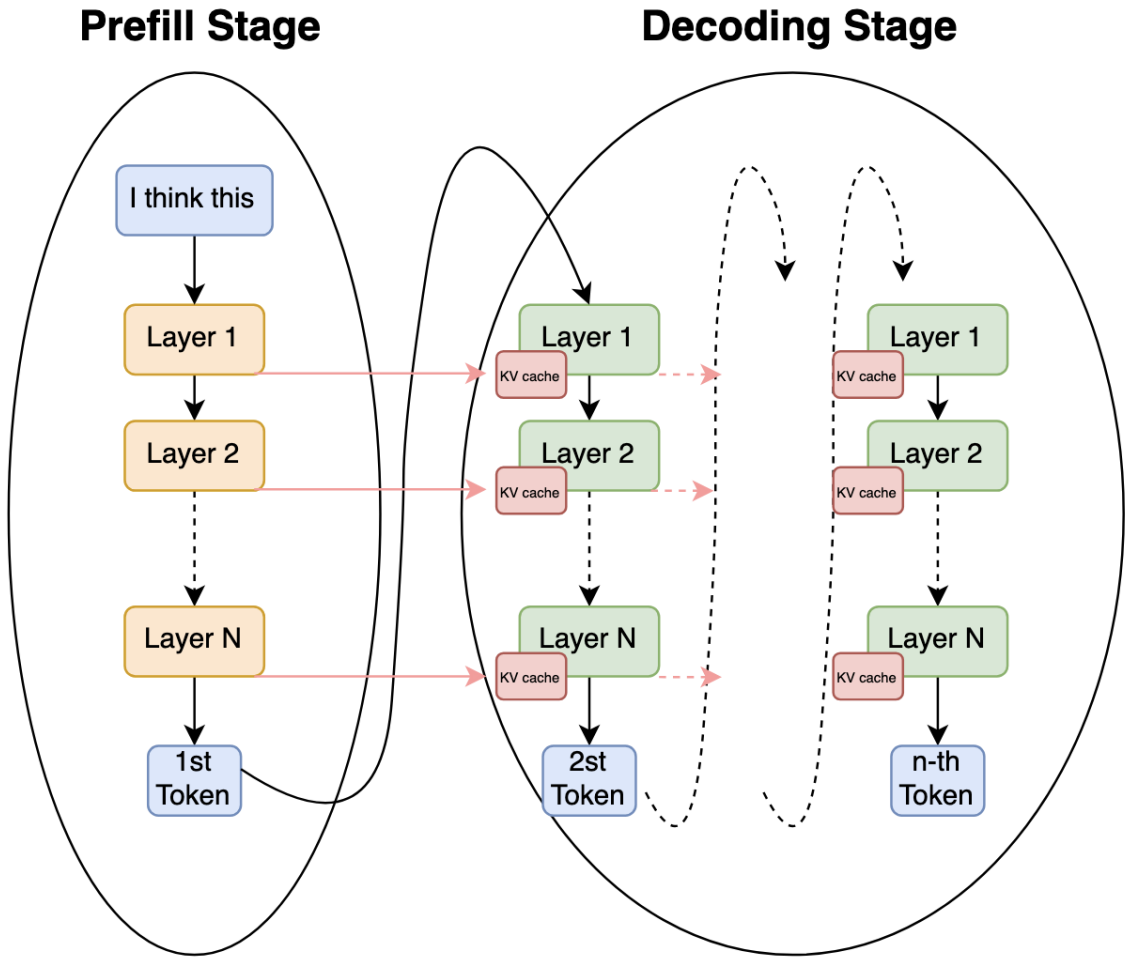
- Inference task
  - Given an input sequence X, use LLMs to generate next n-continuation Y.
  - E.g.) X= I think this, Y_1 = is, Y_2, …
  - use Transformer architecture (with embeddings & precidion head) to generate next token in an autoregressive manner

- Two-step solution
  - Step 1. Prefill stage
  - Step 2. Decoding stage



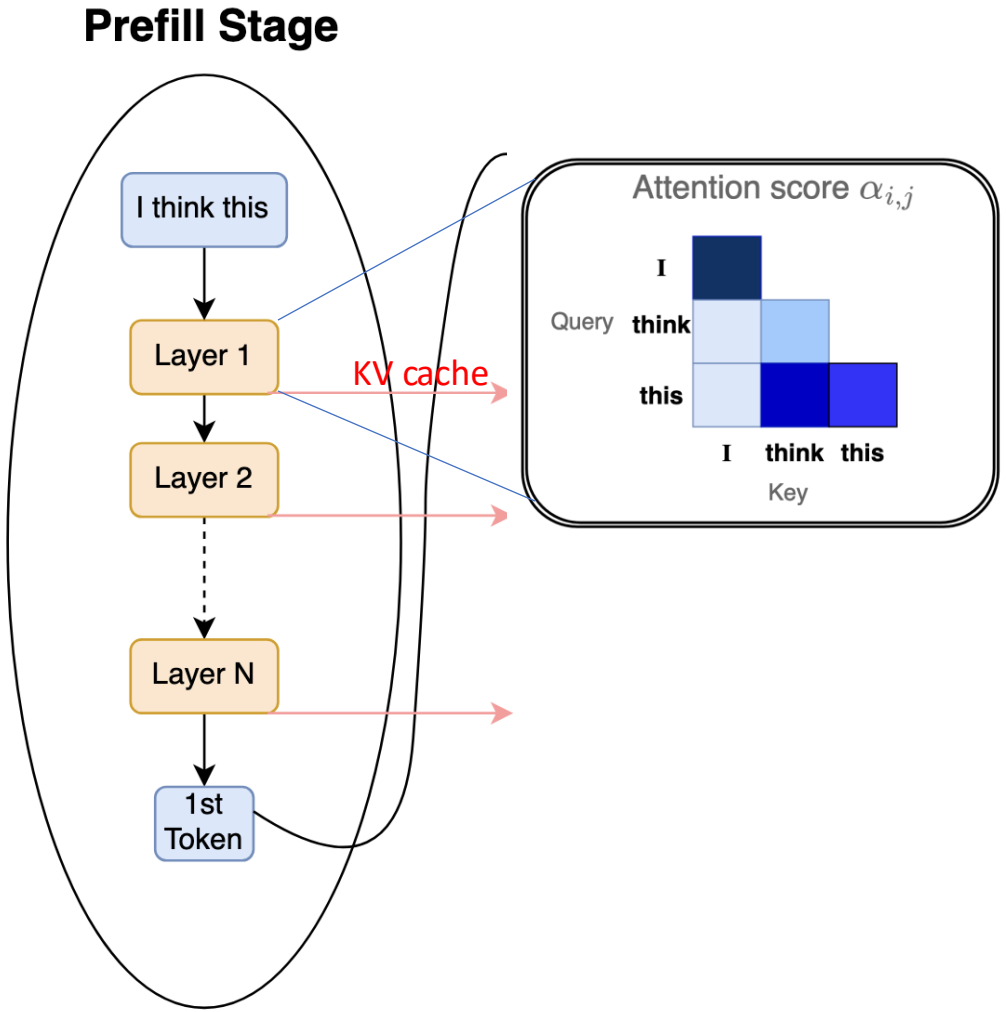
Inference Task & Computations

- 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



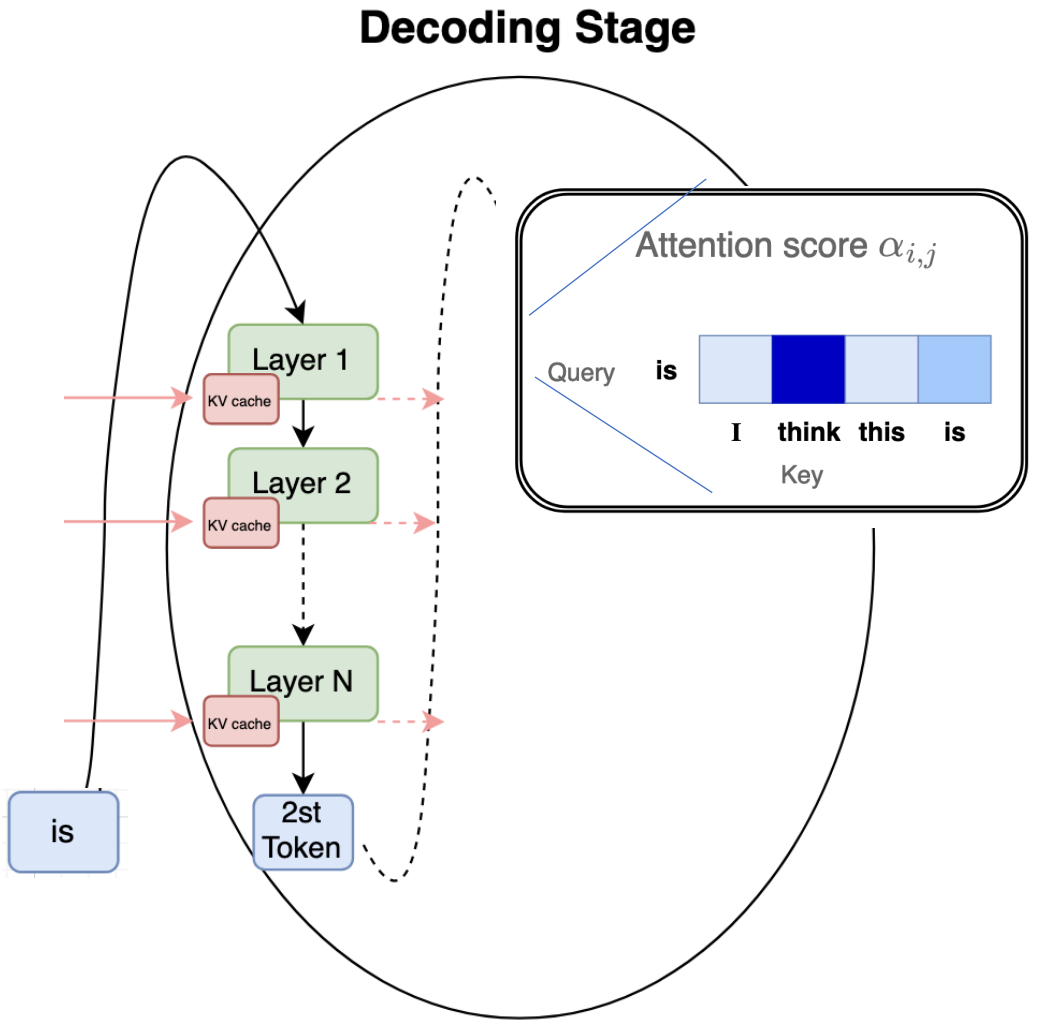
Inference Task & Computations

- 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
    - save for each token the key and value activations in each layer, ie., KV cache, for the decoding stage (covered later)
    - The representations of all tokens during the prefill stage is computed in parallel.

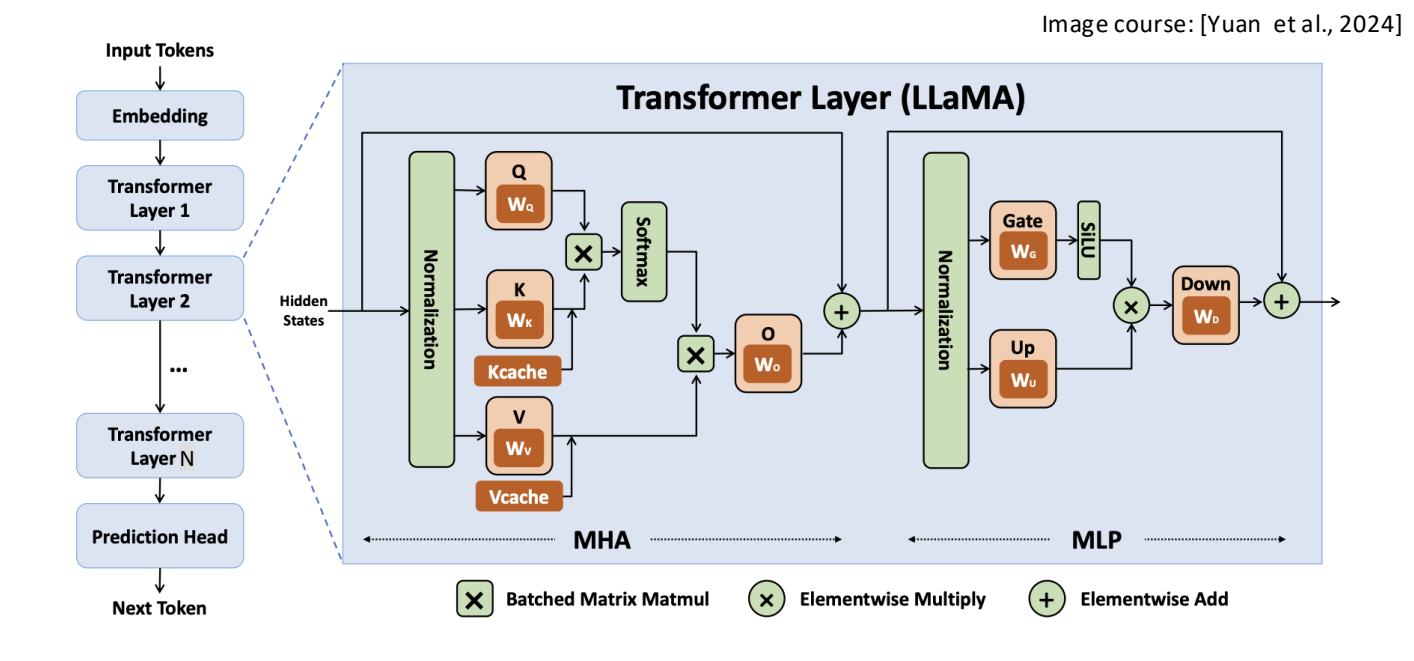


Inference Task & Computations

- 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



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.

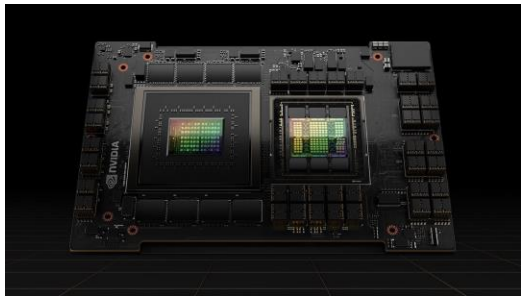
$$4d^2 + Ld + 8d^2 = 12d^2 + Ld$$

Overview of AI Hardware

- What are AI Accelerators?
- Compute Engines in AI Accelerators
- Memory Hierarchy in AI Accelerators
- Kernel Programming
- Scale Up Networking: From Chip to Server
- Scale Out Networking: From Server to Datacenter

What are AI Accelerators?

- GenAI workloads (e.g., Transformer computations) have highly regular computational demands
- Requires **high speed communication, arithmetic throughput, low latency** and **energy efficiency**
- Modern AI accelerators are **domain-specific hardware** (ASICs) explicitly designed to speed up the heavy linear-algebra (e.g., matrix multiplications)
- Example: **AWS Trainium**, **Google TPUs**, **NVIDIA GPUs** *co-design the entire vertical stack*, i.e., silicon, compiler, runtime, and interconnect around the computational demands of machine learning workloads.



Hardware Specifications

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

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

NeuronCore-v4

Each Trainium3 chip consists of the following components:

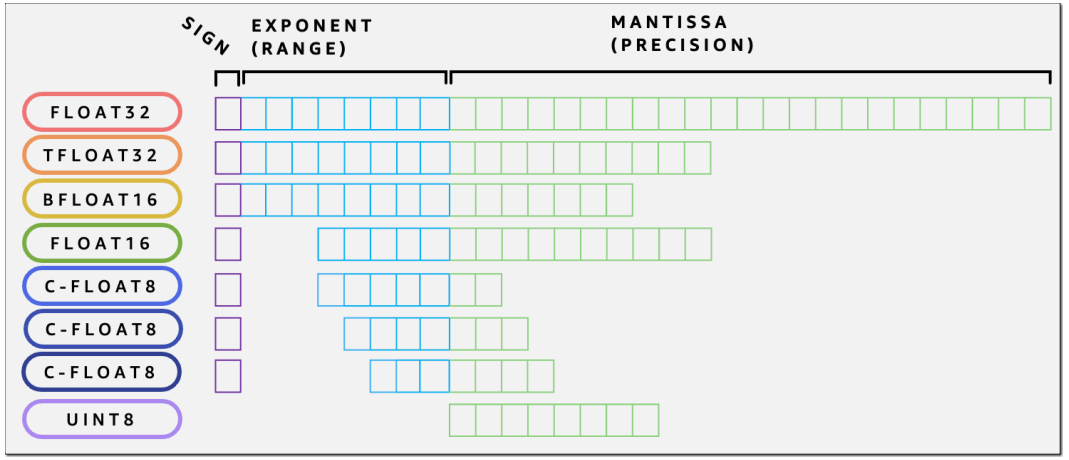
Compute	Eight NeuronCore-v4 cores that collectively deliver: • 2,517 MXFP8/MXFP4 TFLOPS • 671 BF16/FP16/TF32 TFLOPS • 2,517 FP16/BF16/TF32 sparse TFLOPS • 183 FP32 TFLOPS
Device memory	144 GiB of device memory, with 4.9 TB/sec of bandwidth.
Data movement	4.9 TB/sec of DMA bandwidth, with inline computation.
NeuronLink	NeuronLink-v4 for device-to-device interconnect provides 2.56 TB/sec efficient scale-out training, as well as memory pooling between the diffe

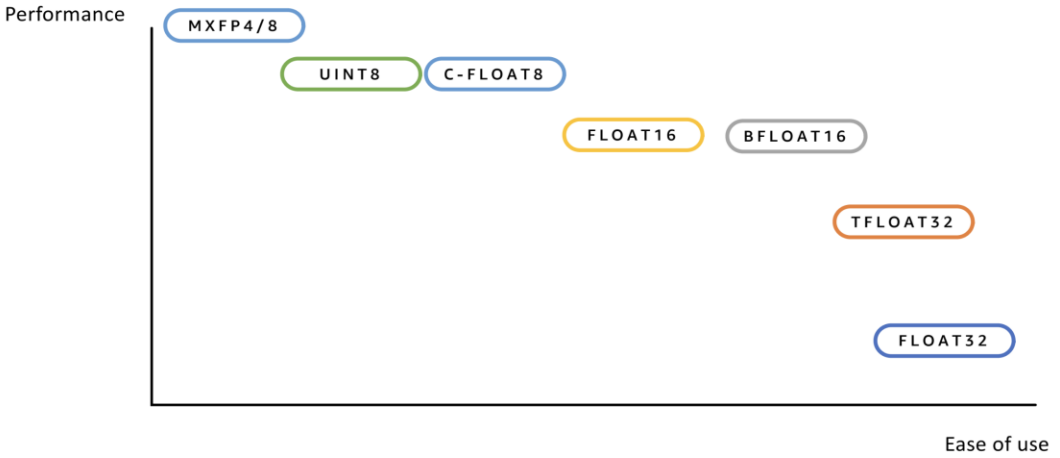
Key elements:

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

Supporting Data Types

- Various precision formats are supported by modern hardware
- Floating-point formats differ in:
  - **Exponent bits** → dynamic range
  - **Mantissa bits** → precision
- From full-precision of FP32 to arising data types (e.g., MXFP8/4, NVFP4)
- High number of bits → Ease of use
- Low number of bits → High performance



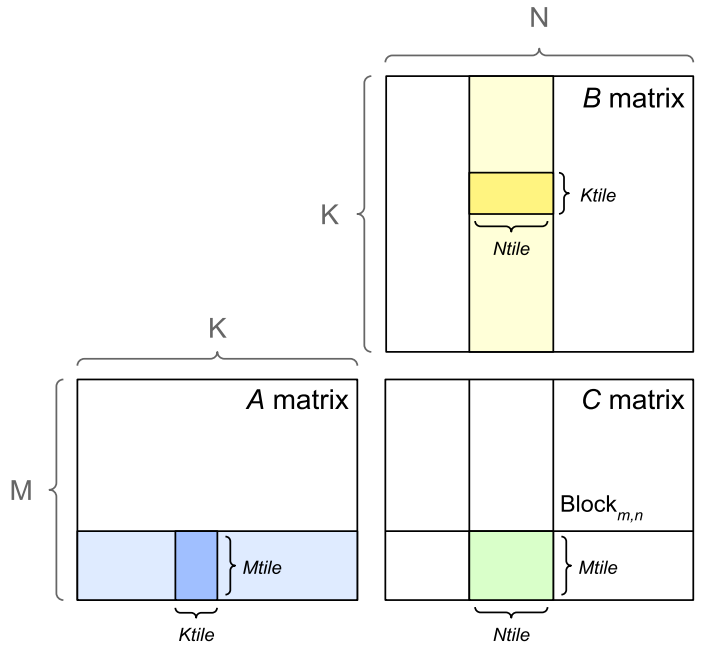


Matrix Multiply-Accumulate: Blocks of GEMM

- Matrix Multiply-Accumulate (MMA) is the basic operation, computing small tiled blocks of C = A @ B + C

- Different chips support diff M, K, N in their tensor core on BF16
  - Nvidia: M16 x N8 x K8, M16 x N8 x K16 [NVIDIA CUDA PTX]
  - AMD: M16 x N16 x K8, M32 x N32 x K4 [AMD Matrix Cores]
  - Google TPU: M128 x N128 x K128 [source]
  - AWS Trainium: M128 x N512 x K128 [source]

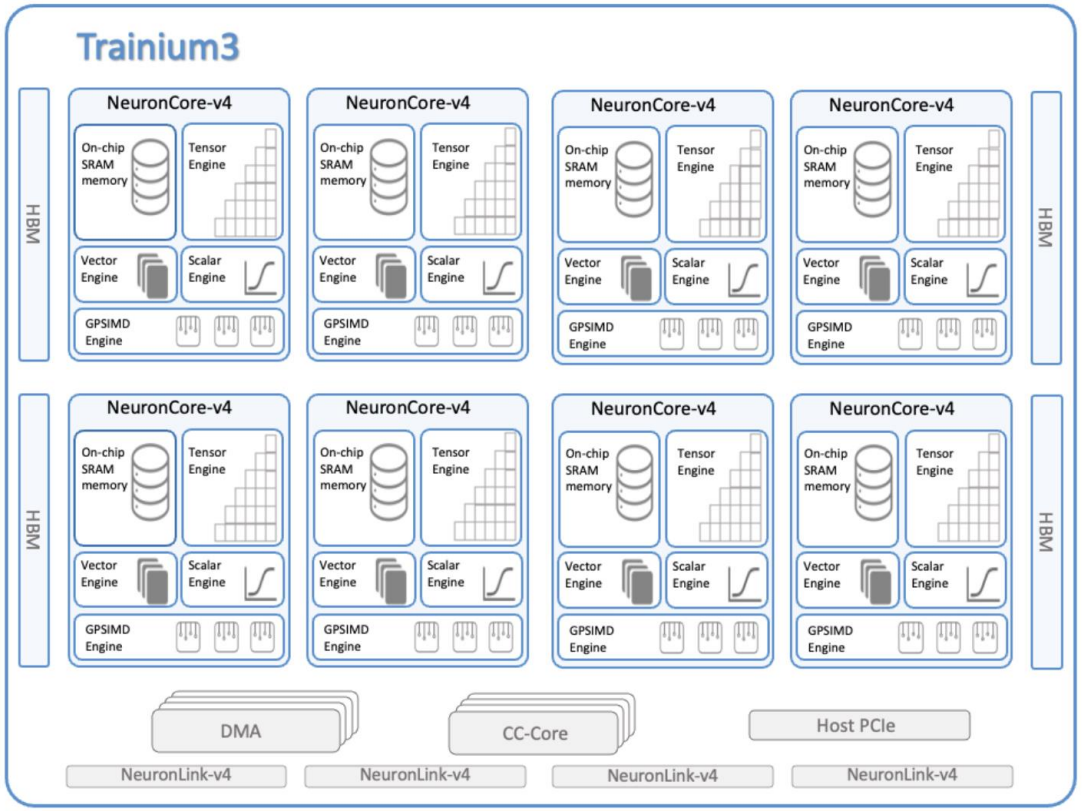
- Design of MMA units trades off between flexibility vs peak MFU



Figures are from NVIDIA

Case Study: AWS Trn3 AI Accelerators

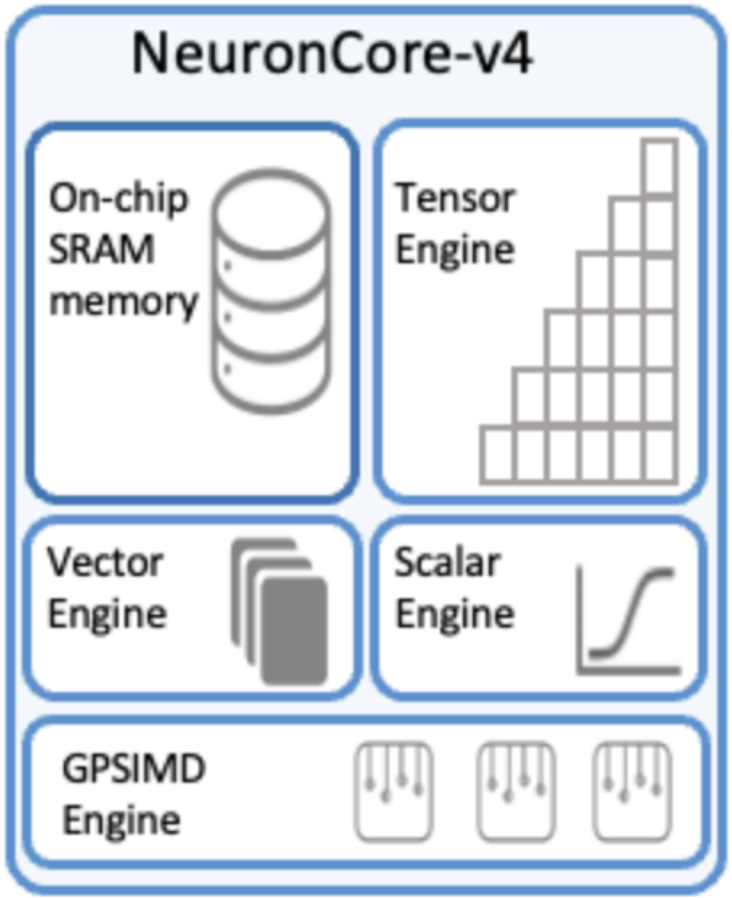
- Case Study: **Trainium3** (*concepts are generally applicable to other chips*)
- This shows a block diagram of a Trainium3 device, which consists of:
  - 8 NeuronCores (v4).
  - 4 HBM stacks with a total device memory capacity of 144 GiB and bandwidth of 4.7 TB/s.
  - 128 DMA (Direct Memory Access) engines to move data within and across devices.
  - 20 CC-Cores for collective communication.
  - 4 NeuronLink-v4 for device-to-device collective communication.



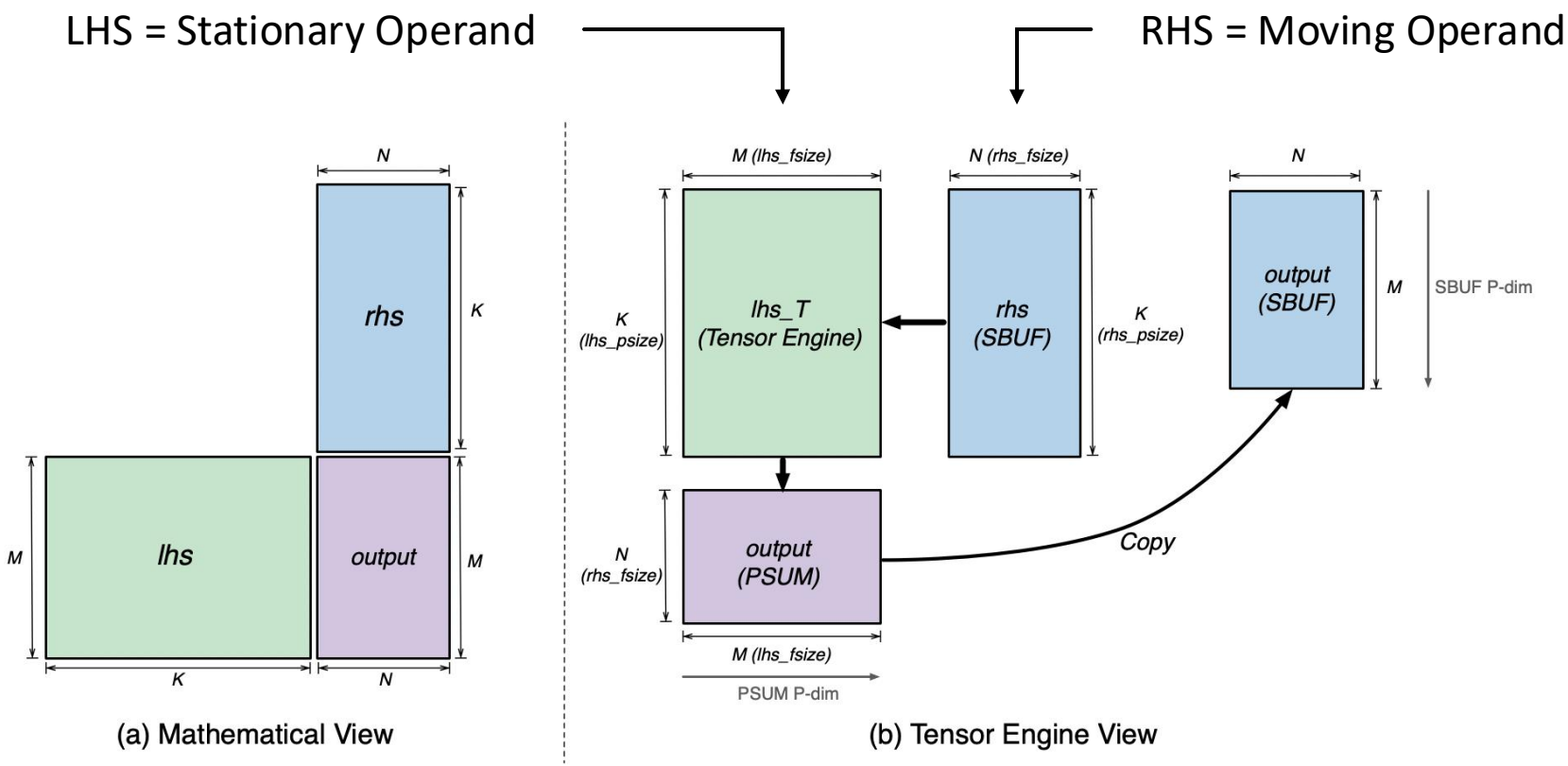
amazon | science

Compute Engines in AI Accelerators

- **NeuronCore**: Fundamental compute unit in Trn
- **Tensor Engine**:
  - Responsible for accelerating the dense linear-algebra operations.
  - Trn3 chips delivers 315 MXFP8/MXFP4 TFLOPS, 79 BF16/FP16/TF32, or 20 FP32 TFLOPS of tensor computations.
  - Responsible for the vast majority of FLOPs in LLM inference.
- **Vector Engine**:
  - Responsible for wide-variety of vectorized operations
  - Examples include axpy operations (Z=aX+Y), Layer Normalization, activation functions, and Pooling operations.
- **Scalar Engine**:
  - Computations in which every element of the output is dependent on one element of the input (e.g., activations, absolute value, etc.)
- **GPSIMD Engine**: Everything else (general-purpose code, implementing and executing cusotm operators)
- **All engines can run in parallel (high throughput)**

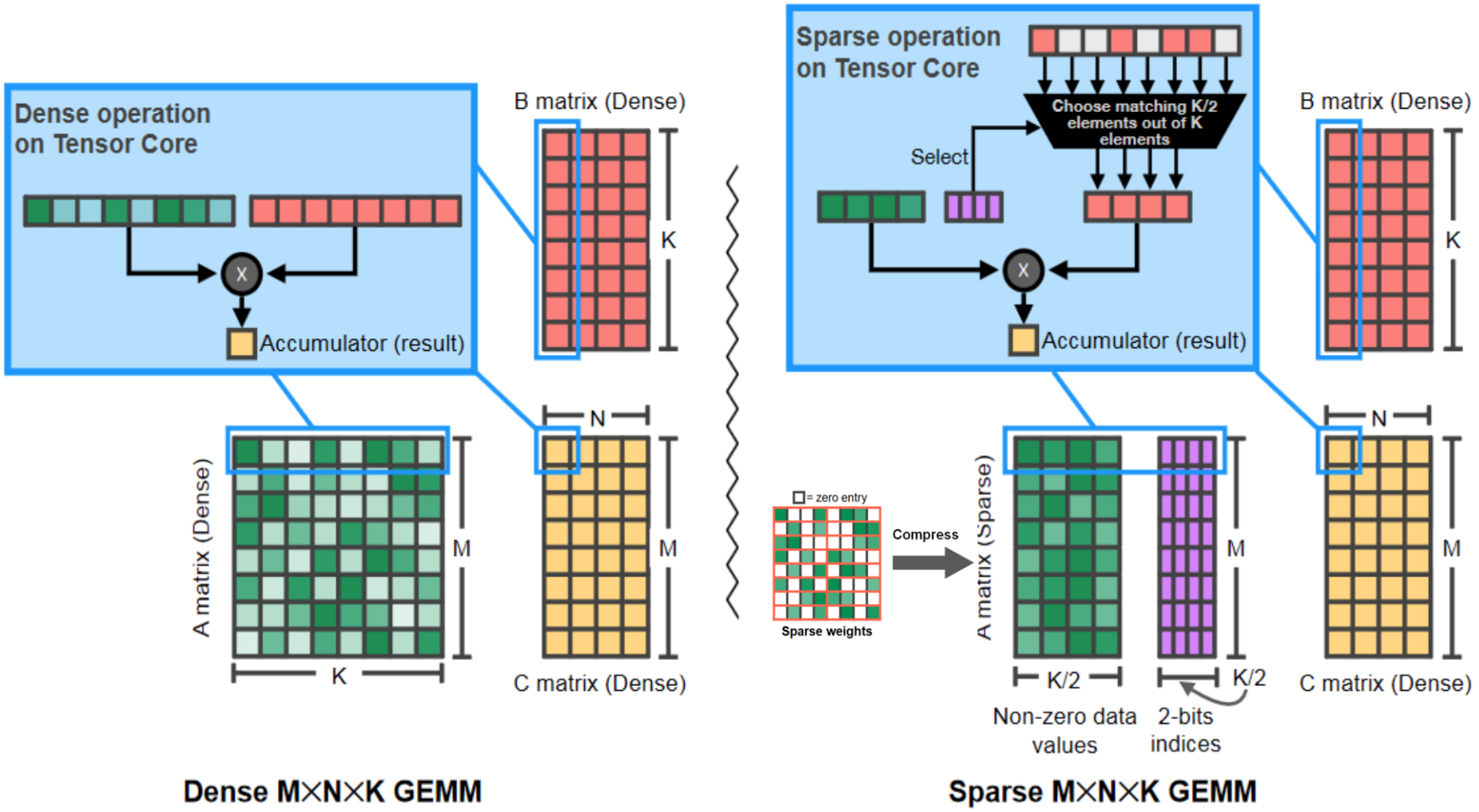


Matrix Multiplication on the Tensor Engine

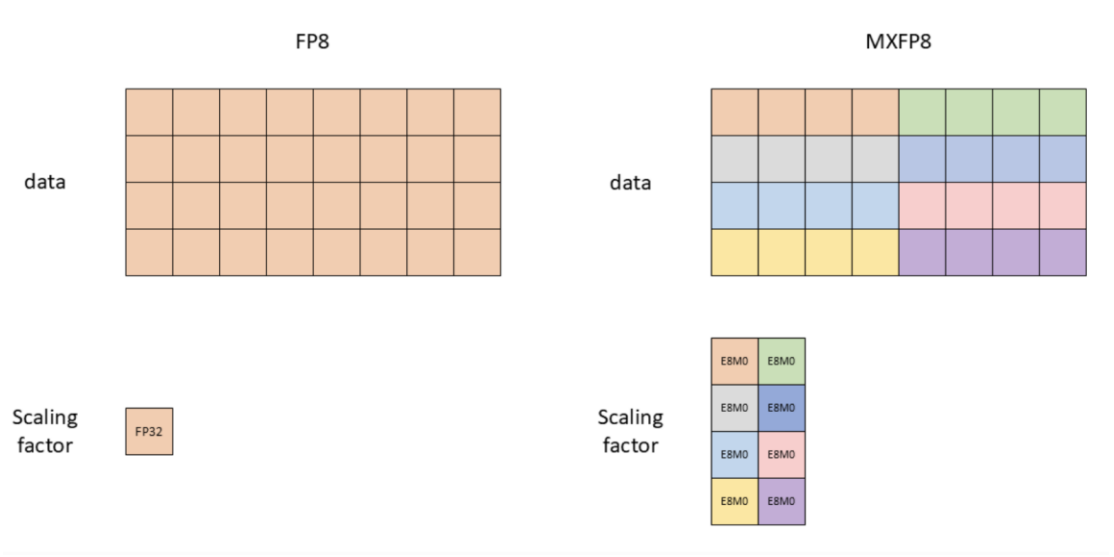


To compute the matrix product $A.B$ of shape $[M, N]$, must map into physical layout of the memory first, i.e., providing $A^{\intercal}$ (shape $[K, M]$) and B (shape $[K, N]$) to the Tensor Engine

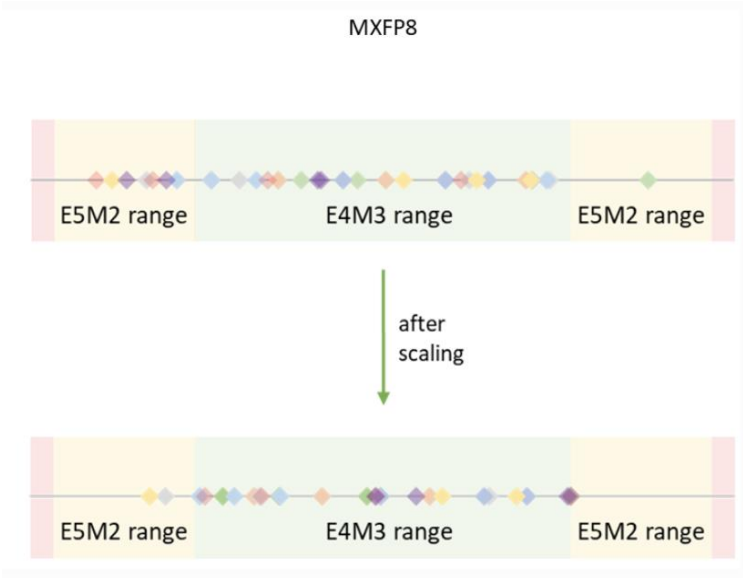
Dense/Sparse Matrix Multiplications



Micro-scaling (MX) Data Type



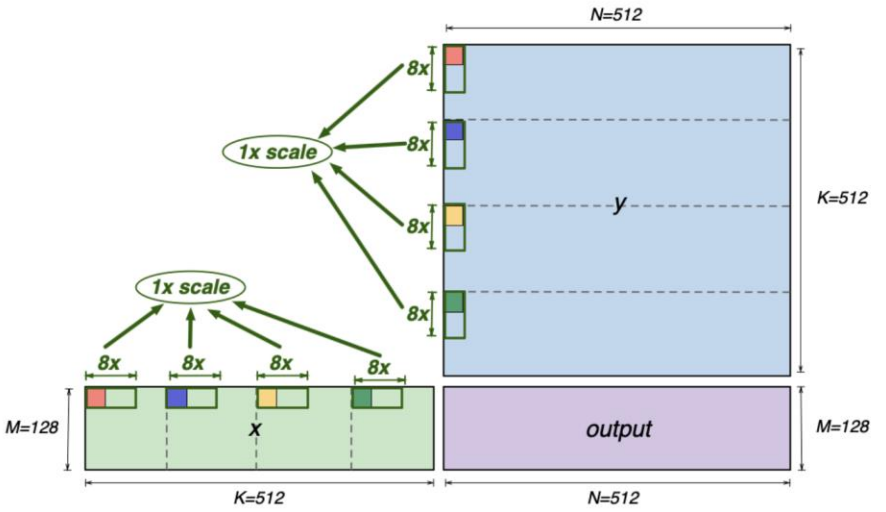
FP8 vs MXFP8 Format



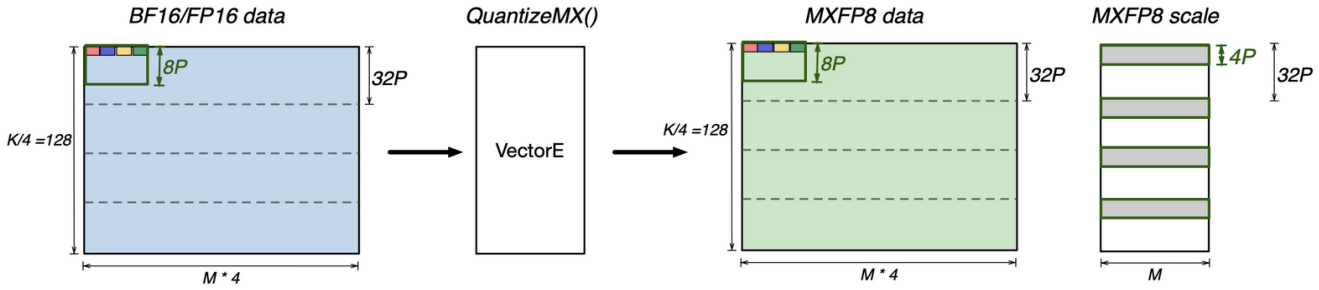
- **MX (microscaling)** is an OCP-defined quantization scheme: Applies fine-grained **absmax** scaling to small groups
- MXFP8 consists of 8 bits (1 sign, E exponent, M mantissa, E+M=7) plus 1 shared (power of 2) scale per block
- With MX, data point outside of E4M3 falls in MX-E4M3 range after the scaling, getting more accuracy

Micro-scaling Matrix Multiplication

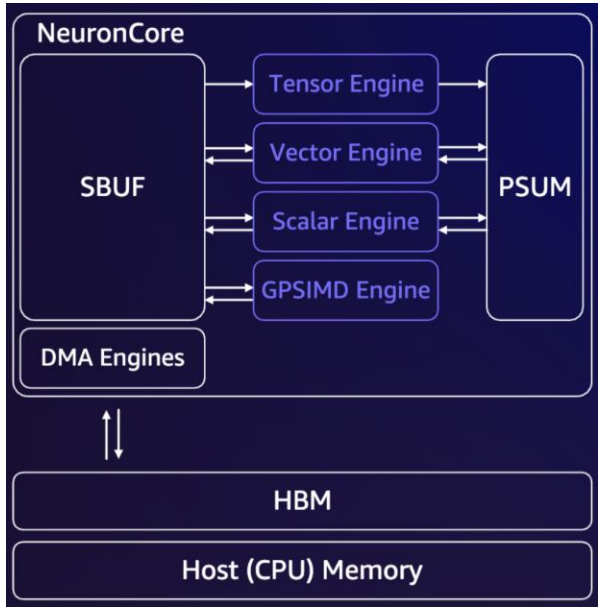
- MX matmul on Trn3:
  - 128×512×512 MMA
  - K=512 reshaped in SBUF as 128 (partition) × 4 (fast free dim),
- Tensor Engine requires quad-packed data:
  - NKI exposes this via x4 MX datatypes
  - Both stationary and moving operands following the same quad layout.
- Dequantization is fused into MX matmul

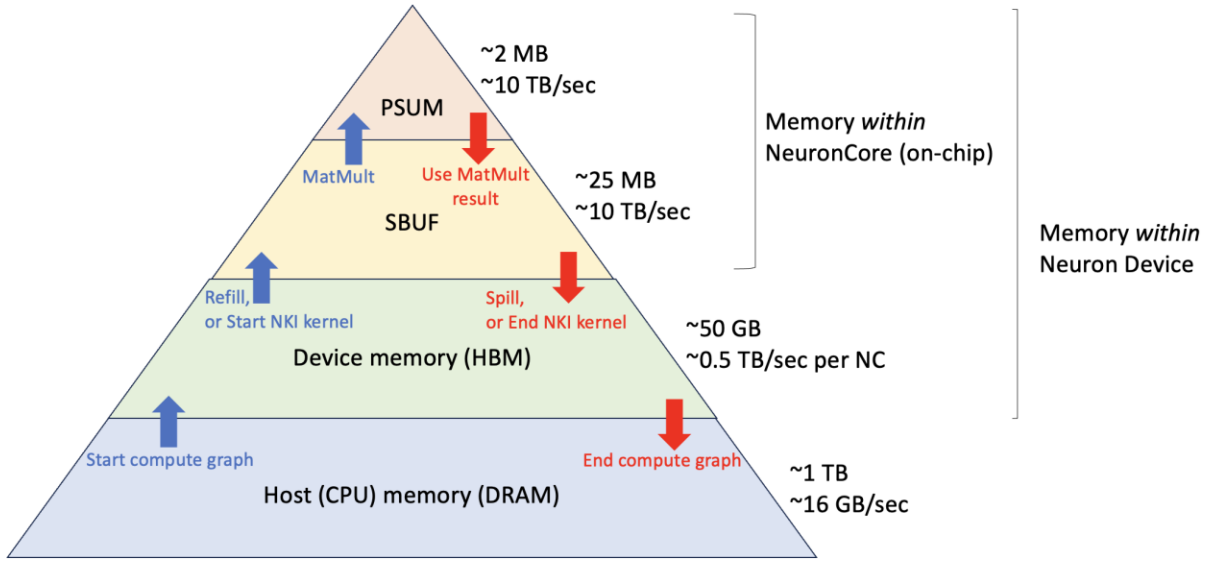


Mathematical View of a Scaling Group



Memory Hierarchy in AI Accelerators

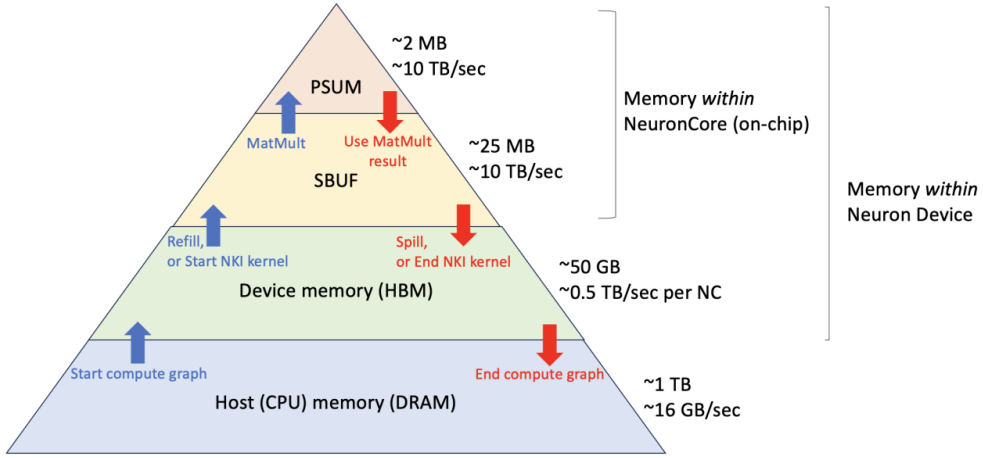


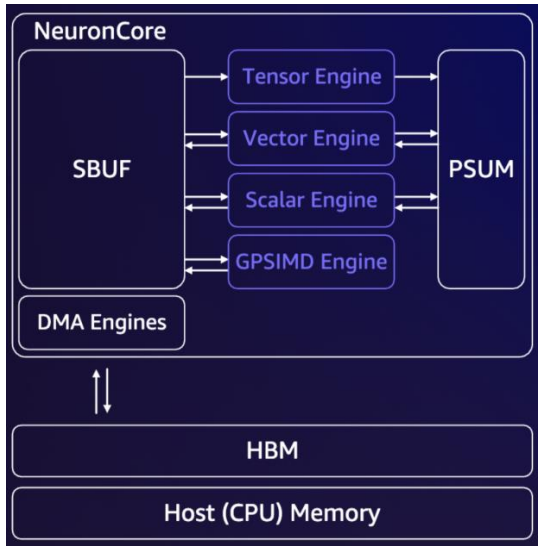


- Compute engines need to be consistently kept supplied with data to avoid keeping them idle (wasted compute)
- Peformance of an accelerator depends not only on raw compute FLOPs, but also on efficiency of memory hierarchy that moves model weights and input/output activations of each layer through the chip
- On-chip memories (SBUF, PSUM aka on-chip SRAM): Closer, **Higher bandwidth**, but **much smaller capacity**
- Off-chip memories (HBM, DRAM): **Larger capacity** but **Slow**
- **DMA Engines** move data between HBM and SBUF in parallel with computation (latency hiding).

External Memory: Host DRAM and Device HBM

- **Host DRAM:**
  - Main system memory attached to the host CPU
  - Serves as the *primary memory* for the operating system, ML frameworks, and application-level data structures
  - Model checkpoints are often initially loaded into host DRAM from storage before being transferred to accelerator-attached memory (HBM)
  - Commonly used to manage request queues, tokenize inputs, batch requests, and orchestrate data movement
- **High-Bandwidth Memory (HBM):**
  - Specialized DRAM that is present on the accelerator device
  - Primary device memory that stores model weights, activations, KV caches, and intermediate tensors during execution.
  - Significantly faster than host DRAM, it is still much slower and higher-latency than on-chip SRAM (SBUF and PSUM)
  - Many optimization strategies (e.g., quantization, KV cache compression) are techniques to reduce HBM traffic

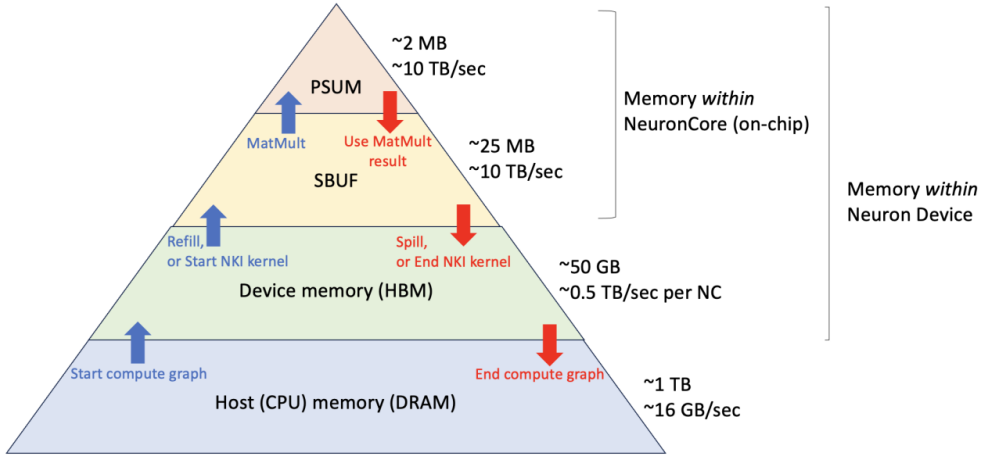


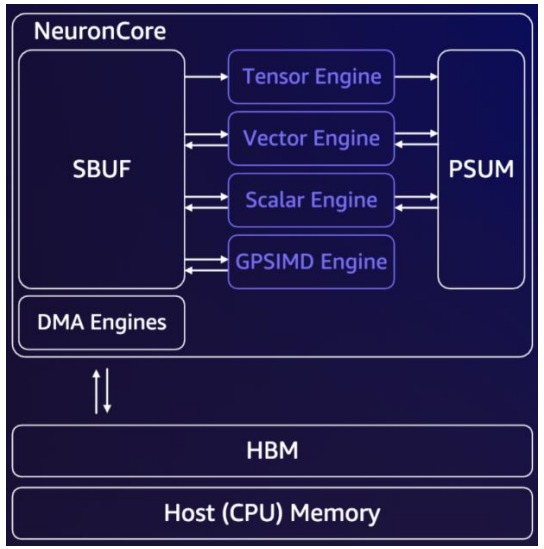


Internal (On-Chip) Memories

- **State Buffer (SBUF):**
  - Main working memory for a compute core
  - Before any computation begins, input tensors must be explicitly loaded from HBM into SBUF, and once computation completes, results must be written back from SBUF to HBM
  - SBUF capacity is limited; so careful management is required to avoid spill backs to slower memory
- **Partial Sum Buffer (PSUM):**
  - Specialized on-chip memory designed specifically to support high-throughput matrix multiplication on the Tensor Engine
  - Essential for large GEMMs, where the final output matrix is produced by accumulating many smaller tiled computations
  - Completed tiles quickly evicted to SBUF for further processing and storage

Understanding the tradeoff between speed and size is essential for writing inference-optimized kernels





amazon | science

Importance of Kernel Programming

- **What is a kernel?** A kernel is simply a function:
  - Written in a domain specific language (DSL) e.g., CUDA, Triton, NKI, etc.
  - Explicitly specifies how computation is mapped onto the available compute engines (Tensor, Vector, Scalar, and GPSIMD) and memory hierarchy (HBM, SBUF, and PSUM)
  - Optimize how tensors are tiled, where data is staged, which engine executes each operation, and when data is moved between memory levels
- **Goal**:
  - Maximize utilization of fast on-chip resources
  - Keep frequently reused data resident in SBUF or PSUM
  - Minimize expensive spill and refill traffic to HBM

Performance of LLM Inference on any accelerator chip depends not only on the capacity of the compute engines, but also on effective mapping of dataflow to this memory hierarchy

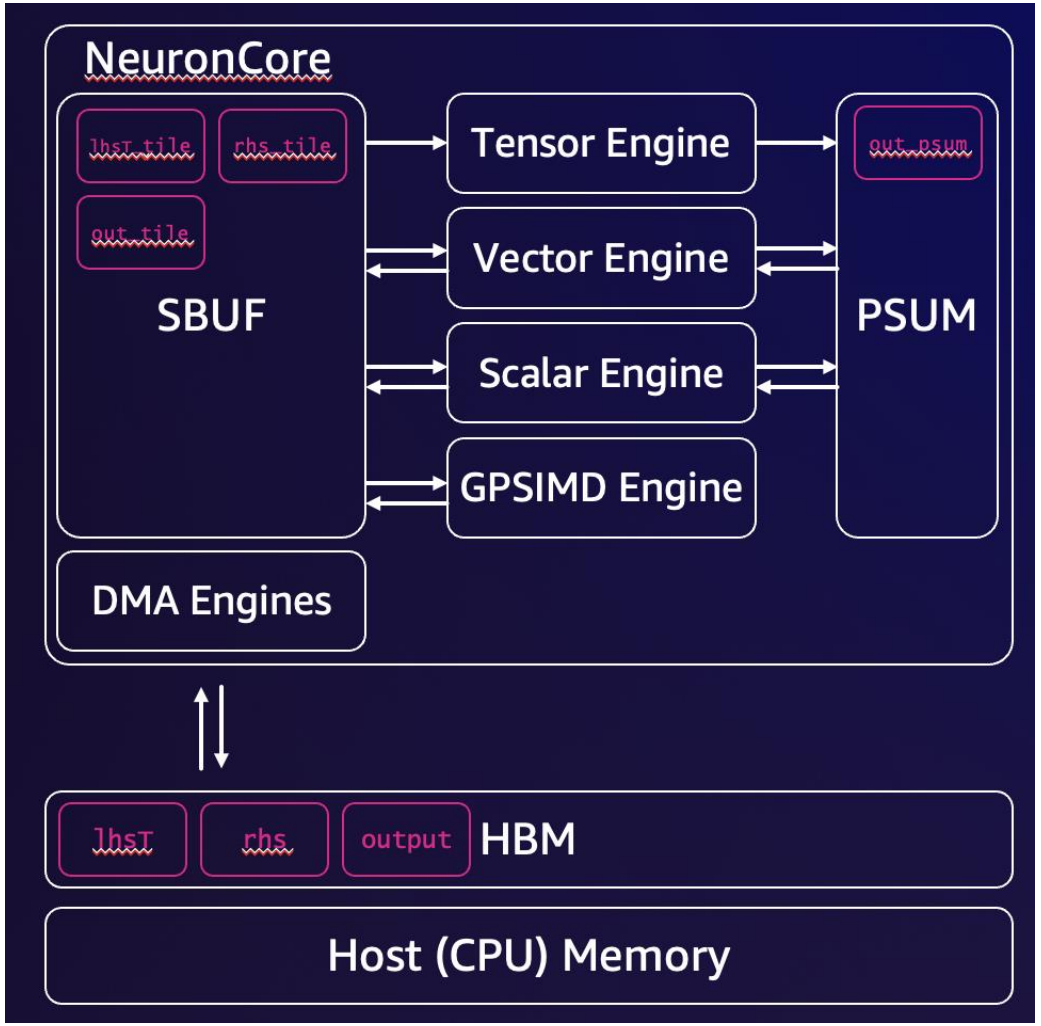
Neuron Kernel Interface (NKI) for Trn

- Python DSL for writing kernels, inspired by NumPy and Triton
- Integrates with PyTorch, JAX, and NumPy scripts
- Directly emits Neuron ISA instructions, enabling close-to-metal development
- Provides both high-level constructs (nki.language) and direct access to Neuron ISA (nki.isa)

```python
from neuronxcc import nki
import neuronxcc.nki.language as nl


@nki.jit
def nki_tensor_add_kernel(a_input, b_input):

    """NKI kernel to compute element-wise addition
    of two input tensors
    """
    assert a_input.shape == b_input.shape
    assert a_input.shape[0] <= nl.tile_size.pmax

    # Load the inputs from device memory to SBUF
    a_tile = nl.load(a_input)
    b_tile = nl.load(b_input)

    # Specify the computation (in our case: a + b)
    c_tile = nl.add(a_tile, b_tile)

    # Create a HBM tensor as the kernel output
    c_output = nl.ndarray(a_input.shape,
                          dtype=a_input.dtype,
                          buffer=nl.shared_hbm)

    # Store the result to device memory
    nl.store(c_output, value=c_tile)

    # Return kernel output as function output
    return c_output
```

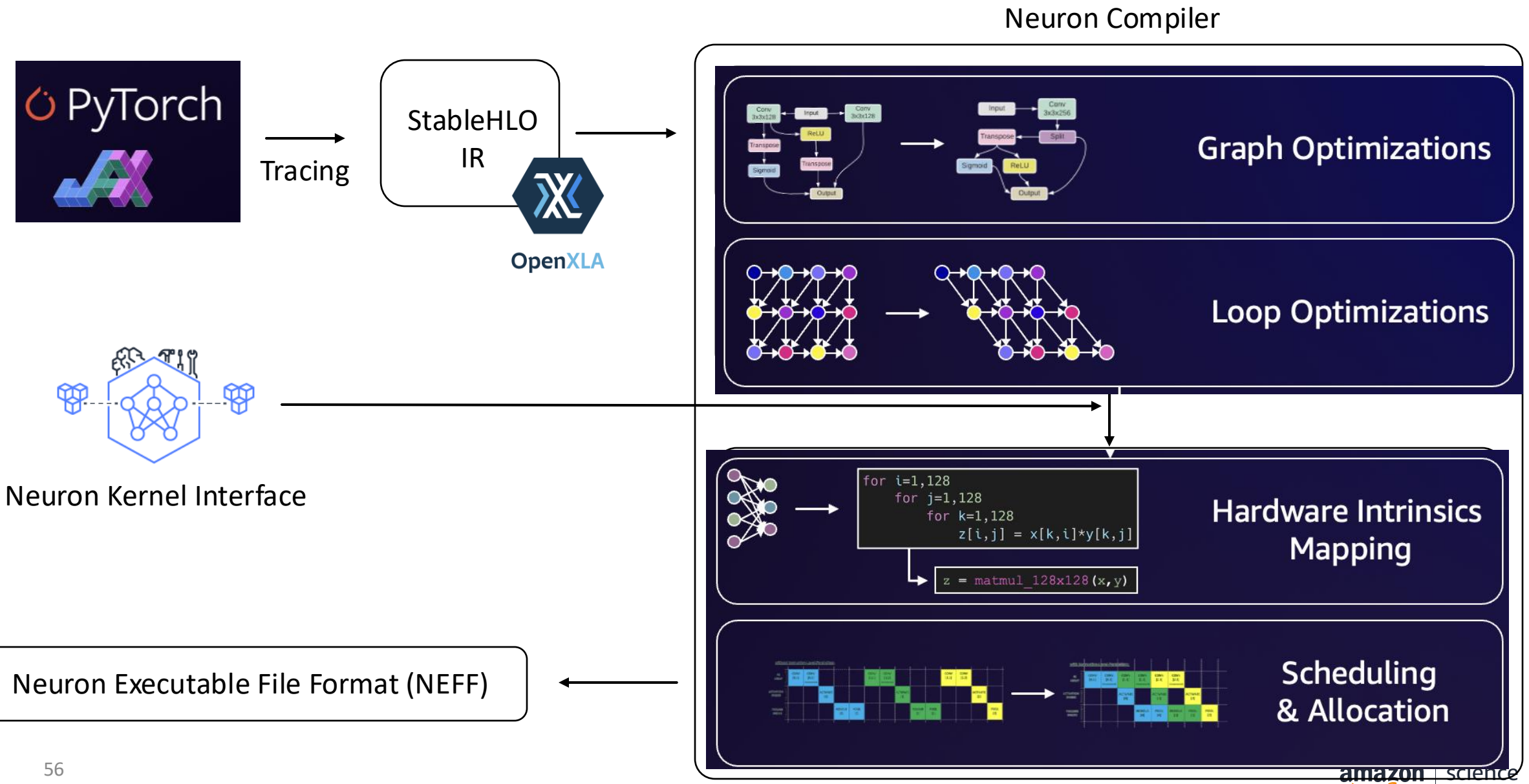
NKI for Matrix Multiplication

```python
@nki.jit
def nki_matmul_basic(lhsT, rhs):
    """Perform matrix multiplication: lhsT: (K, M), RHS: (K, N) → result: (M, N)."""
    _, M = lhsT.shape
    _, N = rhs.shape

    # Create a HBM tensor to store the kernel output
    result = nl.ndarray((M, N), dtype=lhsT.dtype, buffer=nl.shared_hbm)

    # Load the inputs from device memory (HBM → SBUF) using DMA engines
    lhs_tile = nl.load(lhsT)
    rhs_tile = nl.load(rhs)

    # Perform matrix multiplication using the Tensor Engine
    result_psum = nl.matmul(lhs_tile, rhs_tile, transpose_x=True)

    # Copy the result back to SBUF (PSUM → SBUF) through Vector/Scalar Engine
    result_tile = nl.copy(result_psum, dtype=result.dtype)

    # Store the computed result to device memory (SBUF → HBM) using DMA engines
    nl.store(result, value=result_tile)

    # Return kernel output as function output
    return result
```



There are lots of advanced techniques of tiling, indexing, looping, masking, etc. (not covered today)

Where does NKI fit in?



NKI API

Matrix & Tensor Operations

nisa.nc_matmul, nisa.transpose

Reduction Operations

nisa.tensor_reduce, nisa.tensor_partition_reduce, nisa.tensor_scalar_reduce, nisa.tensor_tensor_scan, nisa.activation_reduce, nisa.bn_stats, nisa.bn_aggr

Element-wise Operations

nisa.tensor_tensor, nisa.scalar_tensor_tensor, nisa.tensor_scalar, nisa.activation, nisa.reciprocal, nisa.dropout

Memory & Data Movement

nisa.tensor_copy, nisa.dma_copy, nisa.memset, nisa.iota, nisa.nc_stream_shuffle, nisa.tensor_copy_dynamic_[src|dst]

Selection & Filtering

nisa.affine_select, nisa.range_select, nisa.tensor_copy_predicated

Top-k & Indexing

nisa.max8, nisa.nc_find_index8, nisa.local_gather
nisa.nc_match_replace8

Configuration (Enums)

nisa.engine, nisa.reduce_cmd, nisa.dge_mode
nisa.nc_version

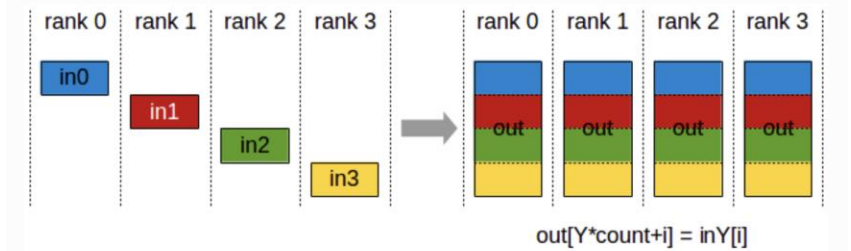
When to use NKI?

- **As an ML Practitioner:** To optimize performance hotspots after profiling.
- **As a Performance Engineer:** To implement high-performance hardware-aware model components.
- **NKI allows to iterate and innovate faster** by reducing dependency on:
  - **Compiler toolchain** – apply optimizations without waiting for compiler updates
  - **Framework-supported ops** – implement missing operators via custom kernels

- But **not** to use if
  - Model is performing well on Neuron: Profile your model first!
  - Don’t deeply understand the Neuron Device architecture
  - Because there is an existing kernel for another hardware architecture: A bottleneck occurring on a given hardware might not occur on Neuron hardware and vice-versa.

Collective Communication Primitives

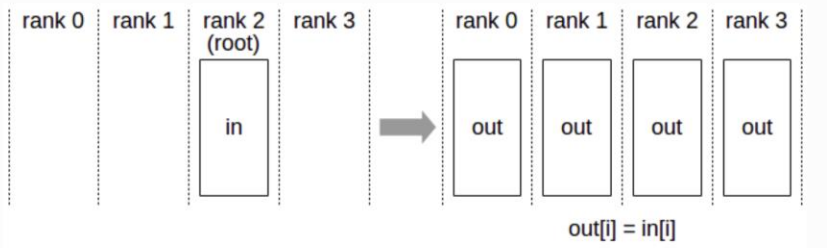
All-Reduce operation: each rank receives the reduction of input values across ranks.



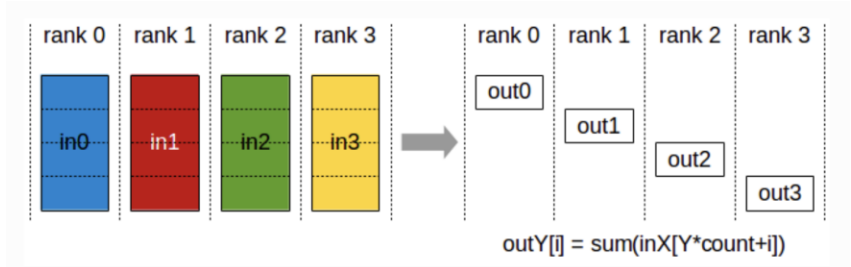
AllGather operation: each rank receives the aggregation of data from all ranks in the order of the ranks.

python

```
torch.distributed.all_reduce(...)
```



Broadcast operation: all ranks receive data from a "root" rank.



Reduce-Scatter operation: input values are reduced across ranks, with each rank receiving a subpart of

```
lax.psum(...)
```

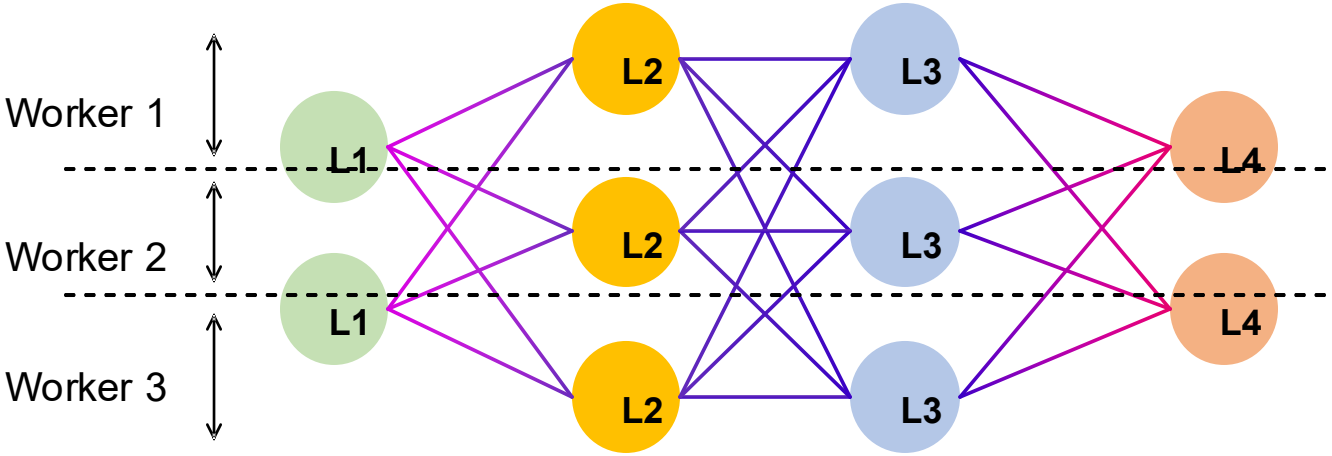
All-reduce API in Pytorch, JAX

Figures from https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html

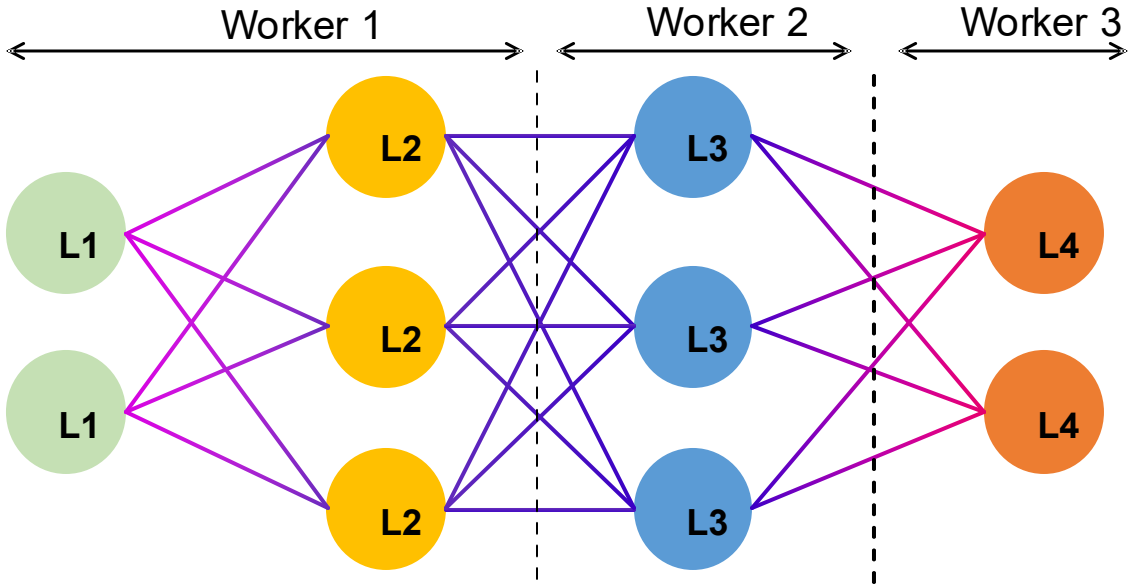
collective operations are available in system libraries that coordinate many such instructions over chips

Distributed Computation & Parallelism

Tensor Parallelism

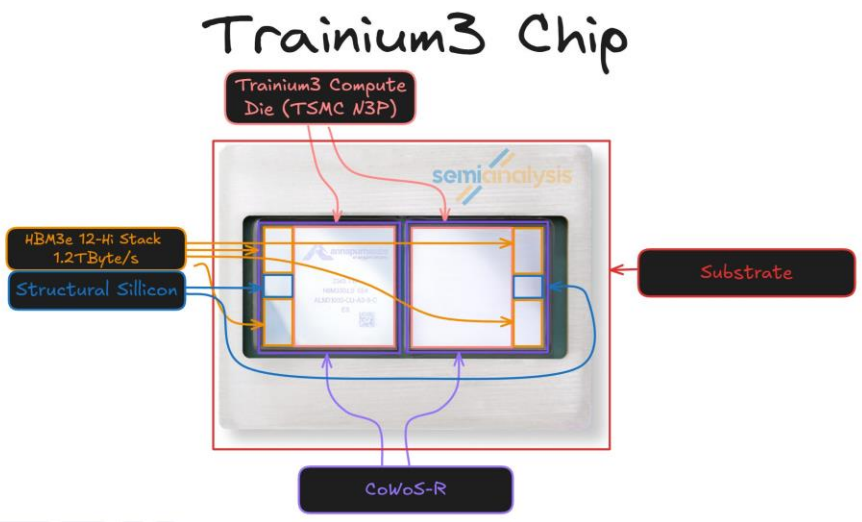


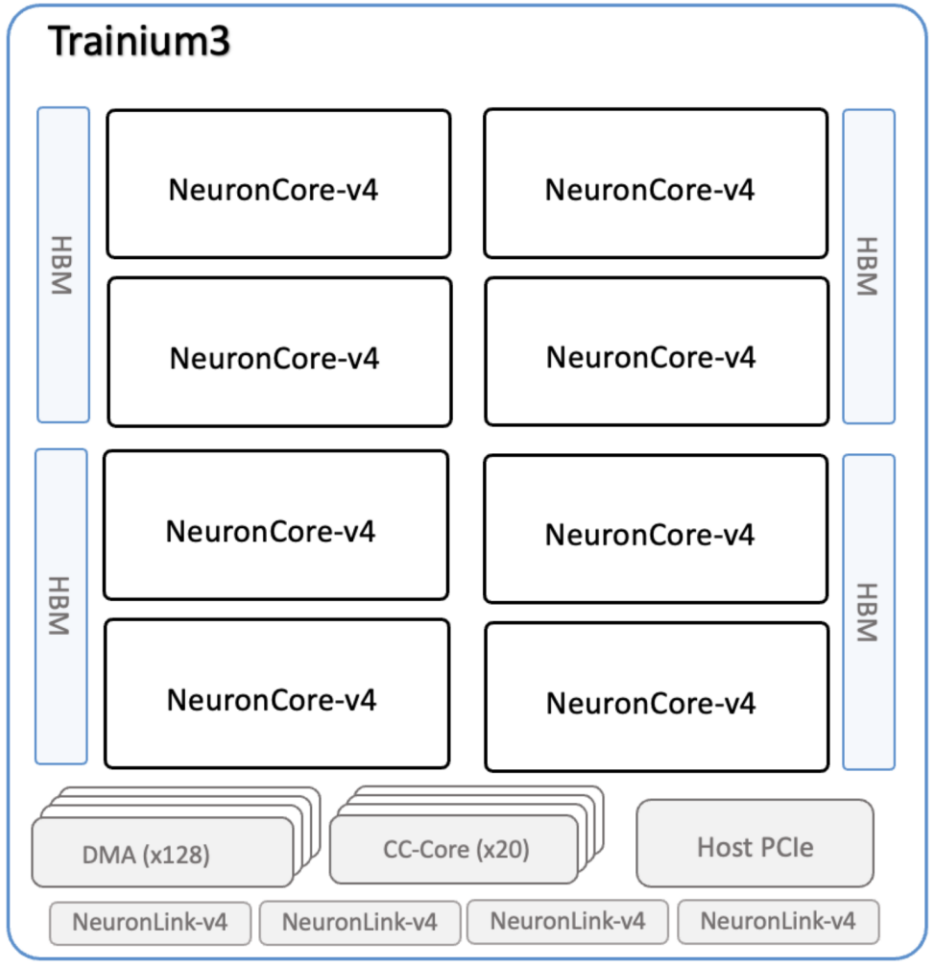
Pipeline Parallelism



Scale Up Networking: From Chip to Server

- Modern LLMs rarely run on a single chip in isolation.
- Model sizes, KV cache growth, and throughput requirements quickly exceed the capacity of one device
- Rectangular package with the compute silicon at the center and multiple HBM stacks placed closely around it, all mounted on a high-density substrate.

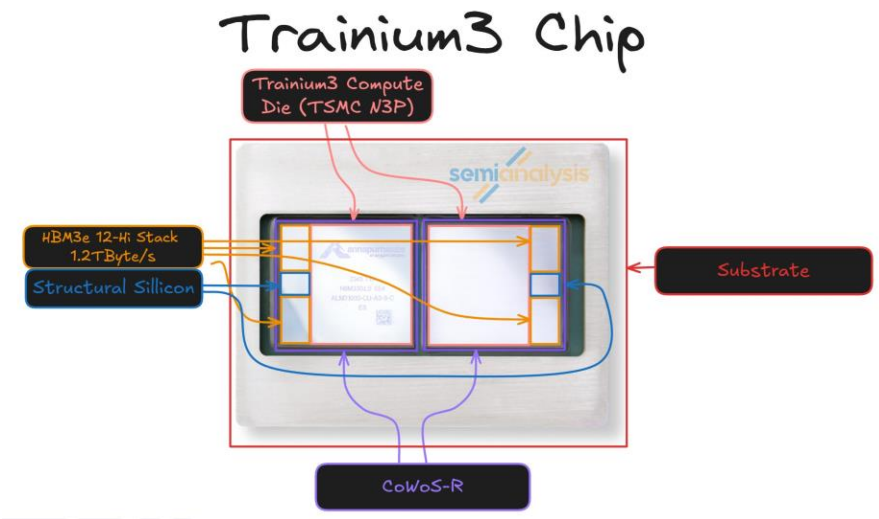


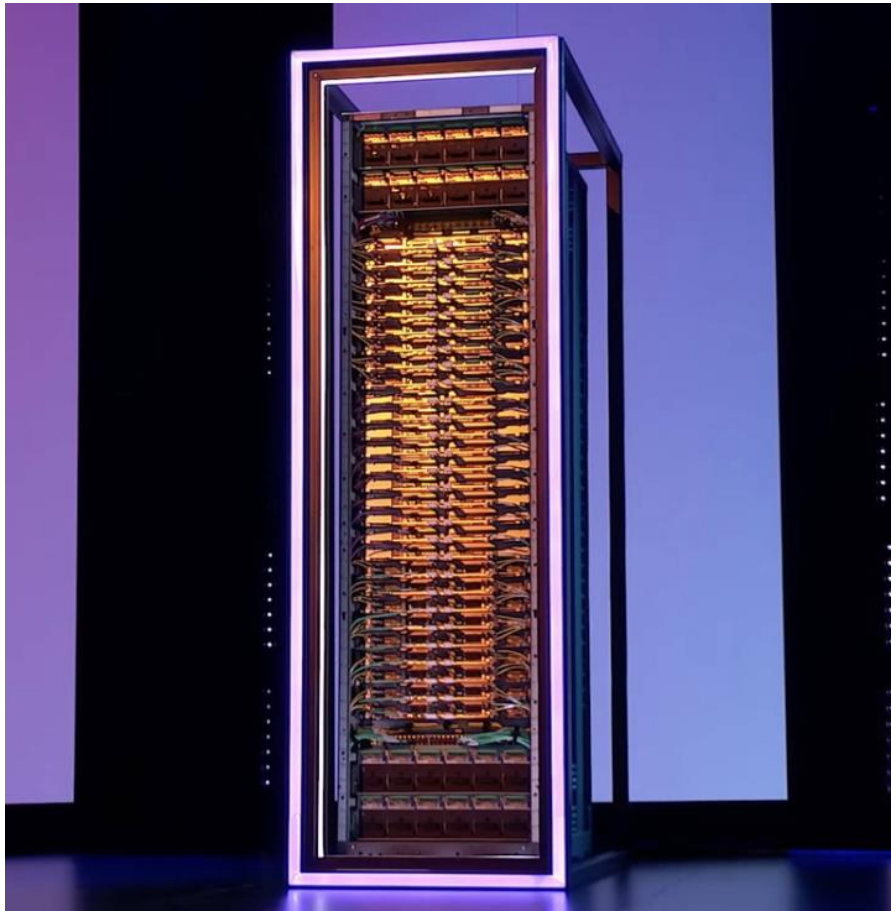


Trainium chip consisting of eight (8) NeuronCores and four (4) banks of HBM (SemiAnalysis & AWS Neuron Docs)

Scale Up Networking: From Chip to Server

- Each Trn chip is deployed as a PCIe accelerator card
- The card exposes high-speed links for host connectivity and device-to-device communication, allowing multiple chips to be wired together within a single server
- Board is designed to deliver power, cooling, and interconnect bandwidth at scale

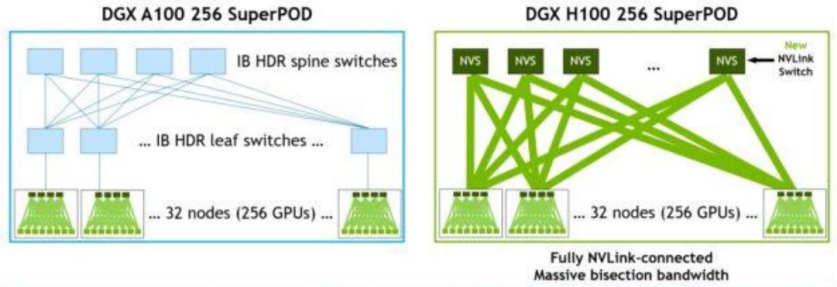




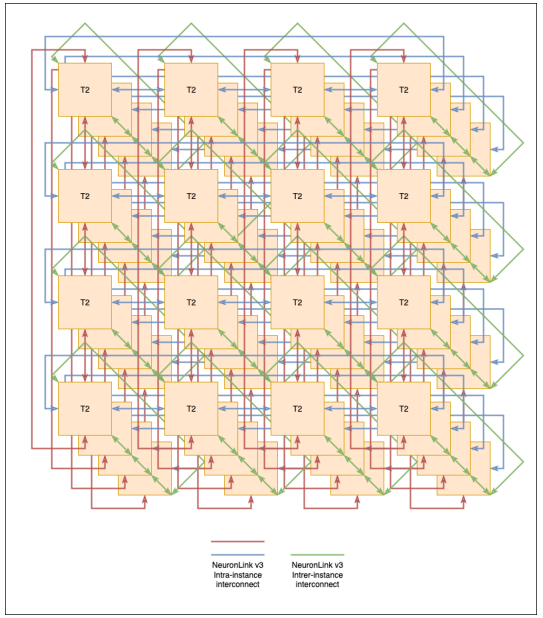
Trn3 ultraserver with several connected Trn chips (credits: AWS re:Invent, 2025)

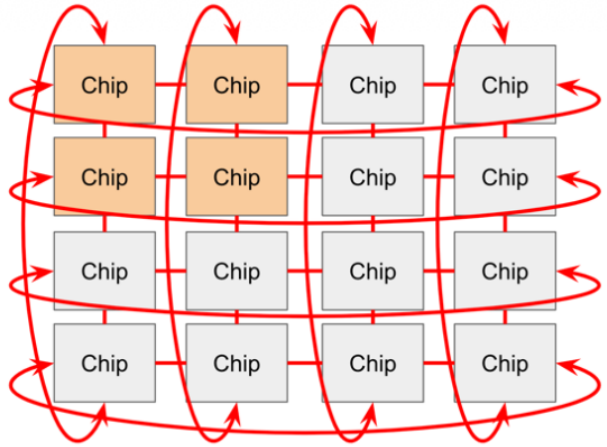
amazon | science

Different Networking Topology



	A100 SuperPod			H100 SuperPod			Speedup	
	Dense PFLOP/s	Bisection [GB/s]	Reduce [GB/s]	Dense PFLOP/s	Bisection [GB/s]	Reduce [GB/s]	Bisection	Reduce
1 DGX / 8 GPUs	2.5	2,400	150	16	3,600	450	**1.5x**	**3x**
32 DGXs / 256 GPUs	80	6,400	100	512	57,600	450	**9x**	**4.5x**



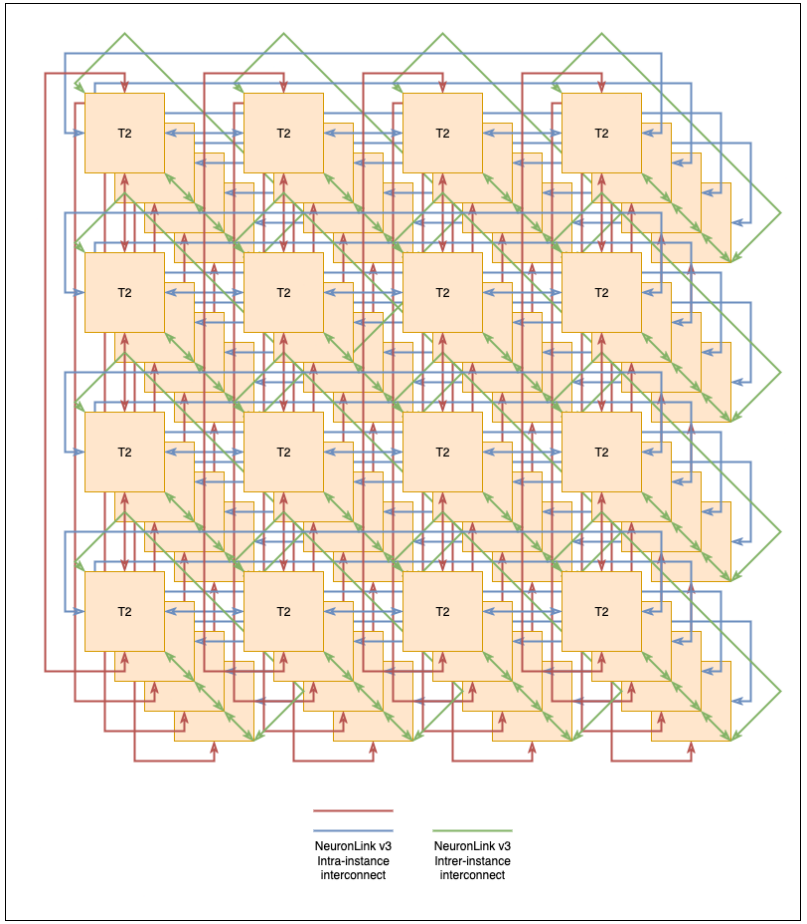


GPU: all-to-all (upto 256)

Trn/TPU: 3D Toroidal mesh

Direct Torus Topology for Scale-Up Networking

- **Torus topology**: Each accelerator is directly connected to a small, fixed set of neighbors (e.g., in 2D or 3D)
- Communication between distant devices proceeds by hop-by-hop routing through intermediate accelerators.
- Work particularly well for workloads with structured communication patterns -- such as pipeline or tensor parallelism
- Example: Prefill phase of LLM inference -- dominated by large batched matrix multiplications
- Each communication step transfers large tensors and is followed by substantial computation, allowing the network to be efficiently pipelined.
- Downside: Hop-based nature of a torus introduces latency that grows with distance

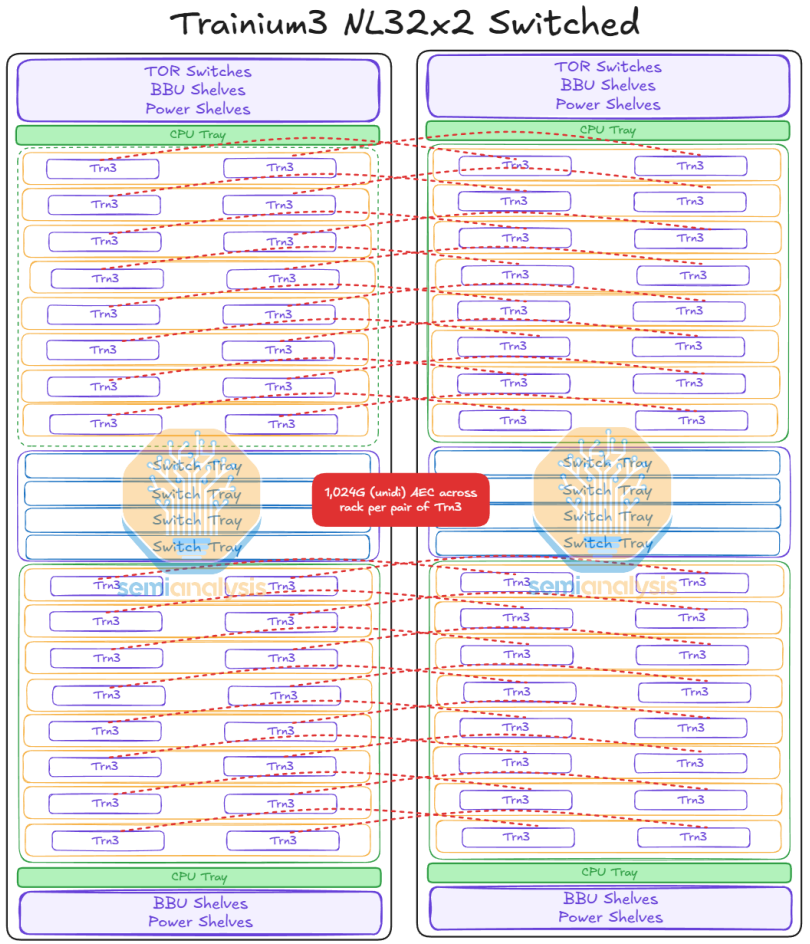


Scaling up with a 3D Torus (Trainium2)

Switched Fabric for Scale-Up Networking

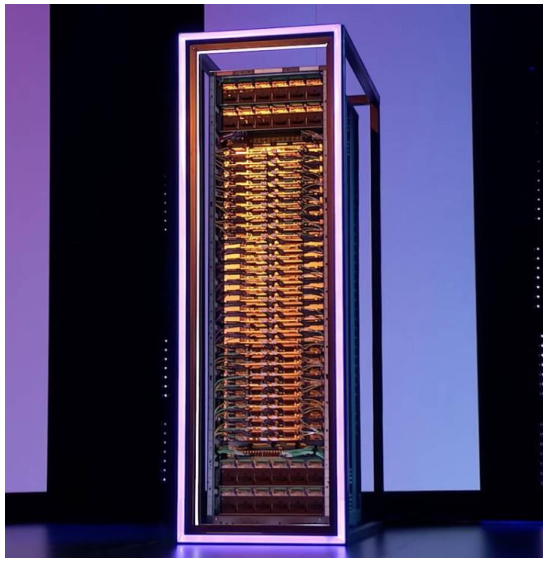
- Inference phases such as **autoregressive decoding** or **Mixture-of-Experts routing** involve smaller, more frequent, and often more global communication events
- Limitations of direct topologies become visible, motivating the use of **switched fabrics, unique feature of Trn3 compared with Trn2**
- **Switched fabric**: Each accelerator connects to one or more high-speed switches
- Any device can communicate with any other device in (ideally) a single hop
- More complex and costly: Additional switch hardware, cabling, and power, increasing system cost, complexity, and thermal overhead.

Accelerator design is heavily influenced by algorithmic and modeling designs

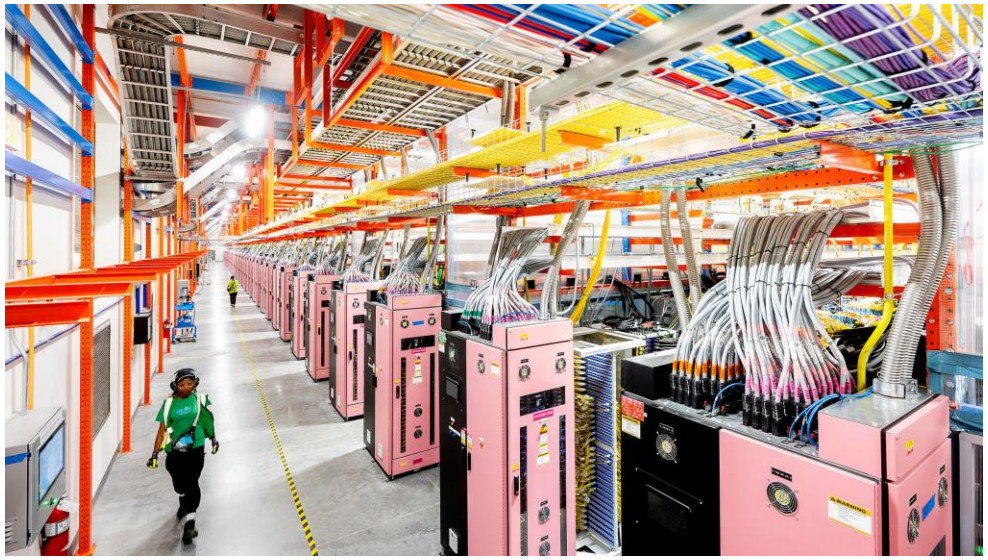


Schematic of a Trn3 NL 32x2 server showing two racks with compute trays, switching trays, CPU, and power sources

Scale Out Networking: From Server to Datacenter



A single Trainium server



Several accelerator-equipped servers are connected to form a single, datacenter-scale inference and training fabric.

- Scale-out networking connects multiple servers across a rack, cluster, or entire datacenter
- Essential once model size, serving throughput, or availability requirements exceed what a single node can provide
- Typically using high-speed ethernet fabric (e.g., Elastic Fabric Adapter (EFA))
- Performance is no longer dominated by FLOPs or memory bandwidth alone; instead, network latency, bandwidth

Overview of Performance Analysis

- **Roofline Analysis**
- **Performance Metrics**

Roofline Analysis: What and Why?

- When an algorithm is run on hardware, there are three dominant constraints:
  - How many floating-point operations the compute engines can perform per second (**Ops/sec**)
  - How many bytes per second the chip (or system) can move in memory or across an interconnect ("**bandwidth**")
  - How much memory you have in total (on-chip, off-chip, network) to hold data
- **Roofline model**: Upper and lower bounds on runtime. The time an operation takes cannot be less than the you'd get if you were *just* limited by compute

$$T_{\text{math}} = \frac{\text{Computation OPs}}{\text{Accelerator OPs/s}}, \qquad T_{\text{comm}} = \frac{\text{Communication bytes}}{\text{Memory Bandwidth bytes/s}}$$

- Upper bound on the latency of any operation: $T_{\text{math}} + T_{\text{comm}}$
- Usually computation can be overlapped with communication.
- With perfect overlap, lower bound on the latency: $\max\left(T_{\text{math}}, T_{\text{comm}}\right)$

Memory and Compute Bound Regimes

$$T_{\text{math}} = \frac{\text{Computation OPs}}{\text{Accelerator OPs/s}}, \qquad T_{\text{comm}} = \frac{\text{Communication bytes}}{\text{Memory Bandwidth bytes/s}}$$

- Assuming communication and computation overlap perfectly:
  - **Compute-bound** when $T_{\text{math}} > T_{\text{comm}}$ (Hardware fully utilized)
  - **Communication bound** when $T_{\text{math}} < T_{\text{comm}}$ (Compute capacity is idle while waiting for data)

$$\text{Accelerator intensity} = \frac{\text{Accelerator OPs/s}}{\text{Bandwidth bytes/s}}$$

$$\text{Computation intensity} = \frac{\text{Computation OPs}}{\text{Communication bytes}}$$

- **Accelerator intensity**: Characteristic of an accelerator hardware
- **Computation intensity**: Function of an operation/algorithm
- High computation intensity (i.e, A-intensity < C-intensity) → Compute-bound (Desirable)\

Memory and Compute Bound Regimes

$$\text{Accelerator intensity} = \frac{\text{Accelerator OPs/s}}{\text{Bandwidth bytes/s}}$$

$$\text{Computation intensity} = \frac{\text{Computation OPs}}{\text{Communication bytes}}$$

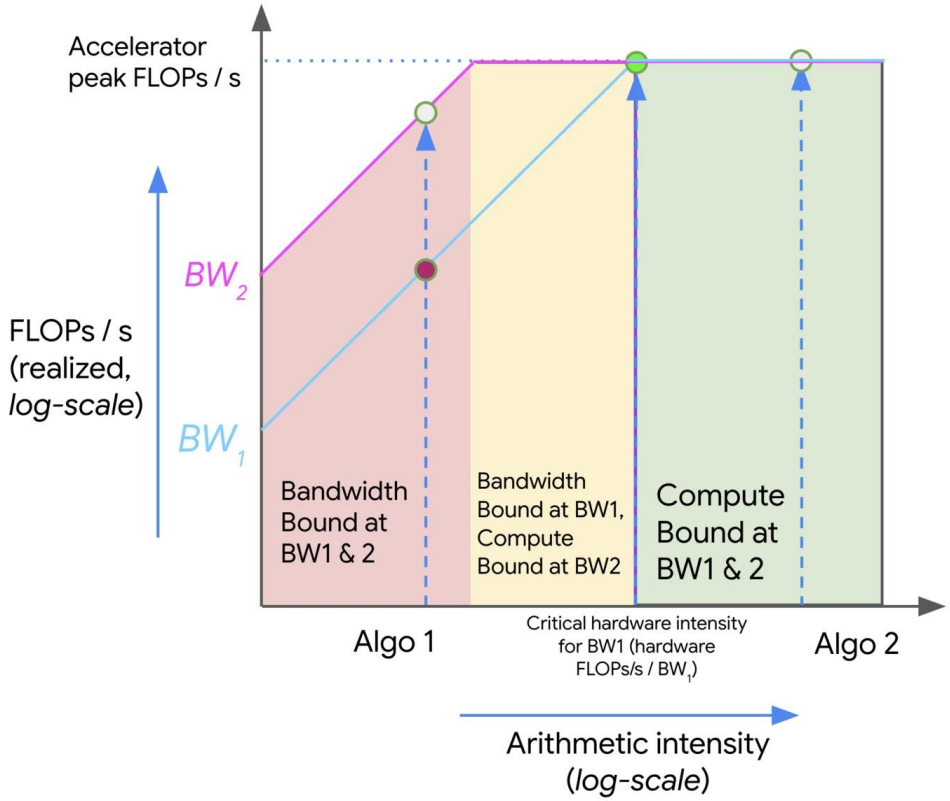
- Case study of matmul on TPU v5e MXU
  - TPU can perform 1.97e14 FLOPs/s and load 8.2e11 bytes/s from HBM. Accelerator intensity is 240 FLOPs/byte
  - For matmul between X (size of B x D) and Y (size of D x F), and assuming B << D, F
  - Compute intensity is

$$\text{Intensity(matmul)} = \frac{2BDF}{2BD + 2DF + 2BF} = \frac{BDF}{BD + DF + BF} \approx \frac{BDF}{DF} = B$$

  - To fall into compute bound, B > 240 is desirable for matmul algorithm

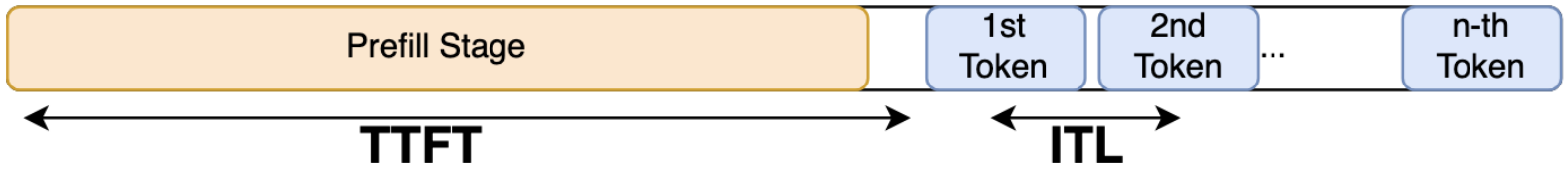
Visualizing Rooflines

- Two algorithms: Algo 1 and Algo 2, are plotted against two bandwidths (BW1 and BW2)
- For low computation intensity, performance is limited by how fast data can be moved (bytes/second)
- At high computation intensity, performance is limited by the peak OPs/sec of the hardware.
- **Algo 1 (memory-bound)**: Reducing bytes moved, increasing data reuse, improving memory locality, or boosting bandwidth, will improve performance more that increasing raw compute.
- **Algo 2 (compute-bound)**: Further increasing arithmetic intensity or bandwidth helps only marginally, as we are essentially limited by compute. Further gains require more compute capacity (or new precision formats, etc.)



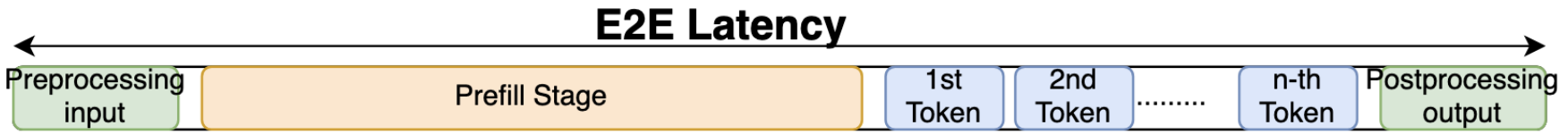
Visualization of roofline analysis (credits: How To Scale Your Model)

Performance Metrics



- Time-to-first-token (TTFT or **Prefill latency**): Elapsed time between submitting a prompt and receiving the first generated token in response
  - Captures the model's initial latency, encompassing tokenization, prompt encoding, etc.
  - Especially important for interactive or streaming applications such as chatbots: Perceived responsiveness directly affects user experience.
  - Generally, **compute-bound**: Latency is dominated by dense matrix multiplications and attention operations over long input sequences.
- OTPS (or **Decoding throughput**): How quickly an LLM generates tokens after the first token has appeared.
  - OTPS can also be measured by its inverse, **inter-token latency (ITL)**
  - 6 English words/second may be sufficient for chatbots. Higher is preferred for agentic communications
  - Generally, **memory-bound**: Need to repeatedly access and update large key–value (KV) caches

Performance Metrics



- For each request, E2E latency to complete inference including prefill, decoding, pre/post-processing
- Production systems such as chat services, or model endpoints handle many concurrent prompts arriving continuously, each with varying sequence lengths and response demands.
- **Throughput** captures this aggregate behavior by measuring the total number of tokens generated per second across all concurrent inferences.

$$\text{Throughput}_{\text{system}} = \frac{\text{Total tokens generated across all requests}}{\text{Total wall-clock duration}}$$

- Measures how effectively the model server converts hardware capacity into useful output
- Systems that overlap the prefill and decoding phases, such as those employing continuous batching (e.g., vLLM) maintain high throughput even when TTFT varies across requests.

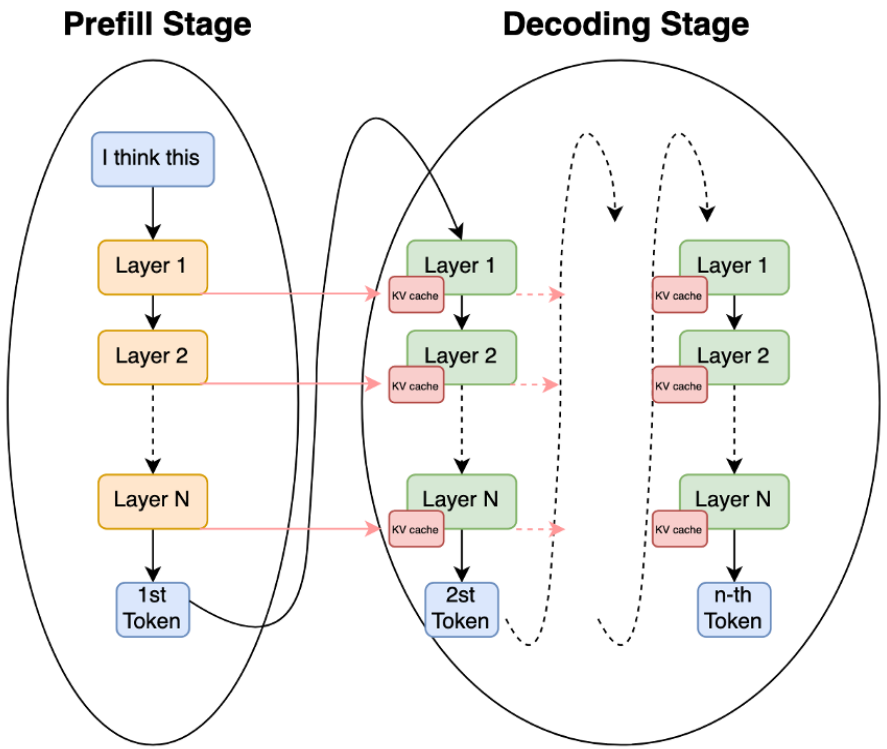
Tutorial Outline

- ~~Primer: Foundations of Generative Inference~~
- **Algorithmic and Modeling-Level Inference Optimizations**
- Systems-Level Optimizations
- Open-Source Frameworks and Tools

Algorithmic and Modeling-Level Inference Optimizations

- **Key-Value (KV) Caching**
- Inference-Aware Model Architectures
- Model Compression
- Speculative Decoding

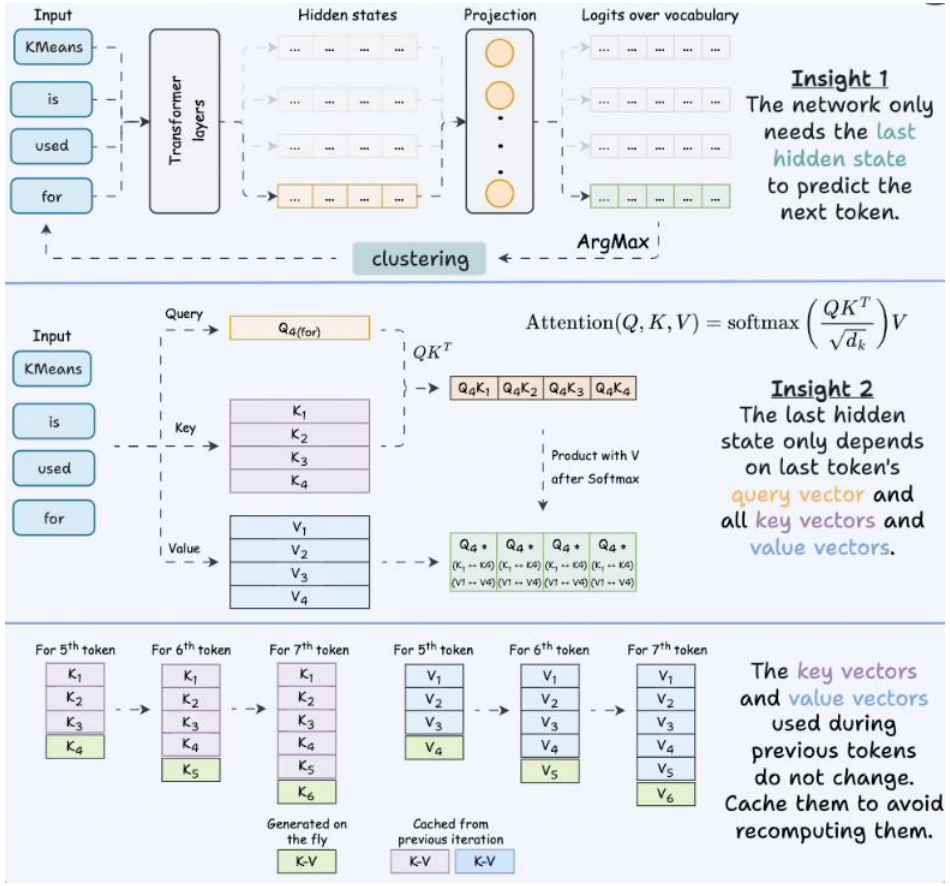
KV Caching in Autoregressive Inference



- Transformer inference has two phases:
  - **Prefill**: process the prompt
  - **Decoding**: generate tokens one-by-one
- Without KV caching, attention would be **recomputed over all past tokens** at every decoding step.
- This leads to **quadratic cost** in sequence length
- **KV caching stores keys and values once** and reuses them across decoding steps.
- Shifts autoregressive decoding stage of inference from **compute-bound** to **memory-bandwidth-bound**

KV caching avoids redundant computation and makes real-time decoding feasible

Casual Attention Enables Computation Reuse



KV cache is initially populated during prefill and subsequently appended during autoregressive decoding (image: dailydoseofds)

- During **prefill**:
  Each layer computes & stores **K and V for all input tokens**
- During **decoding**:
  Only **Q, K, V for the new token** are computed
  Past keys and values are **loaded from the KV cache**
- **Causality**: Current token depends only on past tokens
- Causal masking ensures:
  Past token representations **never change**
  Recomputing them would be redundant
- KV cache is incrementally appended as decoding proceeds

Past tokens are immutable, so their attention states can be reused

KV Cache: Compute and Memory

- **Compute scaling:**

$$\text{Cost (prefill)} = \mathrm{O}(LDT_{\text{input}}^2)$$

Prefill dominated by large GEMMs. Compute-bound. Determines TTFT

$$\text{Cost (without KV cache)} = \sum_{u=1}^{t+1} \mathrm{O}(uD) = \mathrm{O}(t^2 D)$$

Quadratic → **Linear** in sequence length

$$\text{Cost (with KV cache)} = \mathrm{O}(tD + D^2)$$

Decoding reads KV cache at every step. Memory-bound. Determines OTPS

- **Memory footprint:** $\text{KV Cache memory} = 2 \cdot b \cdot L \cdot T \cdot D$

Linear growth in sequence length

Key implications:

- Prefill and decoding stress different system bottlenecks.
- Optimizing inference requires separate strategies for TTFT and OTPS

Example: Memory Requirement

- **Model parameters**: #parameters x #precision (bytes)
- **KV-cache per token**: 2 x hidden_dim (D) x #layers (N) x #precision (bytes)
- **Others**: intermediate tensors, CUDA reserved memory, fragmentation
- **Example** with **Llama2-7B** on **NVIDIA RTX-4090** with **24 GiB** memory
  - Parameters: 7B x 2 (float16) = 14 GiB
  - KV-cache per token: 2 x 4096 x 32 x 2 (float16) = **0.5 MiB/token**
  - Assume: 1GB of cache memory, we can host Llama-2-7B on one RTX-4090 with at most 2k tokens
  - Note: 5k tokens (for remaining 10 GiB) can only serve < 5 requests for 1k sequence length

Algorithmic and Modeling-Level Inference Optimizations

- ~~Key-Value (KV) Caching~~
- **Inference-Aware Model Architectures**
- Model Compression
- Speculative Decoding

Inference-Aware Model Architectures

Inference cost is often dominated by **attention** and **feed-forward** layers

- **Efficient attention architectures** limit the **quadratic scaling** in **sequence length**
- **Mixture-of-Experts (MoE)** increases model capacity via sparse compute – only a small subset of parameter are activated per token (FFN layers)

General goal: **Lower latency** and **memory footprint** without sacrificing quality

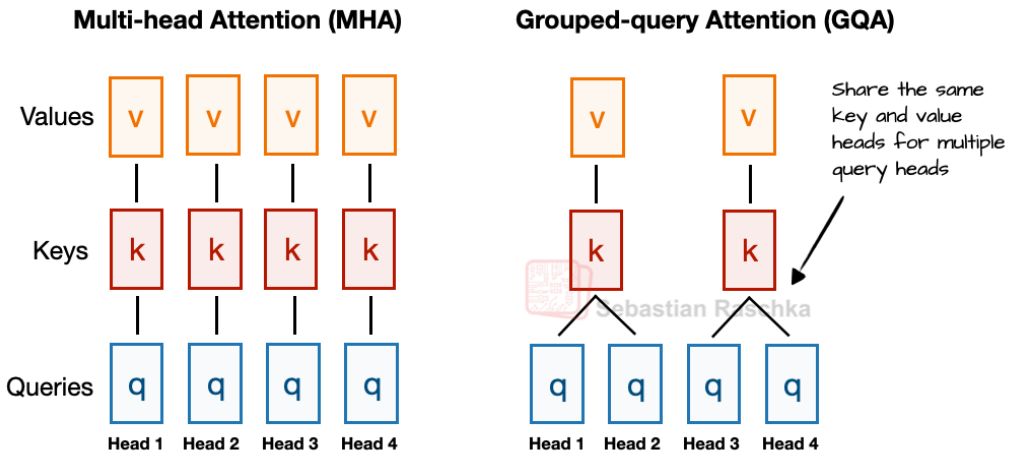
Key Question: How do these choices translate into real inference gains ?

Efficient Attention Variants

- Standard Multi-Head Attention (MHA) is expressive but costly.
- Inference bottlenecks arise from KV cache memory, bandwidth, and scaling with context.
- Efficient attention variants trade redundancy, structure, or locality for efficiency.
- Focus on three representative design dimensions:
  - **Parameter sharing** (MHA → Grouped Query Attention (GQA))
  - **Latent compression** (Multi-head Latent Attention (MLA))
  - **Sparsity and Locality** (Sliding window & Sparse attention)

Multi-Head vs. Grouped Query Attention

- **Multi-Head Attention (MHA)** computes attention using multiple parallel heads – each head has its own query, key, and value projections
- This allows different heads to focus on different positions or patterns in the sequence
- **Grouped Query Attention (GQA)** modifies MHA by reducing the number of key/value heads
- Query heads remain unchanged; **keys and values are shared across groups of queries**
- Designed to improve **inference efficiency** *(next slide)* while preserving expressivity



Multi-Head vs. Grouped Query Attention

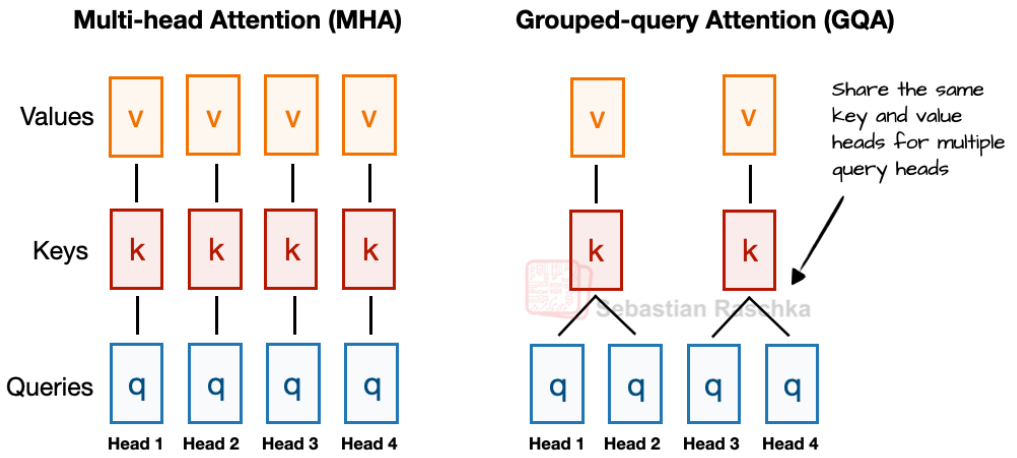
MHA: One KV head per query head vs. GQA: multiple query heads share the same KV head

$$\text{KV cache} \propto B \times T \times L \times d_{\text{head}} \times G$$

- B: batch size
- T: sequence length
- L: number of layers
- $d_{\text{head}} = d_{\text{model}}/H$
- G: number of KV heads

Inference impact:

- KV cache memory reduces by a factor of H/G compared to MHA
- Lower KV cache → faster autoregressive decoding
- Enables long context or larger models under fixed memory budgets (Llama-2 70B and Llama-3)



MHA vs. GQA (image: Sebastian Raschka)

Multi-Head Latent Attention (MLA)

MLA saves KV cache memory further (than GQA) by compressing the KV into a latent space before caching

Latent vector per token:

$$Z_t = X_t W_Z, \quad W_Z \in \mathbb{R}^{d \times d_\ell}, \quad d_\ell \ll d.$$

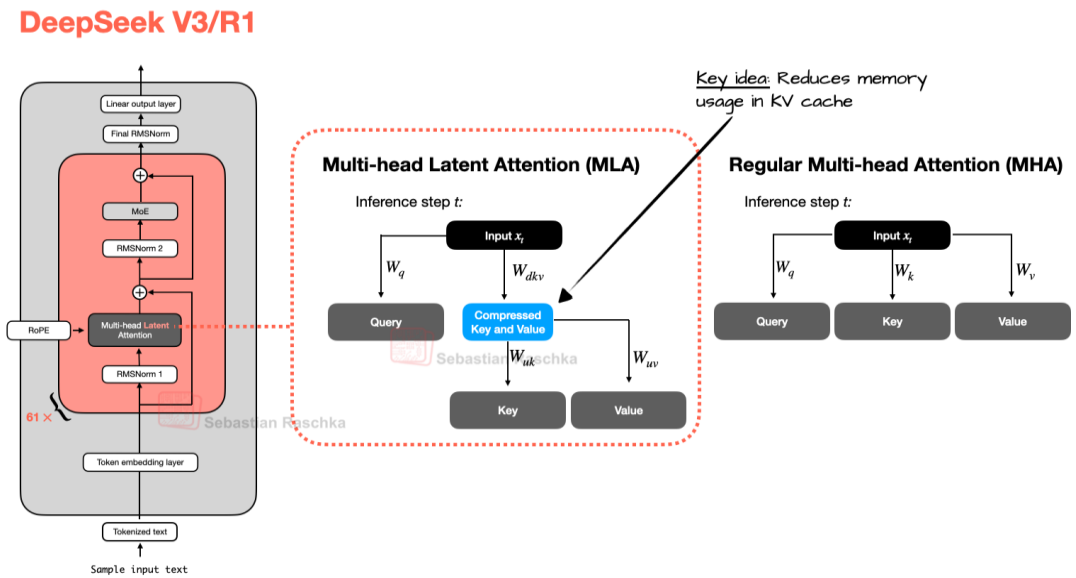
Cache latent and reconstruct:

$$K_t^{(h)} = Z_t W_K^{(h)}, \quad V_t^{(h)} = Z_t W_V^{(h)}$$

Queries remain head-specific

Inference impact:

- KV cache scales with **latent dimension**
- Extra projection FLOPs for reconstruction during inference are negligible since decoding is **memory-bound**



MLA was introduced in DeepSeek (image: Sebastian Raschka)

Sliding Window Attention (SWA)

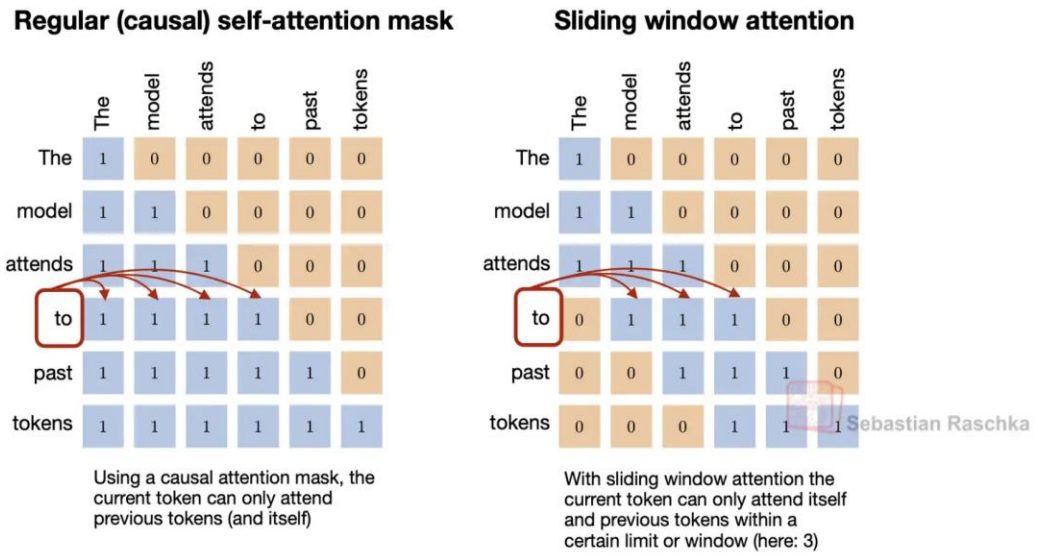
SWA reduces the cost of attention computation by **restricting attention to a fixed local window**

Full attention allows each token to attend to all previous tokens, resulting in **quadratic cost** in sequence length

SWA compute complexity: $\mathcal{O}(N^2) \rightarrow \mathcal{O}(Nw)$

KV cache memory reduction factor:

$$W / \text{sequence length}$$

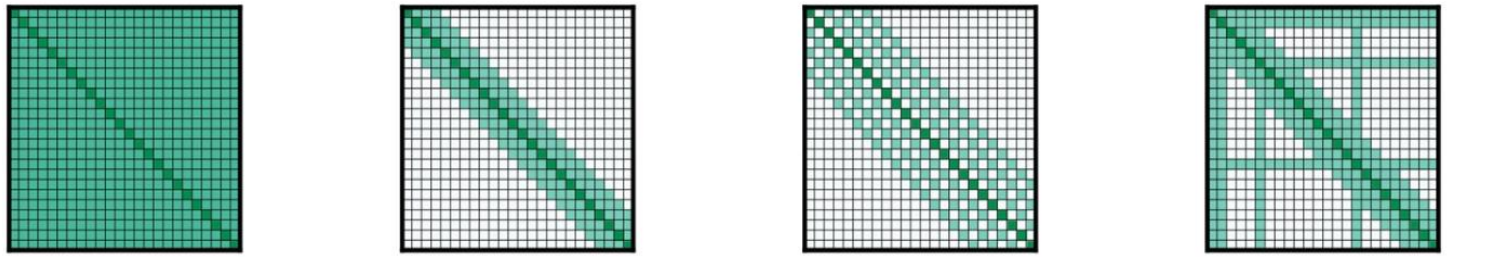


SWA is used in GPT-OSS model (image: Sebastian Raschka)

Inference impact:

- SWA trades global context for locality, yielding linear scaling in memory and compute.
- Faster prefill and decoding

Sparse Attention



(a) Full n^2 attention (b) Sliding window attention (c) Dilated sliding window (d) Global+sliding window

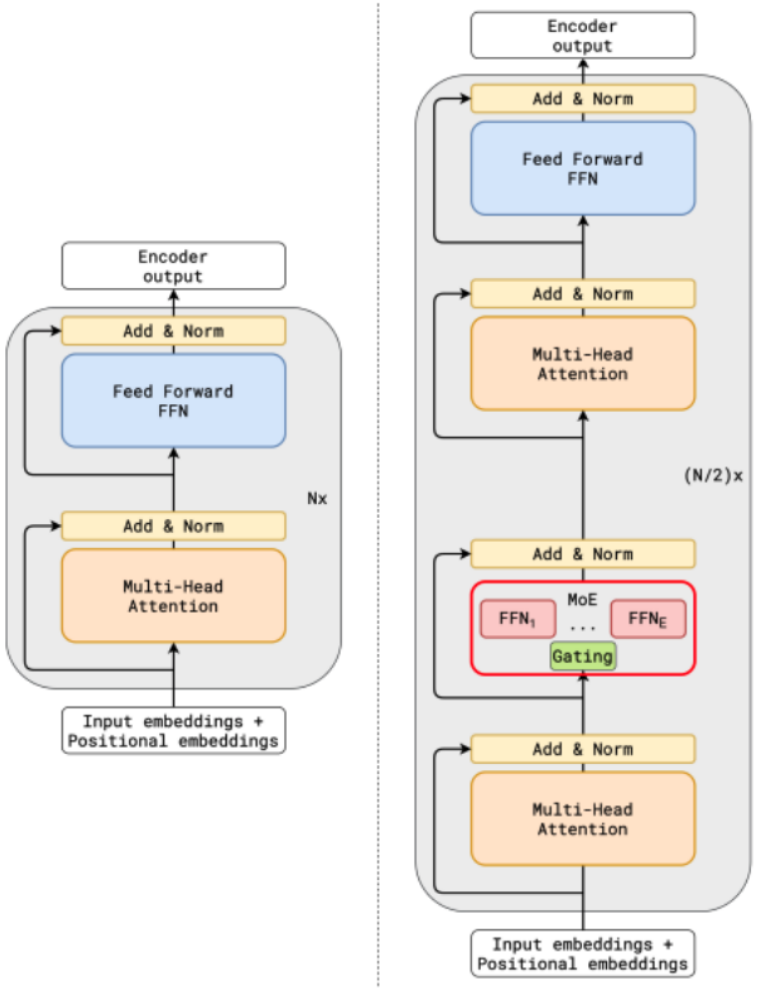
Sparse attention structures extend SWA

- **From locality to general sparsity**: Sliding window is a **specific sparsity pattern**: local, uniform, regular. Sparse attention allows **arbitrary but structured subsets** of token interactions
- **Key idea**: Reduce the effective size of the attention matrix while preserving important context, by restricting which token pairs can attend to each other.
- **Increases expressivity**: Combine local windows with **global, strided, or block-structured links.**
- **Additional systems challenges**: Irregular sparsity → non-uniform memory access, and often requires **specialized kernels and runtime support**

Mixture-of-Experts: Conditional Computation

- Efficient attention reduces inference cost by **limiting or compressing attention**
- However, **feed-forward (FFN) layers still dominate per-token compute**
- Scaling dense models increases **quality and cost together**
- **Mixture of Experts (MoE)** addresses this by activating **only a subset of parameters per token**

MoE: Architecture and Scaling



- In dense transformers, **every token executes all FFN parameters**
- Scaling dense models improves quality but **linearly increases inference cost**
- **MoE replaces FFN layers** with a bank of independent MLPs
- A lightweight **router (gate)** selects the top-k experts per token (typically k=1 or k=2)

$$g = W_g h, \quad \text{where} \quad W_g \in \mathbb{R}^{E \times d}$$

$$G(h) = \text{Softmax}\left(\text{TopK}(g, k)\right)$$

- Only the selected experts are executed → **conditional computation**

$$\text{MoE}(h) = \sum_{i=1}^{n} G(h)_i E_i(h), \text{ where } E_i(h) \text{ is the output of expert } i$$

Top-K Routing

- Router computes expert scores: $g = W_g h, \quad \text{where} \quad W_g \in \mathbb{R}^{E \times d}$
- Select top-K experts and normalize routing scores: $G(h) = \text{Softmax}\left(\text{TopK}(g, k)\right)$
- MoE output: $\text{MoE}(h) = \sum_{i=1}^{n} G(h)_i E_i(h), \text{ where } E_i(h) \text{ is the output of expert } i$

- **Implications for inference**:
  - **Sparse activation**: Only K experts are evaluated per token. Constant compute (independent of E)
  - **Dynamic control flow**: Expert selection varies per token. Prevents fixed execution graphs
  - **Routing overhead**: Tokens must be routed to remote experts (all-to-all). Often dominates latency.
  - **Fused MoE kernels**: Groups tokens by experts. Then, builds compact expert batches and ensures kernel launch overhead is minimized.

MoE inference requires fast expert compute, and efficient routing and communication to ensure sparsity actually pays off

MoE: Inference Efficiency

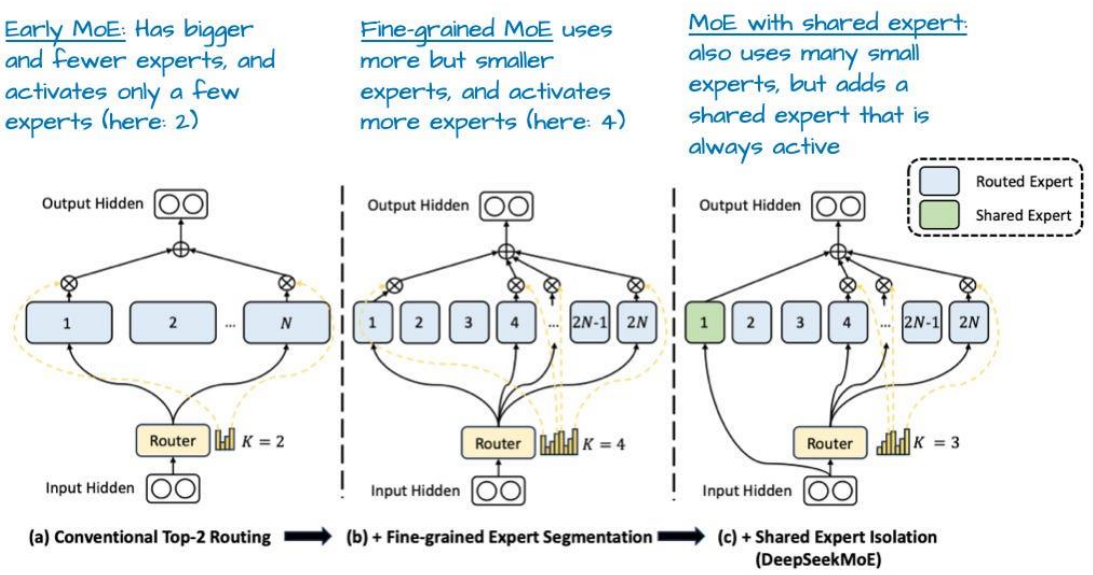
Compute scaling: Dense FFNs scale as $\mathcal{O}(3rd^2)$ whereas MoE FFNs scale as $\mathcal{O}(3krd^2)$, keeping per-token compute constant as the number of experts increases.

Parallelism strategies:

- **Expert Parallelism (EP):** shard experts across devices, route tokens via all-to-all
- **Tensor Parallelism (TP):** shard large experts across devices using collectives
- Choice depends on expert granularity: many small (DeepSeek) vs few large experts (Llama-4)

Inference Optimizations:

- **Fused MoE kernels** combine gate, up, activation, and down projections
- **Quantized expert weights** (e.g., INT4 / MXFP4) reduce HBM bandwidth pressure



Algorithmic and Modeling-Level Inference Optimizations

- ~~Key-Value (KV) Caching~~
- ~~Inference-Aware Model Architectures~~
- **Model Compression**
- Speculative Decoding

Model Compression

- **Motivation**
- Model Quantization
- Key-Value (KV) Cache Compression
- Knowledge Distillation

Model Compression: Motivation

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**

Current trend: Larger models tend to do better on benchmarks. Image source: The Llama 3 Herd of Models (https://llama.meta.com/)

Compression is not just about making models smaller – it's about making inference practical !

- Larger models tend to do better on benchmarks (scaling laws)
- At inference time, this leads to high **memory footprint, bandwidth pressure, and latency**
- Model compression reduces **storage, memory traffic, and compute** without retraining from scratch

Qwen3-235B-A22B-Instruct

- ≈ 470 GB in native precision (BF16)
- Need at least two P4d instances (each with eight 40 GiB A100 GPUs, i.e., 320 GiB)
- Annual on-demand cost: $384,739 (at $21.96 per hour)

Model Compression: Motivation

Challenge: Inference is expensive and slow with large models

- **Slower decoding (higher OTPS)**

  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.

Model Compression: Motivation

Challenge: Inference is expensive and slow with large models

- **Slower prefill (higher TTFT)**

  Number of flops increases as the model size increases

  Attention compute grows linearly with number of layers and model dimension but quadratically in input sequence length

  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.

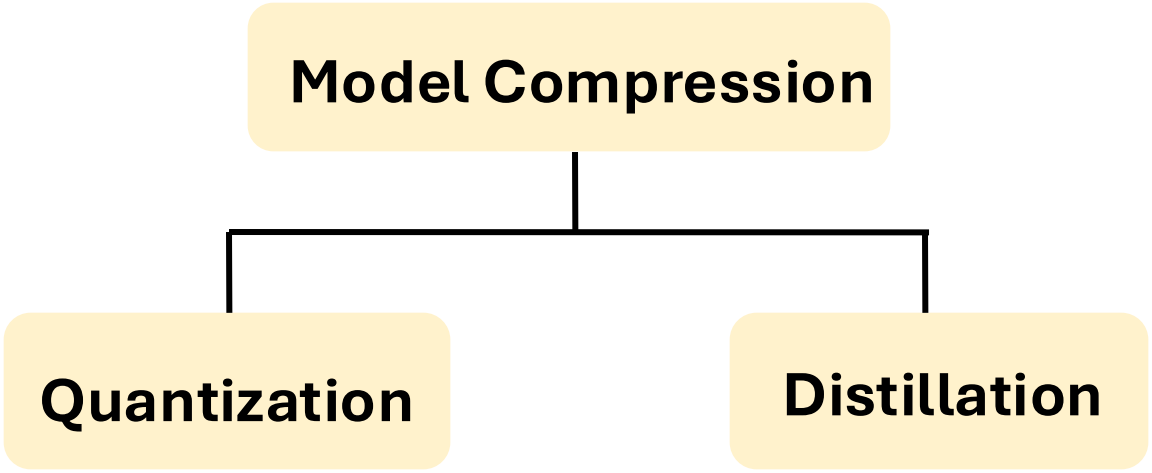
Model Compression

- Compress a large model into a smaller model:

  Quantization: Use low-precision data types

  Distillation: Use fewer parameters

- Trade-off between **accuracy** and **performance**
- Reduces the memory footprint of LLMs: Applies to *all* model compression methods (helps in memory-bandwidth bound regimes)
- Reduces the compute FLOPs: Applies to *some* compression methods (helps in compute-bound regimes)



Model Compression

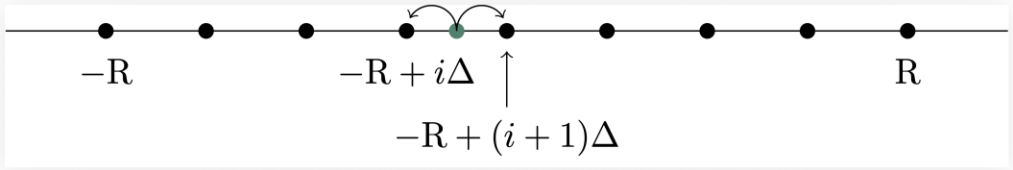
- ~~Motivation~~
- **Model Quantization**
- Key-Value (KV) Cache Compression
- Knowledge Distillation

Model Quantization

- **Low-Precision Data Types: What and Why?**
- **Post-Training Quantization (PTQ)**
- **Quantization-Aware Training/Distillation (QAT/D)**

What is Model Quantization?

- Quantization maps **high-precision** values to **lower-precision** representations
- Reduces:
  - Model size and memory footprint
  - Memory bandwidth during inference
  - Energy and Latency
- Widely used for LLM inference deployment
- Core idea: Mapping to a finite set $Q : S \to T, \quad |T| \ll |S|$



Uniform scalar quantization (B bits → 2^B discrete points)

Fewer bits per parameter → cheaper inference.

How Quantization Works (Intuition)

- Implemented via **scaling + rounding**

$$x_q = Q(x) = \text{round}(x/\Delta) - z$$

- Dequantization reconstructs:

$$\hat{x} = \Delta \cdot (x_q + z)$$

Δ: scale (step size)

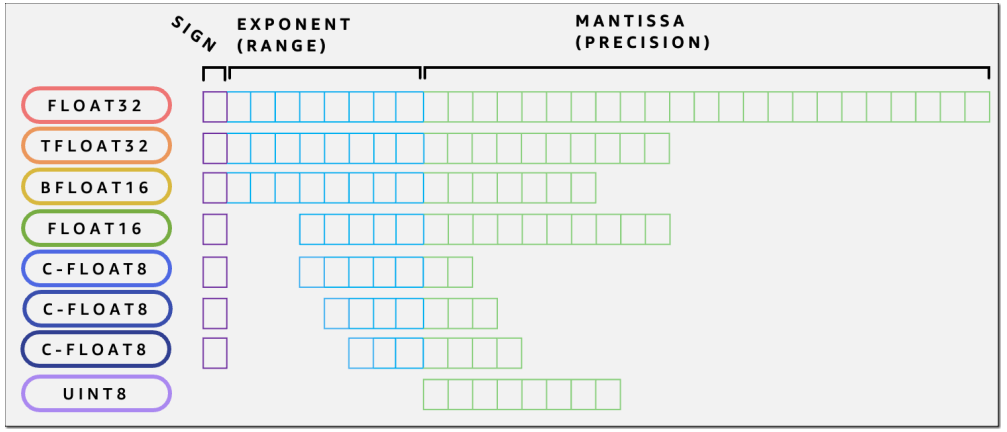
z: zero-point (for asymmetric ranges)

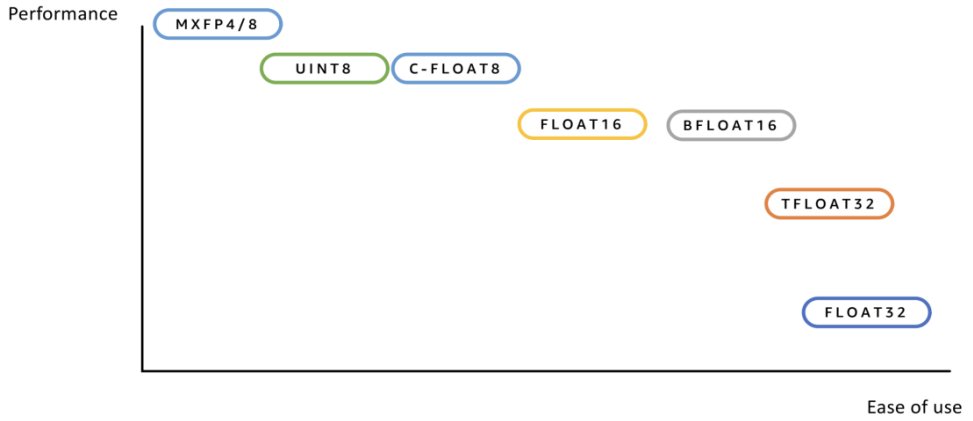
Quantization error: $\epsilon = |x - \hat{x}|$

Compression is achieved by trading numerical precision for efficiency

Data Types and Precision Trade-Offs

- Several different precision formats supported by current hardware
- Floating-point formats differ in:
  - **Exponent bits** → dynamic range
  - **Mantissa bits** → precision
- Common inference formats:
  - FP16, BF16
  - FP8 (E4M3, E5M2) (Trn3/Blackwell: MX formats)
  - INT8 / INT4
- Fewer mantissa bits → larger rounding error
- Fewer exponent bits → overflow / underflow risk

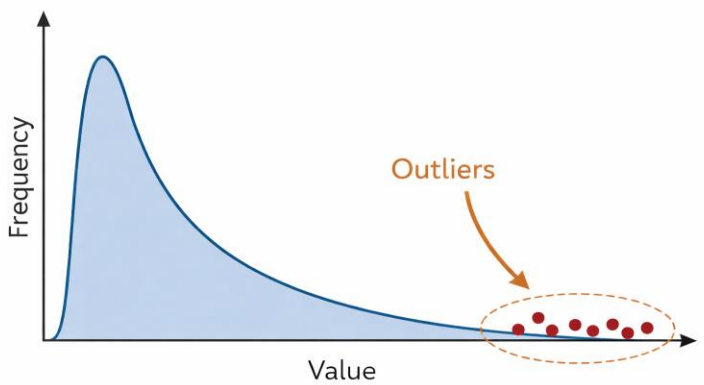




Quantization is not just about bit-width, but *where* the bits are spent.

Where Quantization Error Comes From

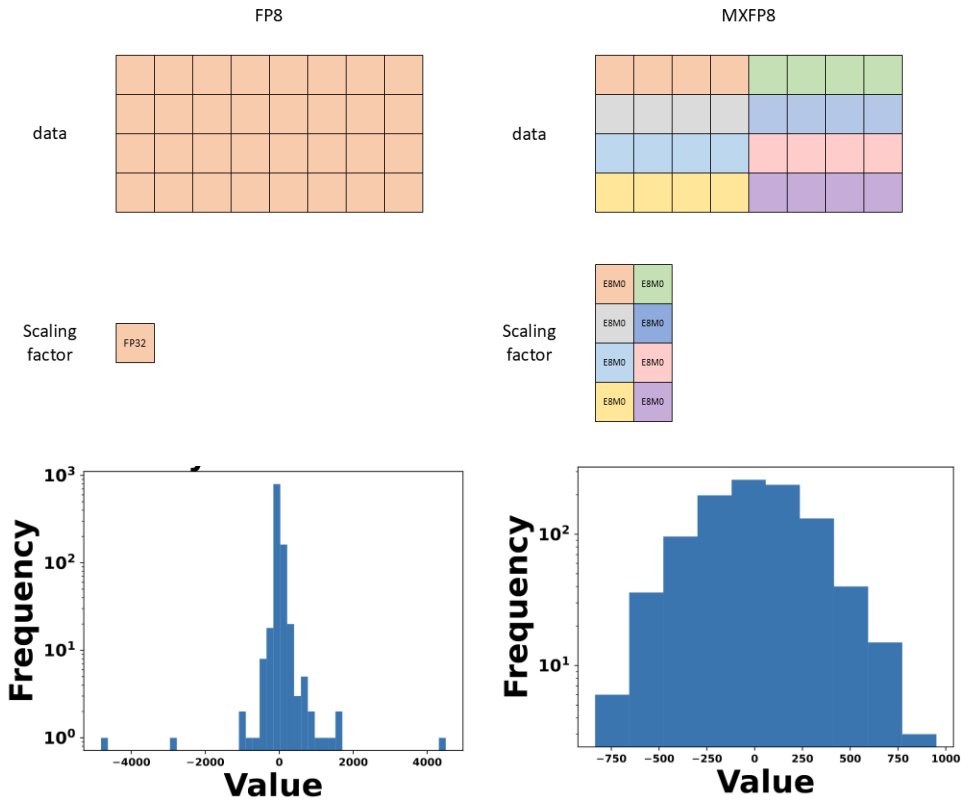
- **Reduced dynamic range**
  - Values exceeding the target range **saturate** (overflow)
  - Small-magnitude values **collapse to zero** (underflow)
  - Particularly harmful for activations with long-tailed distributions
- **Outliers**
  - Rare large values dominate scale selection
  - Force coarse quantization for the majority of values
  - Even a small fraction of outliers can cause large accuracy loss
- **Reduced precision**
  - Limited mantissa bits cause **rounding collisions**
  - Distinct values map to the same quantized value
  - Errors accumulate across layers in deep transformers



Uncontrolled quantization can silently break model behavior.

Controlling Quantization Error

- **Granularity choices**
  - Tensor-wise vs channel-wise vs group-wise
- **Micro-scaling (MX) formats**
  - Fine-grained scaling (blocks of 16–32)
  - Prevent single outliers from dominating an entire tensor
  - Increasingly supported by modern accelerators (Trn3/Blackwell). GPT-OSS MoE weights in MXFP4.
- **Distribution shaping**
  - Random Hadamard Transform (RHT) spreads large values across dimensions
  - Reduces peak magnitudes without changing model semantics
  - Improves utilization of the quantization codebook



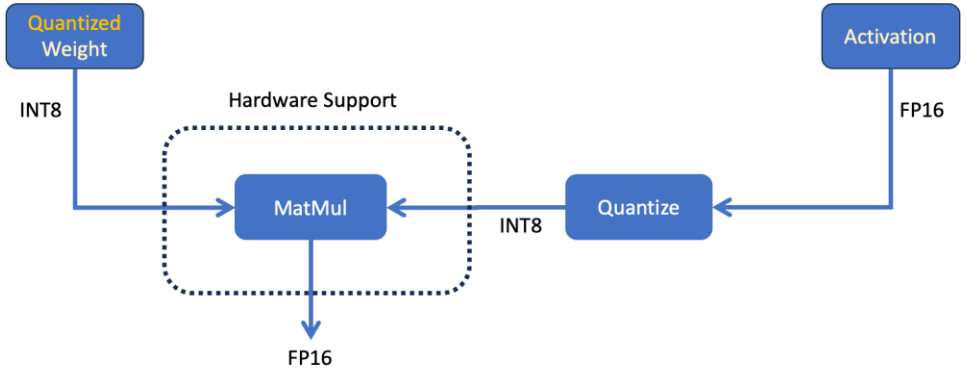
Heavy-tailed vector

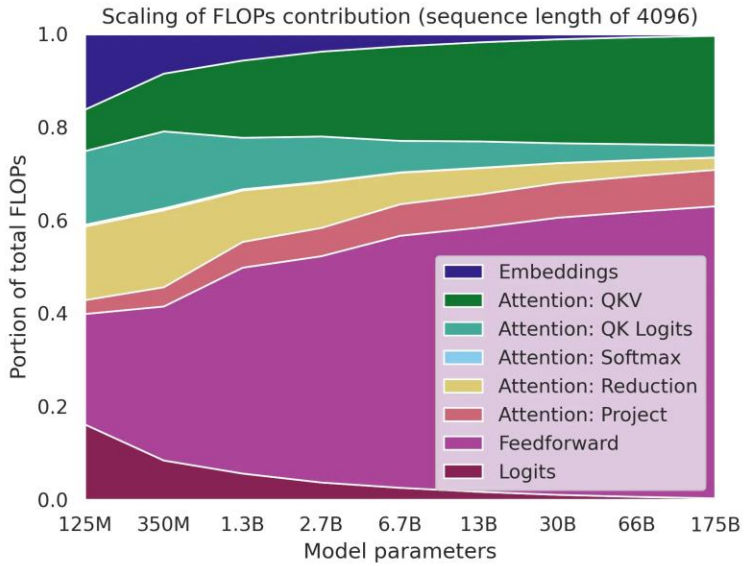
RHT transformed vector

Most modern quantization algorithms aim to reduce the error, *not* lower bits alone.

Quantization in Practice: Performance & Methods

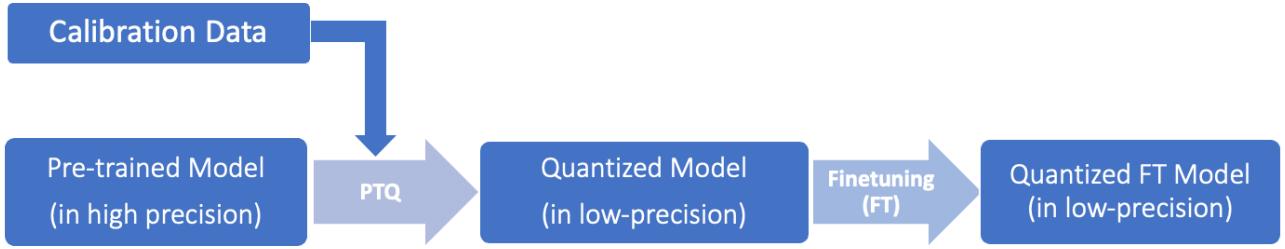
- **Weight-only quantization**
  - Major memory savings
  - Limited compute speedup (dequantize before matmul)
- **Weight + activation quantization**
  - Enables low-precision matmuls
  - Larger speedups, harder to stabilize
- **KV cache quantization**
  - Reduces decoding bandwidth
  - Critical for long-context inference
- Hardware matters: Low-precision MMA (FP8, FP4) → much faster
- Different model components have **different size & sensitivity**





Quantization delivers real speedups only when it aligns with hardware support and targets the right parts of the model.

Post-Training Quantization (PTQ)



- **PTQ:** Quantizes a **pre-trained full-precision model**. Requires no retraining and minimal compute
- May be data-free but typically requires a **small calibration dataset** (often a few hundred to a few thousand samples) to estimate activation ranges and scaling factors
- Significant accuracy degradation (typically) when using fewer than 8-bits. Fine-tuning may be required to recover accuracy.
- Weight quantization: **Offline**. Activation quantization: Dynamic (scale & zero-point computation done online) or Static (offline using calibration dataset)

PTQ makes quantization practical for large-scale inference.

Recent PTQ Methods

- **SmoothQuant (**Xiao et al., ICML 2023**):**
  - Scales activation matrix per channel to mitigate outliers, and scales weight in the inverse direction
  - Makes quantization overall easier by *smoothing* outliers
- **QTIP** (Tseng et al., NeurIPS 2024):
  - *Weight-only PTQ* for high-dimensional vector quantization while avoiding exponential codebook cost
  - Randomized transforms + Hardware-efficient Trellis quantization for fast inference
- **SpinQuant** (Liu et al., ICLR 2025):
  - Learns rotation matrices to transform weight/activation to reduce outliers, making them easier to quantize
  - 4-bit quantization of weights, activations, and KV cache
- **SVDQuant** (Li et al., ICLR 2025)
  - Absorbs outliers into a high-precision low-rank branch using SVD for diffusion models
  - Fused kernels for low-rank processing with the low-bit branch (Nunchaku)

Modern PTQ relies on sophisticated calibration, scaling, and optimization techniques (not simple rounding) to preserve model accuracy.

SmoothQuant – Deep Dive

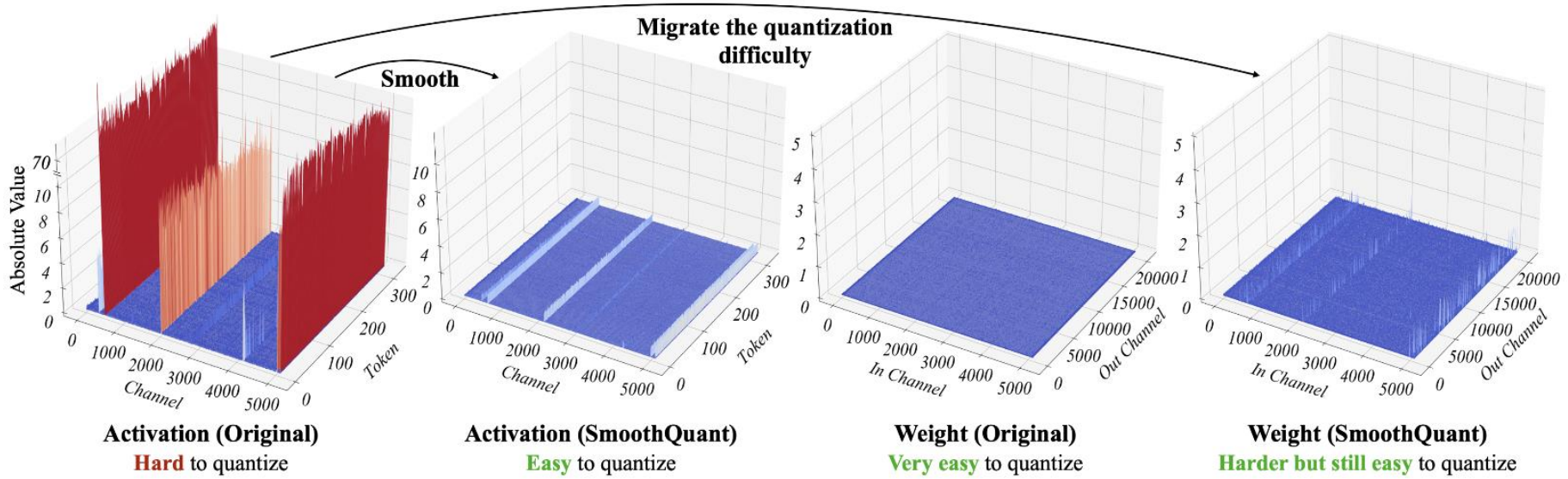


Image Source: Xiao et al., SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models, ICML 2023

- **Observation 1**: Weights are easy to quantize, but activations are hard due to outliers (persist in fixed channels).
- **Observation 2:** Small variance inside each channel (e.g., hidden dimension), but large across each token
- SmoothQuant **migrates** the quantization difficulty from activations to weights by scaling activation matrix per channel, and scaling weight in the inverse direction

SpinQuant – Deep Dive

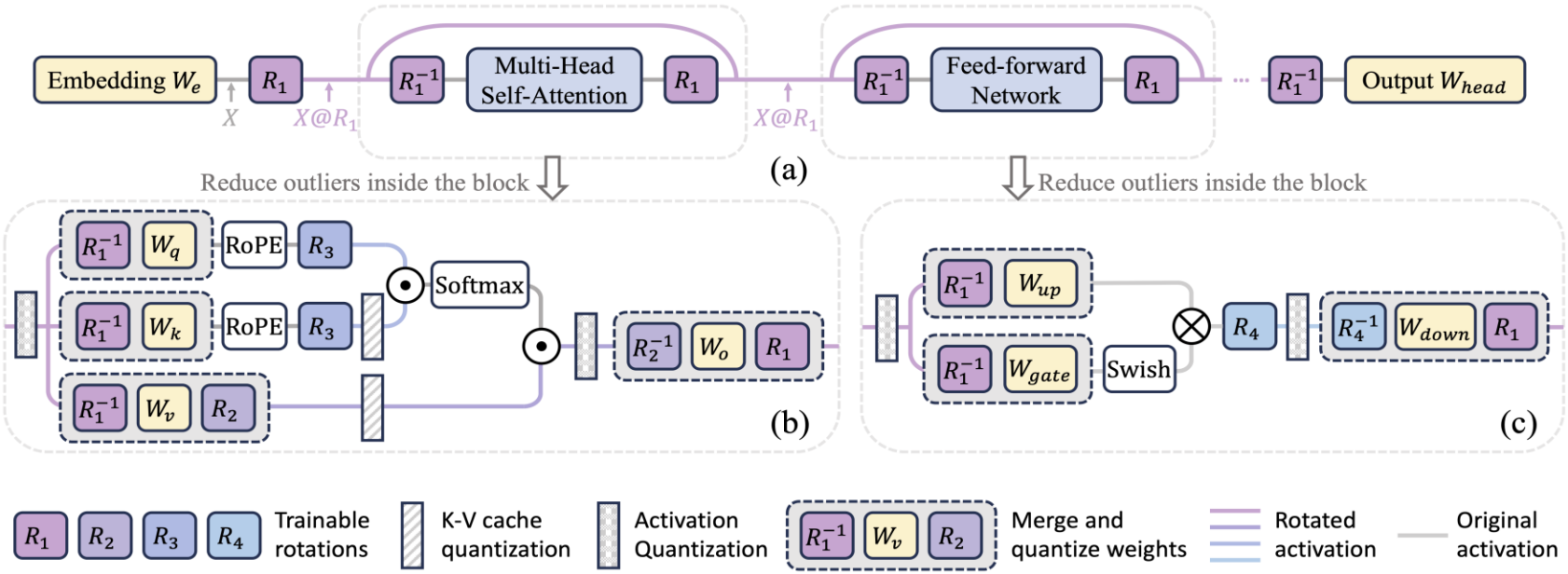


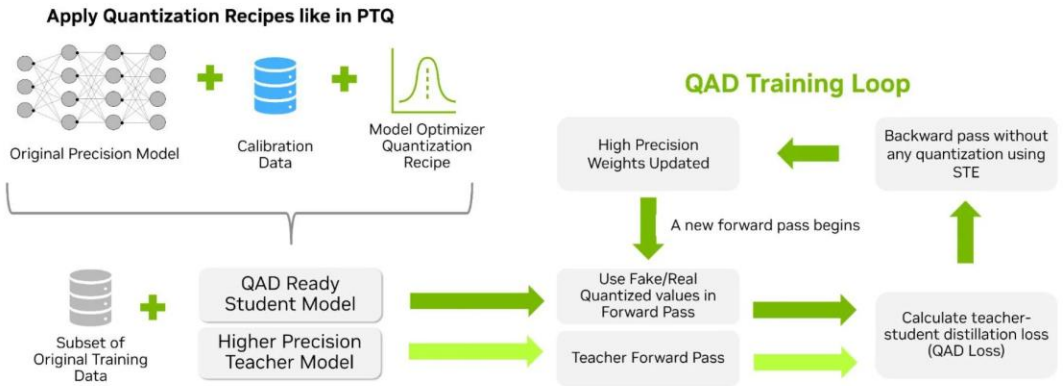
Image Source: Liu et al., SpinQuant: LLM quantization with learned rotations, ICLR 2025

- **Rotation** is a principled way to remove outliers in the LLMs and assist quantization
- Not all rotations help equally, and random rotations produce a large variance in quantized models
- Learning rotation with **Cayley optimization** greatly enhance the final performance
- SpinQuant **narrows the accuracy gap** of W4-A4-KV4 quantization with full precision (w.r.t SmoothQuant)

Quantization-Aware Training/Distillation (QAT/D)

QAT simulates quantization during training

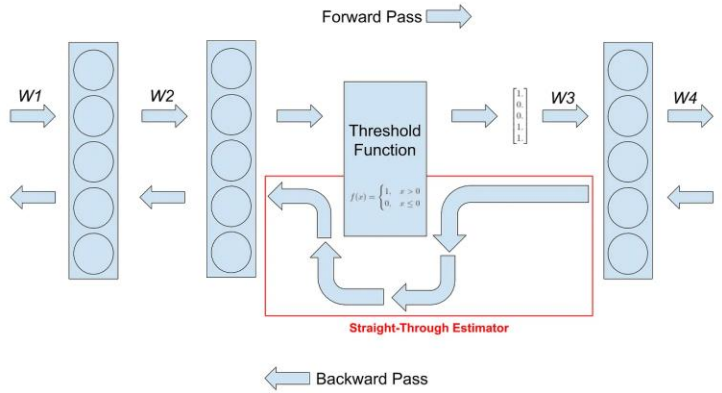
- Start with a higher-precision pretrained model
- Insert **fake-quantize / dequantize ops** in the forward pass
- The forward pass uses quantized weights and activations (simulated)
- Backward pass uses high-precision gradients with **straight-through estimators (STE)**
- Learn to **compensate for quantization errors**



Source: NVIDIA blog: how-quantization-aware-training-enables-low-precision-accuracy-recovery

QAD (distillation loss): **Teacher model** (which is the original full-precision model) guides a **student model** (which uses simulated low-precision compute in the forward pass).

- Expensive to run but QAT/QAD can match or exceed PTQ accuracy for low-bit formats



amazon | science

Model Compression

- ~~Motivation~~
- ~~Model Quantization~~
- **Key-Value (KV) Cache Compression**
- Knowledge Distillation

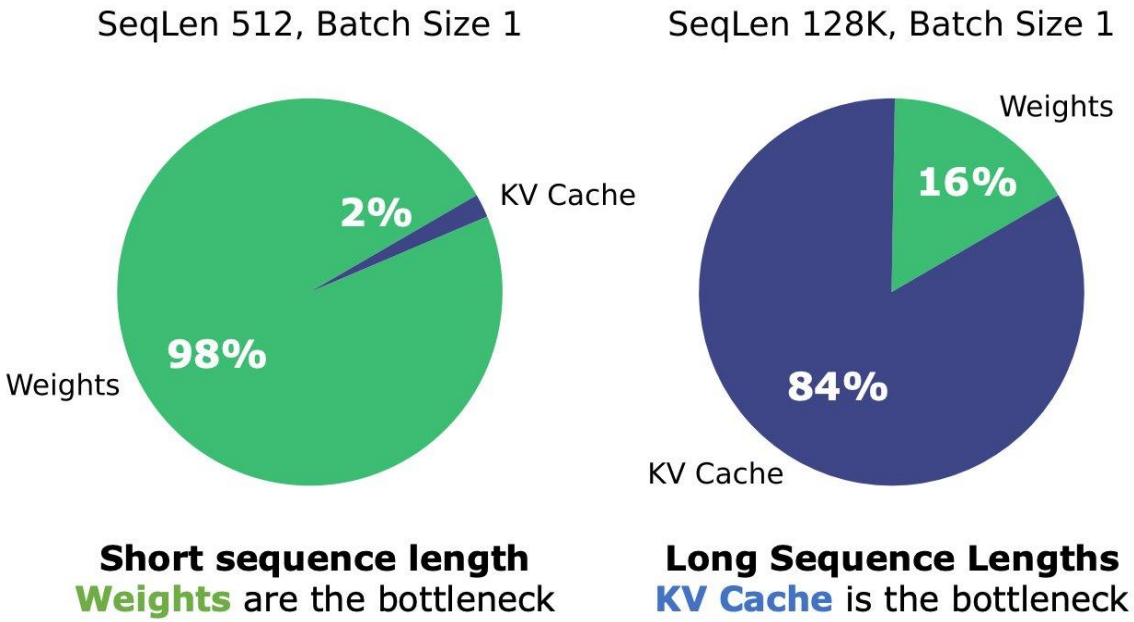
KV Cache Compression

- Why KV Cache Compression Matters
- Prefill vs. Decoding: When is KV Compressed?
- Taxonomy of KV Cache Compression Methods
- Explicit Lossy KV Compression

Why KV Cache Compression Matters

- Autoregressive decoding is **memory-bandwidth bound** due to KV cache growth
- KV cache grows **linearly with sequence length**
- Reducing KV cache size directly:
  - Lowers GPU memory footprint
  - Improves decoding latency (OTPS)
- KV cache compression often trades more complex reconstruction and potential accuracy risk for significantly lower memory traffic

KV cache compression targets the dominant bottleneck in long-context inference.

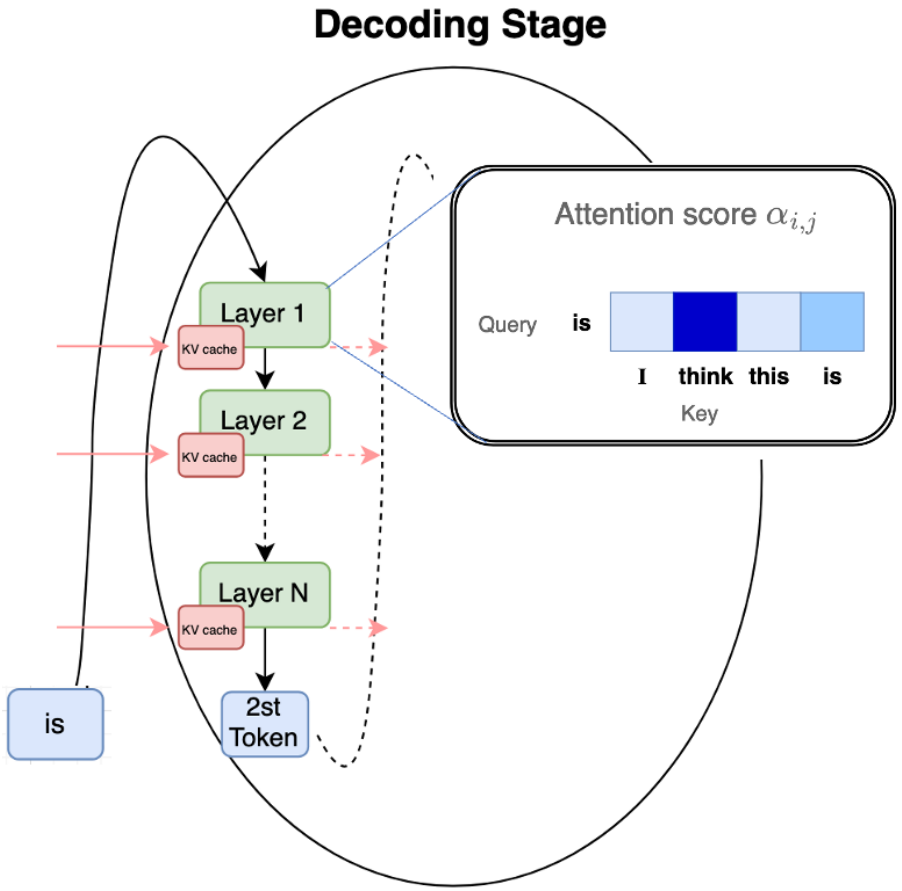


amazon | science

Prefill vs. Decoding: When is KV Compressed?

- Recall: Transformer inference has two phases:
  - **Prefill**: Build the full KV cache
  - **Decoding:** Repeatedly read and update KV cache
- **KV quantization**: Typically applied **only during decoding**
- **KV sparsification**: Tokens are evicted during prefill or after prefill (enable input longer sequence length)
- Post-prefill eviction is generally **more robust**
- Many methods target a **fixed-size** KV cache

KV cache compression strategies differ fundamentally in whether they act during prefill, decoding, or both



Taxonomy of KV Cache Compression Methods

- **Training-stage modification:**
  - Architectural changes like GQA, SWA reduce the size of KV cache
  - State-space models do not have a KV cache (e.g., Mamba): Less suitable for retrieval-related tasks
- **Deployment-stage optimization**:
  - Paged attention (seen later)
  - Chunked prefill: Reuse KV cache between different dialogues. Avoids repeated calculation.
  - Infinigen (Lee et al., 2024): Offload KV cache to CPU with speculation (saves HBM space)
- **KV cache eviction**:
  - Static policies (e.g., Sliding Window Attention)
  - Dynamic policies (e.g., TOVA: TOken Eviction Via Attention), H2 eviction (using accumulated normalized attention scores)
  - **PyramidKV** (Cai et al., 2025) and **PyramidInfer** (Yang et al., 2024): Layerwise approach that shortens the length of the KV cache in deeper layers
  - **SparQ attention** (Ribar et al., 2024): Reduce the amount of data transferred from HBM to SRAM during decoding by fetching selective key-value pairs

Explicit Lossy KV Compression

- **KV cache quantization:** **KVQuant** (Hooper et al., 2025), **WKVQuant** (Yue et al., 2024), **QAQ** (Dong et al., 2024) -- Key and Value cache have different sensitivities.
- **Low-rank + Hybrid:**
  - **LESS** (Dong et al., 2024): Combines sparse eviction with low-rank memory. Maintains a constant-size low-rank cache -- accumulating history from token information before they are discarded from KV cache.
  - **GEAR** (Kang et al., 2024): Low rank + Quantization + Sparsity for KV cache
  - **MiKV** (Yang et al., 2024): Retains evicted KV pairs in reduced; maintains important KV pairs in higher precision.

With quantization, KV cache size is still proportional to the context length (even though it is reduced by a factor of precision). Orthogonal to token eviction, attention sparsity, and model quantization

Lossy KV cache compression trades precision for long context support

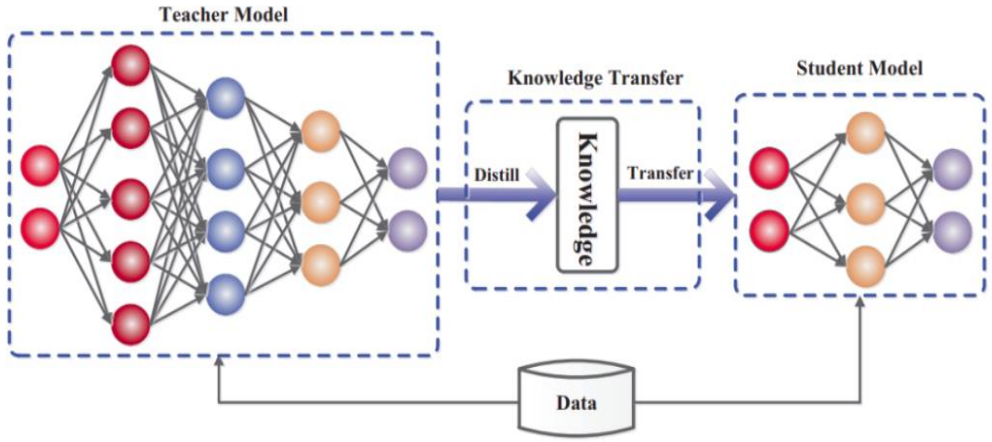
Model Compression

- ~~Motivation~~
- ~~Model Quantization~~
- ~~Key-Value (KV) Cache Compression~~
- **Knowledge Distillation**

Knowledge Distillation

- Motivation and Big-picture
- How and Why Distillation Works
- Logit (Soft-Label) Distillation
- Sequence-Level Distillation
- Feature-Level Distillation: Powerful But Costly

Distillation: Motivation and Big Picture

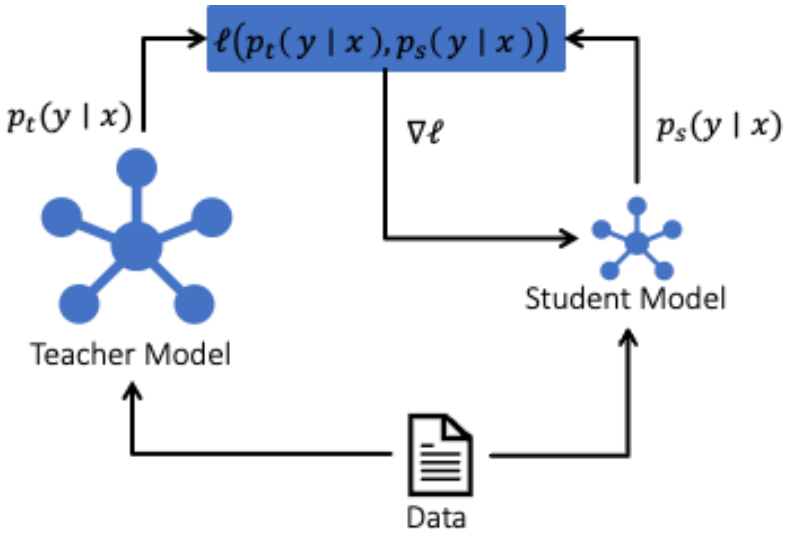


- **KD** trains a smaller *student* model to imitate a larger *teacher* (Hinton et al., NeurIPS 2014)
- Student model can be of same or different architecture, but with fewer parameters
- Student trained with KD consistently outperforms the same architecture trained from scratch
- Achieves lower latency (both prefill/decoding) and memory footprint → Reduces serving cost

KD is more effective than post-training compression, but also expensive

How and Why Distillation Works

- Teacher outputs a **soft probability distribution**, not just the correct label
- Goal: Train the student to match the teacher's accuracy via a **distillation loss** (as opposed to one-hot supervision in SFT)
- Benefits:
  - Smoother optimization landscape
  - Lower effective sample complexity
- Analogy: Learning from an expert vs. Reinventing from scratch *(a college student can learn calculus from a professor in months but would not reinvent calculus on their own even in years.)*
- Student learns relative likelihoods between tokens and subtle correlations invisible in one-hot supervision



The teacher encodes knowledge that is hard to extract directly from data. Distillation replaces hard supervision with richer, more informative signals.

Logit (Soft-Label) Distillation

- Student learns from the teacher's **soft probability distribution**
- Uses a weighted combination of:
  - **Cross-entropy loss** with ground-truth labels
  - **KL divergence** between teacher and student distributions

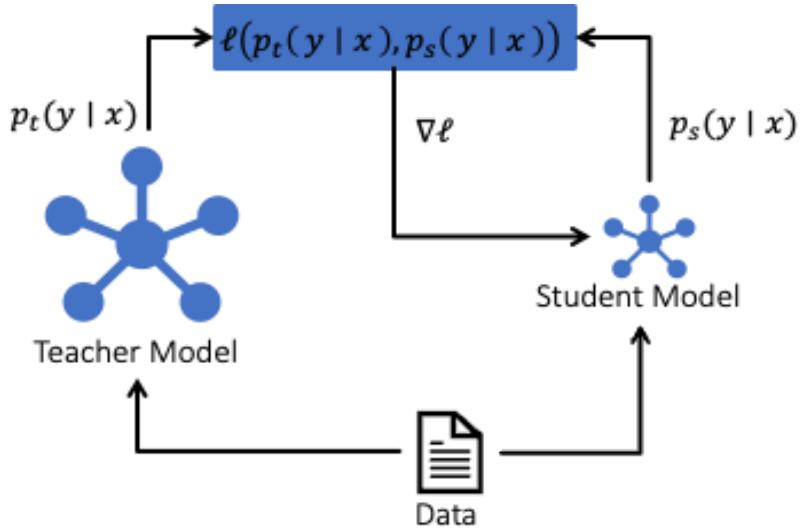
$$L_{\text{KD}} = (1-\alpha)\, L_{\text{CE}} + \alpha\, T^2\, L_{\text{KL}}$$

$$L_{\text{CE}} = -\sum_i y_i \log p_s(i)$$

$$L_{\text{KL}} = \sum_i p_t(i)\, \log \frac{p_t(i)}{p_s(i)}$$

$$p_t = \text{Softmax}\left(\frac{z_t}{T}\right), \quad p_s = \text{Softmax}\left(\frac{z_s}{T}\right)$$

T controls the distribution smoothness; α balances the ground-truth supervision and teacher guidance



Soft labels convey relative confidence across tokens, not just the correct answer

Sequence-Level Distillation

- Teacher generates full output sequences
- Human datasets can be noisy. Teacher-generated datasets provide cleaner supervision
- Student trained via standard cross-entropy on teacher outputs.
- Benefits:
  - Cheap generation of **large** synthetic datasets
  - Transfers **reasoning traces** and **formatting preferences**
- Especially useful when teacher logits are **unavailable** (**closed models**)

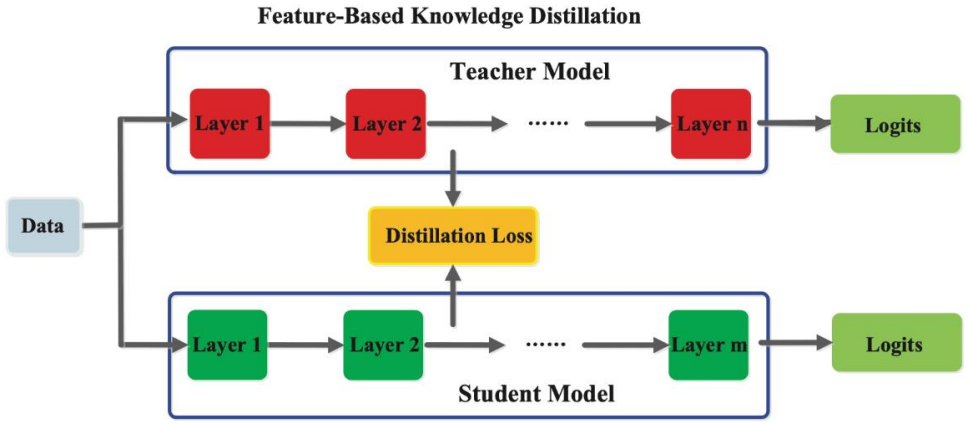
$$y_t = (y_{t,1}, y_{t,2}, \ldots, y_{t,L}) \quad \text{(teacher-generated sequence)}$$

$$L_{\text{seq}} = -\sum_{i=1}^{L} \log p_s(y_{t,i} \mid x, y_{t,<i})$$

Students may not match the teacher's distributions, but can still imitate its behavior

Feature-Level Distillation: Powerful but Costly

- Student matches (w.r.t. e.g., Frobenius norm):
  - Hidden states
  - Attention maps
  - Intermediate embeddings
- Intuitively, captures hierarchical knowledge:
  - Lexical → syntactic → semantic → abstract
- Challenges: **Very high** memory and compute cost
- In practice:
  - Rare for large LLs (≤ 3B parameters is feasible)
  - More common in vision models or small language models



Source: Gou et al., "Knowledge Distillation: A Survey", IJCV 2021

$$L_{\text{feat}} = \sum_{(l,m)\in\mathcal{M}} \left\| \phi\left(H_s^{(m)}\right) - H_t^{(l)} \right\|_2^2$$

or

$$L_{\text{attn}} = \sum_{(l,m)\in\mathcal{M}} \left\| A_s^{(m)} - A_t^{(l)} \right\|_F^2$$

Feature-level KD is powerful, but logit- and sequence-level KD scale better for LLMs

Knowledge Distillation: Takeaways

- Distillation transfers **behavioral patterns** learned by large models into smaller ones
- Enables strong accuracy-latency tradeoffs beyond what architectural scaling allows
- Complements other inference optimizations:
  - Quantization
  - KV cache compression
  - Sparse or efficient architectures
- Choice and granularity of distillation strategy depends on:
  - Access to teacher internals (logits vs text only)
  - Target model size and precision
  - Deployment cost and latency budgets

Algorithmic and Modeling-Level Inference Optimizations

- ~~Key-Value (KV) Caching~~
- ~~Inference-Aware Model Architectures~~
- ~~Model Compression~~
- **Speculative Decoding**

Speculative Decoding

- **Motivation: Why Speculative Decoding?**
- **Core Idea of Speculative Decoding**
- **Acceptance-Rejection Mechanism**
- **Choosing the Draft Model**
- **Optimal Speculation Length**

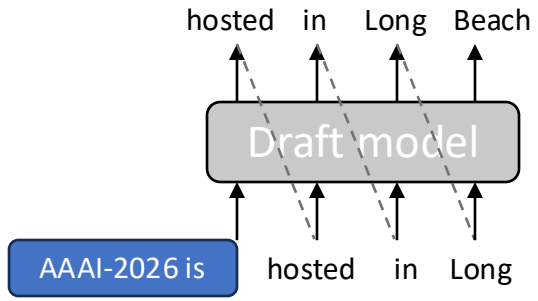
Motivation: Why Speculative Decoding?

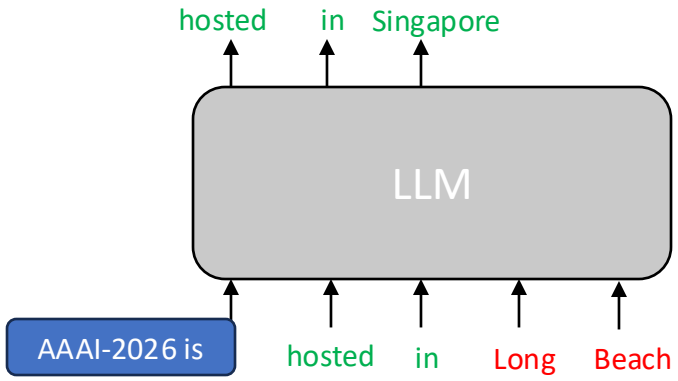
- LLM decoding is autoregressive (one token at a time), and memory-bound
- Latency is dominated by serial dependency between tokens and repeated KV cache reads
- **Question**: Can we generate **multiple tokens per step** without changing the model output?
- **Answer**: Speculative decoding *(Leviathan et al., 2023)*

Speculative decoding is a lossless technique that attacks the *sequential bottleneck* of autoregressive decoding

Core Idea of Speculative Decoding

- Two models:
  - **Draft model** (small and fast, but possibly inaccurate)
  - **Target model** (large and accurate)
- Workflow:
  - Draft model proposes a block of tokens
  - Target model verifies them **in parallel**
  - Accepted tokens are committed
- Example (on the right): 2 out of 4 drafted tokens are accepted
- Guarantees: Final output distribution matches the target model exactly
- Key-insight: Verification can be batched and parallelized

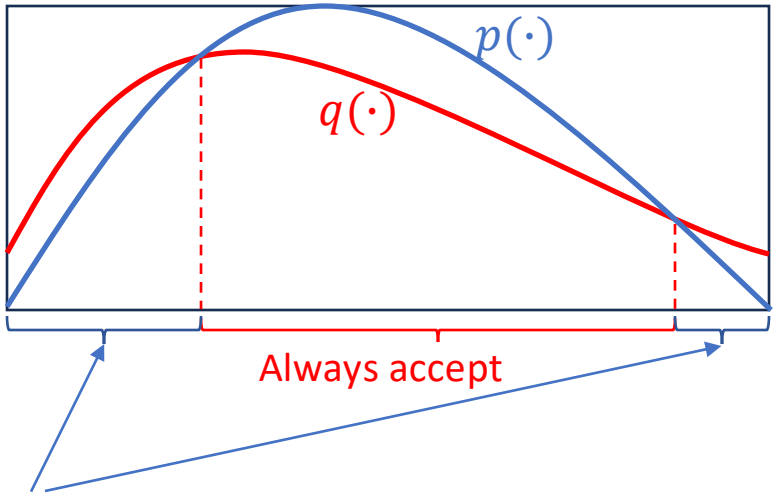




SD trades extra computation for fewer sequential decoding steps (memory bound → compute bound)

Acceptance-Rejection Mechanism

- Draft tokens from a smaller model: $x \sim q(x)$
- Verify (in parallel) the drafted tokens using the target LLM: $p(x)$
- Greedy: Reject if $\arg\max p(x)$ does not equal the drafted
- More generally, rejection sampling based
  - Draw $x \sim q(x)$
  - Accept x with probability $\min\left\{\frac{p(x)}{q(x)}, 1\right\}$, implying always accept when $q(x) < p(x)$
  - Reject x with probability $1 - \min\left\{\frac{p(x)}{q(x)}, 1\right\}$ and redraw $x \sim \text{Normalize}(\max\{p(V) - q(V), 0\})$
- Guaranteed to be equivalent to sampling from $p(\cdot)$

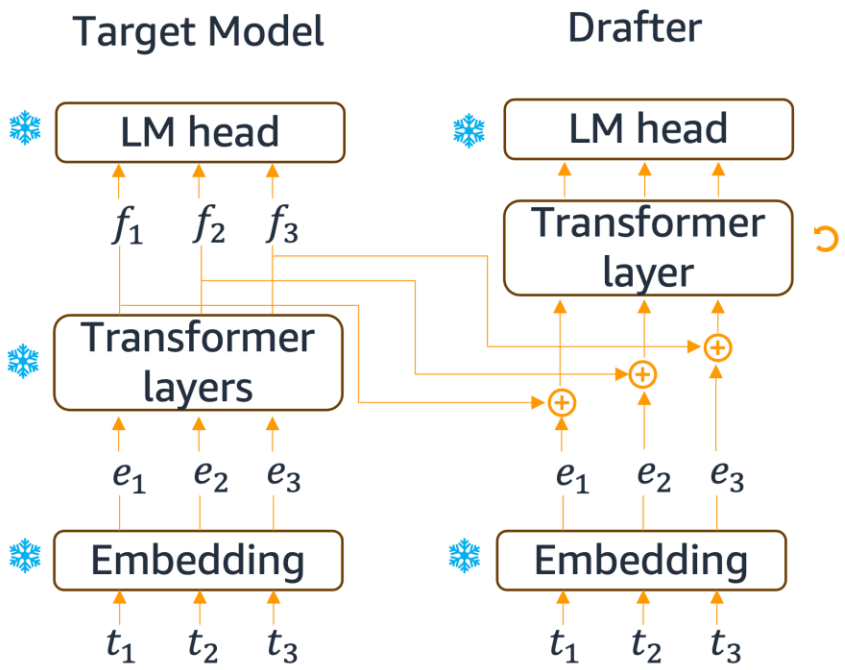


Accept with probability $\frac{p(x)}{q(x)}$; redraw if rejected

Better drafts → Longer accepted runs → Higher speedup.

Choosing the Draft Model

- Speed up depends on:
  - Closeness between $q(\cdot)$ and $p(\cdot)$
  - Speed ratio between the draft model and the LLM
- Up to 2-4x speedup reported
- How to choose the Draft Model?
  - **Smaller model in the same family** as the target (e.g., Tiny-Llama for Llama2-70B)
  - **Train drafter** from target using knowledge distillation (make $q(\cdot)$ and $p(\cdot)$ closer with a distillation-loss)
  - **Eagle** (Li et al., 2024): Reuses intermediate hidden states and parameters of the target model, dramatically reducing training overhead while achieving high acceptance rates.



Optimal Speculation Length

- Too short draft length may not fully exploit the power of the draft model
- Speculation length can be longer if:
  - $q(\cdot)$ and $p(\cdot)$ are close
  - Draft model is much faster than the target LLM
- The closeness between $q(\cdot)$ and $p(\cdot)$ varies
  - Depending on the prefix
    (generic semantics are easy to speculate, specialized knowledge is not)
  - Quality of the drafter (mismatch between pre-training data of target LLM and training data used to train the drafter)

Speculative Decoding: Takeaways

- Speculative Decoding is **lossless**! (Unlike most other optimizations seen before)
- Reduces number of target model forward calls and **increases arithmetic intensity**
- Supported by most popular frameworks (eg., vLLM, Neuron, etc.)
- Various methods to generate drafters 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.
- Best suited for **long generations** and **high-throughput serving**
- Limitations:
  - Limited gains for short outputs
  - Complex control flow in serving systems

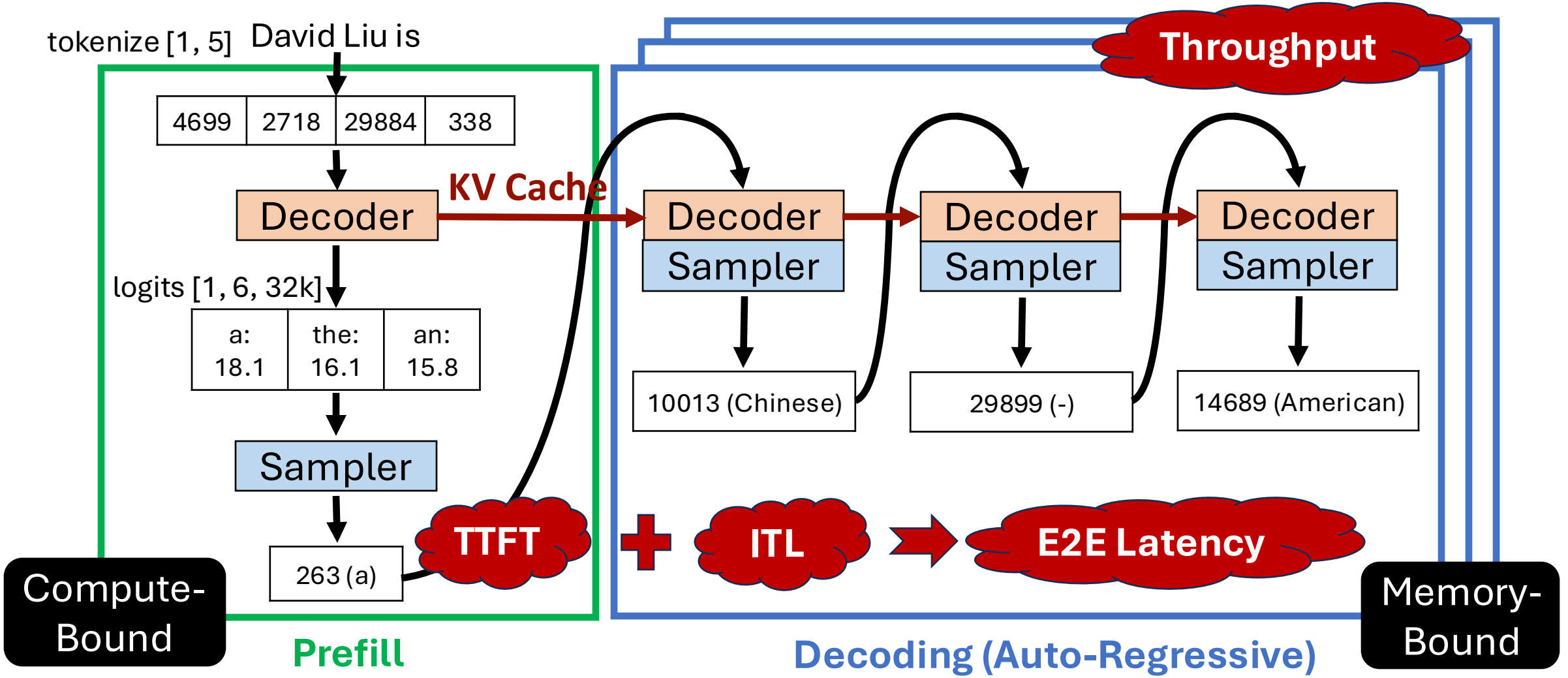
Tutorial Outline

- ~~Primer: Foundations of Generative Inference~~
- ~~Algorithmic and Modeling-Level Inference Optimizations~~
- **Systems-Level Optimizations**
- Open-Source Frameworks and Tools

Systems-Level Optimizations

- **FlashAttention**
- Continuous Batching
- PagedAttention
- Chunked Prefill
- Disaggregated Inference
- Prefix Caching
- Multi-LoRA Serving
- Compilers

Inference Workload Overview

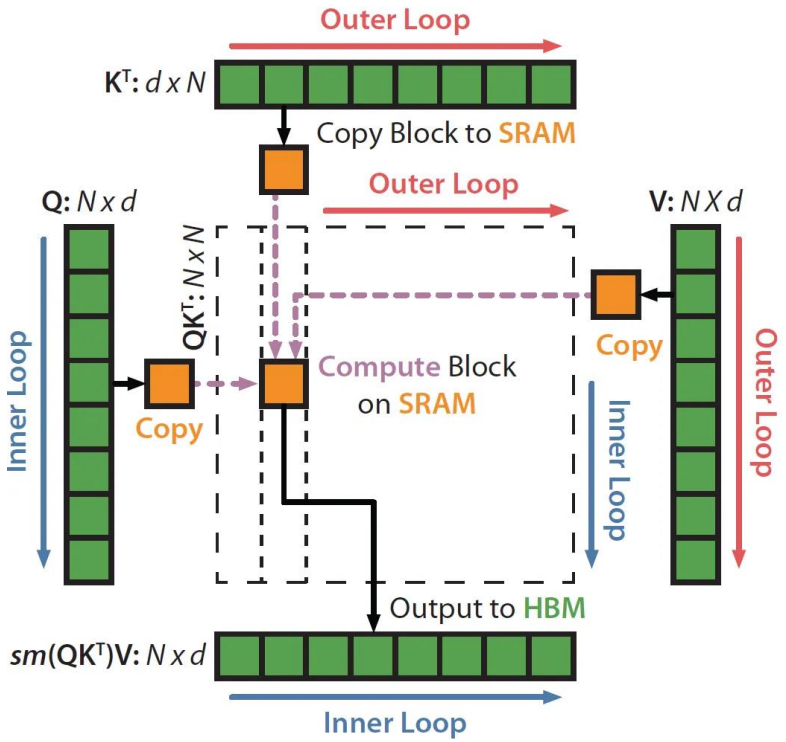


Self Attention is Memory-Hungry

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

- $S = QK^T$ → read QK ($\mathbb{R}^{Nd}$) and write S ($\mathbb{R}^{N^2}$)
- $P = softmax(S)$ → read S ($\mathbb{R}^{N^2}$) and write P ($\mathbb{R}^{N^2}$)
- $O = PV$ → read PV ($\mathbb{R}^{N^2}, \mathbb{R}^{Nd}$) and write O ($\mathbb{R}^{Nd}$)

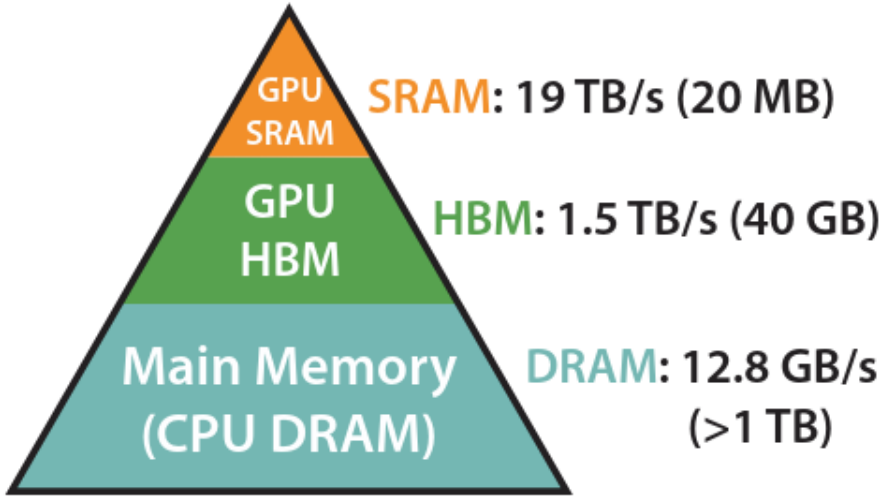
- Standard self attention is slow and memory-hungry
- Memory complexity: $O(N^2)$ where N= sequence length
  - Bandwidth becomes the bottleneck for long sequences



Source: Dao et al., "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness", NeurIPS 2022

FlashAttention: Motivation

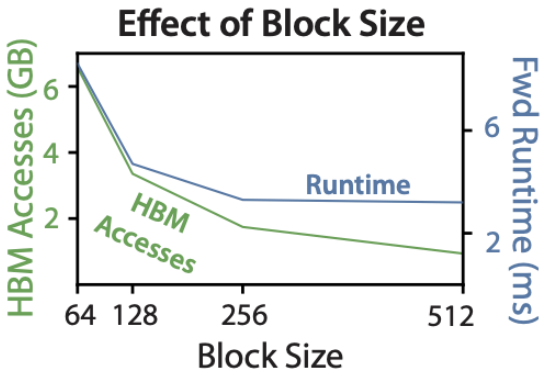
- Standard self attention incurs frequent reads and writes of large intermediate tensors to HBM
- Not all memory is created equal; a small amount of fast memory exists on chip
  - E.g. GPU SRAM, Trn SBUF
- **Key idea: keep data in fast on-chip memory as much as possible.**
- **FlashAttention** resolves memory bandwidth bottleneck through I/O-aware tiling
  - Reduced data transfers between HBM and SRAM

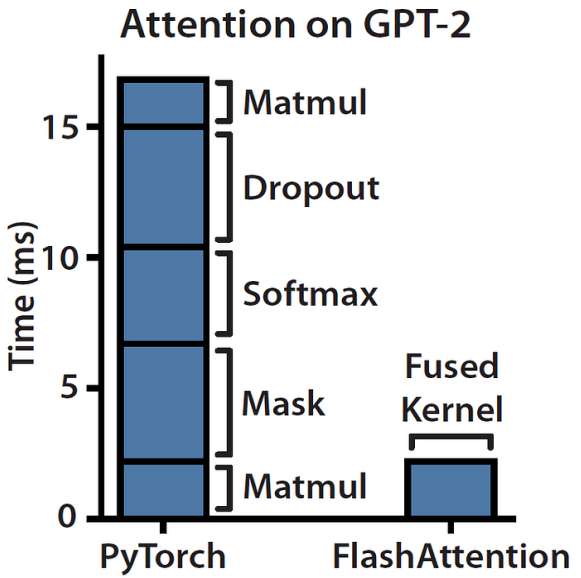


Source: Dao et al., "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness", NeurIPS 2022

FlashAttention

- Decompose (tiling) softmax by scaling
  - Break Q, K, and V into tiles
  - Load tiles into SRAM
  - Compute attention on blocks incrementally
- Fuse S, P, and O computation tile-by-tile
- Tile size depends on SRAM size
  - GPU: 40MB L2 shared
  - Trainium: 24MB per core





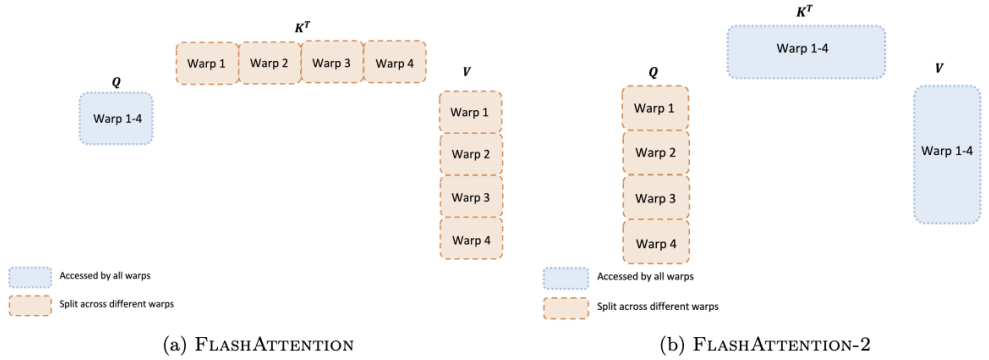
Source: Dao et al., "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness", NeurIPS 2022

FlashAttention-2

FlashAttention-2 introduces three main improvements over FlashAttention:

1. Reduces non-matrix multiplication (rescaling) operations in online softmax
   - Delays normalization until the end
2. Adds parallelism over sequence length in addition to batches/heads
   - Processes Q tiles independently (reorders loop)
3. Improves work partitioning
   - Partitions Q tiles across warps instead of K, V and avoids synchronization of warps

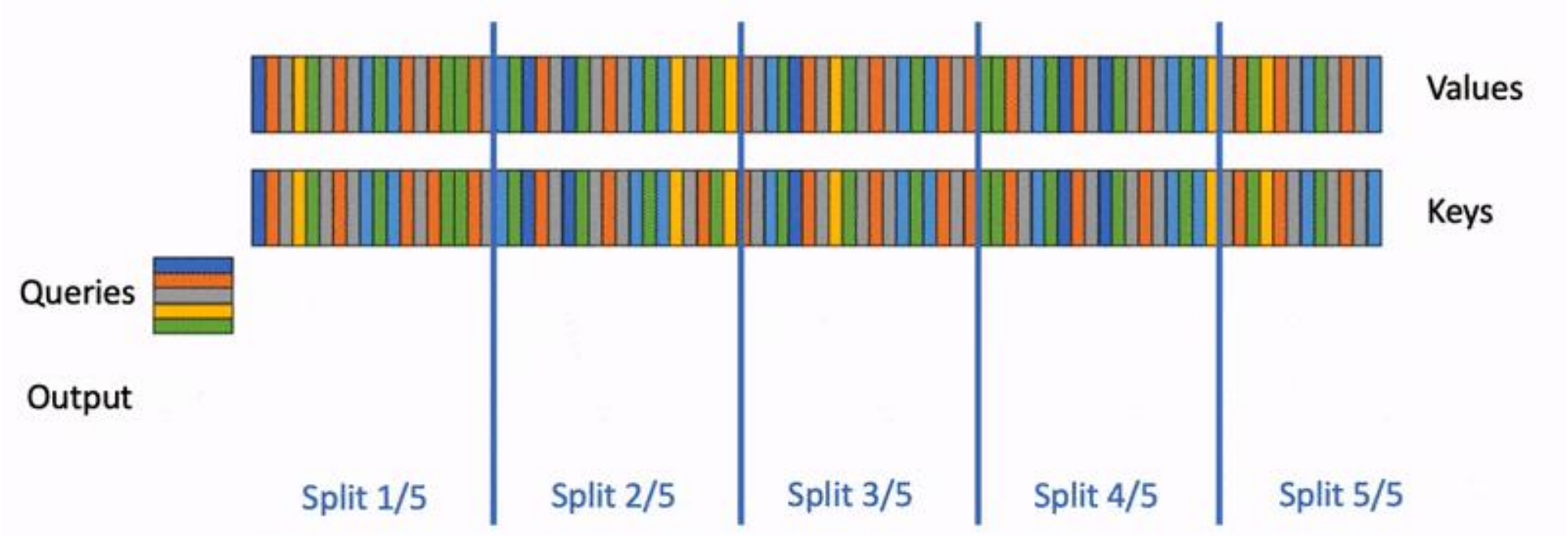
- ~2x faster than original FlashAttention
- Achieves 50-73% of theoretical max FLOPs/s on A100
  - FlashAttention only achieved 25-40%



Source: Dao et al., "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning", 2023

Flash-Decoding

- **Key idea: parallelize over sequence length by splitting KV cache into chunks along sequence dimension and computing partial attention for each chunk in parallel.**



Source: Dao et al., “Flash-Decoding for Long-Context Inference”, 2023

Systems-Level Optimizations

- ~~FlashAttention~~
- **Continuous Batching**
- PagedAttention
- Chunked Prefill
- Disaggregated Inference
- Prefix Caching
- Multi-LoRA Serving
- Compilers

Batching in LLMs

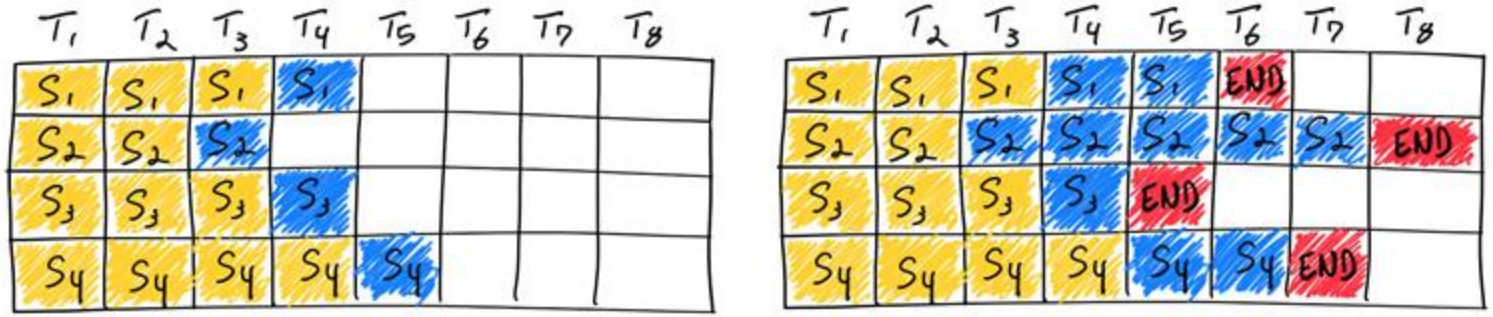
- Batching is a fundamental performance optimization that processes multiple input samples together in a single forward pass
  - Increases throughput, improves device utilization, and shortens queueing delay
- Autoregressive decoding in LLMs poses difficulties for batching
- Frameworks expect batches to have identical structure (shape, execution path, outputs in lockstep), but LLMs vary in context length and generation output length
  - Output can be a few tokens (short answers) or hundreds to thousands of tokens (stories, code generation)

Static Batching

- A simple batching technique is static batching via padding
  - Wait for a batch of requests to arrive and decode tokens synchronously step by step until the longest request finishes
- Short requests may finish early, but cannot exit the queue until longer requests finish
- As some requests finish earlier, the effective batch size deceases over time causing the device to run smaller, less efficient compute operations
  - System throughput decreases
  - Latency for short requests is high
- Static batching is simple to implement, but mismatched with the nature of LLM decoding

Continuous Batching

- **Key idea: operate on batches at the iteration level (decoding step) instead of the request level.**
- Each decoding step runs on all active requests



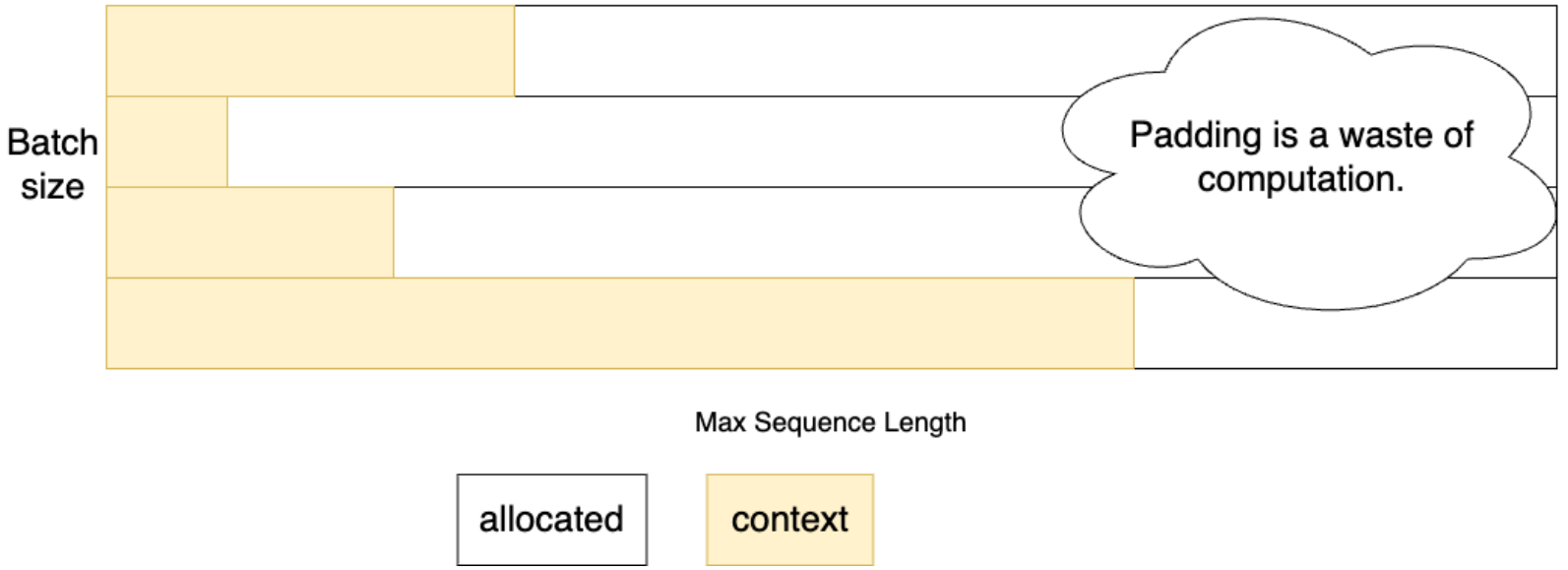
- However, each request maintains a growing KV cache for attention to handle different lengths
  - Orca (Yu et al. OSDI ‘22) allocates a large batched KV buffer and designs attention mechanisms for variable context lengths
  - vLLM (Kwon et al. SOSP ‘23) uses flexible memory management techniques

Systems-Level Optimizations

- ~~FlashAttention~~
- ~~Continuous Batching~~
- **PagedAttention**
- Chunked Prefill
- Disaggregated Inference
- Prefix Caching
- Multi-LoRA Serving
- Compilers

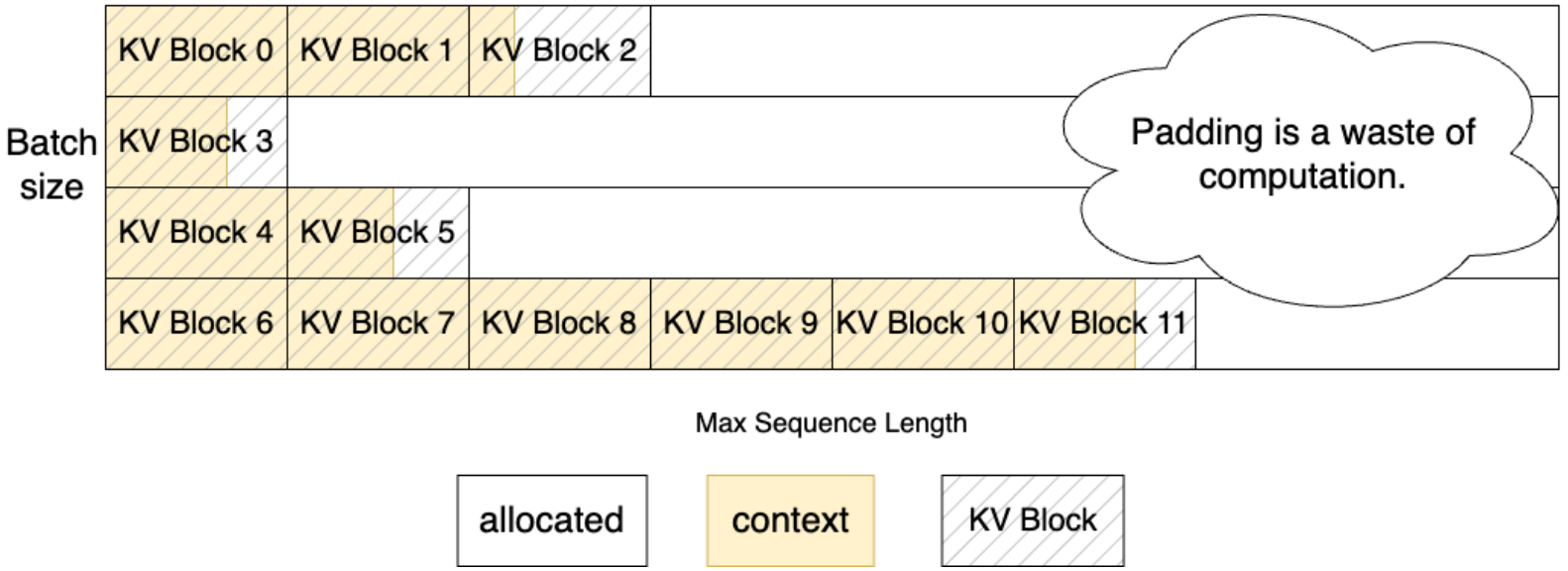
KV Cache Management

- The simplest way to support continuous batching is to preallocate contiguous KV memory to the longest model sequence length for every request



KV Cache Management

- The simplest way to support continuous batching is to preallocate contiguous KV memory to the longest model sequence length for every request
- This naïve method results in serious internal memory fragmentation and reduces the number of requests that can fit in the device memory



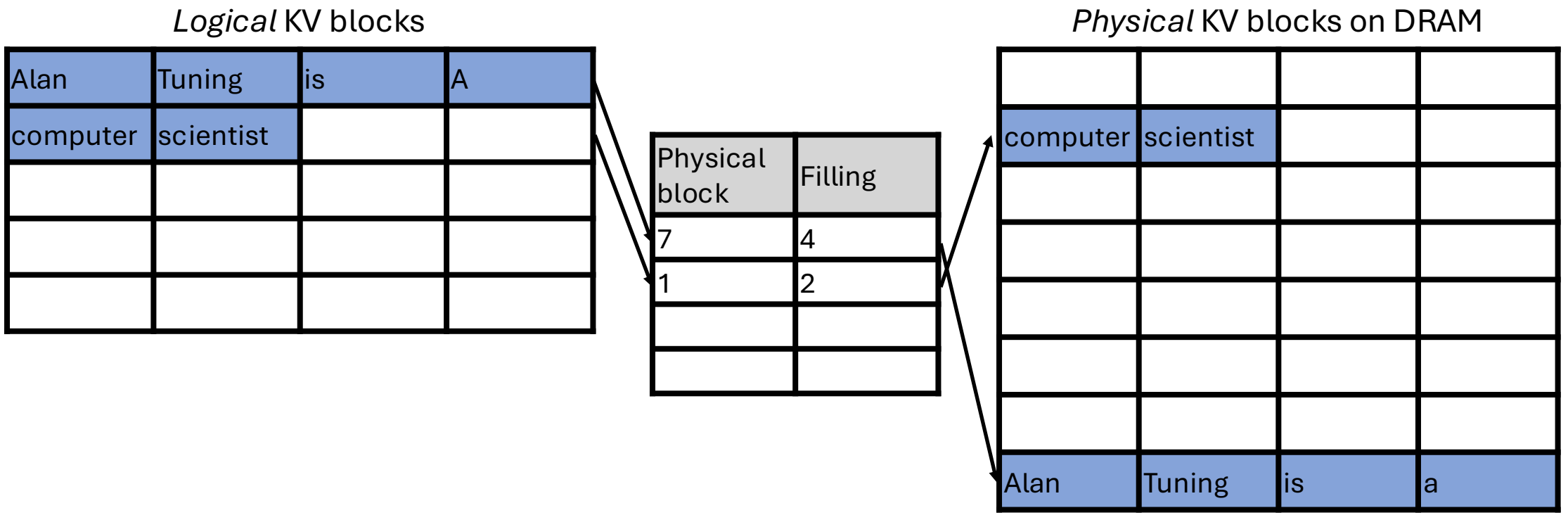
PagedAttention

- **Key idea: extend the OS idea of paging to managing KV memory and treat each request's KV cache as a virtual address space backed by fixed-size blocks of device memory.**
- Blocks form a global pool shared by all requests and are allocated on demand
  - Short requests use fewer blocks and long requests use more blocks → memory usage scales with decoding progress
- The KV cache appears contiguous to the model, but is physically scattered across blocks
- Block size can impact memory efficiency and performance
  - Large block size: fewer lookups and better loading efficiency, but more internal fragmentation
  - Small block size: better memory utilization, but more block-table overhead and scattered accesses

PagedAttention Demo

Prompt: Alan Tuning is a computer scientist

Decode: and

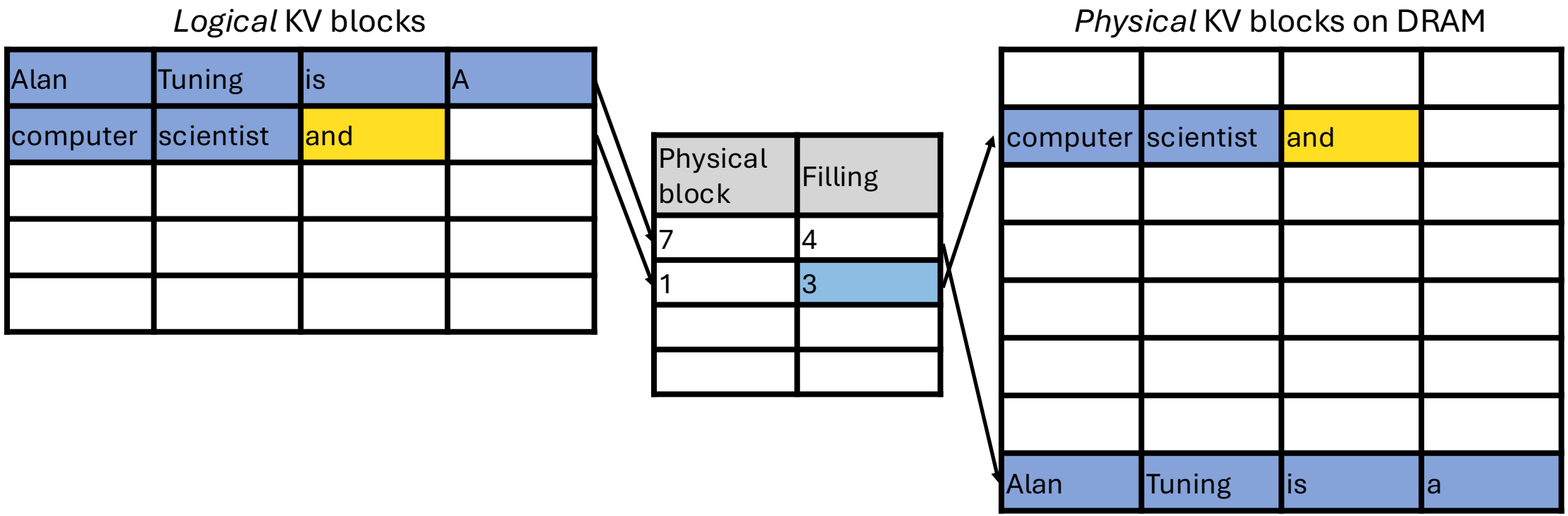


Source: Kwon et al., "Efficient Memory Management for Large Language Model Serving with PagedAttention", SOSP 2023

PagedAttention Demo

Prompt: Alan Tuning is a computer scientist

Decode: and

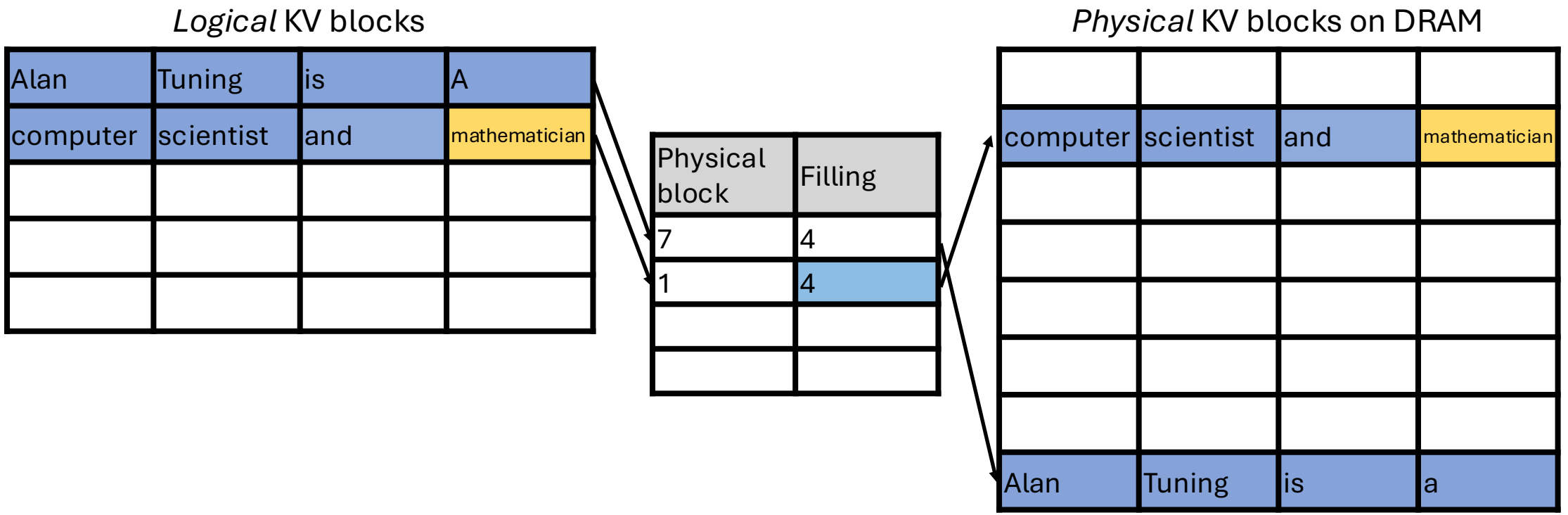


Source: Kwon et al., "Efficient Memory Management for Large Language Model Serving with PagedAttention", SOSP 2023

PagedAttention Demo

Prompt: Alan Tuning is a computer scientist

Decode: and mathematician

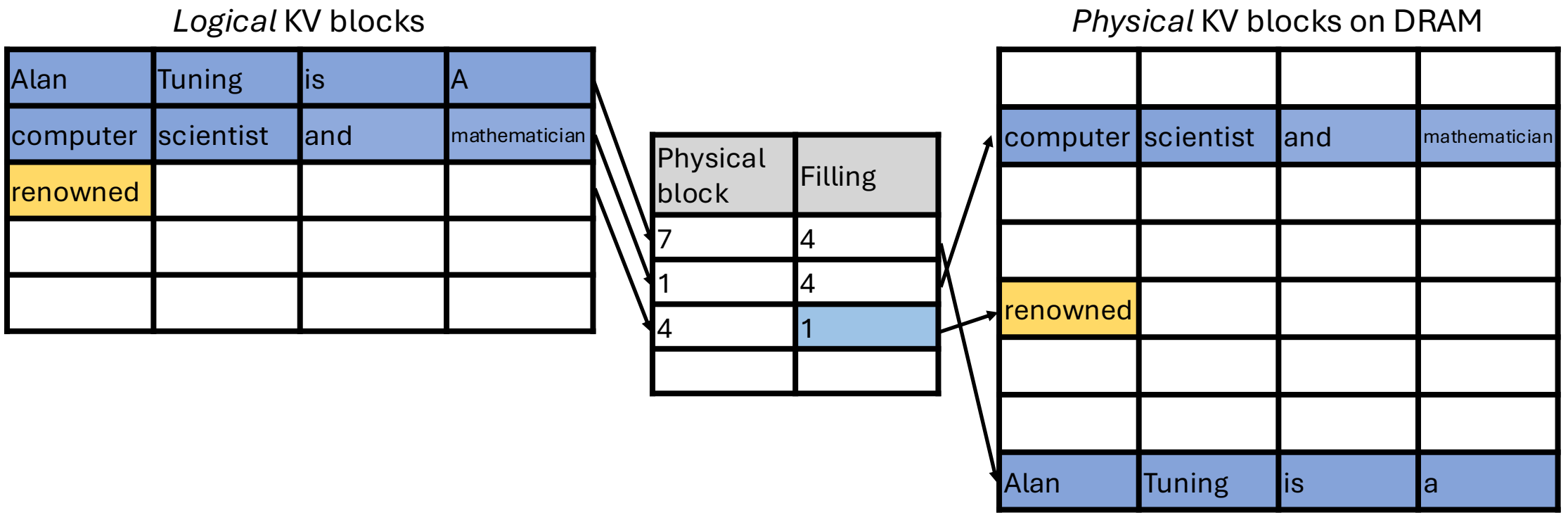


Source: Kwon et al., "Efficient Memory Management for Large Language Model Serving with PagedAttention", SOSP 2023

PagedAttention Demo

Prompt: Alan Tuning is a computer scientist

Decode: and mathematician renowned



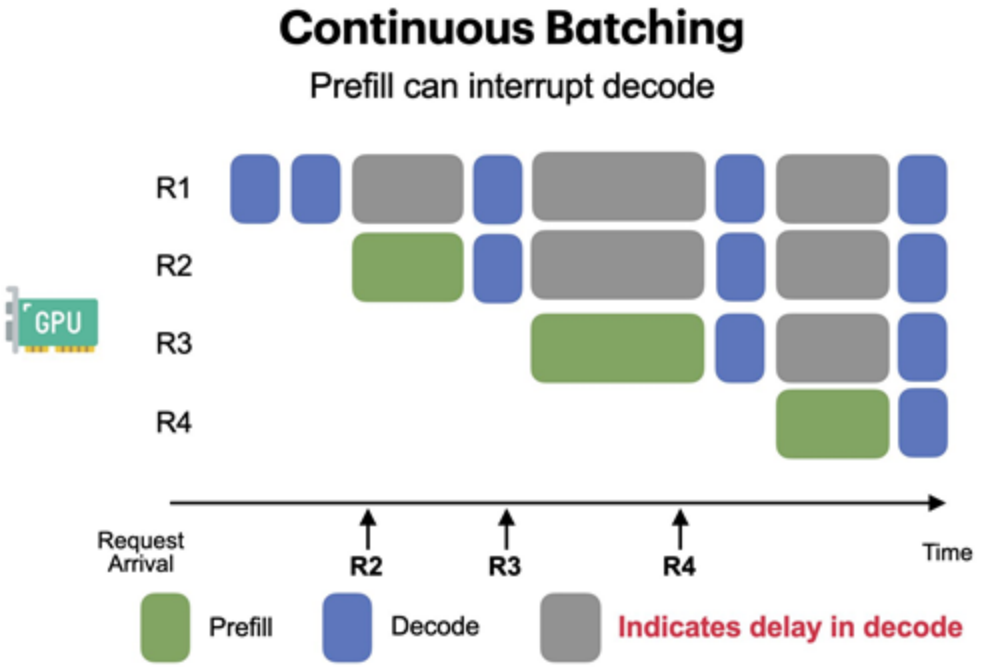
Source: Kwon et al., "Efficient Memory Management for Large Language Model Serving with PagedAttention", SOSP 2023

Systems-Level Optimizations

- ~~FlashAttention~~
- ~~Continuous Batching~~
- ~~PagedAttention~~
- **Chunked Prefill**
- Disaggregated Inference
- Prefix Caching
- Multi-LoRA Serving
- Compilers

Challenges with Prefill

- The default scheduling strategy of many serving systems prioritizes prefill over decode
  - A request cannot join the decode batch until its prefill is finished
- This strategy breaks down when prefill becomes large or irregular
  - Long prompts become expensive and prefill can block decoding

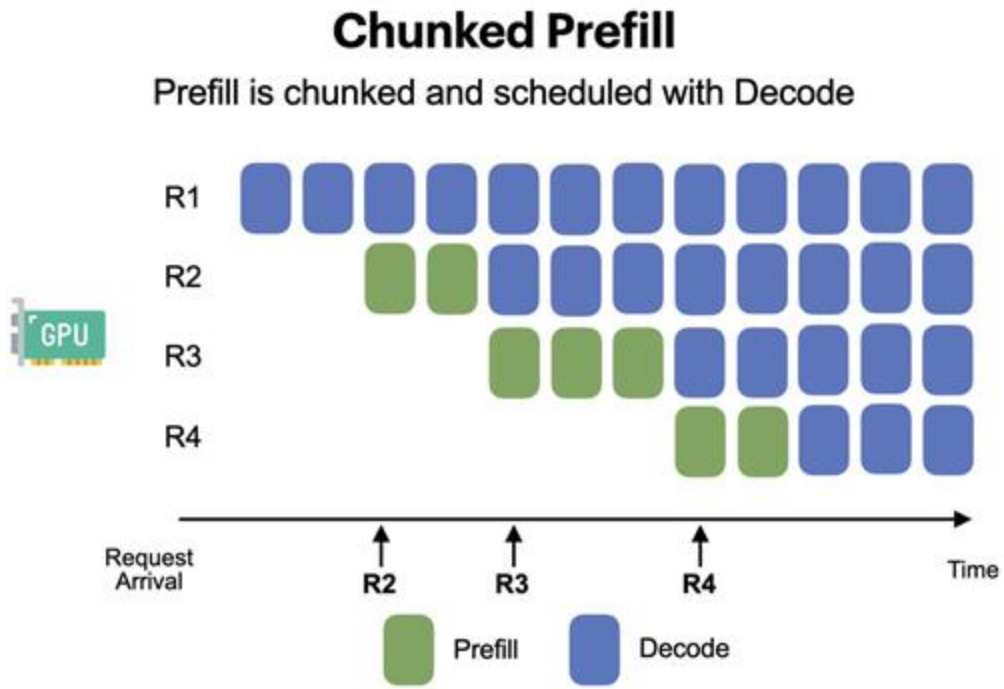


Agrawal et al., "Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve", OSDI 2024

Chunked Prefill

Key Idea: break a long prefill into many small prefills (chunks).

- Tokenize the prompt and divide it into smaller segments (e.g. 512 tokens)
- Process each chunk and append KV cache entries
- For each chunk, decode phases can be piggybacked



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

Chunked Prefill Tradeoffs

- Stability under long prompts: avoids latency spikes and prevents long requests from stalling device
- Interactive performance: decode steps can run between chunks
- Scheduling flexibility: prefill becomes preemptible and makes fair device sharing easier
- Peak memory usage: each chunk's intermediate activations are freed immediately

- Longer Time-To-First-Token (TTFT): prefill is split into many small iterations, each separated by scheduling overhead so the overall prefill latency may increase
- Lower prefill throughput: chunking into smaller fragments reduces per-chunk utilization

Systems-Level Optimizations

- ~~FlashAttention~~
- ~~Continuous Batching~~
- ~~PagedAttention~~
- ~~Chunked Prefill~~
- **Disaggregated Inference**
- Prefix Caching
- Multi-LoRA Serving
- Compilers

Prefill vs. Decode

Prefill

- Compute-bound: requires large matrix multiplications over entire input sequence
- Memory-intensive: generates large intermediate activations and attention matrices
- Batch-friendly: benefits from processing multiple long sequences together
- Throughput-optimized: performant when device is saturated with large, uninterrupted workloads

Decode

- Memory-bandwidth-bound: small per-token computation, limited by KV cache access
- Latency-sensitive: low inter-token latency for interactive applications
- Dynamic batching: requires fine-grained scheduling to handle variable-length sequences
- Responsiveness-optimized: performant with frequent, small scheduling opportunities

When both phases run on the same device, they compete for resources that create unavoidable tradeoffs.

Disaggregated Inference

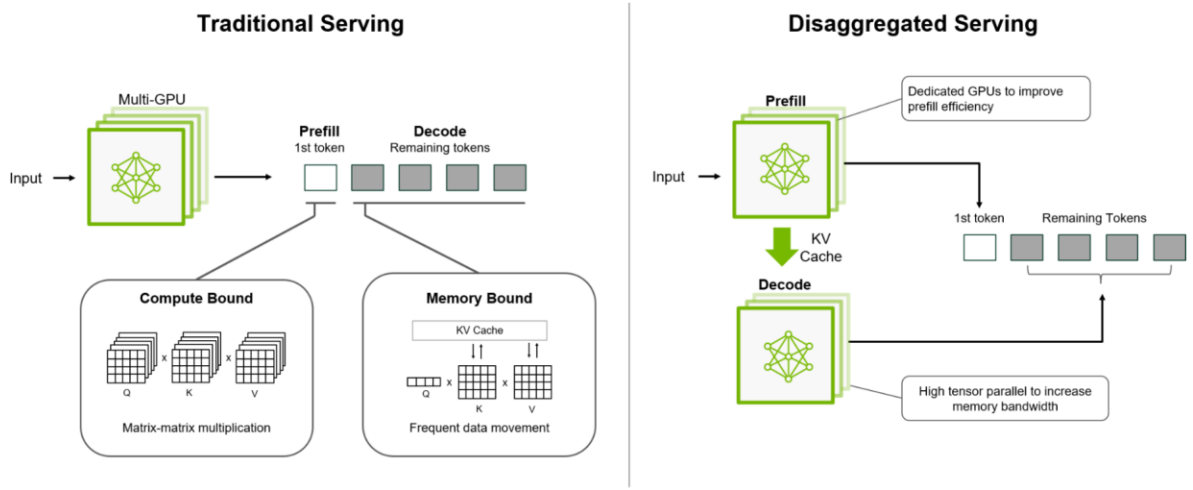
Key idea: execute prefill and decode on separate devices or hardware tiers to optimize for specific workload characteristics.

- Prefill devices can prioritize compute, decode devices can prioritize memory bandwidth
- Different device types can be used for each stage to improve resource utilization
  - E.g. high-compute device for prefill, memory-optimized device for decode
- Long prefill operations no longer block decode requests

Zhong et al., “DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving”, OSDI 2024

Disaggregated Inference (cont.)

- Two logical clusters
  - Route request to prefill device
  - Transfer KV cache to decode device
- Each cluster uses independent scheduling
  - Prefill batches long prompts together
  - Decode uses continuous batching
- Distribute incoming requests across prefill devices
- Distribute completed prefill requests among decode devices



Source: NVIDIA, "NVIDIA Dynamo, A Low-Latency Distributed Inference Framework for Scaling Reasoning AI Models", https://developer.nvidia.com/blog/introducing-nvidia-dynamo-a-low-latency-distributed-inference-framework-for-scaling-reasoning-ai-models/

Challenges with Disaggregated Inference

State Transfer Overhead

- After prefill completes, the KV cache must be transferred from prefill devices to decode devices

System Complexity

- Two-tier scheduling, load balancing across prefill and decode clusters, failure handling, state management

Resource Fragmentation

- Separating into two clusters can lead to fragmentation (one cluster may be underutilized while the other is overloaded, resources allocated to one cluster cannot be easily repurposed for the other)

Cost Considerations

- Infrastructure overhead, network costs, operational complexity, etc.

Systems-Level Optimizations

- ~~FlashAttention~~
- ~~Continuous Batching~~
- ~~PagedAttention~~
- ~~Chunked Prefill~~
- ~~Disaggregated Inference~~
- **Prefix Caching**
- Multi-LoRA Serving
- Compilers

Prefix Caching

- Each request recomputes KV cache for entire prompt
- Many scenarios have shared prefixes, including chatbots, RAG, code completion
- **Key idea: cache and reuse KV computations for shared prefixes.**
  - Hash prefix tokens, check cache for prefix token hash, continue from cached state

```
Request 1: [Compute KV: System+Examples] → [Compute KV: Query1] → Generate
Request 2: [Compute KV: System+Examples] → [Compute KV: Query2] → Generate
Request 3: [Compute KV: System+Examples] → [Compute KV: Query3] → Generate
           └──────────── Redundant ──────────────┘
```

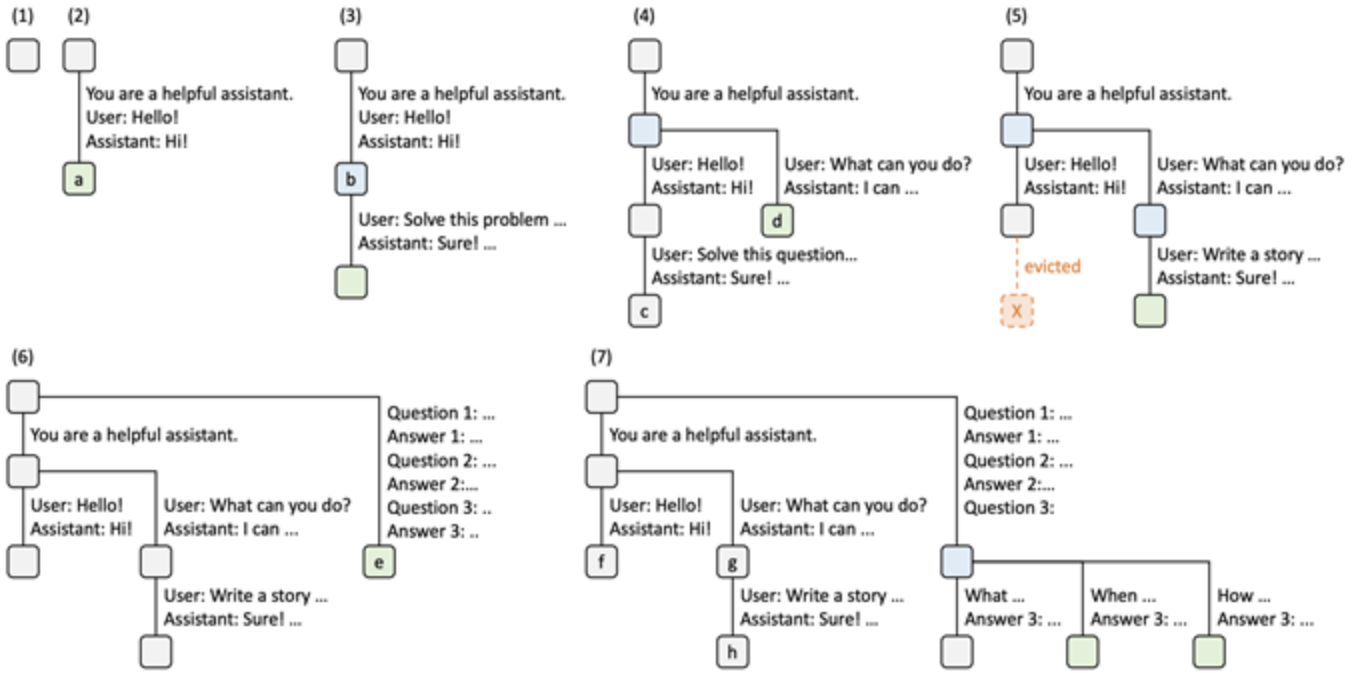
Without prefix caching

```
First request:  [Compute KV: System+Examples] → Cache

Request 1: [Load cached KV] → [Compute: Query1] → Generate  ✓ Fast
Request 2: [Load cached KV] → [Compute: Query2] → Generate  ✓ Fast
Request 3: [Load cached KV] → [Compute: Query3] → Generate  ✓ Fast
```

With prefix caching

RadixAttention

- Using a radix tree to store the KV cache enables efficient (linear-time) prefix matching



Zheng et al., "Efficiently Programming Large Language Models using SGLang", NeurIPS 2024

Summary

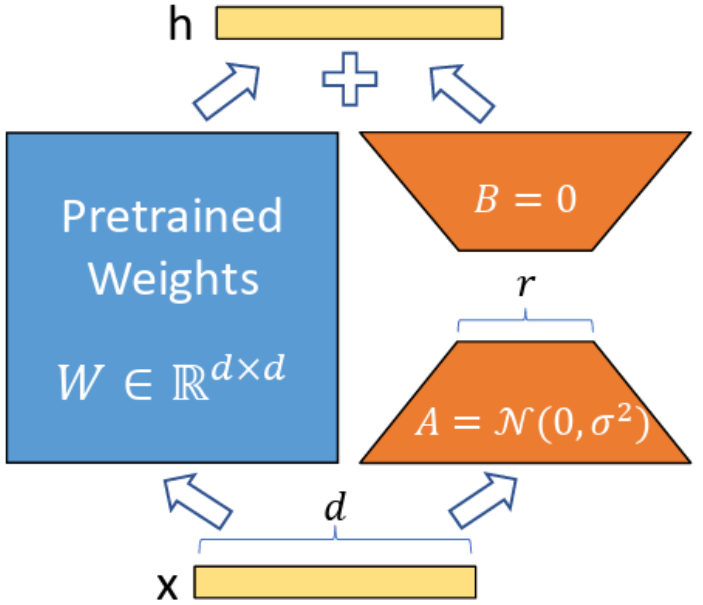
Technique/Feature	Target Metric	Method
FlashAttention	TTFT	Reduce memory traffic
FlashDecoding	ITL	Increase computation utilization
Continuous Batching	Throughput	Increase computation utilization
PagedAttention	Throughput	Increase memory utilization
Chunked Prefill	ITL	Increase computation utilization
Disaggregated Inference	TTFT, ITL, throughput	Increase resource utilization
Prefix Caching	Latency	Reduce duplicated computation

Systems-Level Optimizations

- ~~FlashAttention~~
- ~~Continuous Batching~~
- ~~PagedAttention~~
- ~~Chunked Prefill~~
- ~~Disaggregated Inference~~
- ~~Prefix Caching~~
- **Multi-LoRA Serving**
- Compilers

Low-Rank Adaptation (LoRA)

- Deploying separate full models for individual use cases can be prohibitively expensive
- **Low-Rank Adaptation (LoRA)** is a parameter-efficient fine-tuning (PEFT) technique to adapt LLMs for specific tasks
- **Key idea: freeze original model weights and inject adapters (pairs of small matrices) into model layers**
- LoRA provides computational and memory efficiency
  - LoRA adapter size is typically less than 1% of base model size



Source: Hu et al., "LoRA: Low-Rank Adaptation of Large Language Models", ICRL 2022

LoRA Computation During Inference

LoRA adapters can be applied in one of two ways:

1. Runtime merging (online merging): compute the base model and LoRA contribution separately, then combine the two

$$y = Wx + \alpha \cdot (AB)x = Wx + \alpha \cdot A(Bx)$$

   - Keeps base model weights frozen, enables dynamic adapter swapping, and adds computational overhead (extra matrix multiplication and addition every layer)

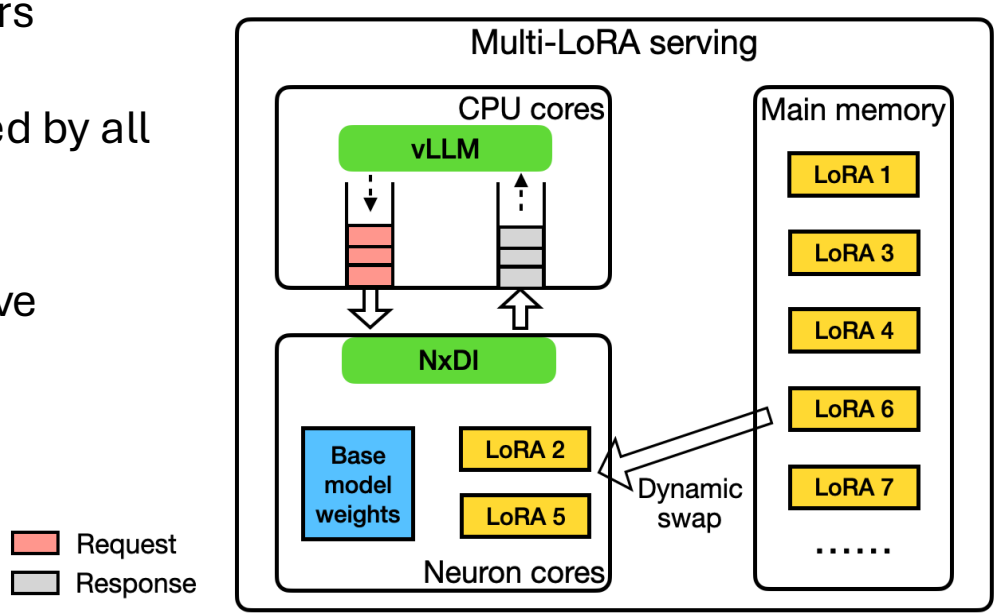
2. Pre-merged weights (offline merging): precompute a new merged weight matrix offline and use the merged weights directly

$$y = (W + \alpha \cdot AB)x$$

   - Eliminates runtime overhead, requires separate base model per adapter, and increases memory footprint (negates memory benefits of LoRA)

Multi-LoRA Serving

- In multi-LoRA serving, a single base model serves multiple specialized variants by dynamically loading adapters
- The base model weights are loaded once and shared by all adapters, but device memory is limited
- To serve hundreds or thousands of adapters, inactive adapters can be offloaded to CPU memory
- **Static multi-LoRA serving:** adapters are preloaded before serving begins (no swapping)
- **Dynamic multi-LoRA serving:** adapters can be loaded and unloaded on demand



Challenges with Multi-LoRA Serving

- Adapter loading latency: transferring adapter weights from CPU host to device can take 10-100 ms
  - Loads can be parallelized across devices, tensors can be coalesced to better utilize memory bandwidth
- Cache management: which adapters should reside in device memory?
  - LRU, LFU, size-aware eviction, popularity-based preloading policies
- Prefetching: predicting which adapters will be needed in the future
  - Request-based prefetching, predictive prefetching, batch-aware prefetching
- Batching across adapters: how to efficiently batch requests using different adapters?

Multi-LoRA Serving Tradeoffs

Memory vs. Latency

- Static serving: higher memory usage, lower latency
- Dynamic serving: lower memory usage, higher latency (due to swapping)

Throughput vs. Flexibility

- Few adapters, large batches: higher throughput per adapter
- Many adapters, small batches: lower throughput per adapter, but serves more use cases

Complexity vs. Efficiency

- Simple static serving: easy to implement, but memory-limited
- Sophisticated dynamic serving: complex memory management, but enables large-scale deployment

Systems-Level Optimizations

- ~~FlashAttention~~
- ~~Continuous Batching~~
- ~~PagedAttention~~
- ~~Chunked Prefill~~
- ~~Disaggregated Inference~~
- ~~Prefix Caching~~
- ~~Multi-LoRA Serving~~
- **Compilers**

Kernel Fusion

Key idea: combine multiple operations into single kernel to minimize traffic between on-chip memory (SRAM) and device memory (HBM).

- Without fusion, each operation in a computation graph reads input tensors from HBM to on-chip memory, performs computation, and writes output tensors back to HBM
- With fusion, intermediate results are kept in fast on-chip memory to eliminate roundtrips to HBM
  - On-chip memory provides ~20x higher bandwidth than HBM
- Tiling can be used to process large tensors in cache-friendly blocks
  - Applied to FlashAttention

Compilation Techniques: Define-Then-Run

define-then-run (static graph): builds a static shape computation graph, runs after everything is defined

- Graph is fixed before any data flows through
- Preferred when there are fixed model shapes or can be well predicted at compilation time (e.g., CNNs, dense LLMs such as LLaMA3.x)
- Benefits: optimized compute graph (kernel fusion, memory planning, scheduling)
- Examples: TensorFlow/XLA, PyTorch XLA (TPU), Jax (@jit)
- Challenges with MoE: dynamic token dispatch, and irregular communication
    - TPU/XLA: pad each expert inputs to a fixed “capacity” (wasting compute)
    - Mitigation: dynamic control flow to skip padding, memory is still allocated

Compilation Techniques: Define-By-Run

define-by-run (dynamic graph / eager execution): operations run immediately, no precompiled global graph, shapes resolved dynamically as graph builds

- Each run can have a different graph structure
- Preferred when tensor shapes vary (e.g. variable-length sequences, adaptive computation in MoE)
- Benefits: flexible, debuggable, and research-friendly
- Examples: PyTorch eager mode, PyTorch 2.0 with torch.compile (captures sub-graphs for optimization, overall dynamic, still prefers stable shapes to avoid overhead)

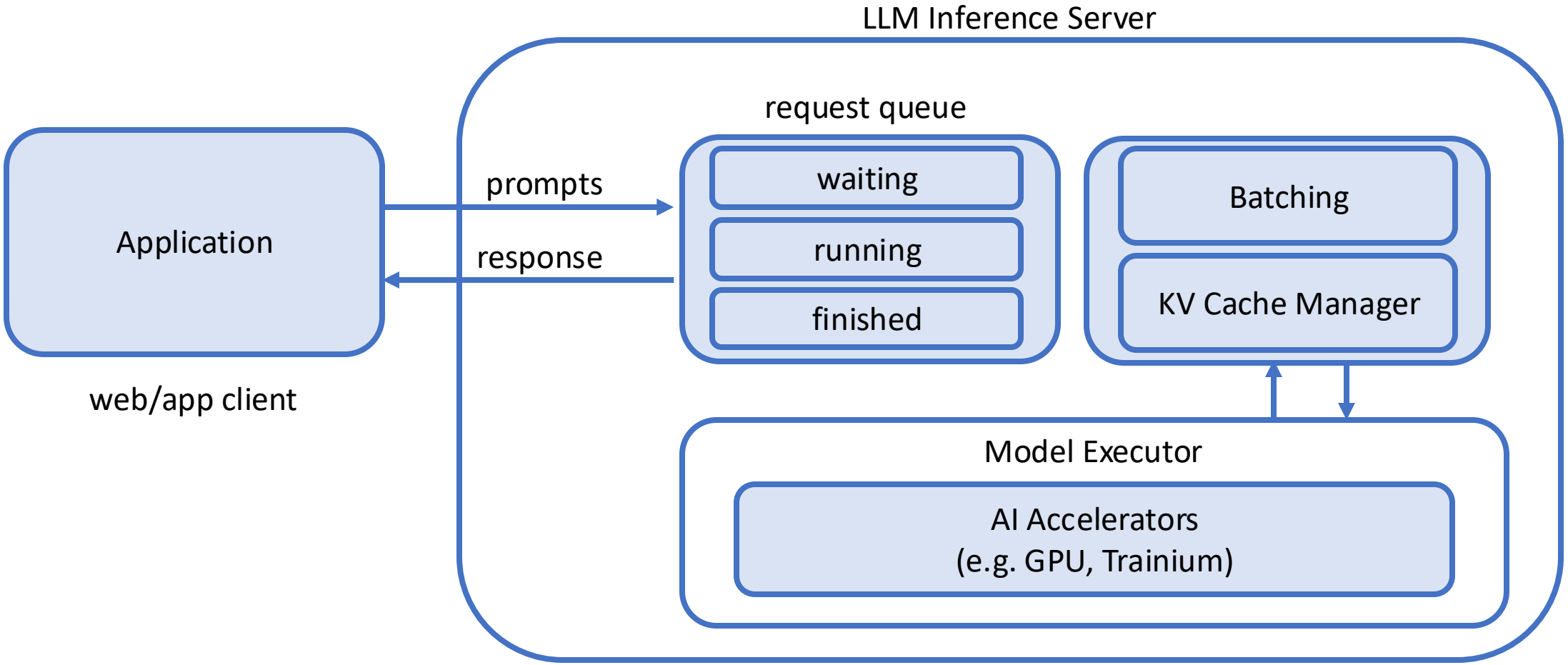
Tutorial Outline

- ~~Primer: Foundations of Generative Inference~~
- ~~Algorithmic and Modeling-Level Inference Optimizations~~
- ~~Systems-Level Optimizations~~
- **Open-Source Frameworks and Tools**

Open-Source Frameworks and Tools

- **Inference Server**
- vLLM
- SGLang
- Case Study

Inference Server



Open-Source Frameworks and Tools

- ~~Inference Server~~
- **vLLM**
- SGLang
- Case Study

vLLM

vLLM is an open-source, high-performance inference engine designed to optimize Large Language Model (LLM) deployment. Its core features are:

- Tensor, pipeline, data, and expert parallelism for distributed inference
- Continuous batching, **PagedAttention**, chunked prefill
- OpenAI-Compatible API
- Quantization (GPTQ, AWQ, INT4, INT8, FP8)
- Multi-LoRA serving, heterogeneous batching
- Speculative decoding

vLLM is known for its high throughput and near zero wasted memory

vLLM, “East, fast, and cheap serving for everyone”, https://docs.vllm.ai/en/latest/

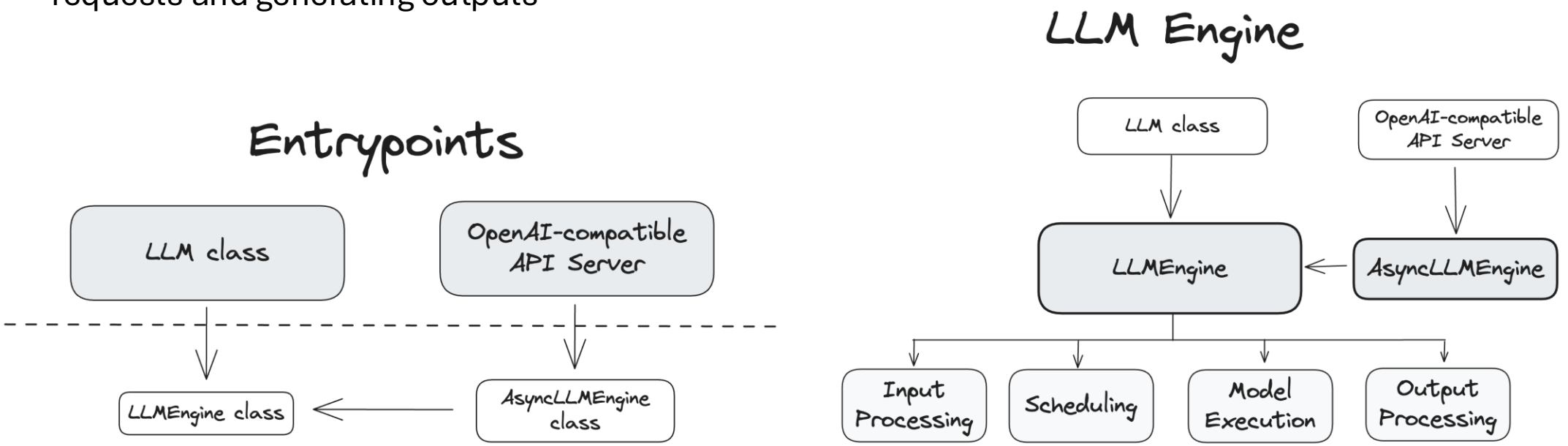
vLLM: Model Support

- Supports a wide range of models, including the Llama, Mistral, and Qwen families and other LLMs
- Excels in many scenarios:
  - Chatbots & AI assistants: handling thousands of concurrent users with low latency
  - Content generation: high-throughput tasks, such as code generation or document summarization
  - Multimodal models: supports vision-language models (e.g. LLaVA) via extensions

vLLM, “East, fast, and cheap serving for everyone”, https://docs.vllm.ai/en/latest/

vLLM Architecture

- vLLM provides a class with a Python interface (offline serving) and an OpenAI-compatible server
- The `LLMEngine` class is the core component of the vLLM engine responsible for receiving client requests and generating outputs



vLLM, "East, fast, and cheap serving for everyone", https://docs.vllm.ai/en/latest/

Open-Source Frameworks and Tools

- ~~Inference Server~~
- ~~vLLM~~
- **SGLang**
- Case Study

SGLang

SGLang is another open-source, high-performance serving framework for large language models and multimodal models. Its core features are:

- Tensor, pipeline, data, and expert parallelism for distributed inference
- **RadixAttention** for prefix caching
- Continuous batching, PagedAttention, chunked prefill
- Disaggregated inference
- Quantization (GPTQ, AWQ, INT4, INT8, FP8)
- Speculative decoding

SGLang is also known for its high throughput and efficient serving

Zheng et al., “SGLang: Efficient Execution of Structured Language Model Programs”, NeurIPS 2024

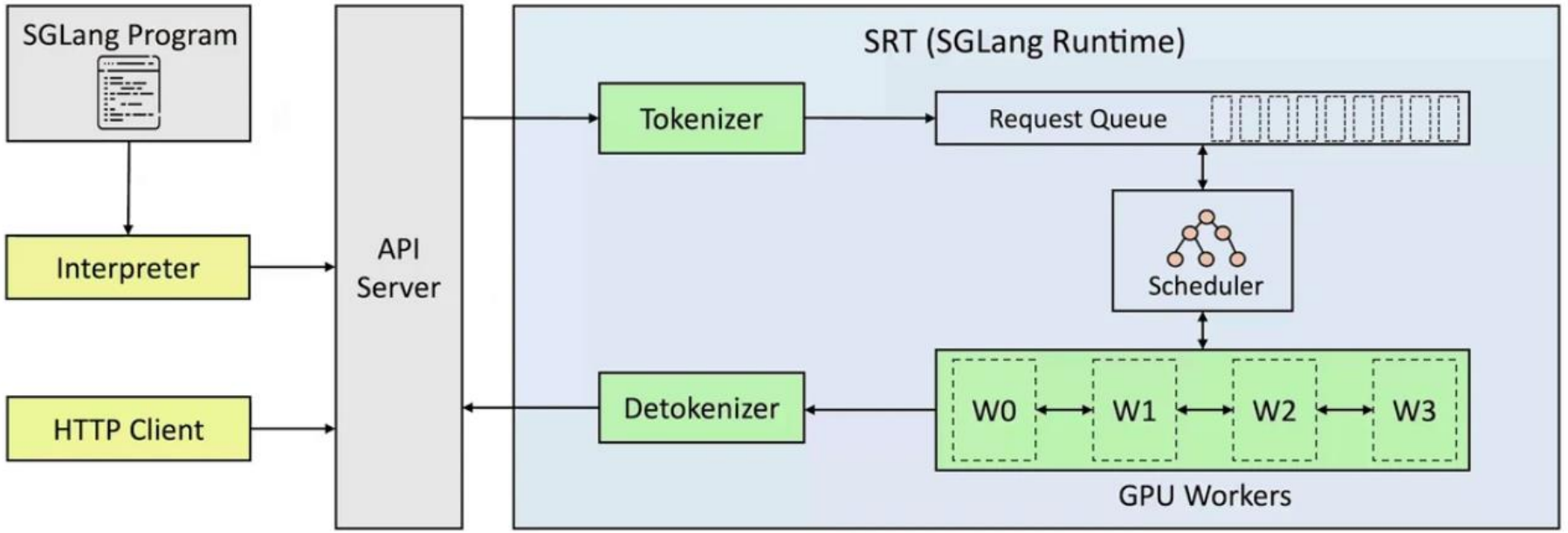
SGLang: Model Support

- Supports a wide range of models, including the Llama, DeepSeek, Kimi, GLM, GPT, Gemma, Mistral, and Qwen families and other LLMs
- Excels in many scenarios:
  - Heavy prefix sharing patterns (best-in-class prefix caching)
  - Complex multi-step LLM applications
  - Agentic workflows

Zheng et al., “SGLang: Efficient Execution of Structured Language Model Programs”, NeurIPS 2024

SGLang Architecture

- Frontend language (Python-embedded DSL) makes complex LLMs easy to write
- Backend runtime contains optimizations to make LLMs fast to execute



Zheng et al., "SGLang: Efficient Execution of Structured Language Model Programs", NeurIPS 2024

Framework Comparison

Framework	Attention Backend	Continuous Batching	KV Cache Management	Chunked Prefill	Prefix Caching	Disaggregated Inference	Multi-LoRA Serving
vLLM	FlashAttention	yes	PagedAttention	yes	block-level hash	yes	yes
SGLang	FlashInfer	yes	RadixAttention	yes	radix tree	yes	yes
HuggingFace Transformers	FlashAttention	no	static/pool-based allocation	no	none	no	yes
Text Generation Interface (TGI)	FlashAttention	yes	PagedAttention	yes	yes	no	yes

Open-Source Frameworks and Tools

- ~~Inference Server~~
- ~~vLLM~~
- ~~SGLang~~
- **Case Study**

Case Study

Using Speculative Decoding and Quantization to Improve Llama3.1 405B Performance on Trn2

Speculative Decoding and Quantization for Llama3.1 405B on Trn2

We will demonstrate how to optimize inference performance for the Llama3.1 405B model on a **`trn2.48xlarge`** instance using speculative decoding and quantization (bf16 --> FP8). We will compile and load the model into a vLLM server and measure performance using LLMPerf.

Steps:

1. Compile the model
2. Start the vLLM server
3. Measure performance using LLMPerf

Source: Using Speculative Decoding and Quantization to Improve Llama3.1 405B Performance on Trn2

Compile the Model

Baseline

```
# Replace this with the path where you downloaded and saved the model files.
MODEL_PATH="/home/ubuntu/models/Llama-3.1-405B-Instruct/"
# This is where the compiled model will be saved. The same path
# should be used when launching vLLM server for inference.
COMPILED_MODEL_PATH="/home/ubuntu/traced_model/Llama-3.1-405B-Instruct/"

NUM_CORES=128
TP_DEGREE=64
LNC=2

export NEURON_RT_VIRTUAL_CORE_SIZE=$LNC
export NEURON_RT_NUM_CORES=$((NUM_CORES/NEURON_RT_VIRTUAL_CORE_SIZE))
export NEURON_RT_EXEC_TIMEOUT=600


inference_demo \
    --model-type llama \
    --task-type causal-lm \
        run \
        --model-path $MODEL_PATH \
        --compiled-model-path $COMPILED_MODEL_PATH \
        --torch-dtype bfloat16 \
        --start_rank_id 0 \
        --local_ranks_size $TP_DEGREE \
        --tp-degree $TP_DEGREE \
        --batch-size 1 \
        --max-context-length 12288 \
        --seq-len 12800 \
        --on-device-sampling \
        --top-k 1 \
        --fused-qkv \
        --sequence-parallel-enabled \
        --qkv-kernel-enabled \
        --attn-kernel-enabled \
        --mlp-kernel-enabled \
        --cc-pipeline-tiling-factor 1 \
        --pad-token-id 2 \
        --enable-bucketing \
        --context-encoding-buckets 2048 4096 10240 12288 \
        --token-generation-buckets 12800 \
        --prompt "What is annapurna labs?" 2>&1 | tee log
```

Speculative Decoding + Quantization

```
# Replace this with the path where you downloaded and saved the model files.
MODEL_PATH="/home/ubuntu/models/Llama-3.1-405B-Instruct-FP8-rescaled/"
# Replace this with the path where you downloaded and saved the draft model files.
DRAFT_MODEL_PATH="/home/ubuntu/models/Llama-3.2-1b-instruct/"
# This is where the compiled model (.pt file) and sharded checkpoints will be saved. The same path
# should be used when launching vLLM server for inference.
COMPILED_MODEL_PATH="/home/ubuntu/traced_model/Llama-3.1-405B-Instruct/"
# Add a modules to not convert json file to the model path to specify non quantized modules.
MTNC_FILE_PATH="/home/ubuntu/models/Llama-3.1-405B-Instruct-FP8-rescaled/modules_to_not_convert.json"

NUM_CORES=128
TP_DEGREE=64
LNC=2


export NEURON_RT_VIRTUAL_CORE_SIZE=$LNC
export NEURON_RT_NUM_CORES=$((NUM_CORES/NEURON_RT_VIRTUAL_CORE_SIZE))
export NEURON_RT_EXEC_TIMEOUT=600
export XLA_HANDLE_SPECIAL_SCALAR=1
export UNSAFE_FP8FNCAST=1

inference_demo \
    --model-type llama \
    --task-type causal-lm \
    run \
        --model-path $MODEL_PATH \
        --compiled-model-path $COMPILED_MODEL_PATH \
        --torch-dtype bfloat16 \
        --start_rank_id 0 \
        --local_ranks_size $TP_DEGREE \
        --tp-degree $TP_DEGREE \
        --batch-size 1 \
        --max-context-length 12288 \
        --seq-len 12800 \
        --on-device-sampling \
        --top-k 1 \
        --fused-qkv \
        --sequence-parallel-enabled \
        --qkv-kernel-enabled \
        --attn-kernel-enabled \
        --mlp-kernel-enabled \
        --cc-pipeline-tiling-factor 1 \
        --draft-model-path $DRAFT_MODEL_PATH \
        --enable-fused-speculation \
        --speculation-length 7 \
        --quantized-mlp-kernel-enabled \
        --quantization-type per_channel_symmetric \
        --rmsnorm-quantize-kernel-enabled \
        --prompt "What is annapurna labs?" \
        --modules-to-not-convert-file $MTNC_FILE_PATH \
        --context-encoding-buckets 2048 4096 10240 12288 \
        --token-generation-buckets 12800 2>&1 | tee compile_and_generate_log
```

Start vLLM with the Compiled Model

```
export NEURON_RT_VIRTUAL_CORE_SIZE=2

MODEL_PATH="/home/ubuntu/models/Llama-3.1-405B-Instruct/"
COMPILED_MODEL_PATH="/home/ubuntu/traced_model/Llama-3.1-405B-Instruct/"

export VLLM_NEURON_FRAMEWORK="neuronx-distributed-inference"
export NEURON_COMPILED_ARTIFACTS=$COMPILED_MODEL_PATH  # Re-use the compiled artifacts
VLLM_RPC_TIMEOUT=100000 python -m vllm.entrypoints.openai.api_server \
    --model "$MODEL_PATH" \
    --max-num-seqs 1 \
    --max-model-len 12800 \
    --tensor-parallel-size 64 \
    --no-enable-prefix-caching \
    --port 8000 & PID=$! 2>&1 | tee llama405b_bf16.log
echo "vLLM server started with PID $PID"
```

Baseline

Speculative Decoding + Quantization

```
export NEURON_RT_INSPECT_ENABLE=0
export NEURON_RT_VIRTUAL_CORE_SIZE=2
export XLA_HANDLE_SPECIAL_SCALAR=1
export UNSAFE_FP8FNCAST=1


MODEL_PATH="/home/ubuntu/models/Llama-3.1-405B-Instruct-FP8-rescaled"
DRAFT_MODEL_PATH="/home/ubuntu/models/Llama-3.2-1b-instruct"
COMPILED_MODEL_PATH="/home/ubuntu/traced_models/Llama-3.1-405B-Instruct_fp8"


export VLLM_NEURON_FRAMEWORK="neuronx-distributed-inference"
export NEURON_COMPILED_ARTIFACTS=$COMPILED_MODEL_PATH
VLLM_RPC_TIMEOUT=100000 python -m vllm.entrypoints.openai.api_server \
    --model $MODEL_PATH \
    --max-num-seqs 1 \
    --max-model-len 12800 \
    --tensor-parallel-size 64 \
    --device neuron \
    --speculative-max-model-len 12800 \
    --speculative-model $DRAFT_MODEL_PATH \
    --num-speculative-tokens 7 \
    --use-v2-block-manager \
    --override-neuron-config "{\"enable_fused_speculation\":true, \"quantized-mlp-kernel-enabled\":true, \"quantization-type\":\"per_channel_symmetric\", \"skip_warmup\": true}" \
    --port 8000 & PID=$!
echo "vLLM server started with PID $PID"
```

Measure Performance Using LLMPerf

```
# This should be the same path to which the model was downloaded (also used in the above
MODEL_PATH="/home/ubuntu/models/Llama-3.1-405B-Instruct"
# This is the name of directory where the test results will be saved.
OUTPUT_PATH=llmperf-results-sonnets

export OPENAI_API_BASE="http://localhost:8000/v1"
export OPENAI_API_KEY="mock_key"

python token_benchmark_ray.py \
    --model $MODEL_PATH \
    --mean-input-tokens 10000 \
    --stddev-input-tokens 0 \
    --mean-output-tokens 1500 \
    --stddev-output-tokens 0 \
    --num-concurrent-requests 1\
    --timeout 3600 \
    --max-num-completed-requests 50 \
    --additional-sampling-params '{}' \
    --results-dir $OUTPUT_PATH \
    --llm-api "openai"
```

Compare Performance

Baseline

```
inter_token_latency_s
    p25 = 0.03783673520494379
    p50 = 0.037929154633788834
    p75 = 0.03799374728198055
    p90 = 0.03806084386428147
    p95 = 0.03818095359194858
    p99 = 0.03862880035825585
    mean = 0.03790912092492011
    min = 0.03711292916794487
    max = 0.03867580939426865
    stddev = 0.0002364662521116205
ttft_s
    p25 = 2.437347081664484
    p50 = 2.441959390998818
    p75 = 2.4439403364085592
    p90 = 2.444729209714569
    p95 = 2.445114637189545
    p99 = 79.22927707570342
    mean = 5.451600373298861
    min = 2.427013176959008
    max = 153.00210832804441
    stddev = 21.292646281386 15
end_to_end_latency_s
    p25 = 70.06310007086722
    p50 = 70.09642704750877
    p75 = 70.1557097924524
    p90 = 70.28295350184199
    p95 = 70.56055794338462
    p99 = 148.28325726192182
    mean = 73.19207735829521
    min = 70.00512732309289
    max = 222.50397142698057
    stddev = 21.54750467688136
request_output_throughput_token_per_s
    p25 = 25.417755028050912
    p50 = 25.463487985775544
    p75 = 25.522234144656743
    p90 = 25.6487981126861
    p95 = 25.729858763245502
    p99 = 25.90146713883131
    mean = 25.13808905954906
    min = 8.080754642125802
    max = 26.021214285642255
    stddev = 2.465472136291901
```

```
number_input_tokens
    p25 = 10000.0
    p50 = 10000.0
    p75 = 10000.0
    p90 = 10000.0
    p95 = 10000.0
    p99 = 10000.0
    mean = 10000.0
    min = 10000
    max = 10000
    stddev = 0.0
number_output_tokens
    p25 = 1783.0
    p50 = 1785.0
    p75 = 1789.75
    p90 = 1798.1
    p95 = 1803.55
    p99 = 1816.67
    mean = 1787.92
    min = 1779
    max = 1825
    stddev = 8.54720386310933
Number Of Errored Requests: 0
Overall Output Throughput: 24.421011092151268
Number Of Completed Requests: 50
Completed Requests Per Minute: 0.8195336846889548
```

Speculative Decoding + Quantization

```
inter_token_latency_s
    p25 = 0.008220573497974934
    p50 = 0.008265312568750231
    p75 = 0.008438719224417583
    p90 = 0.00848199803312309
    p95 = 0.008495625438929224
    p99 = 0.011143428944987235
    mean = 0.008419798457414533
    min = 0.008173695931987216
    max = 0.01364151847269386
    stddev = 0.0007612118573477839
ttft_s
    p25 = 2.2543624382815324
    p50 = 2.254961202503182
    p75 = 2.2576071268413216
    p90 = 2.2596270388457924
    p95 = 2.260639927221928
    p99 = 2.2628143909573555
    mean = 2.256157155628316
    min = 2.2534945809748024
    max = 2.2629711360204965
    stddev = 0.0023667267664955545
end_to_end_latency_s
    p25 = 14.586015026085079
    p50 = 14.65608573507052
    p75 = 14.91364526405232
    p90 = 14.977840351965279
    p95 = 15.000083449739032
    p99 = 18.969864878777866
    mean = 14.886235136194154
    min = 14.520539953839034
    max = 22.716861865017563
    stddev = 1.1415236552464672
request_output_throughput_token_per_s
    p25 = 100.64608830743339
    p50 = 102.4148205461138
    p75 = 102.90679421801005
    p90 = 103.02201242683091
    p95 = 103.26614794565539
    p99 = 103.36118277211666
    mean = 101.22055373532301
    min = 66.0742671641385
    max = 103.37081160698546
    stddev = 5.19249551094185
```

```
number_input_tokens
    p25 = 10000.0
    p50 = 10000.0
    p75 = 10000.0
    p90 = 10000.0
    p95 = 10000.0
    p99 = 10000.0
    mean = 10000.0
    min = 10000
    max = 10000
    stddev = 0.0
number_output_tokens
    p25 = 1501.0
    p50 = 1501.0
    p75 = 1501.0
    p90 = 1501.0
    p95 = 1501.0
    p99 = 1501.0
    mean = 1501.0
    min = 1501
    max = 1501
    stddev = 0.0
Number Of Errored Requests: 0
Overall Output Throughput: 100.69986490153724
Number Of Completed Requests: 50
Completed Requests Per Minute: 4.025311055357918
```

TPOT and output token throughput increases by 4x!

Thank You!
Questions?

amazon | science

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)

Reach out to Youngsuk Park (pyoungsu@amazon.com) , Yida Wang (wangyida@amazon.com)