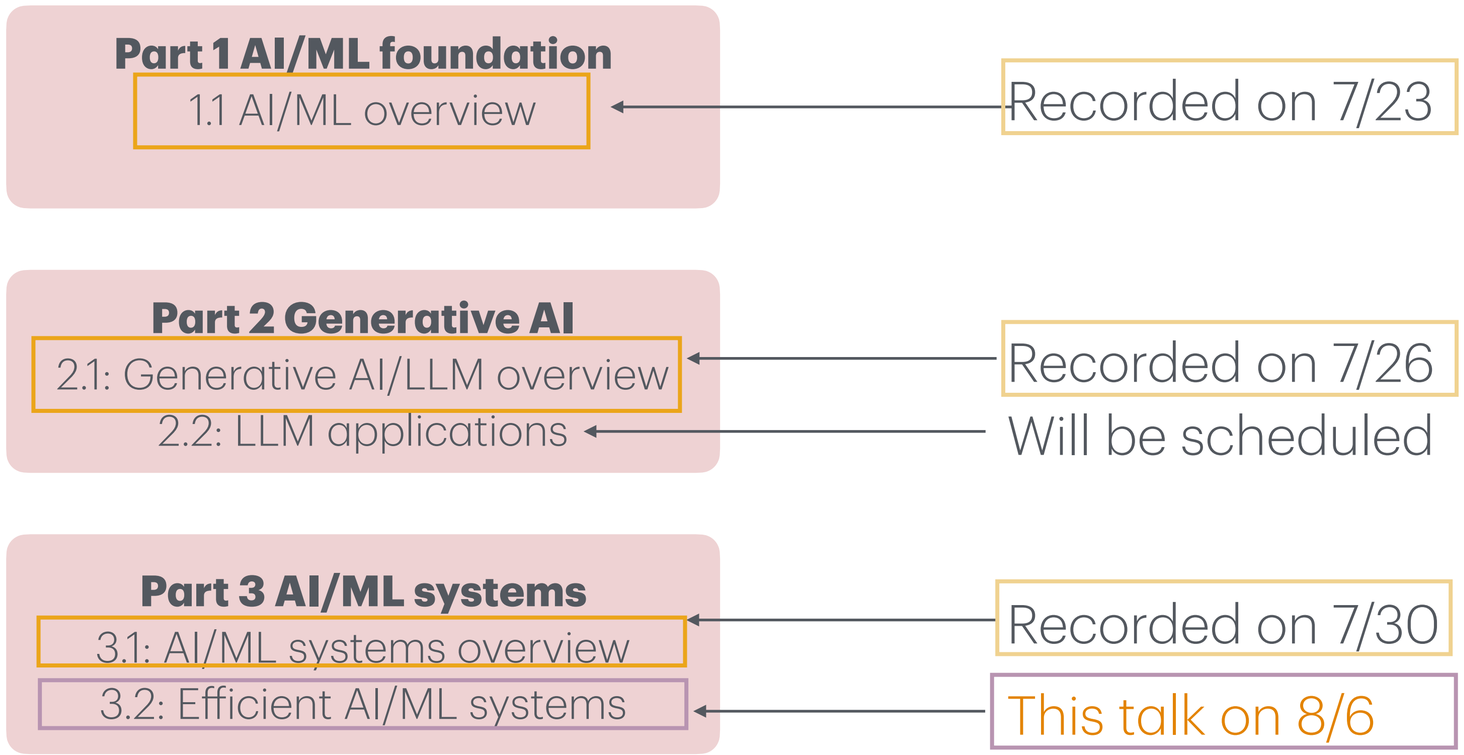
Part 3. AI/ML systems

3.2: Efficient AI/ML systems

Insop S., 8/6/2024, Ver 0.4

AI learning modules



3. AI/ML systems

Contents

3.2: Efficient AI/ML systems

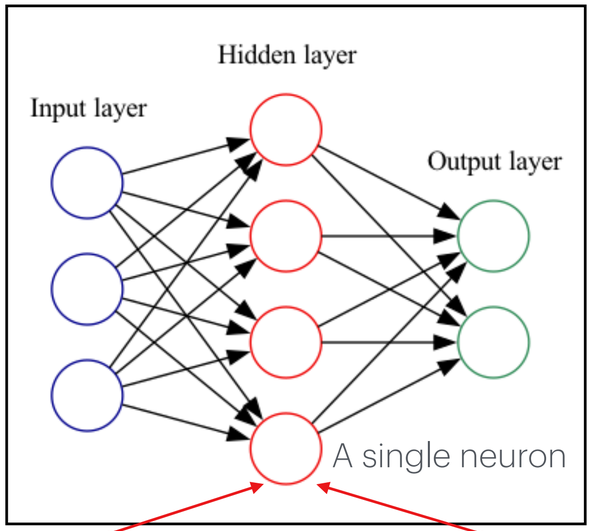
- Efficient LLM inference

Review

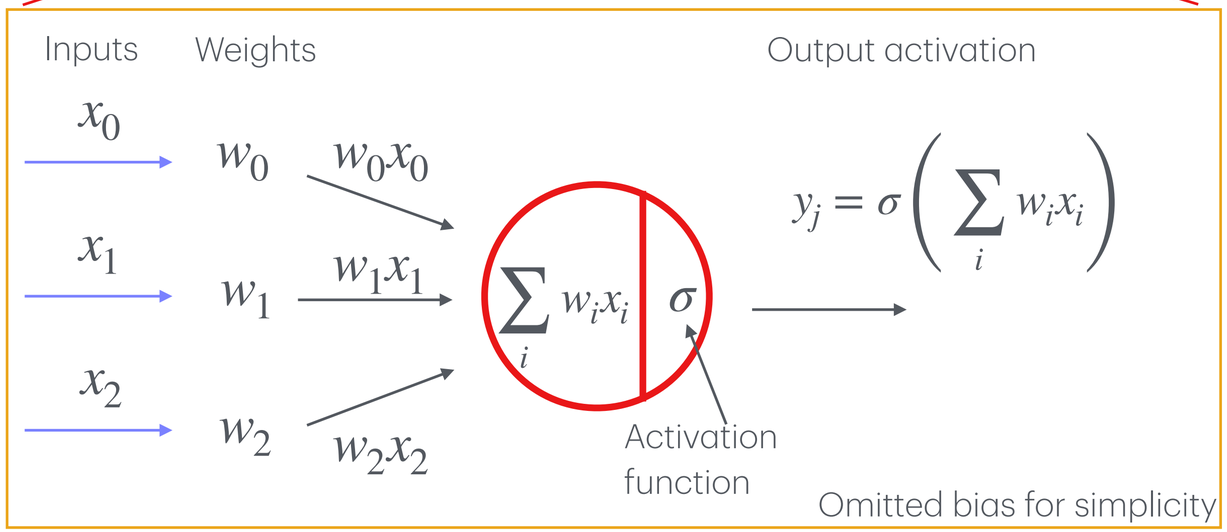
- Machine Learning (ML) learns how to solve problems through data and statistical methods without explicit programming
- Three key ingredients in ML: hypothesis (model), loss (defines the goodness of the model), and optimization (how to find the parameters that minimize the loss).
- A neural network is a basic building block of deep learning and consists of fully connected layers with non-linear activation functions that can learn non-linear boundaries.

ℹ️ Check out the part 1 slide for references, books and neural network code

2 layers neural network with 1 hidden layer

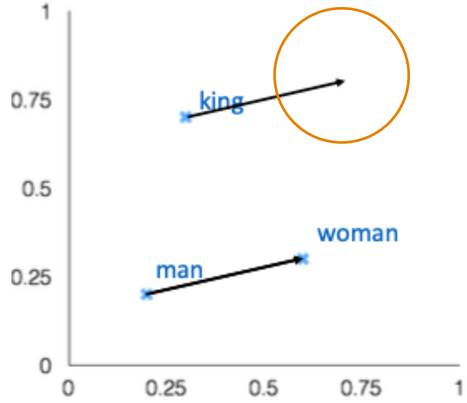


A single neuron internal view

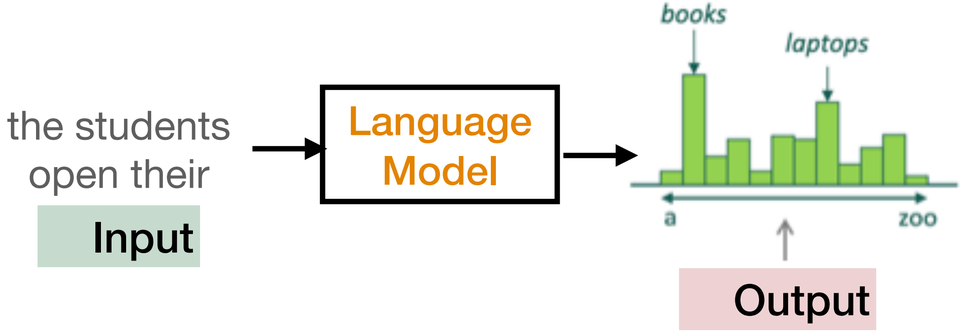


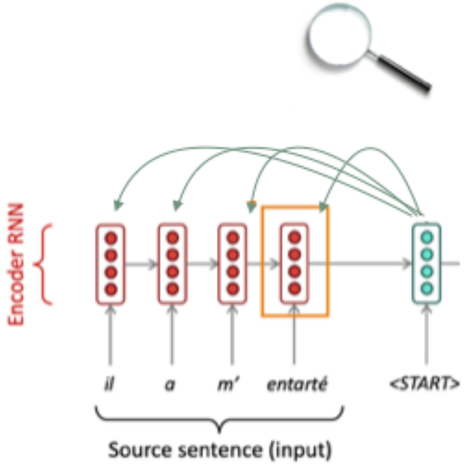
Review

From Part 2.1: Generative AI/LLM overview



- Word2vec: a model that trained to converts words to vector (word embeddings)
- LM (language model): predicts next word given input
- RNN: type of neural network that can process sequential data, which allows stateful computation since it uses hidden state from previous time. Any input sequence length can be processed
- Sequence to sequence model: consists of encoder block and decoder block and used for neural machine translation model
- Attention: a general method in neural network that solves information bottleneck, which directly computes the similarities with all input and focus on the specific part of the input

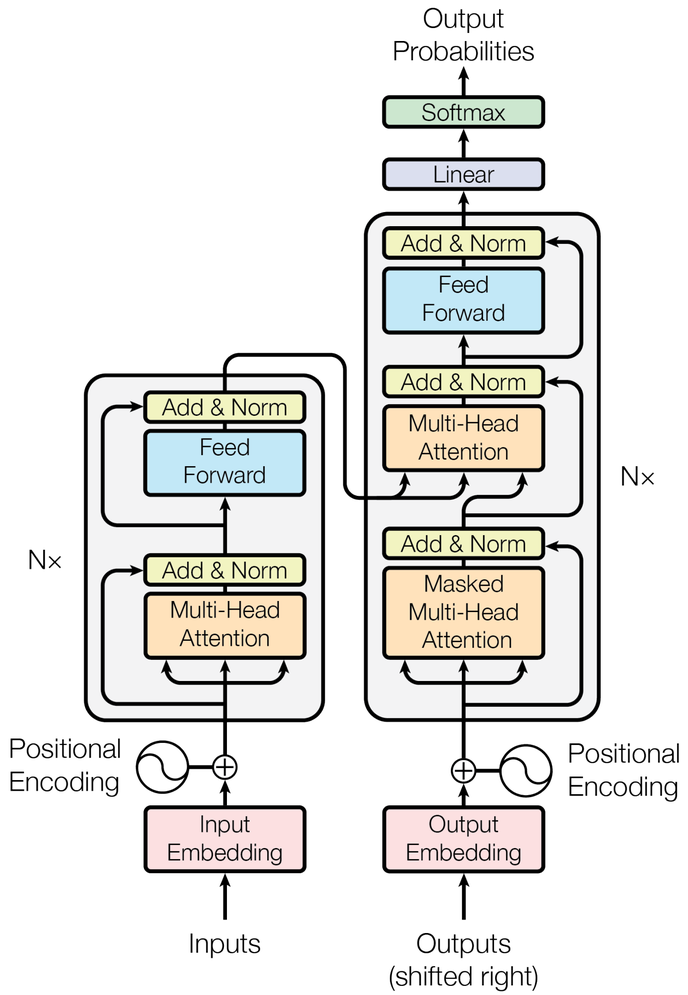




Review

From Part 2.1: Generative AI/LLM overview

- Transformer architecture/model: improves the issues with RNN, such as providing attention paths, enabling parallel computations.
- Transformer block: a core block for the Transformer architecture, which includes *multi-headed self attention* and *feedforward network*.
- Self-supervision: in language model training, it uses the input text to predict the next word without label.
- Pre-training: teaches the model how the world works in general by using self-supervision training
- Fine-tuning: customizes it to a specific task to easier to use.
- Generation (LM inference): the model continues its input (prompt) sequence.



Review

From Part 3.1: AI/ML systems overview

1. Deep learning framework supports:
    - auto differentiation,
    - hardware acceleration,
    - distributed training support
2. Deep learning framework: creates computation graph from model (neural network).
3. Auto differentiation is based on chain rule from calculus.

Review

From Part 3.1: AI/ML systems overview

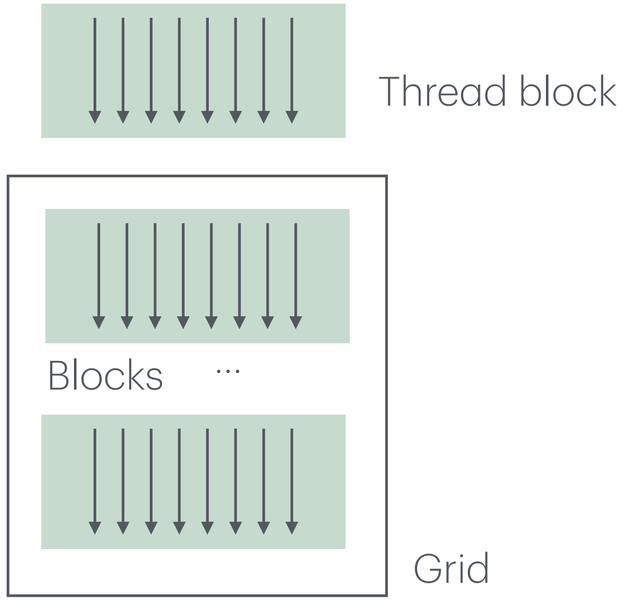
1. GPU: includes many small cores that hides memory latency to produce high throughput

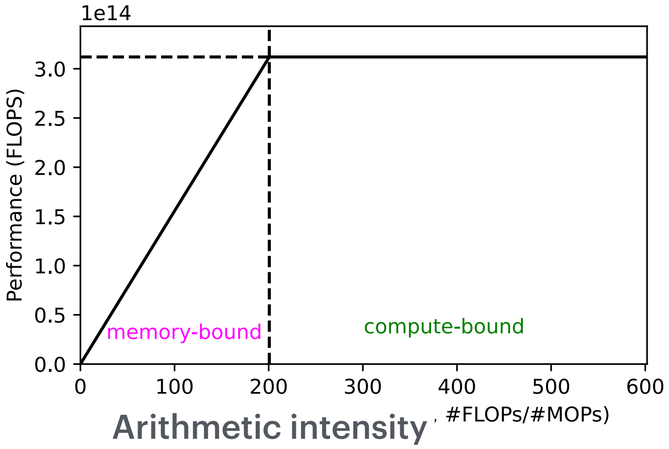
2. GPU programming model: threads are organized as thread blocks and blocks are organized as grid. Threads in a thread block will run the same instructions for different data (SPMD, single program multiple data).

3. Arithmetic intensity: for each floating point operation, how many bytes moved

$$\text{arithemetic intensity} = \frac{\text{Total number of operations}}{\text{Number of bytes read from/written to memory}}$$

4. Roofline model: provides a way to compare application performance against machine performance

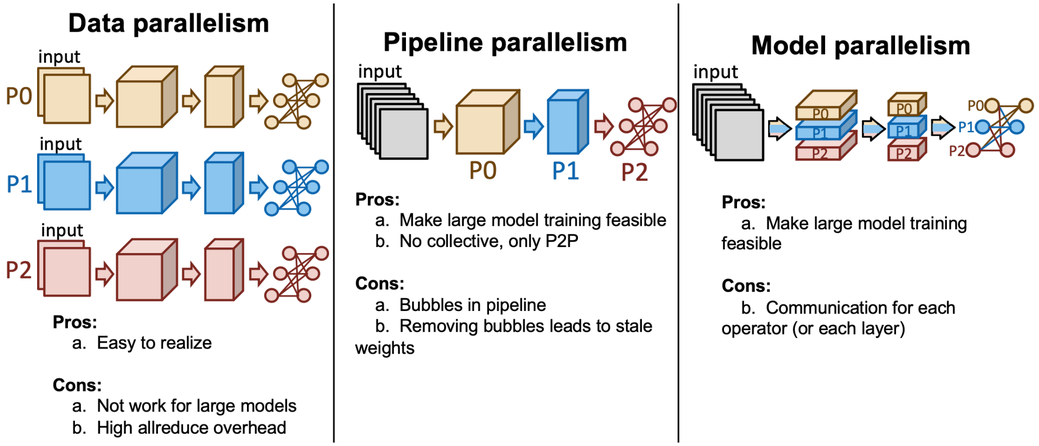




Review

From Part 3.1: AI/ML systems overview

1. Data parallelism: put entire model on multiple processors and feed partitioned data to improve training time
2. Pipeline parallelism: partition model layer boundary and run on multiple processors. It only requires point-to-point communication, but needs to put efforts to reduce bubbles.
3. Tensor parallelism: partition individual layers and run on multiple processors, requires highest network bandwidth and lowest latency
4. To train large scale model requires multiple parallelisms, such as 4D parallelism used by llama3 405b model



3.2: Efficient AI/ML systems

Training and inference cost

Training:

- A large upfront cost for training
- For large models, training could cost tens or hundreds of millions of dollars*

Inference:

- Every time model process input request (such as gpt, copilot generation), it incurs inference cost
- Many applications use LLM generations
- A single LLM API cost is low, but it could accumulate over time in scale

Efficient ML is important: to reduce power used, cost of serving, latency for real-time applications

Ref: [2] *: link

What you will learn about today?

- Challenges in LLM inference
- Recent approaches to the LLM inference challenges

Plan for today

- LLM inference
- Optimization approaches
  - Model compression
  - Algorithm optimization
  - System level optimization
- Hardware aware algorithm design
- Non-transformer based models

LLM inference

LLM inference

LLM inference has two phases:

- Prefill (prompt) phase:
  All input tokens (prompt) are processed *at once in parallel*, 'prefill' tokens for the following generation step
  Prefill phase runs only once

- Decoding (token generation) phase:
  Autoregressive 'decoding' phase, a single next token is generated using all previous tokens *one at a time iteratively*
  Decode phase iteratively generated one token as a time.
  Repeat until i) it reached the pre-defined max length (e.g. 2048 tokens) ii) or generates stop token

Review

Transformer architecture

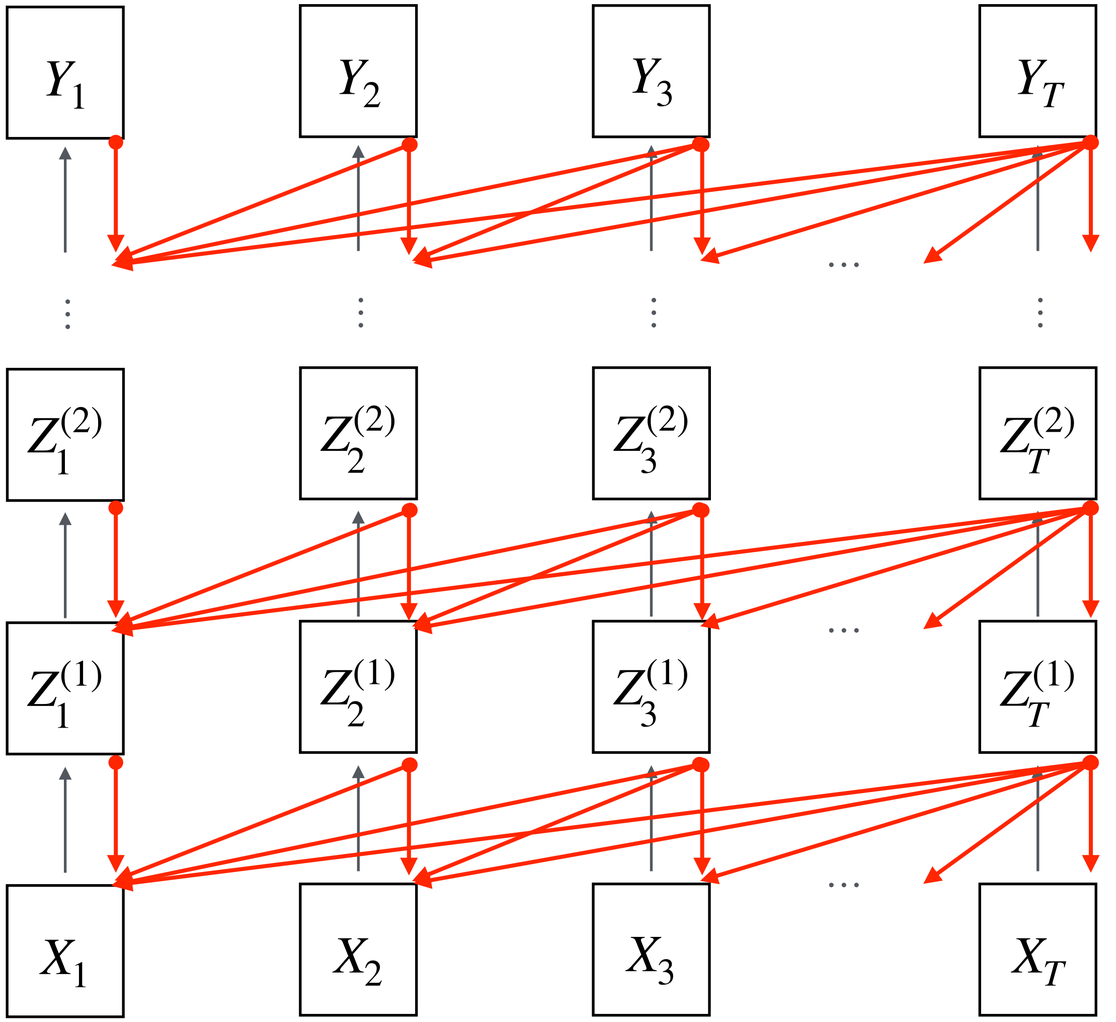
GPT-3 has 96 layers and can handle 2048 time steps

Transformer architecture uses a layers of *transformer block* (Transformer_block()) which has self-attention* and feedforward network layers to process natural language (sequential data).

$$Z_t^{(i+1)} = \text{Transformer\_block}(Z_t^{(i)})$$

Where superscript $i, i+1$th indicate layers, subscript t indicates time step.

For decoder mode language model (an example on the right), self-attention layers attend to all the previous states, which is indicated by red arrow.



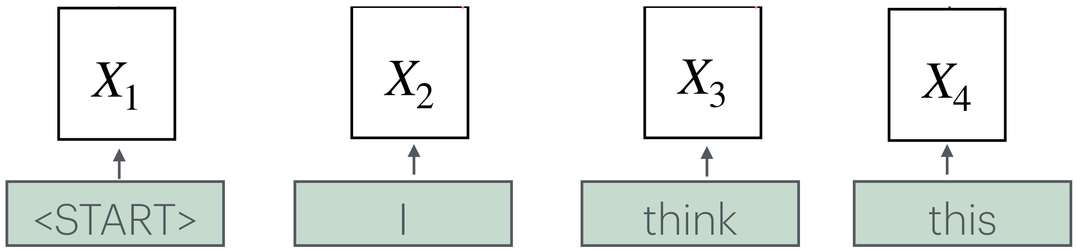
*: for simplicity a single self-attention is used for the description; however, there are multiple self attentions, i.e. multi-head attention (MHA).

Ref: [2]

Prefill phase

Language model is
built with
transformer blocks

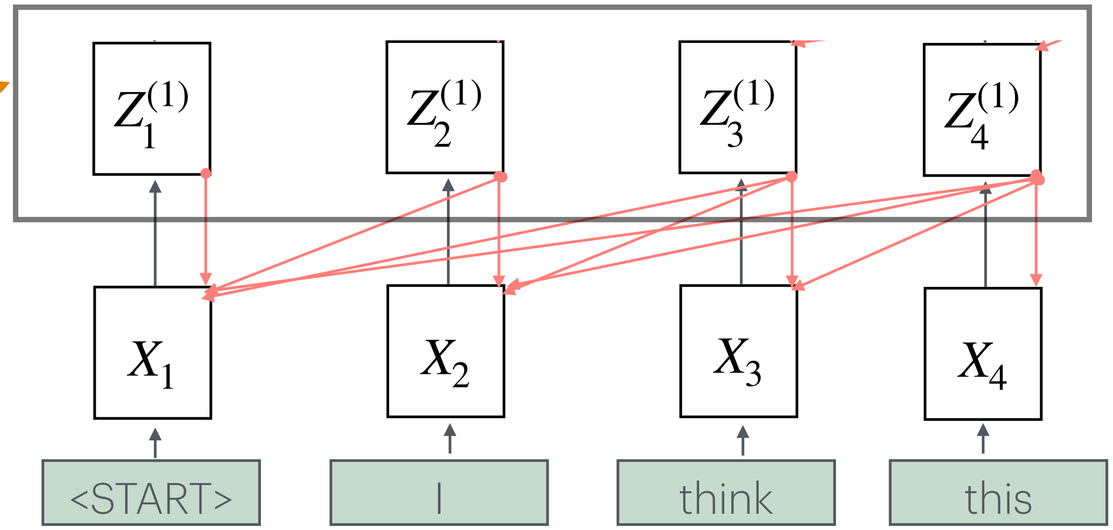
$Z_t^{(i+1)} = \text{Transformer\_block}(Z_t^{(i)})$



Prefill phase

Language model is built with transformer blocks

$Z_t^{(i+1)} = \text{Transformer\_block}(Z_t^{(i)})$



Each layer is transformer block with attention connections to preceding states

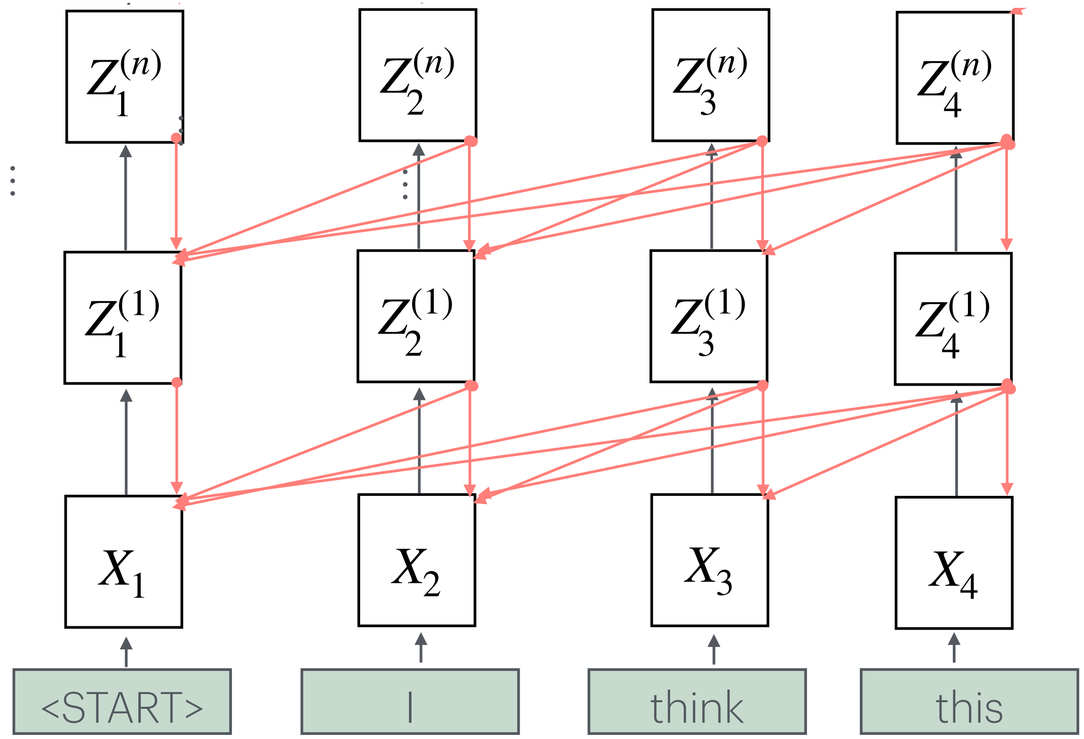
Prompt

During prefill phase, all input tokens are computed together, during the process query, key, values are generated in each layer

Prefill phase

Language model is built with transformer blocks

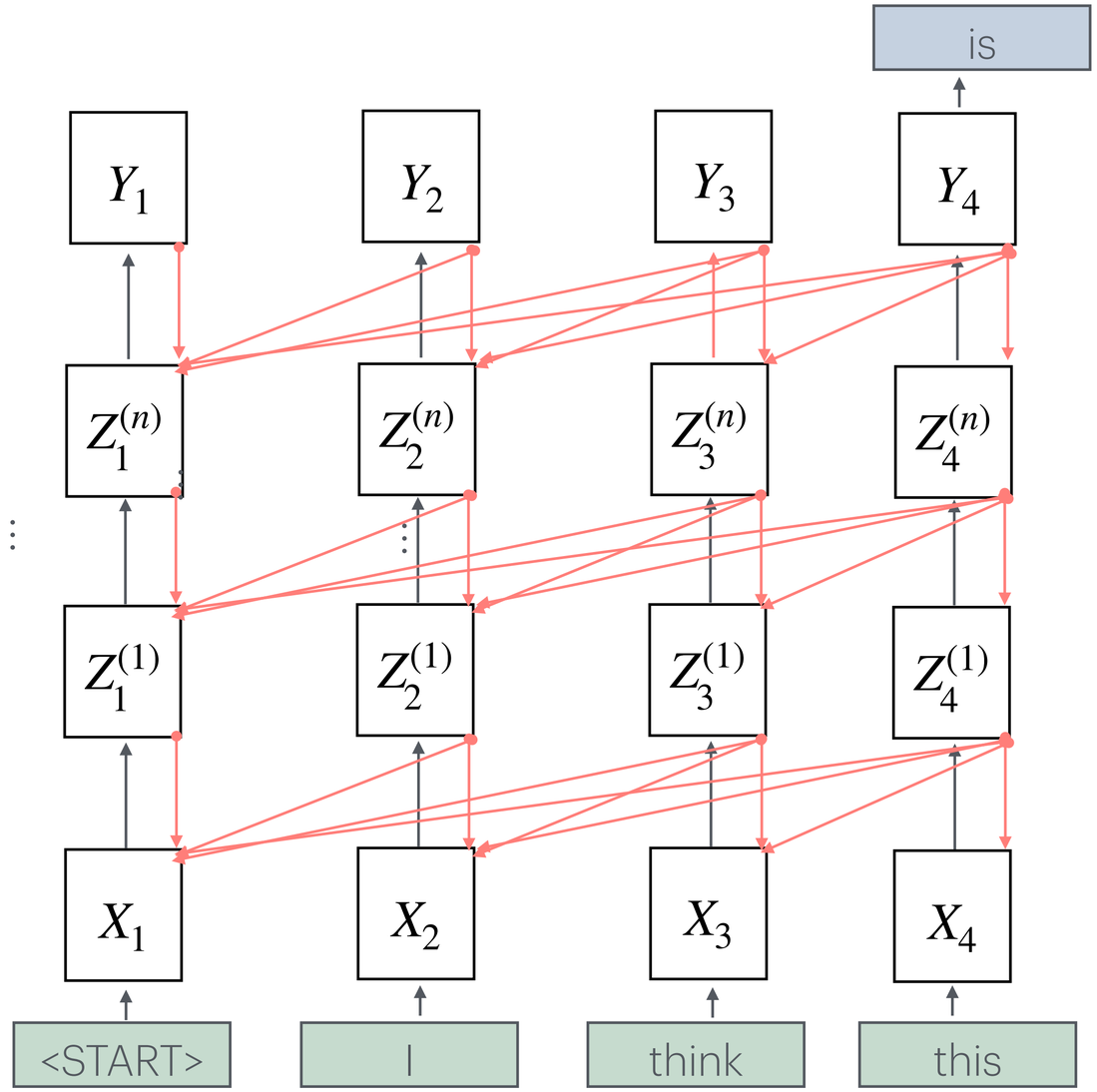
$Z_t^{(i+1)} = \text{Transformer\_block}(Z_t^{(i)})$



Prefill phase

Language model is built with transformer blocks

$Z_t^{(i+1)} = \text{Transformer\_block}(Z_t^{(i)})$



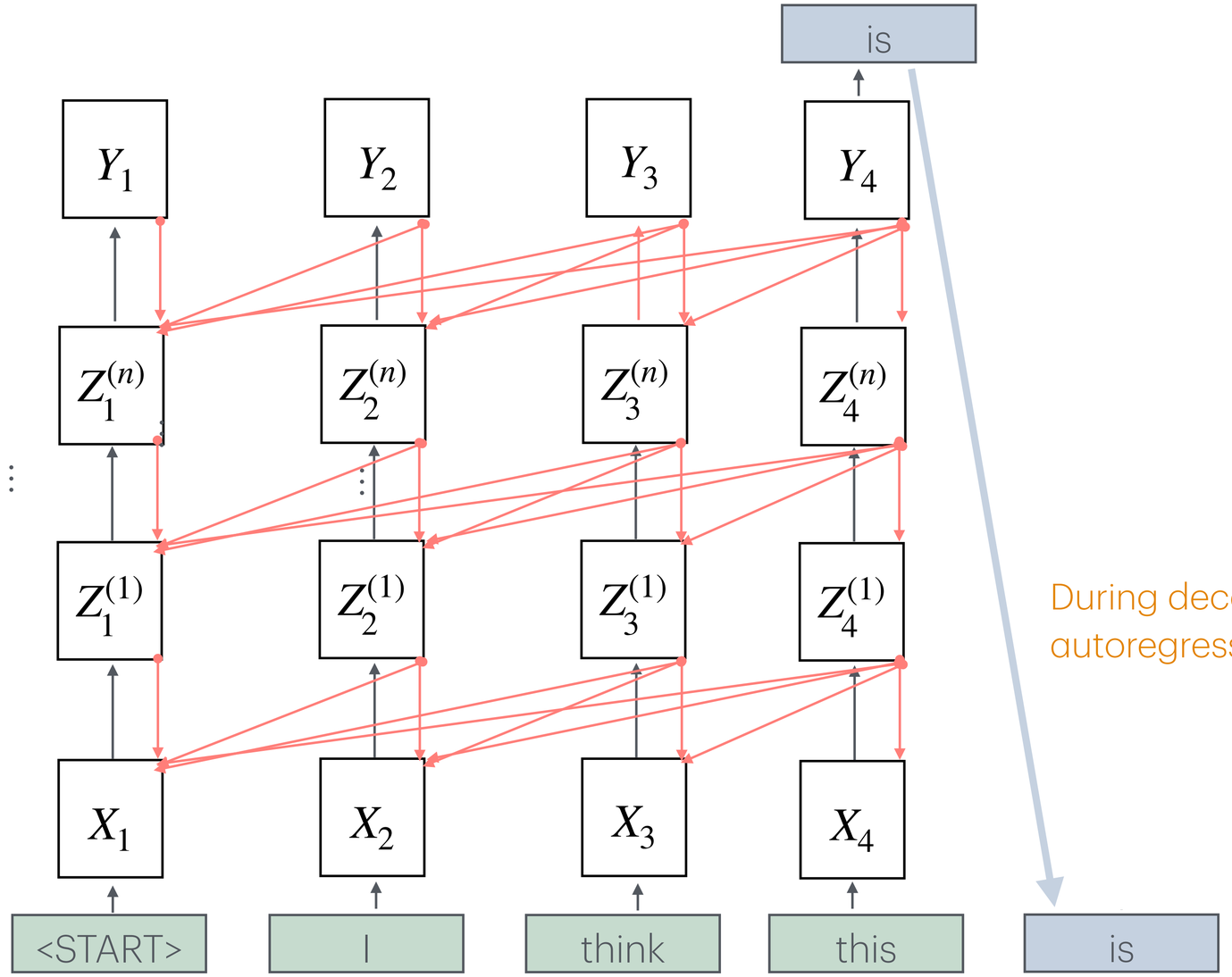
Key and values of all previous steps will not be changed, so cached and use for the next steps

Decode phase

Generated word

Language model is built with transformer blocks

$Z_t^{(i+1)} = \text{Transformer\_block}(Z_t^{(i)})$



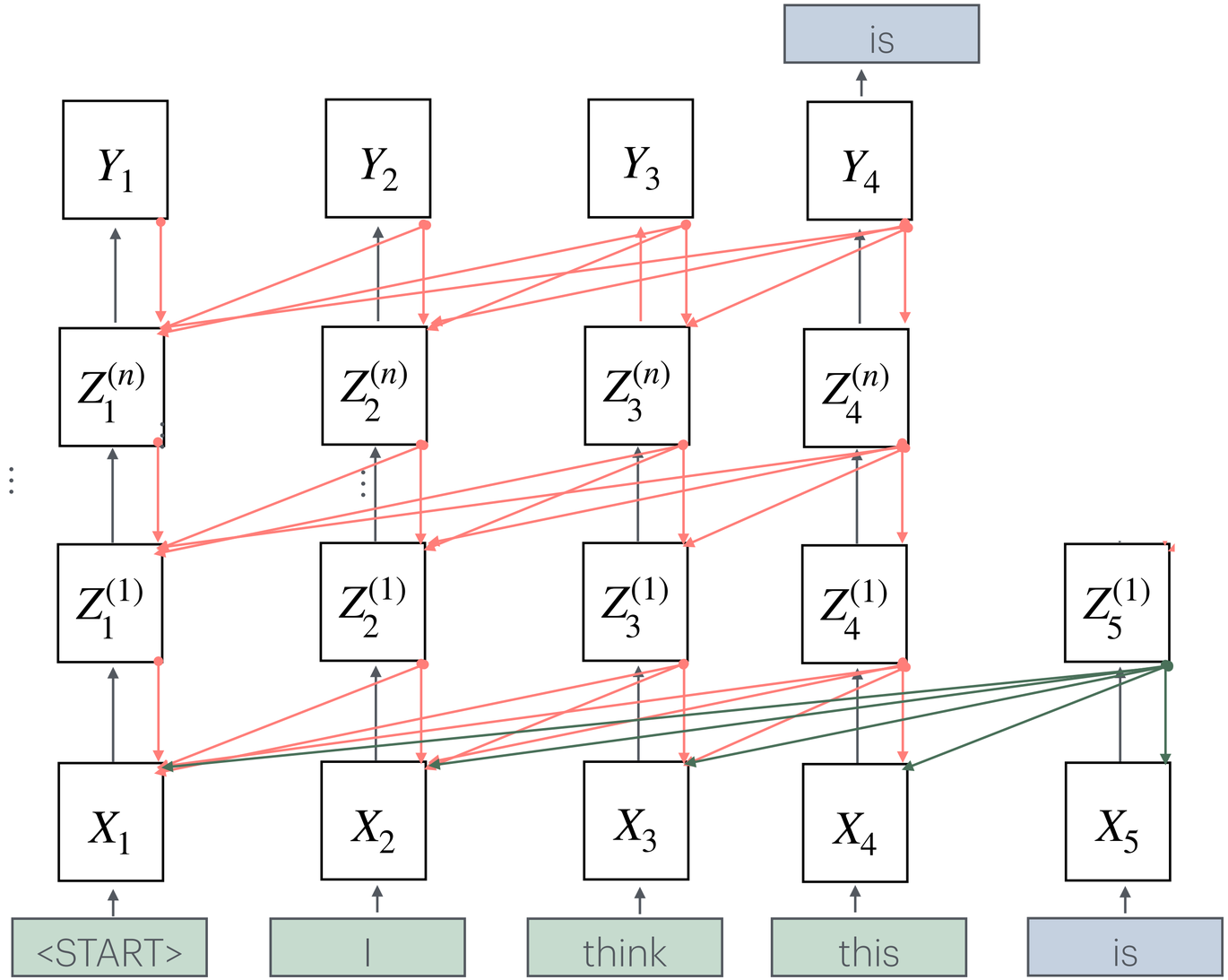
During decode phase, it is autoregressive next token generation

Decode phase

Generated word

Language model is built with transformer blocks

$Z_t^{(i+1)} = \text{Transformer\_block}(Z_t^{(i)})$

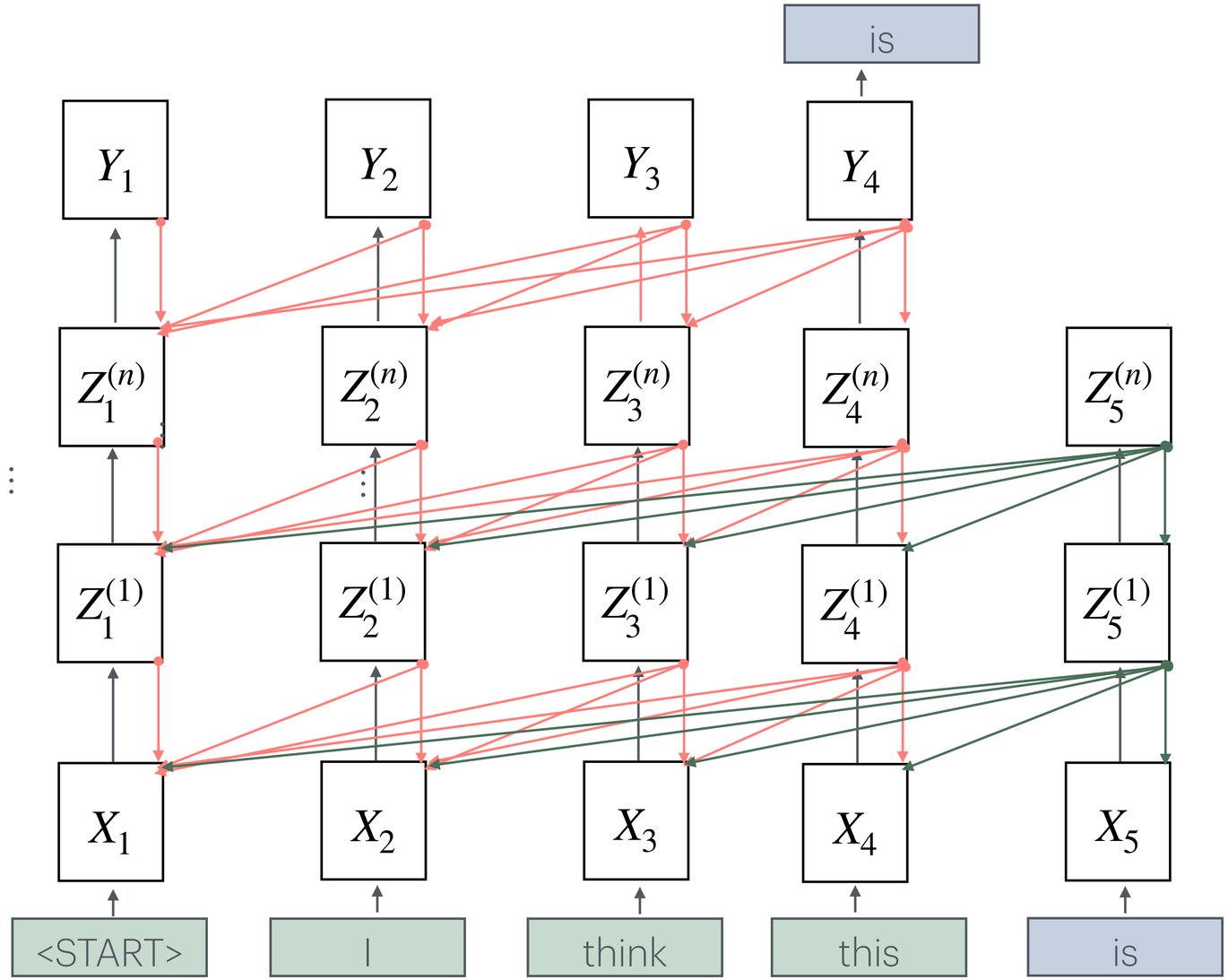


Decode phase

Generated word

Language model is built with transformer blocks

$Z_t^{(i+1)} = \text{Transformer\_block}(Z_t^{(i)})$

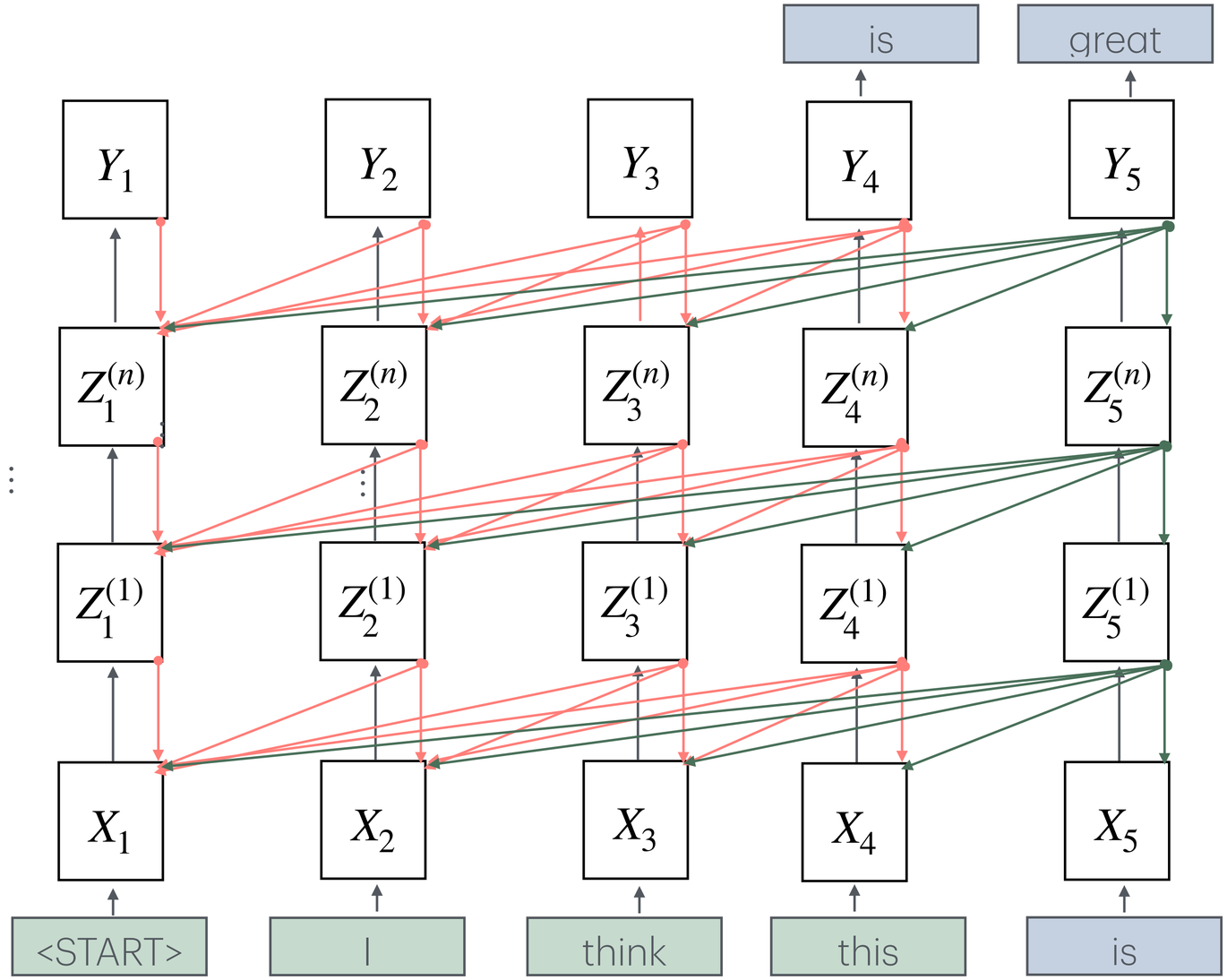


Decode phase

Generated word

Language model is built with transformer blocks

$Z_t^{(i+1)} = \text{Transformer\_block}(Z_t^{(i)})$

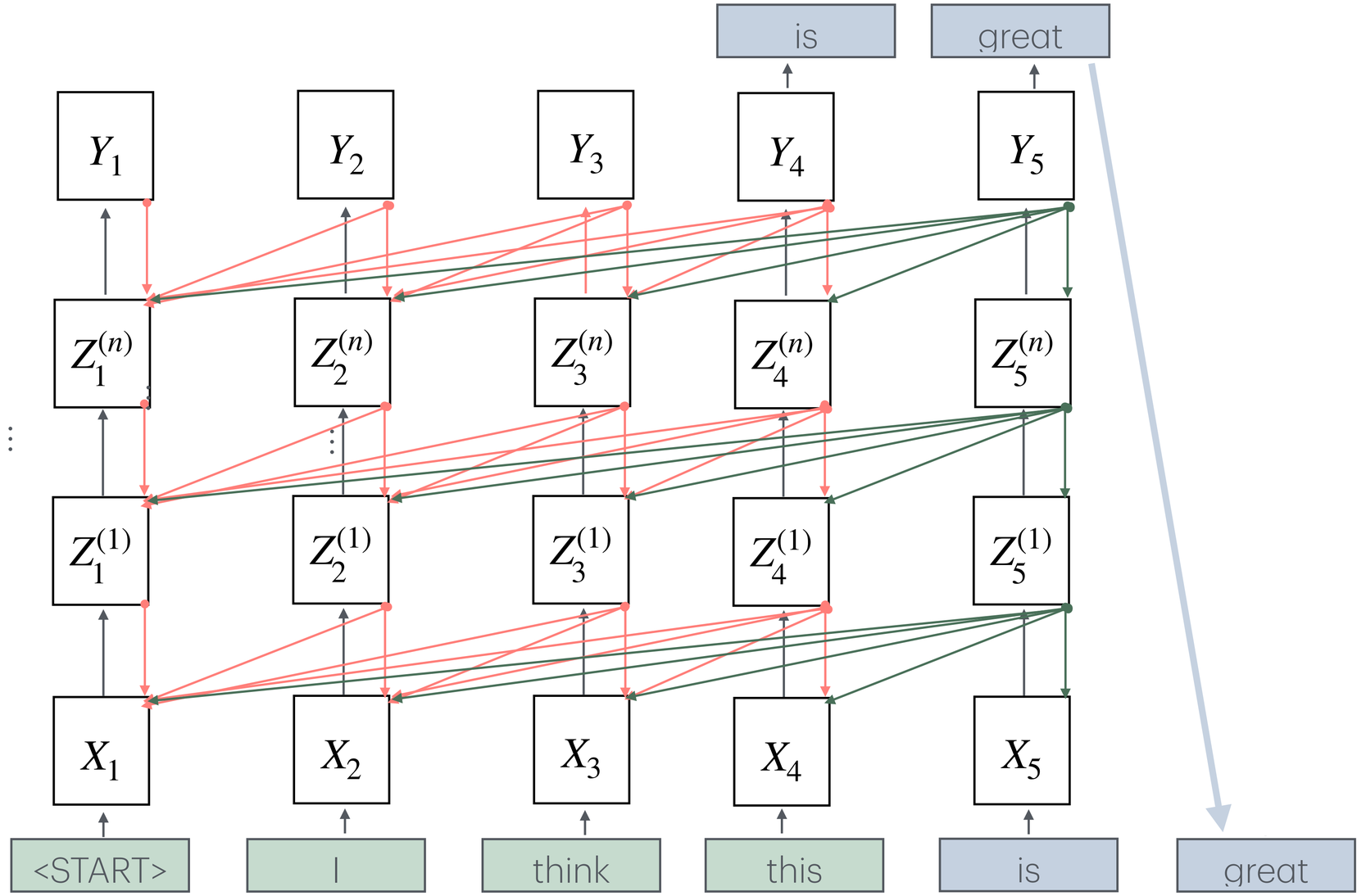


Decode phase

Generated word

Language model is built with transformer blocks

$Z_t^{(i+1)} = \text{Transformer\_block}(Z_t^{(i)})$

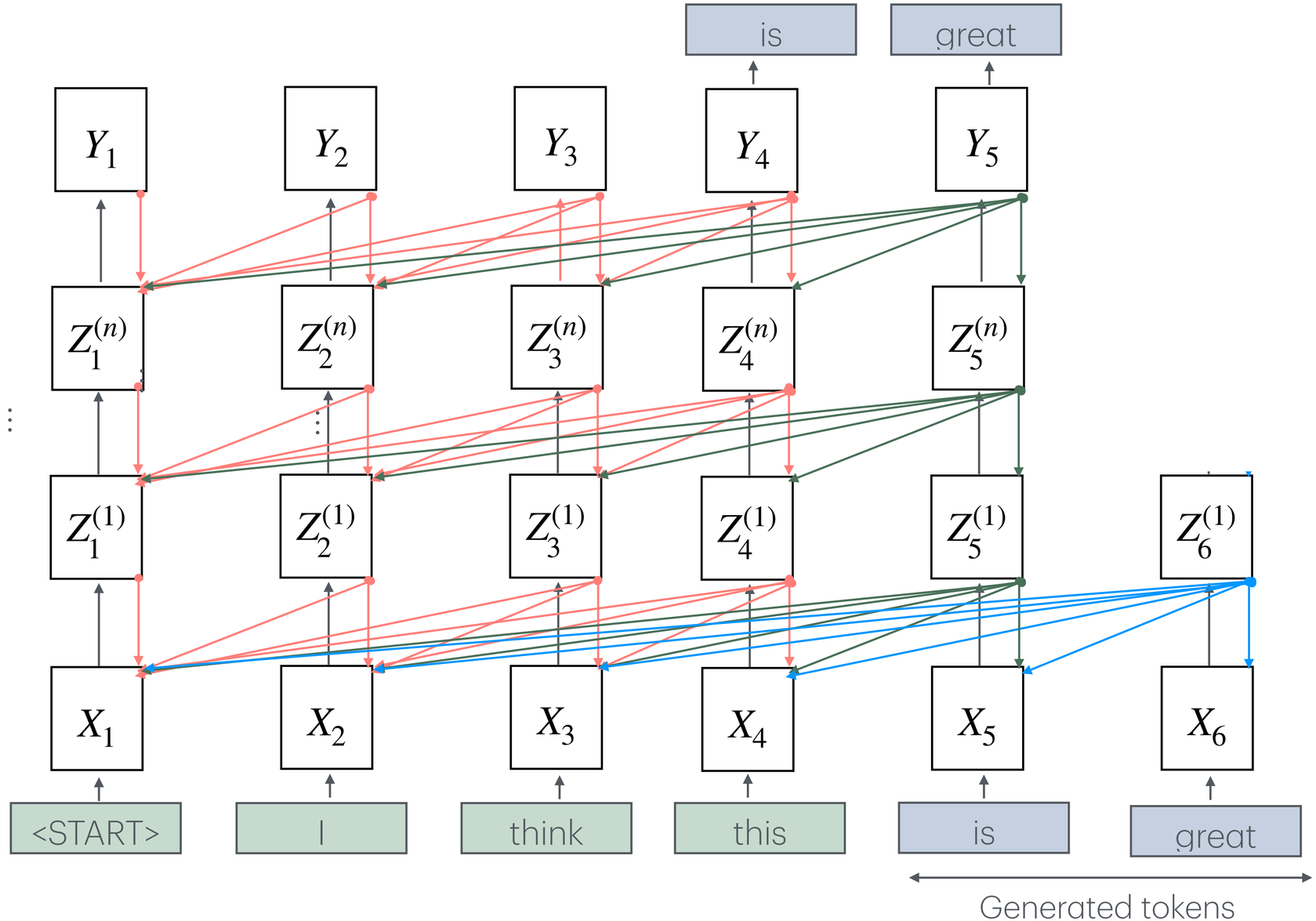


Decode phase

Generated word

Language model is built with transformer blocks

$Z_t^{(i+1)} = \text{Transformer\_block}(Z_t^{(i)})$

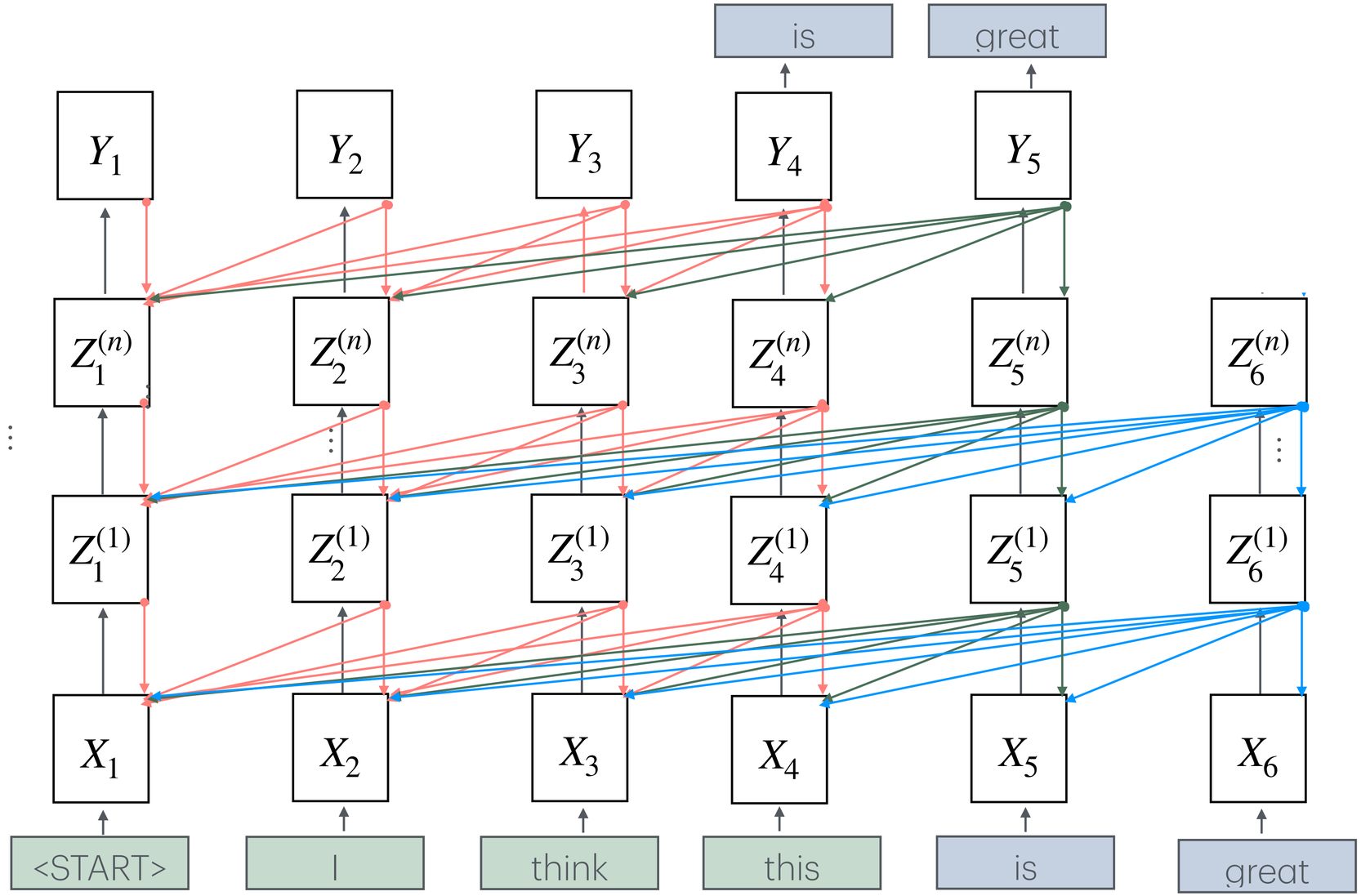


Decode phase

Generated word

Language model is built with transformer blocks

$Z_t^{(i+1)} = \text{Transformer\_block}(Z_t^{(i)})$

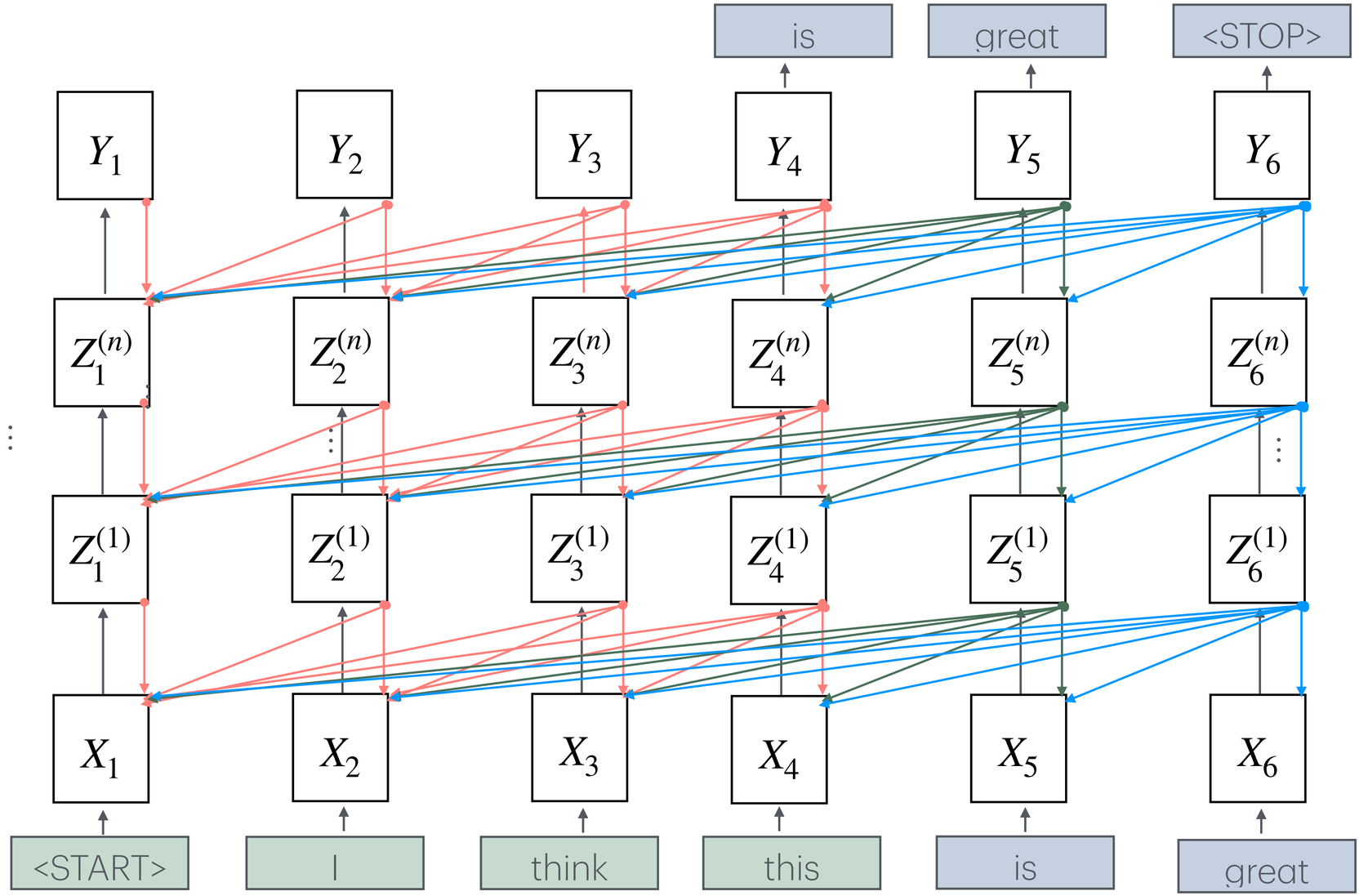


Decode phase

Generated word

Language model is built with transformer blocks

$Z_t^{(i+1)} = \text{Transformer\_block}(Z_t^{(i)})$

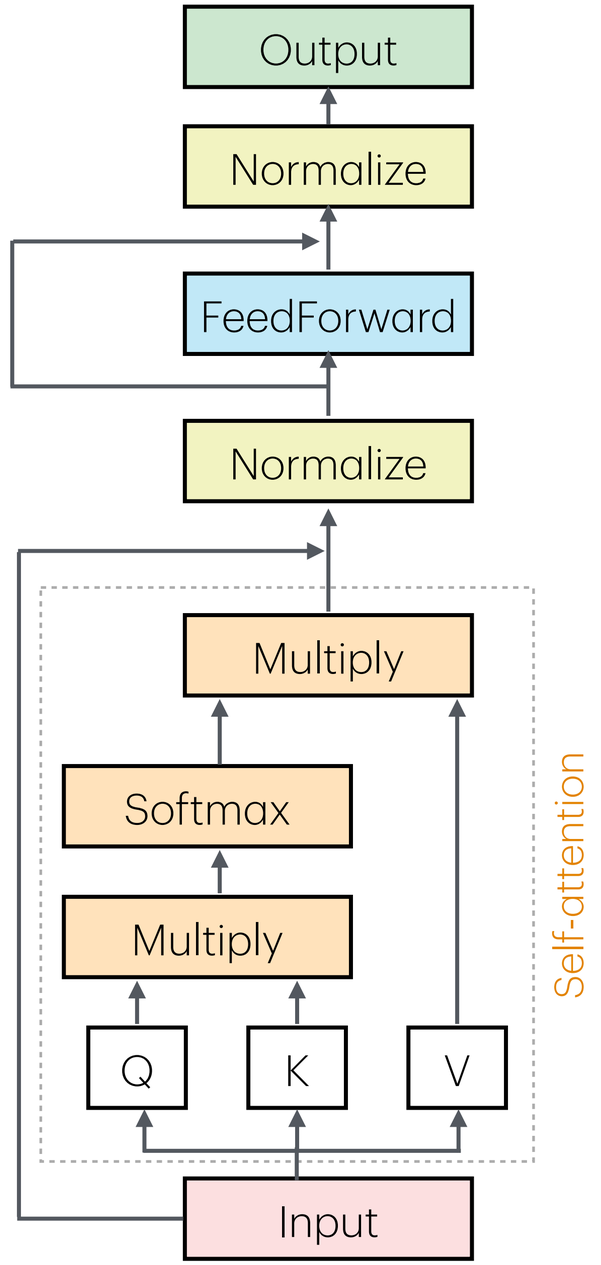


Review

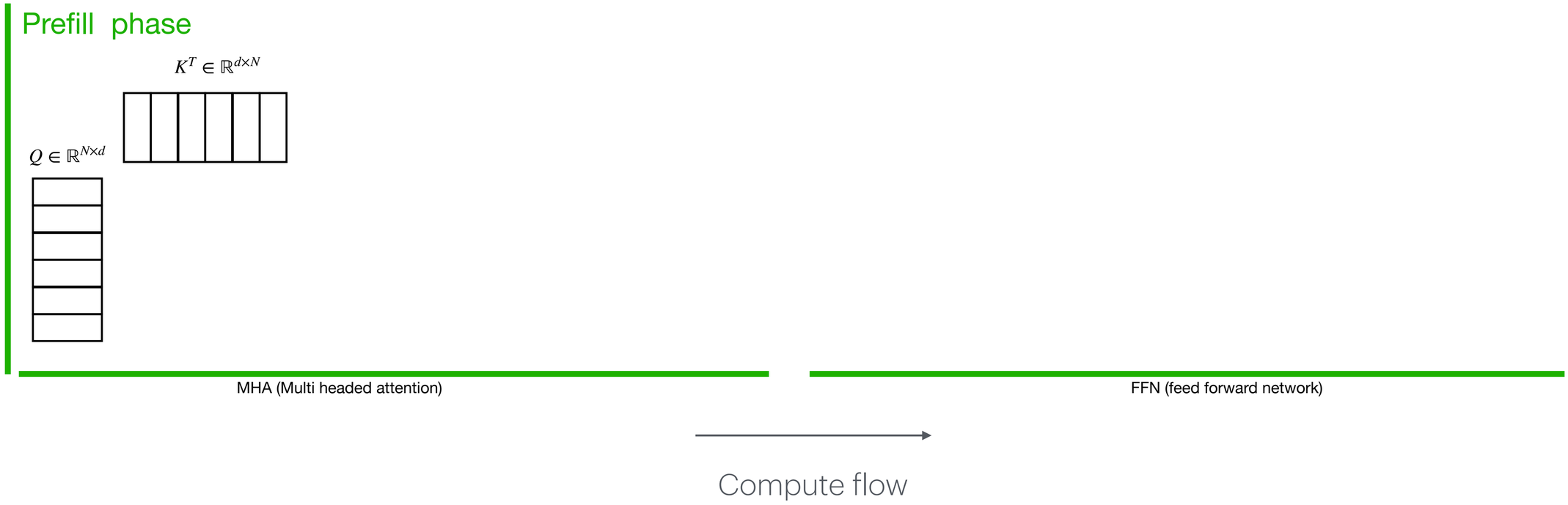
Transformer block

Transformer block consists of self-attention and feedforward network.

- Self-attention
- Feedforward network
- Layer norm and residual connection



Prefill phase, Attention block compute view



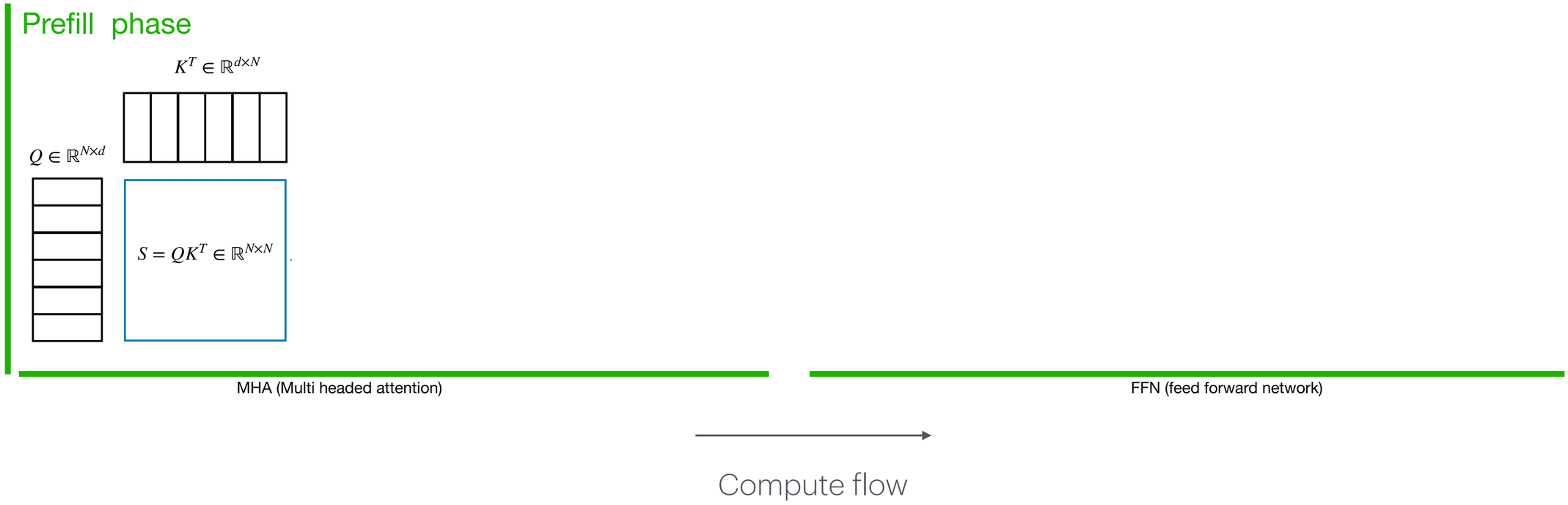
When input prompt is provided query, key, values are full sequence length N dim. N dim matrix & matrix multiplication, which is high arithmetic intensity.

Where N sequence length and d is hidden dimension

* We omit layer norm, residual addition, and causal masking for simplicity.

Ref: [3]

Prefill phase, Attention block compute view



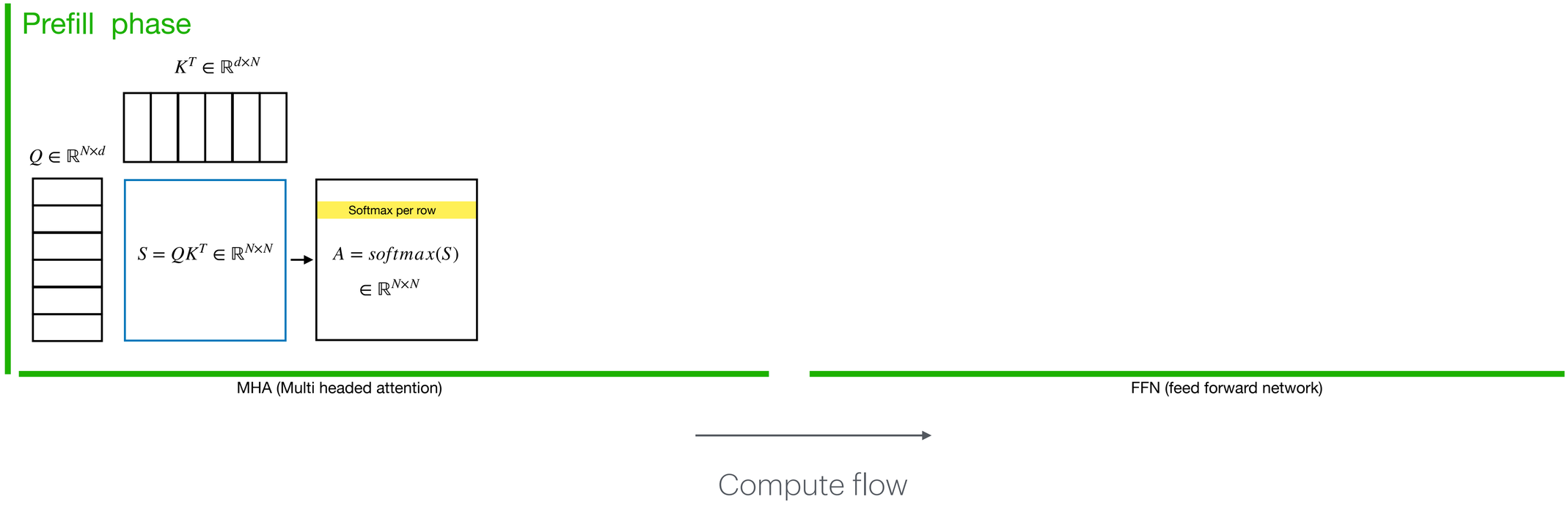
When input prompt is provided query, key, values are full sequence length N dim. N dim matrix & matrix multiplication, which is high arithmetic intensity.

Where N sequence length and d is hidden dimension

* We omit layer norm, residual addition, and causal masking for simplicity.

Ref: [3]

Prefill phase, Attention block compute view



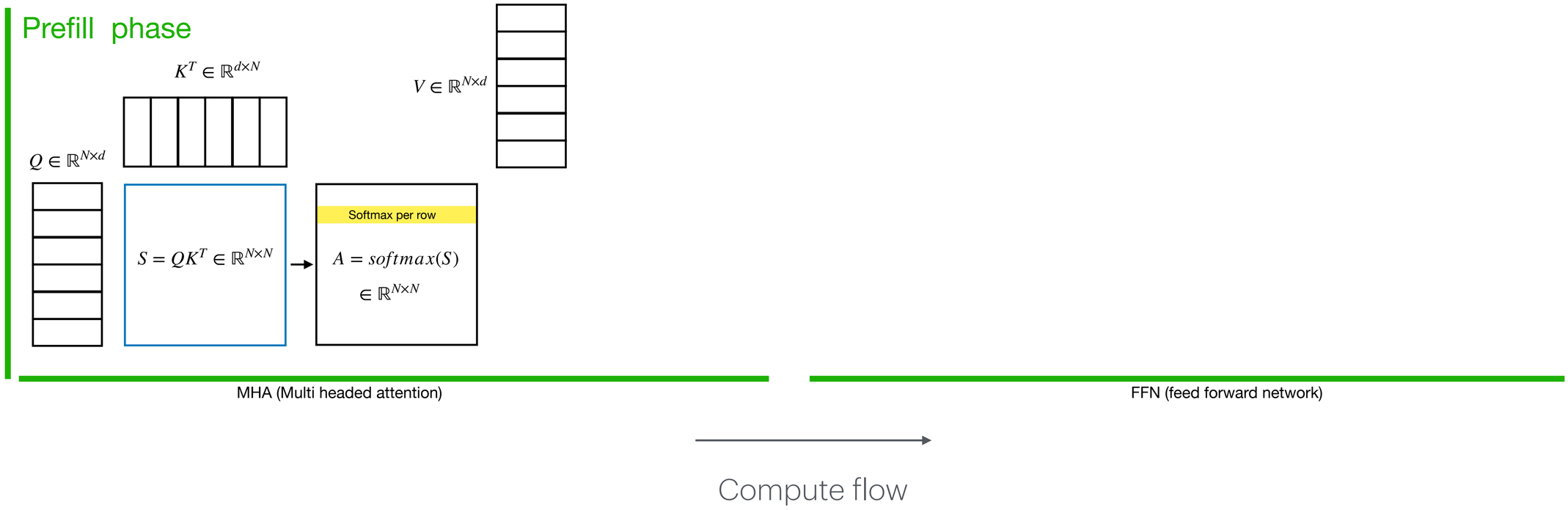
When input prompt is provided query, key, values are full sequence length N dim. N dim matrix & matrix multiplication, which is high arithmetic intensity.

Where N sequence length and d is hidden dimension

* We omit layer norm, residual addition, and causal masking for simplicity.

Ref: [3]

Prefill phase, Attention block compute view



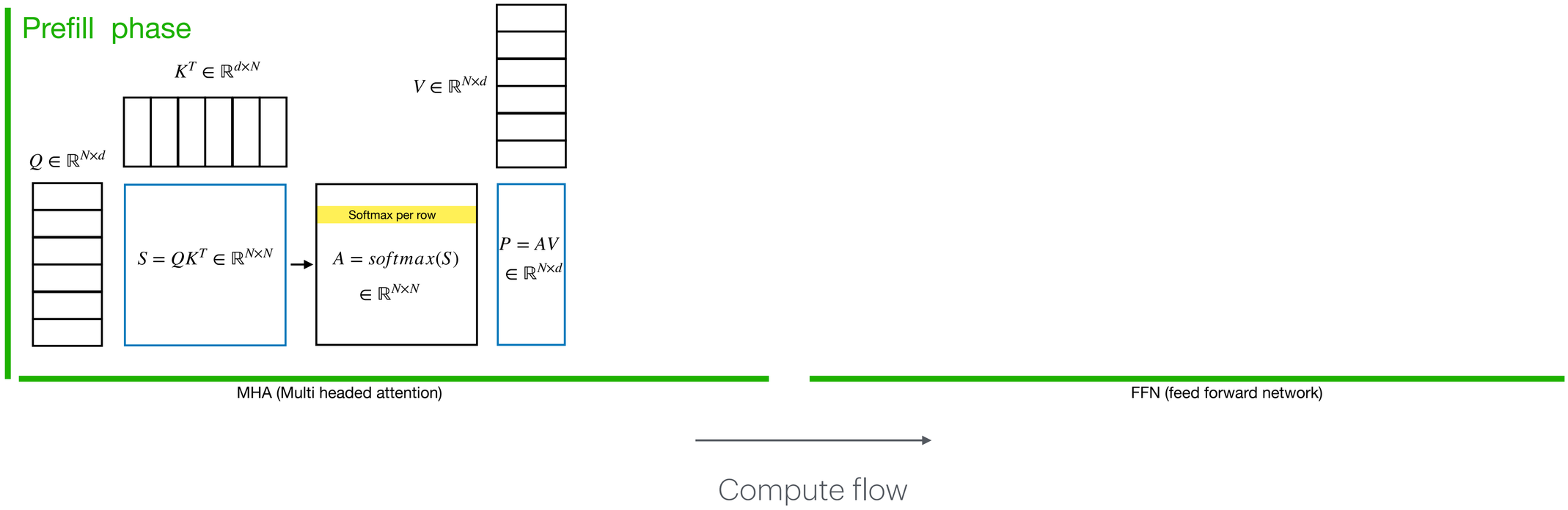
When input prompt is provided query, key, values are full sequence length N dim. N dim matrix & matrix multiplication, which is high arithmetic intensity.

Where N sequence length and d is hidden dimension

* We omit layer norm, residual addition, and causal masking for simplicity.

Ref: [3]

Prefill phase, Attention block compute view



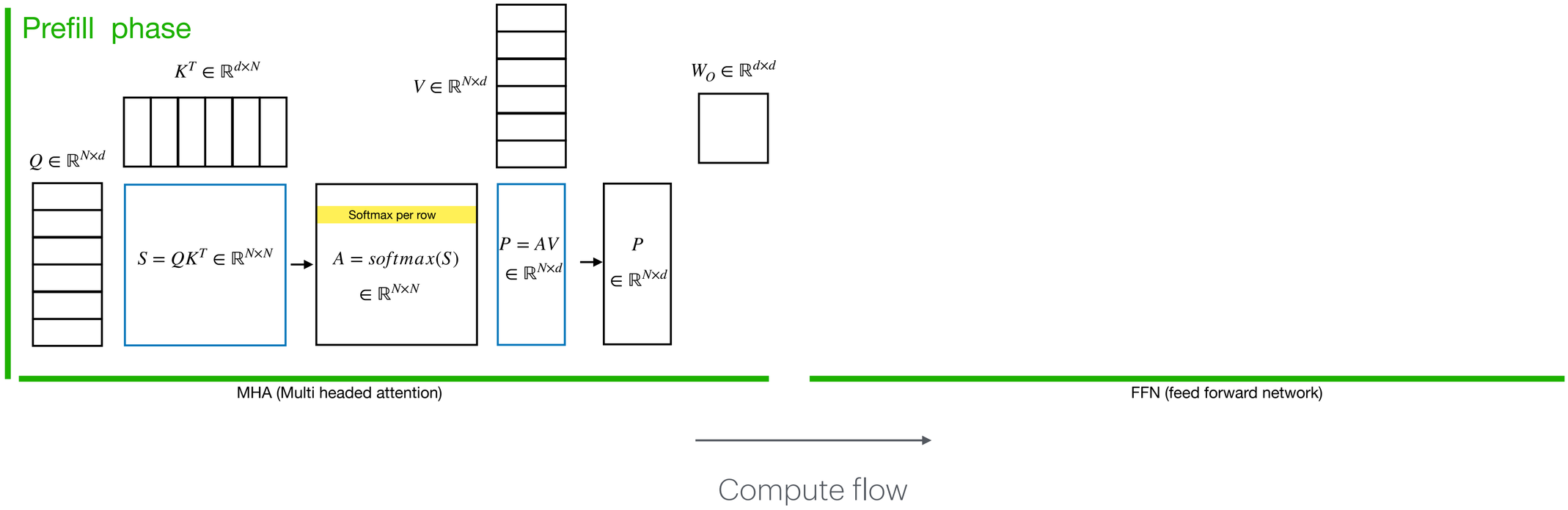
When input prompt is provided query, key, values are full sequence length N dim. N dim matrix & matrix multiplication, which is high arithmetic intensity.

Where N sequence length and d is hidden dimension

* We omit layer norm, residual addition, and causal masking for simplicity.

Ref: [3]

Prefill phase, Attention block compute view



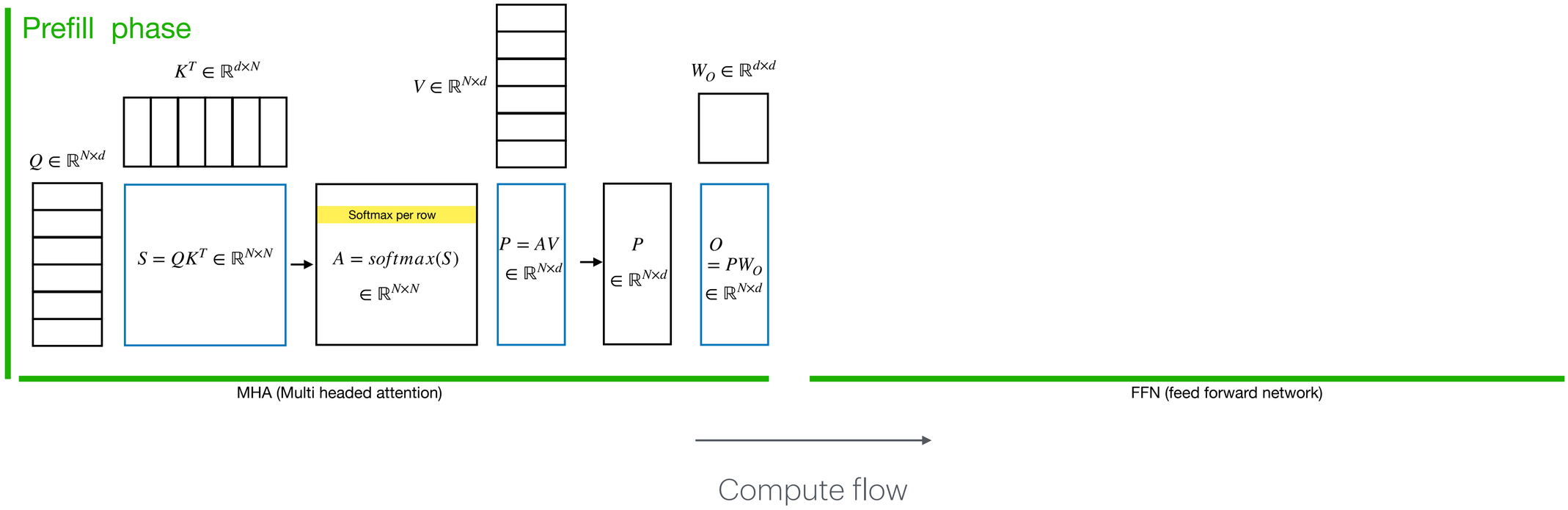
When input prompt is provided query, key, values are full sequence length N dim. N dim matrix & matrix multiplication, which is high arithmetic intensity.

Where N sequence length and d is hidden dimension

* We omit layer norm, residual addition, and causal masking for simplicity.

Ref: [3]

Prefill phase, Attention block compute view



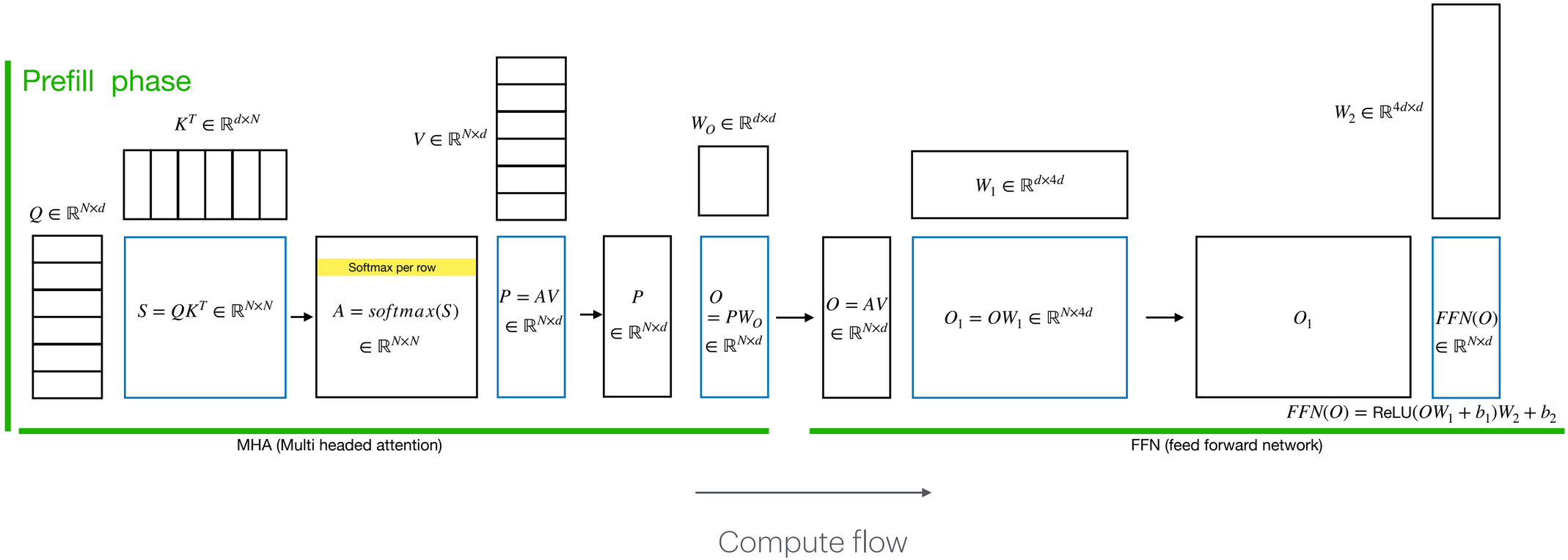
When input prompt is provided query, key, values are full sequence length N dim. N dim matrix & matrix multiplication, which is high arithmetic intensity.

Where N sequence length and d is hidden dimension

* We omit layer norm, residual addition, and causal masking for simplicity.

Ref: [3]

Prefill phase, Attention block compute view



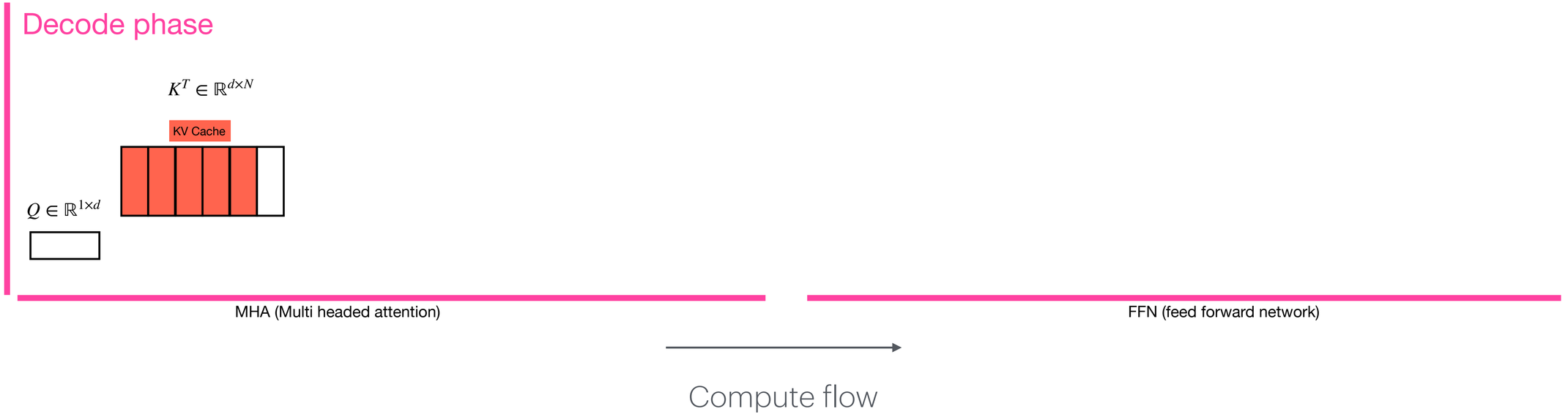
When input prompt is provided query, key, values are full sequence length N dim. N dim matrix & matrix multiplication, which is high arithmetic intensity.

Where N sequence length and d is hidden dimension

* We omit layer norm, residual addition, and causal masking for simplicity.

Ref: [3]

Decode phase, Attention block compute view



During decode phase, query is a vector, while key and values are full N sequence matrix. So it is vector and matrix multiplication, which is low arithmetic intensity.

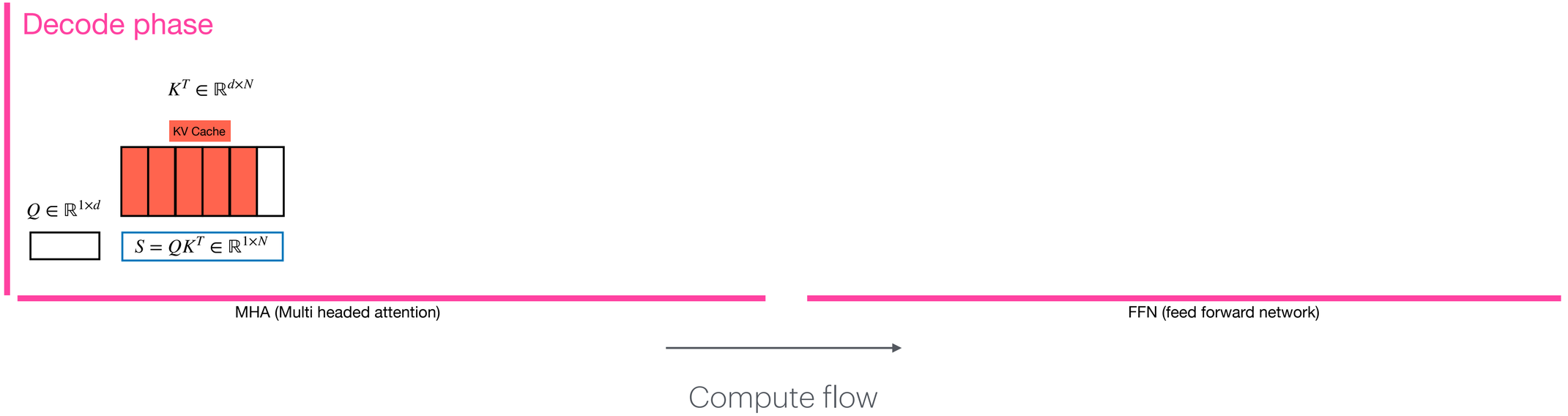
Key and values can be cached instead of recomputed in this phase

Where N sequence length and d is hidden dimension

* We omit layer norm, residual addition, and causal masking for simplicity.

Ref: [3]

Decode phase, Attention block compute view



During decode phase, query is a vector, while key and values are full N sequence matrix. So it is vector and matrix multiplication, which is low arithmetic intensity.

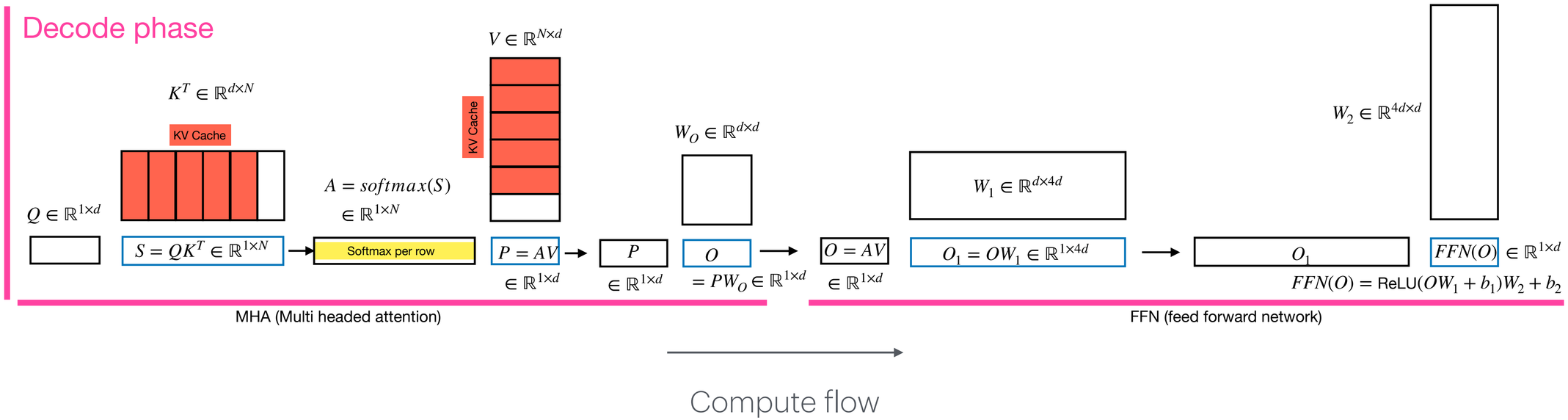
Key and values can be cached instead of recomputed in this phase

Where N sequence length and d is hidden dimension

* We omit layer norm, residual addition, and causal masking for simplicity.

Ref: [3]

Decode phase, Attention block compute view



During decode phase, query is a vector, while key and values are full N sequence matrix. So it is vector and matrix multiplication, which is low arithmetic intensity.

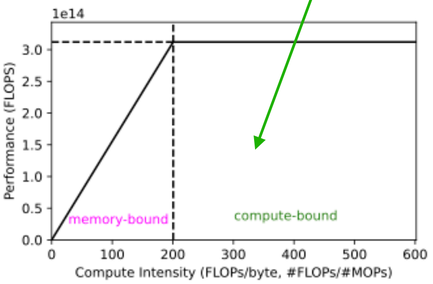
Key and values can be cached instead of recomputed in this phase

Where N sequence length and d is hidden dimension

* We omit layer norm, residual addition, and causal masking for simplicity.

Ref: [3]

Two phases of LLM inference: attention block view



Prefill phase

$Q \in \mathbb{R}^{N\times d}$ $K^T \in \mathbb{R}^{d\times N}$ $V \in \mathbb{R}^{N\times d}$ $W_O \in \mathbb{R}^{d\times d}$ $W_1 \in \mathbb{R}^{d\times 4d}$ $W_2 \in \mathbb{R}^{4d\times d}$

$S = QK^T \in \mathbb{R}^{N\times N}$ → Softmax per row → $A = softmax(S) \in \mathbb{R}^{N\times N}$ → $P = AV \in \mathbb{R}^{N\times d}$ → $P \in \mathbb{R}^{N\times d}$ → $O = PW_O \in \mathbb{R}^{N\times d}$ → $O = AV \in \mathbb{R}^{N\times d}$ → $O_1 = OW_1 \in \mathbb{R}^{N\times 4d}$ → O_1 → $FFN(O) \in \mathbb{R}^{N\times d}$

$FFN(O) = \text{ReLU}(OW_1 + b_1)W_2 + b_2$

MHA (Multi headed attention)

FFN (feed forward network)

Decode phase

$Q \in \mathbb{R}^{1\times d}$ $K^T \in \mathbb{R}^{d\times N}$ KV Cache $V \in \mathbb{R}^{N\times d}$ KV Cache $W_O \in \mathbb{R}^{d\times d}$ $W_1 \in \mathbb{R}^{d\times 4d}$ $W_2 \in \mathbb{R}^{4d\times d}$

$S = QK^T \in \mathbb{R}^{1\times N}$ → Softmax per row → $A = softmax(S) \in \mathbb{R}^{1\times N}$ → $P = AV \in \mathbb{R}^{1\times d}$ → $P \in \mathbb{R}^{1\times d}$ → $O = PW_O \in \mathbb{R}^{1\times d}$ → $O = AV \in \mathbb{R}^{1\times d}$ → $O_1 = OW_1 \in \mathbb{R}^{1\times 4d}$ → O_1 → $FFN(O) \in \mathbb{R}^{1\times d}$

$FFN(O) = \text{ReLU}(OW_1 + b_1)W_2 + b_2$

MHA (Multi headed attention)

FFN (feed forward network)

** This roofline model is based on A100 40G HBM

Where N sequence length and d is hidden dimension

* We omit layer norm, residual addition, and causal masking for simplicity.

Ref: [3]

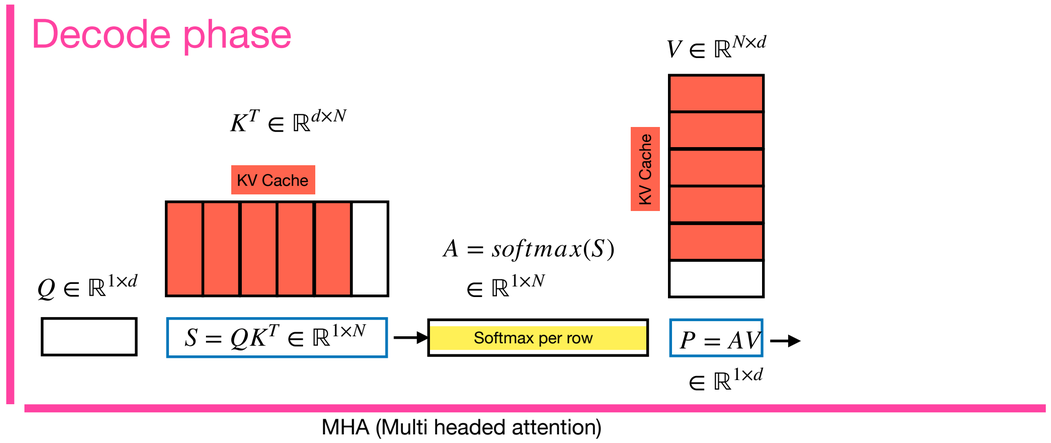
KV (key-value) cache

Keys and values for previous steps in all layers are not changing. We can reuse them for generating a new token.
In each step, new key and value are appended to the KV cache.

Memory requirements for KV cache with FP16

batch_size * seqlen * (d_model/n_heads) * n_layers * 2 (K and V) * 2 (bytes per Float16) * n_kv_heads

Grows linearly with sequence length and batch size.



Ref: link

Summary: Key technical concepts

1. LLM inference consist of two phases

    1. Prefill (prompt) phase:
       All input tokens (prompt) are processed at once, ‘prefill’ tokens for the following generation step

    2. Decoding (token generation) phase:
       Autoregressive ‘decoding’ phase, a single next token is generated using all previous tokens one at a time iteratively

2. KV-cache: Keys and values for previous steps in all layers are not changing. We can reuse them for generating a new token.
   In each step, new key and value are appended to the KV cache.

How to address challenges of LLM inference

Model compression approach

- Keep the model structure and parameters. Compress the model using quantization or pruning

Algorithm optimization

- Use new model structures or added components to improve token generation

System optimization

- Keep the model structure and parameters. Improve scheduling for requests and memory management

Ref: [3]

Model compression

Model with large parameter (over-parameterized) is easier to optimize

Larger models require more computer, more memory, and more energy to operate

Methods to reduce model parameters and/or numerical representations to reduce the amount of memory required for operations:

- Pruning (sparsity): pruning the subset of parameters after training. Pruning makes the model sparse
- Quantization: using quantized parameters by using different numerical representation
- Knowledge distillation: distill knowledge from a larger teacher model to a smaller student model. Train a smaller size model that mimics the quality of the larger model

Ref: [2]

Pruning

Pruning

Reduce the extent of a model by removing superfluous of unwanted parts.

Goal: In order to reduce the cost of inference and/or training.

Like playing Jenga, remove blocks that are not sensitive.

Ref. [5], Image from link

Pruning

Formal definition of pruning

$$\arg\min_{W_p} L(x; W_p)$$

subject to $\|W_p\|_0 < N$

L is loss function,

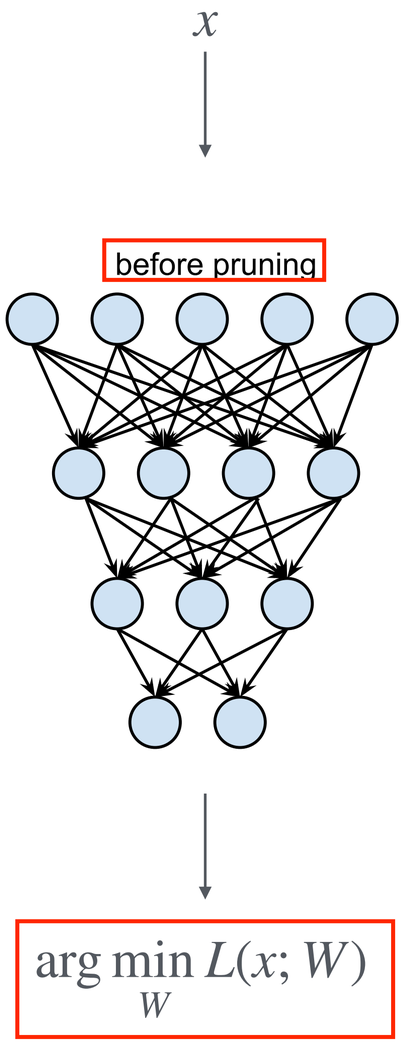
x is input,

W is original weight,

W_p is pruned weight.

$\|W_p\|_0$ is number of non-zeros in $W_{p'}$

N is target number of non-zero parameters.



Ref. [1, 5] and Image source: link

Pruning

Formal definition of pruning

$$\arg\min_{W_p} L(x; W_p)$$

subject to $\|W_p\|_0 < N$

L is loss function,

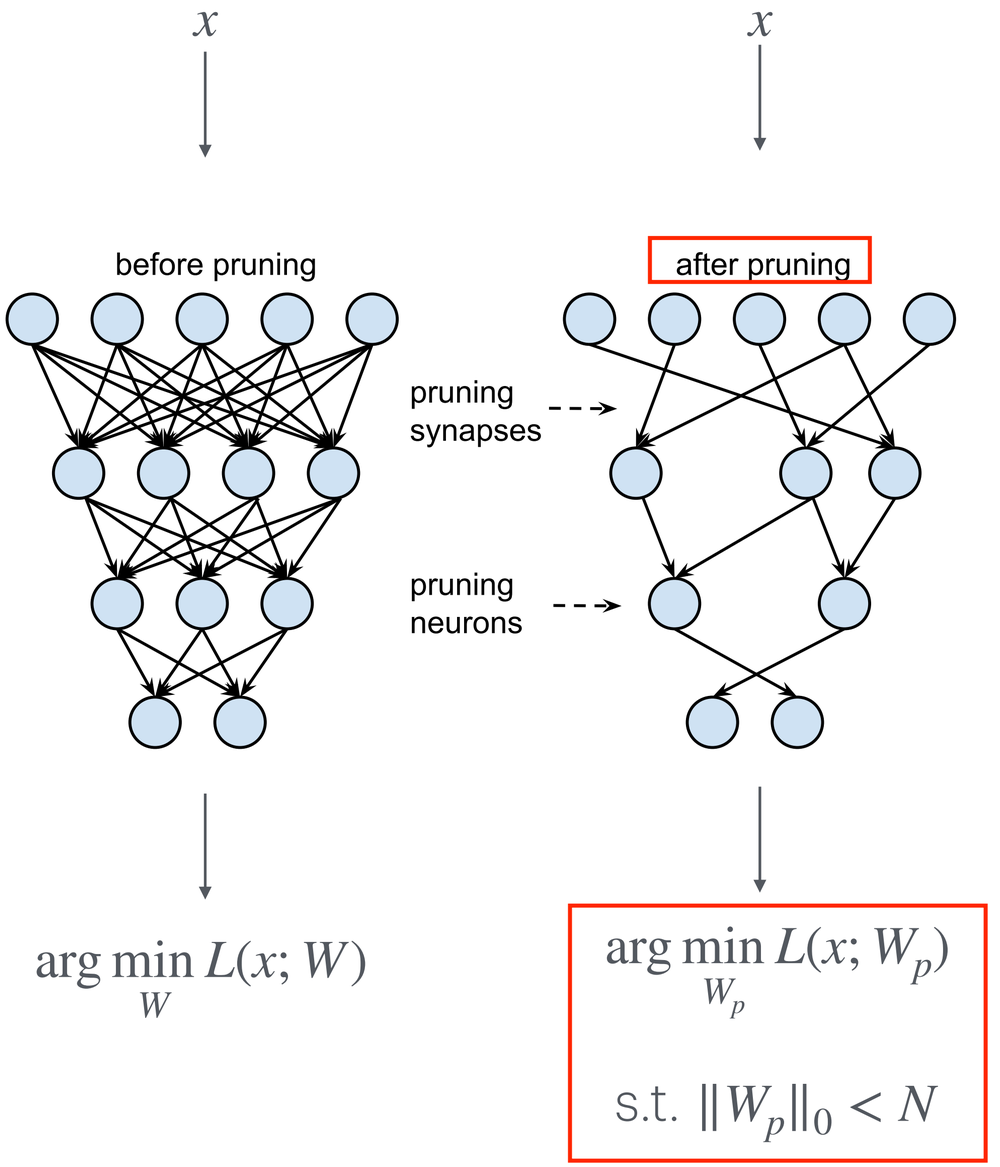
x is input,

W is original weight,

W_p is pruned weight.

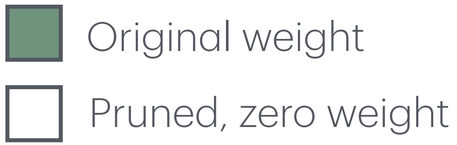
$\|W_p\|_0$ is number of non-zeros in W_p,

N is target number of non-zero parameters.



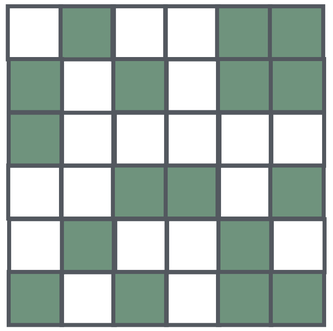
Ref. [1, 5] and Image source: link

Structured vs unstructured pruning



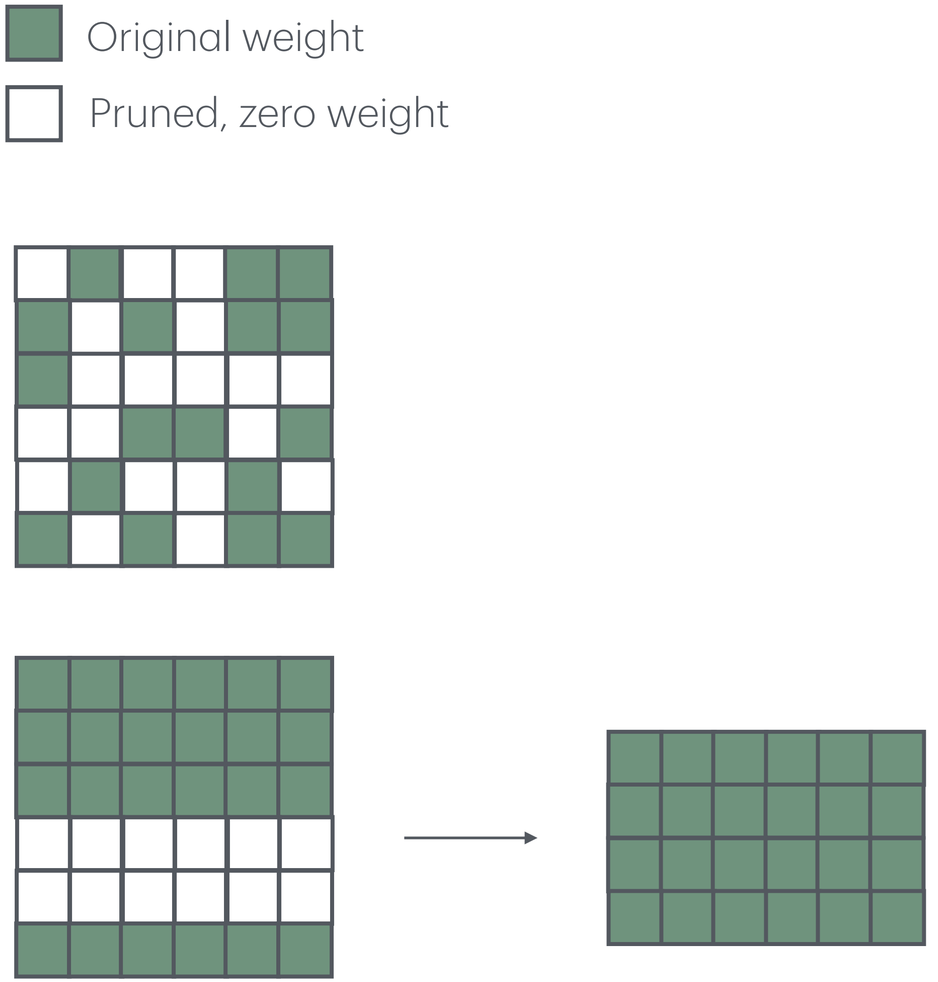
Unstructured pruning

- Flexible pruning selection, irregular patterns
- Hard to accelerate in HW, since you still need to read/write the full weights



Ref. [1, 5]

Structured vs unstructured pruning



Unstructured pruning

- Flexible pruning selection, irregular patterns
- Hard to accelerate in HW, since you still need to read/write the full weights

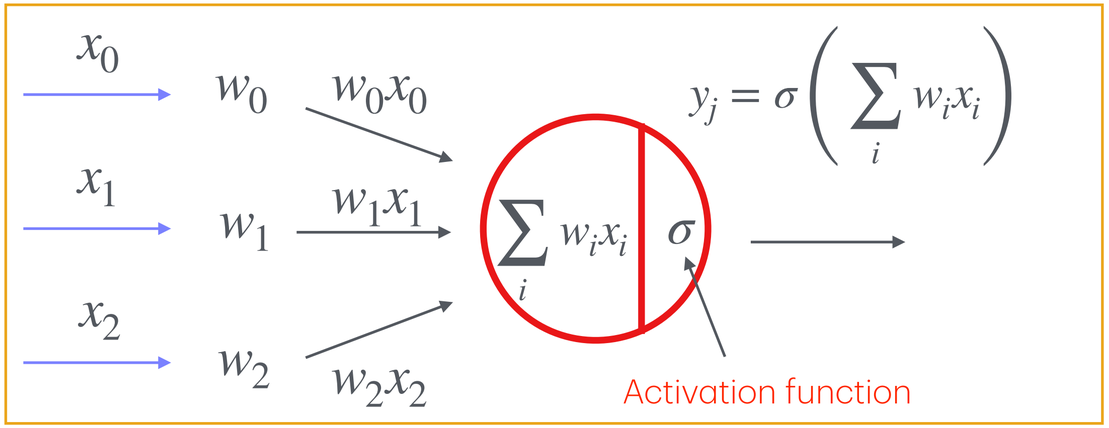
Structured pruning

- Less flexible pruning selection, regular, fixed patterns,
- Easier to accelerate in HW, speed-up in end-to-end run

Ref. [1, 5]

What parameters to prune?

Remove less important parameters to reduce the impact on the performance of the model



$\sigma = \mathrm{ReLU}(x)$,
$w_0 = 10, w_1 = -8, w_2 = 0.1$
$y = \mathrm{ReLU}(10x_0 - 8x_1 + 0.1x_2)$

If one weight is pruned which one will be removed with least impact to the output

Ref. [1], Image from link

What parameters to prune

Pruning criteria, use magnitude as a proxy of importance.

- Magnitude based pruning is used widely
- Element-wise (fine-grained) pruning using absolute value. Importance = $|W|$
- Row-wise (coarse-grained) pruning using L1-norm. Importance $= \sum_{i \in S} |W_i|$, S is structure of W
- Row-wise (coarse-grained) pruning using Lp-norm. Importance $= \sum_{i \in S} (|W_i|^p)^{1/p}$, S is structure of W

Ref. [1]

Hardware support

Hardware supported pattern based pruning. TensorCore with sparsity

- 2:4 sparsity: for each 1x4 weights, set 2 weights with smallest weights to 0

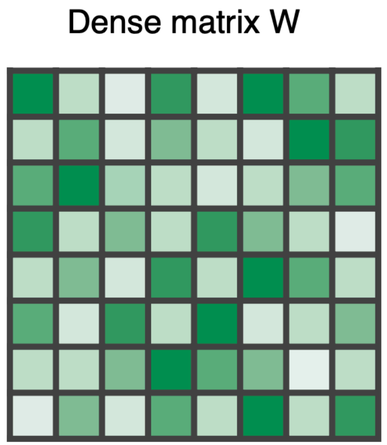


Image from link

Hardware support

Hardware supported pattern based pruning. TensorCore with sparsity

- 2:4 sparsity: for each 1x4 weights, set 2 weights with smallest weights to 0

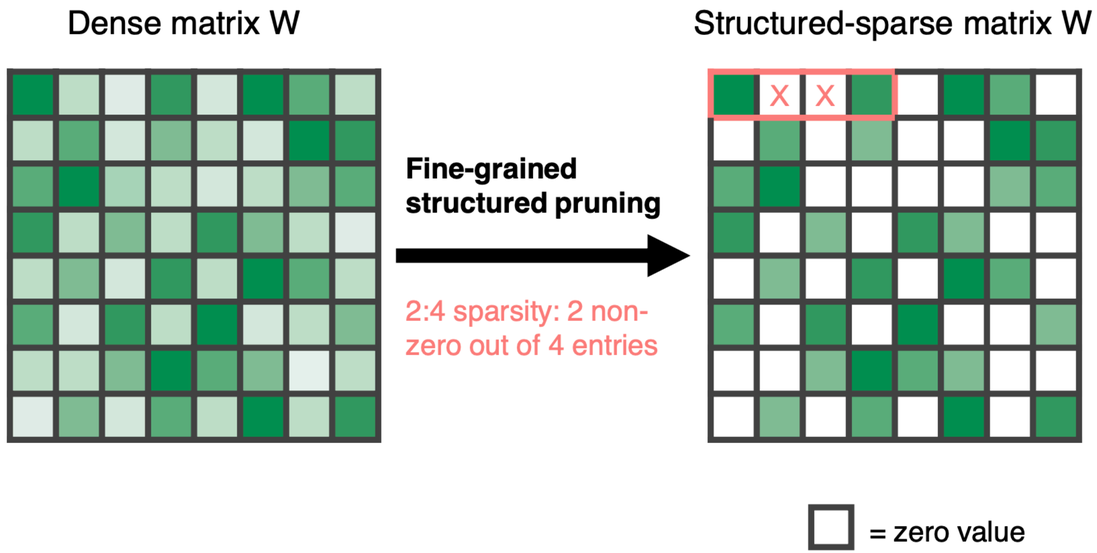


Image from link

Hardware support

Hardware supported pattern based pruning. TensorCore with sparsity

- 2:4 sparsity: for each 1x4 weights, set 2 weights with smallest weights to 0

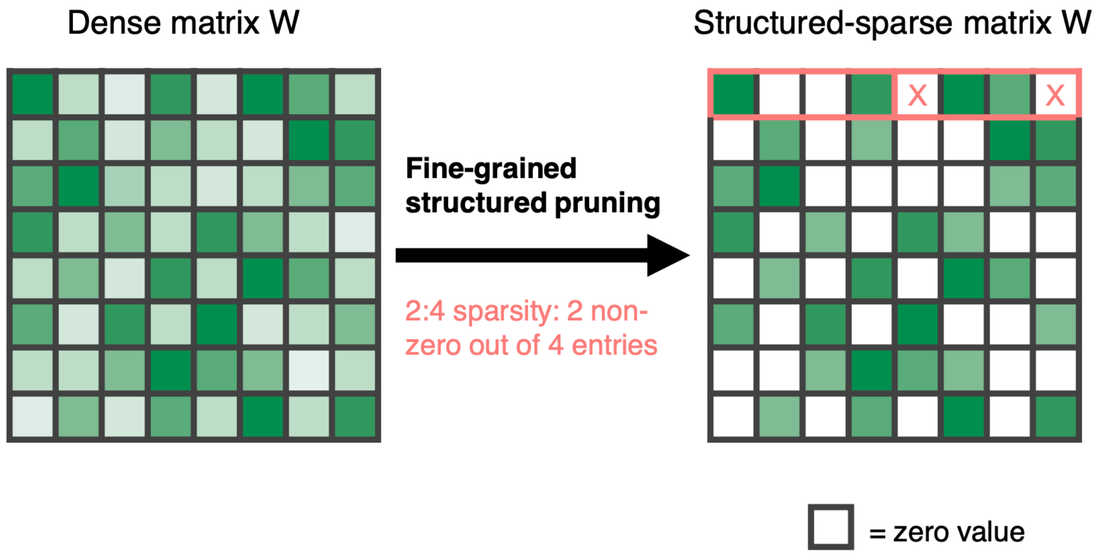


Image from link

Hardware support

Hardware supported pattern based pruning. TensorCore with sparsity

- 2:4 sparsity: for each 1x4 weights, set 2 weights with smallest weights to 0

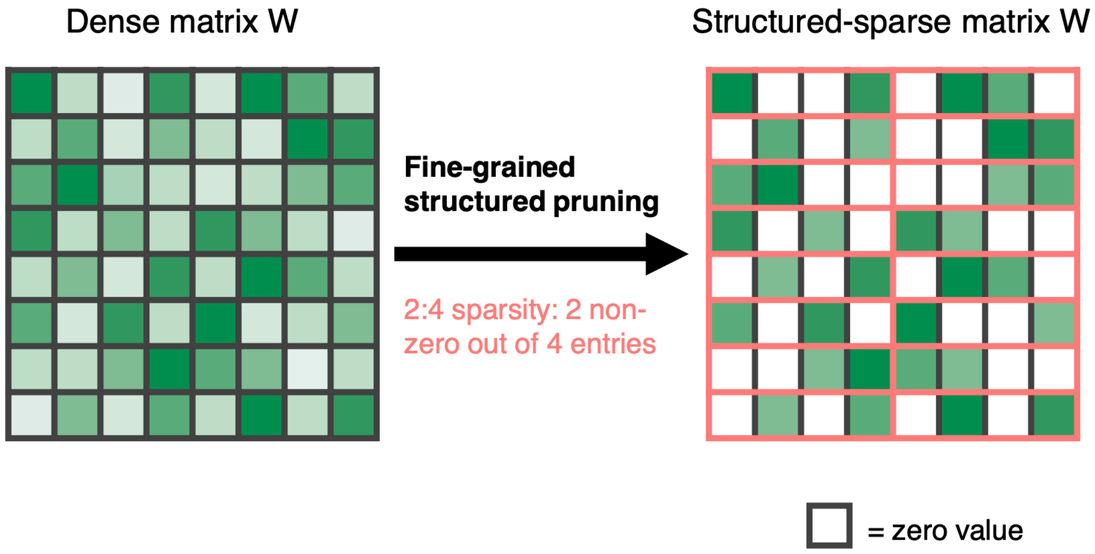
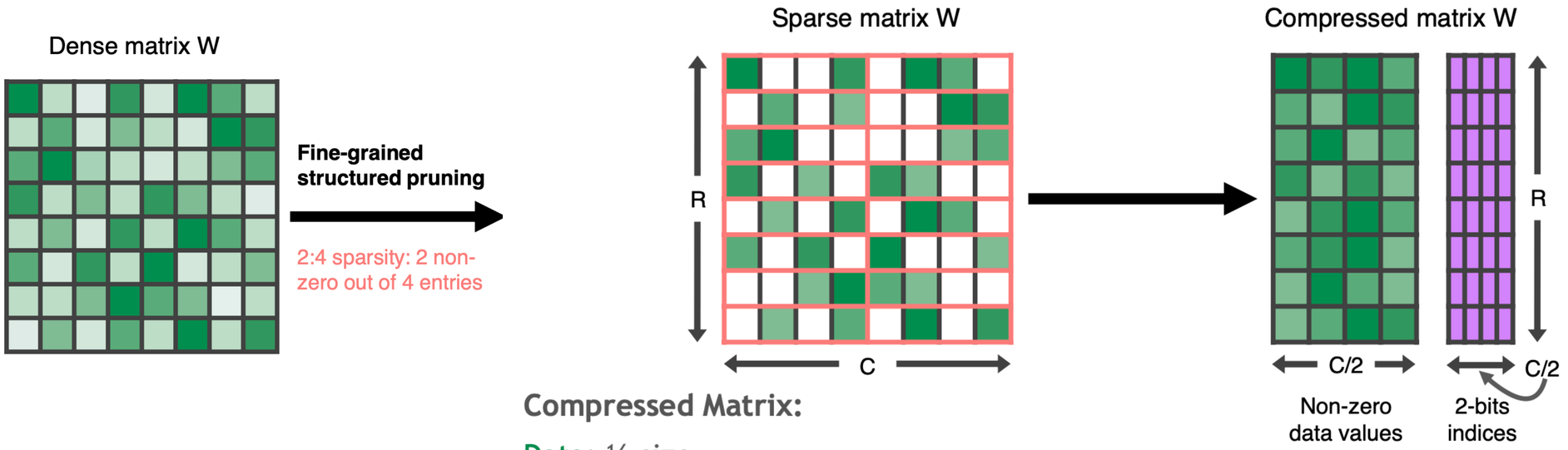


Image from link

Hardware support

Hardware supported pattern based pruning. TensorCore with sparsity

- 2:4 sparsity: for each 1x4 weights, set 2 weights with smallest weights to 0



Compressed Matrix:

Data: ½ size

Metadata: 2b per non-zero element

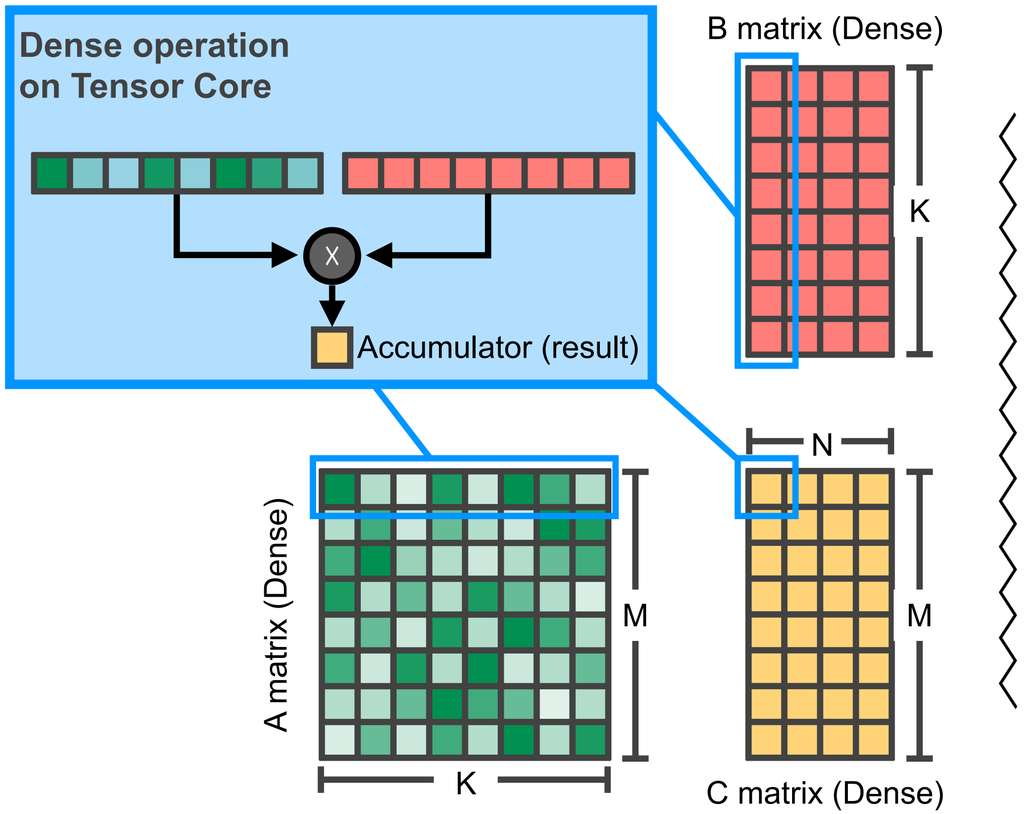
16b data => 12.5% overhead

8b data => 25% overhead

Image from link

Hardware support

Illustration of 2:4 sparse matmul

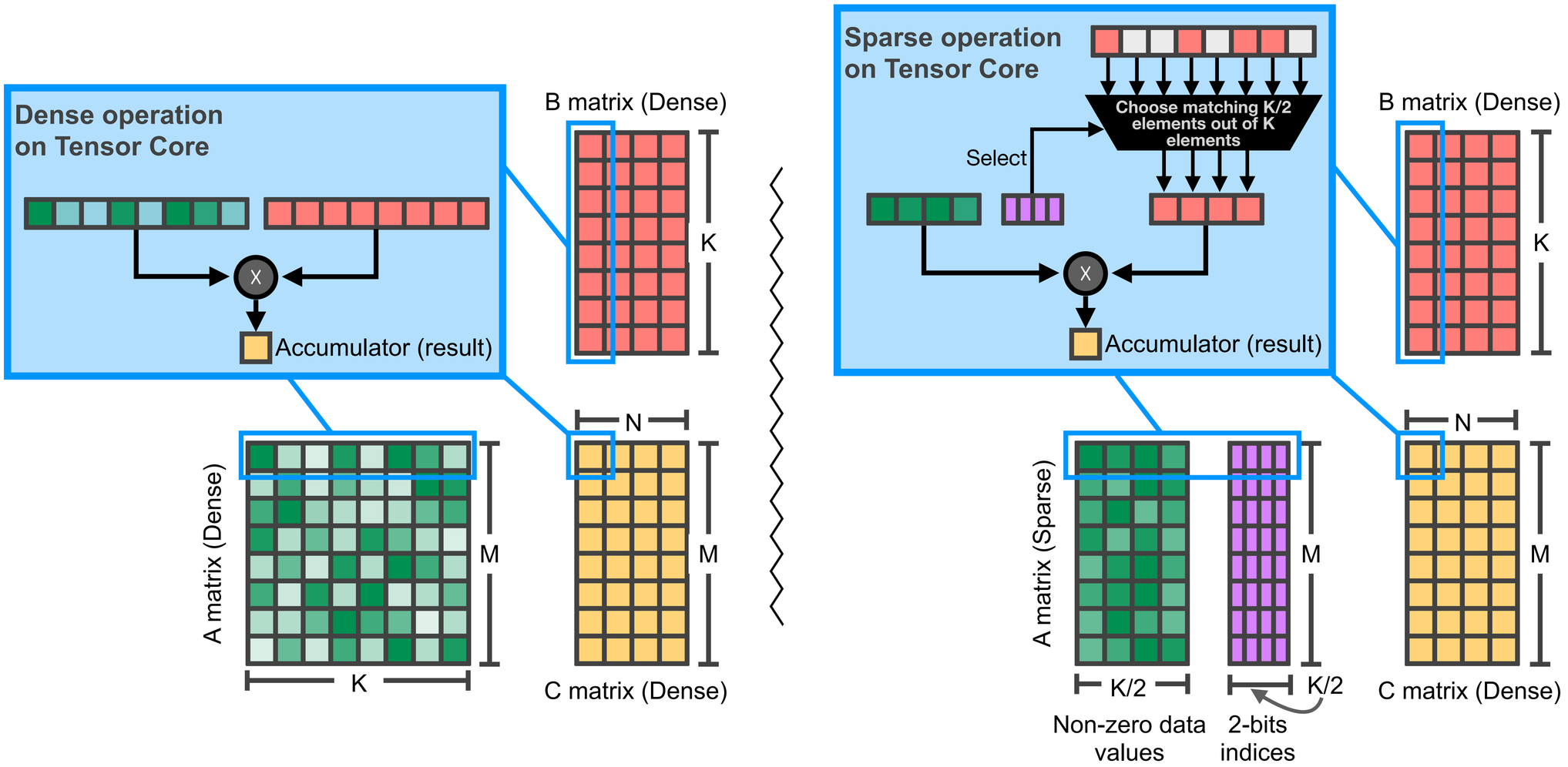


Dense M×N×K GEMM

Image from link

Hardware support

Illustration of 2:4 sparse matmul



Dense M×N×K GEMM

Sparse M×N×K GEMM

Image from link

Hardware support

Hardware supported pattern based pruning. TensorCore with sparsity

- 2:4 sparsity: for each 1x4 weights, set 2 weights with smallest weights to 0
- N:M sparsity: in each contiguous M elements, N of them are pruned
- Supported by Ampere architecture, and delivers ~x2 speed up with low performance impact

Ref. [1]

Pruning playbook

Pruning is used in model inference to reduce memory access and computes.

- Train a (dense) model
- Prune the model
  - based on importance criteria, 2:4 sparsity if hw supports
- Iterative pruning and fine-tuning

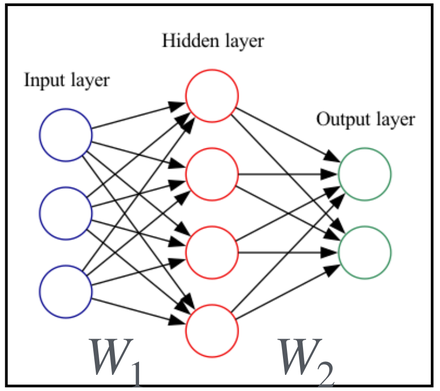
Pruning with hw support can also help model training.

- This blog shows how to use 2:4 sparsity to achieve x1.3 speed up for training vision transformer

Quantization

Review

Parameters, memory, and computation

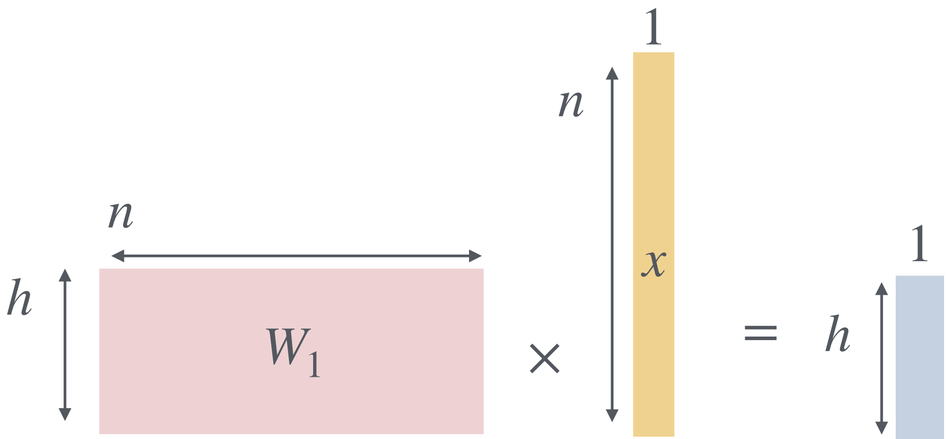


2 layer neural network for single input

$$y = W_2\,\sigma(W_1 x)$$

$$W_1 \in \mathbb{R}^{h \times n}, W_2 \in \mathbb{R}^{k \times h}$$

$$x \in \mathbb{R}^{n}, y \in \mathbb{R}^{k}$$



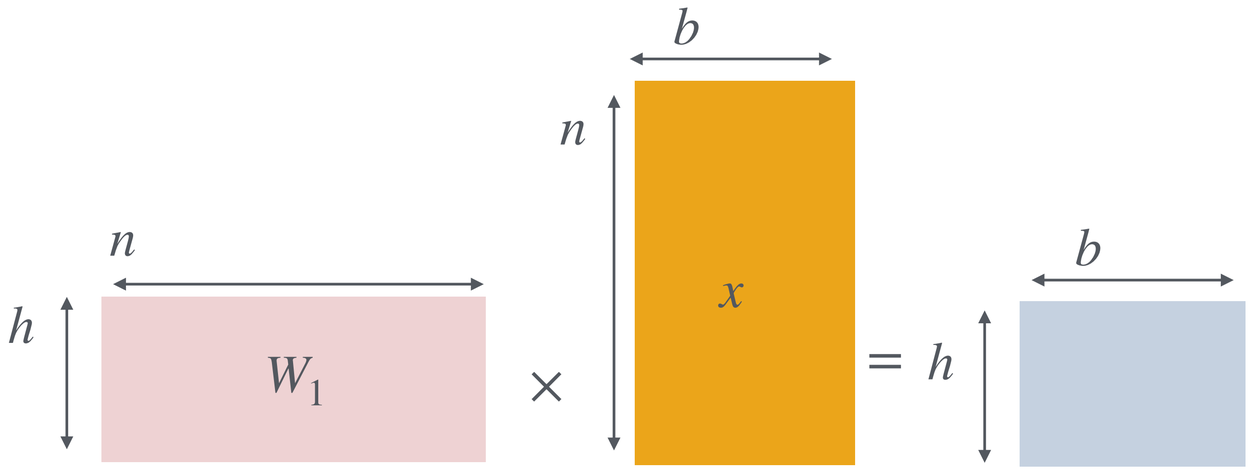
Matrix multiplication

2 layer neural network for mini-*batch* *input (size* b*)*

$$y = W_2\,\sigma(W_1 x)$$

$$W_1 \in \mathbb{R}^{h \times n}, W_2 \in \mathbb{R}^{k \times h}$$

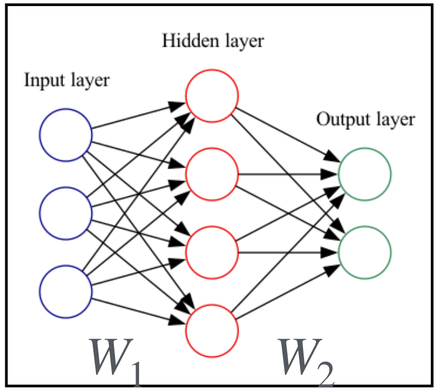
$$x \in \mathbb{R}^{n \times b}, y \in \mathbb{R}^{k \times b}$$



Matrix multiplication

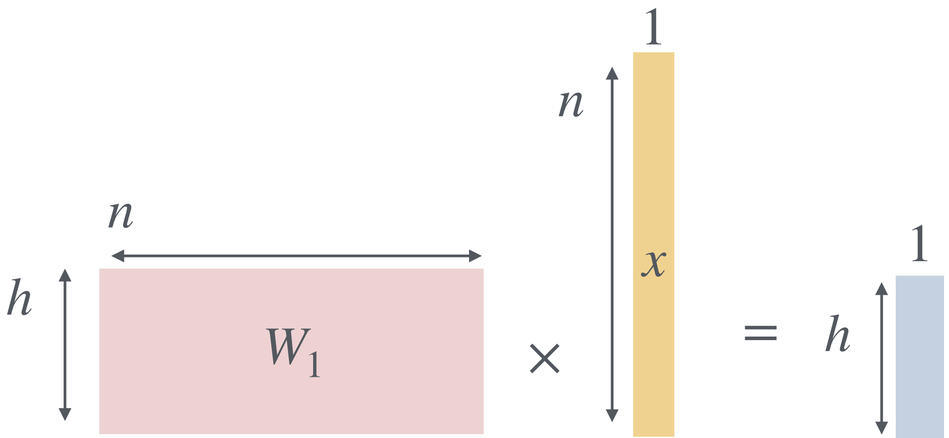
Review

Parameters, memory, and computation



2 layer neural network for single input

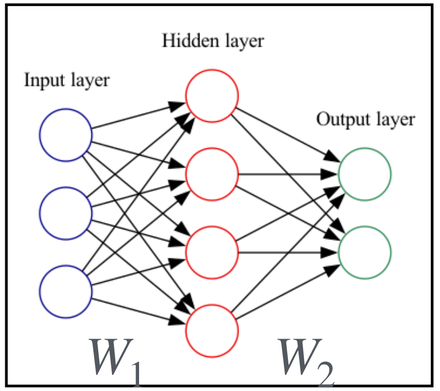
- # parameters for W_1: $h \times n$
- Memory: $2(h \times n + n + h)$, 2 for FP16, hn: weight, n: input activation, h: output activation
- Compute (FLOPs): $2(h \times n)$, 2 for mul + add



Matrix multiplication

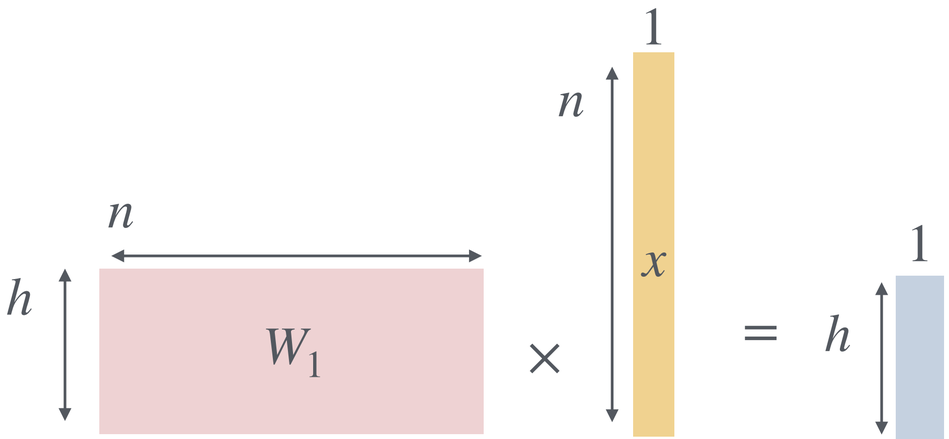
Review

Parameters, memory, and computation



2 layer neural network for single input

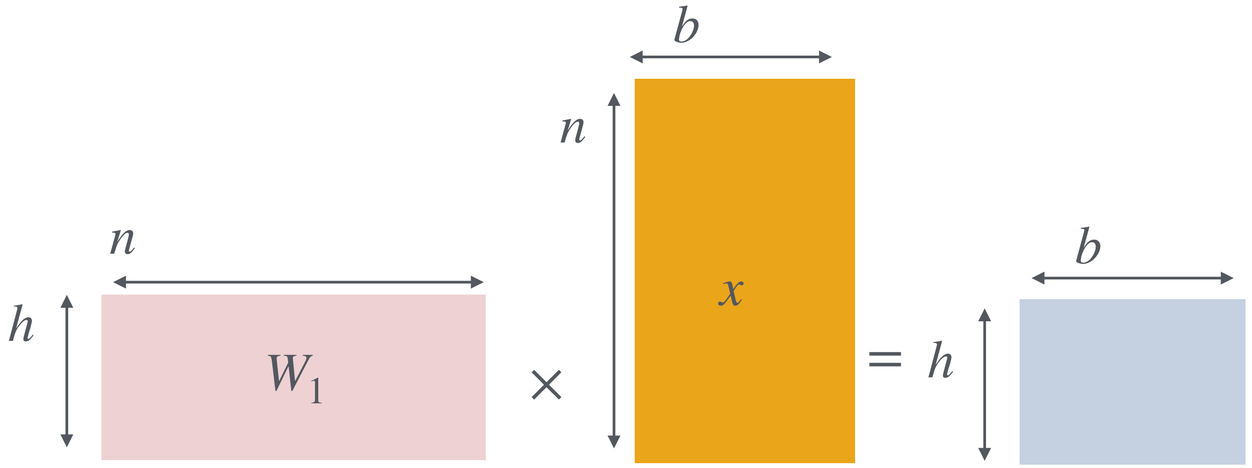
- # parameters for W_1: $h \times n$
- Memory: $2(h \times n + n + h)$, 2 for FP16, hn: weight, n: input activation, h: output activation
- Compute (FLOPs): $2(h \times n)$, 2 for mul + add



Matrix multiplication

2 layer neural network for mini-*batch* *input (size* b*)*

- # parameters for W_1: $h \times n$
- Memory: $2(h \times n + b \times n + b \times h)$, 2 for FP16, hn: weight, bn: input activation, bh: output activation
- Compute (FLOPs): $2(h \times n \times b)$, 2 for mul + add



Matrix multiplication

Memory accesses

As the model size grows, the amount of memory is required also grows for loading parameters, activations, and kv-cache

Decode phase of LLM inference is in memory bound region of the roof line model

Current AI hardware has limitations related memory:

- Memory bandwidth
- Memory capacity
- Power

Ref. [6]

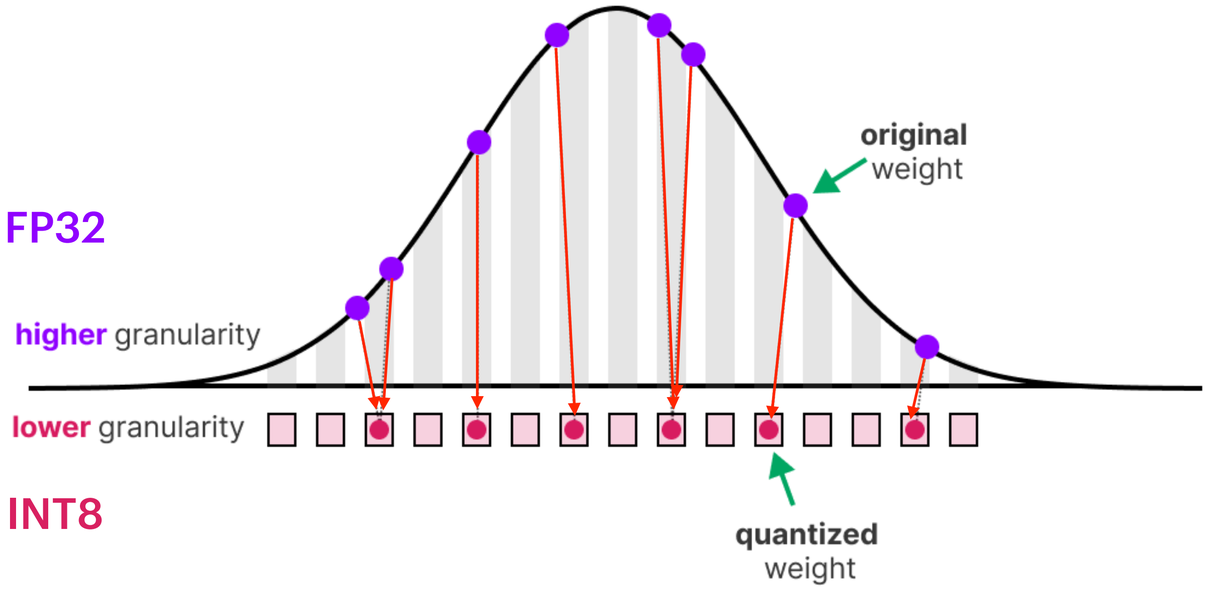
Low precision arithmetic

Traditionally 32 bit or 64 bit floating points are used in ML

Can we use low precision integer or floating point numbers

Quantization: a process of mapping from a continuous signal (or a signal where values come from a larger set) to a discretized signal (or one where values come from a smaller set)

- Quantization error: absolute value of the difference between the original value and quantized value



Ref. [2, 6]. Image source: link

Memory accesses with low precision

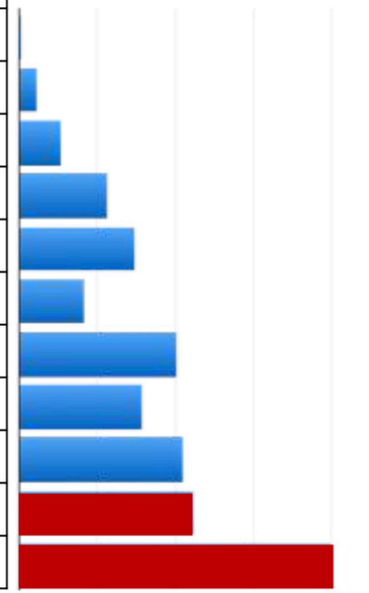
Memory accesses are the principal cost in latency and energy

Low precision weights means that each memory access fetch more data values, less memory bandwidth

Low precision weights takes up less memory space, less memory capacity

Low precision computation requires reduced energy, less power

Operation:	Energy (pJ)
8b Add	0.03
16b Add	0.05
32b Add	0.1
16b FP Add	0.4
32b FP Add	0.9
8b Multiply	0.2
32b Multiply	3.1
16b FP Multiply	1.1
32b FP Multiply	3.7
32b SRAM Read (8KB)	5
32b DRAM Read	640

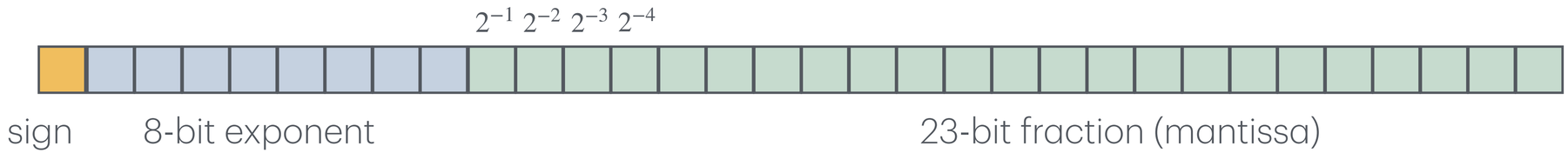


[**Horowitz**, *ISSCC* 2014]

Ref. & image source: [5]

Floating point number

Single precision, FP32



Represented number = $(-1)^{\text{sign}} \times (1 + \text{fraction}) \times 2^{\text{exponent-base}}$

Base = 127 = $2^{8-1} - 1$

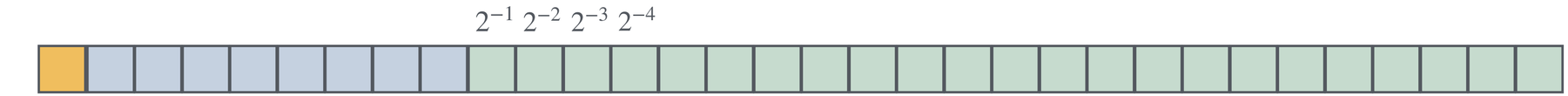
Dynamic range: difference between the max and min values that can be represented

Exponent width: range. Fraction width: precision

Ref. [1, 2, 6]

Floating point number

Single precision, FP32

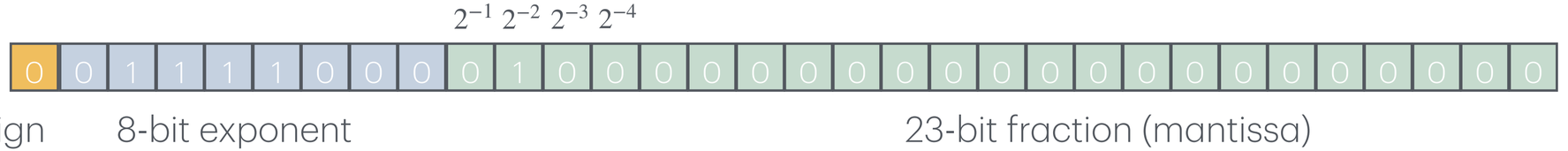


Represented number = $(-1)^{\text{sign}} \times (1 + \text{fraction}) \times 2^{\text{exponent-base}}$

Base = 127 = $2^{8-1} - 1$

Dynamic range: difference between the max and min values that can be represented

Exponent width: range. Fraction width: precision



Exponent: $2^6 + 2^5 + 2^4 + 2^3 = 120$, Base = 127 = $2^{8-1} - 1$, sign=0

Fraction: $2^{-2} = 0.25$

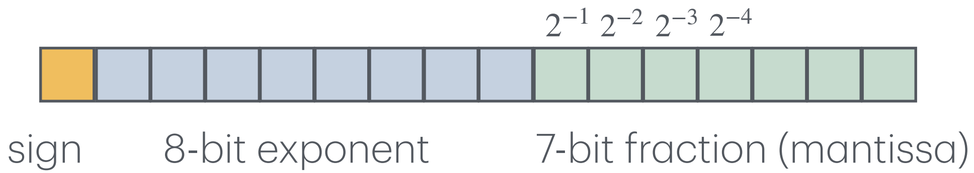
Represented number = $(-1)^{\text{sign}} \times (1 + \text{fraction}) \times 2^{\text{exponent-base}}$

Value = $(-1)^{0} \times (1 + 0.25) \times 2^{120\text{-}127} = 0.009765625$

Ref. [1, 2, 6]

Floating point number

Brain float 16, BF16



FP16 with exponent size 5 has a few drawbacks compared FP32 with exponent width 8 including smaller dynamic range and larger round errors

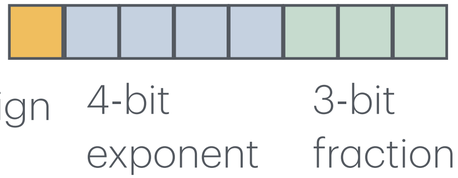
Brain float16 (BF16): dynamic range is now the same as FP16

- This can considered as a truncated FP32
- ML applications are more tolerant to quantization error than they are to overflow (dynamic range)

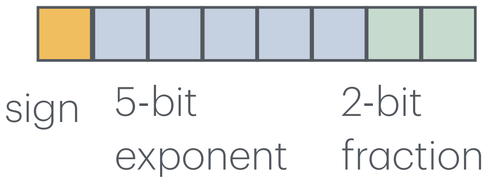
Ref. [1, 2, 6]

Floating point number

FP8 (E4M3, E5M2). H100 supports FP8



FP8 (E4M3)
FP8 E4M3 does not have INF, and S.1111.1112 is used for NaN.
Largest FP8 E4M3 normal value is S.1111.1102 =448.



FP8 (E5M2) for gradient computation in backward propagation
FP8 E5M2 have INF (S.11111.002) and NaN (S.11111.XX2).
Largest FP8 E5M2 normal value is S.11110.112 =57344.

Exponent width determines dynamic range. Fraction width determines precision. They need to be trade-off.

For training, dynamic range is more important.
For inference, precision is more important

Credit to [1]

Symmetric, uniform quantization

Input: r, floating-point

Output: $Q(r)$ in unsigned integer grid $\{0,...,2^b - 1\}$ or signed integer grid $\{-2^{b-1},\ldots,2^{b-1} - 1\}$

$$Q(r) = \text{clamp}(\text{round}\left(\frac{r}{s}\right); \alpha, \beta)$$

$$s = \frac{\beta - \alpha}{2^b - 1}$$

s is quantization step size, or scale (in floating-point)

round() is round-to-nearest integer operator

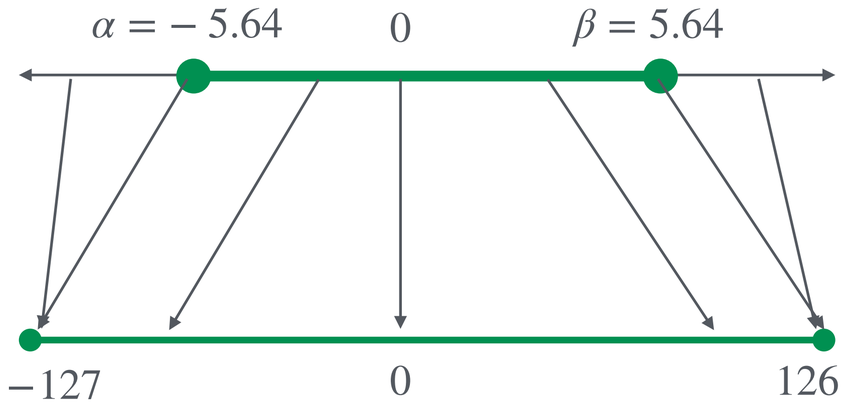
clamp(x; a, c) is clamp operator that clamps x within a and c

b is integer bit widths

De-quantization: $r \approx \hat{r} = s(Q(r))$

FP32 (before quantization)

INT8 (after quantization)



Ref. [4, 7]

Quantization errors

Clipping error:

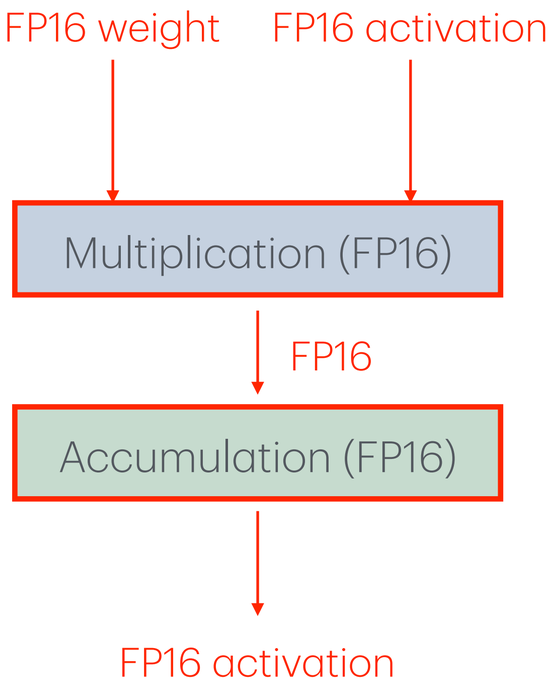
any values of r that lies outside of min and max values (α, β), outlier, is clipped to the limit, where $q_{min} = s(\alpha)$, $q_{max} = s(\beta)$
This error occurs for the outlier values

Rounding error:

Quantized value does not map back to the original input
This error is bounded by $\left\{ -\frac{1}{2}s, \frac{1}{2}s \right\}$ when the input r lies in the between two quantization grids

Ref. [7]

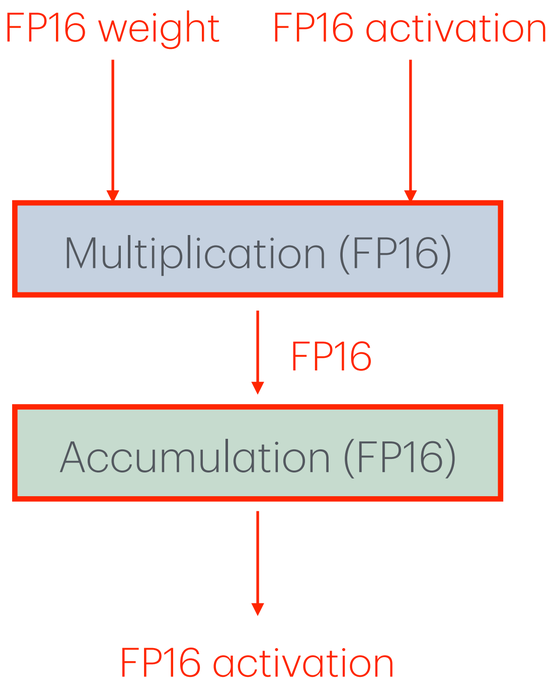
Quantized inference



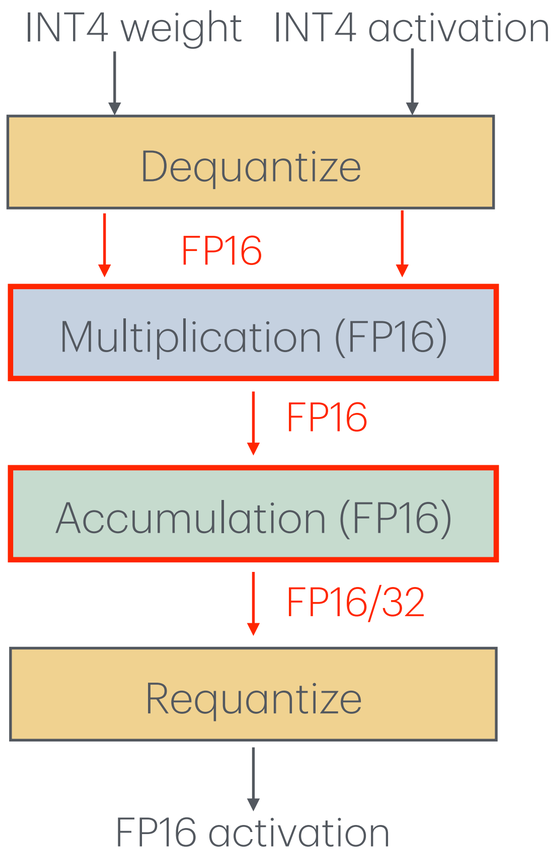
FP16, No quantization

Ref. [7]

Quantized inference



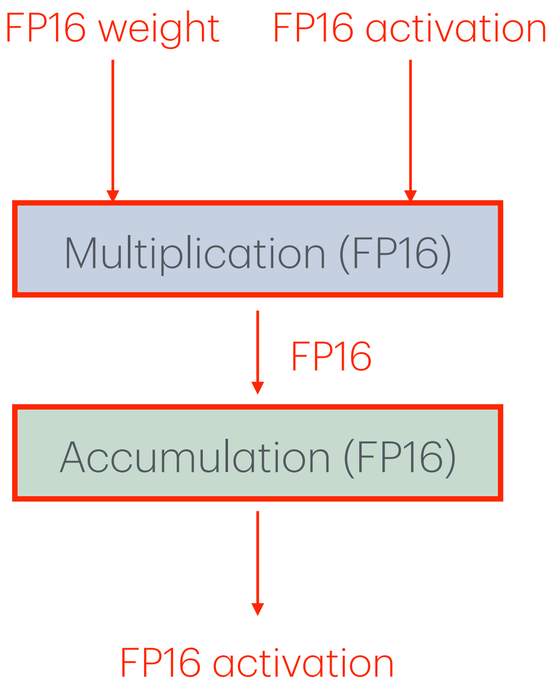
FP16, No quantization



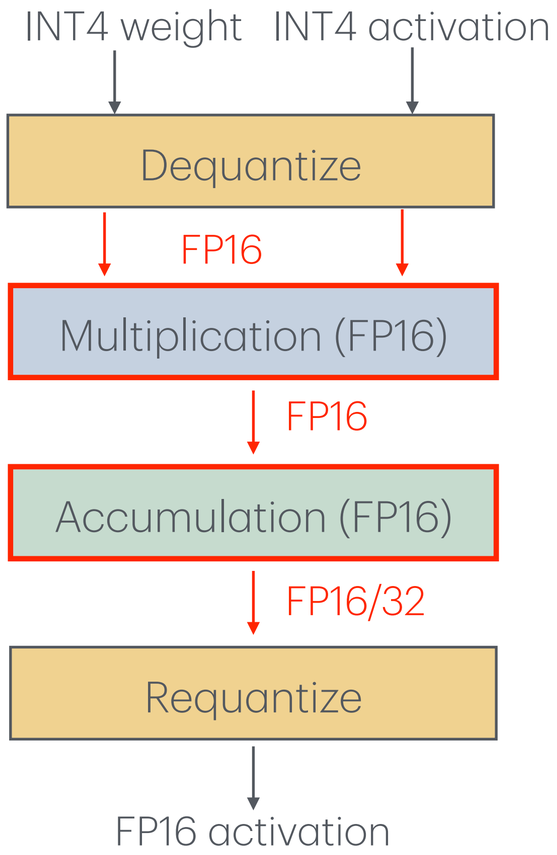
INT4 simulated quantization (fake quantization)

Ref. [7]

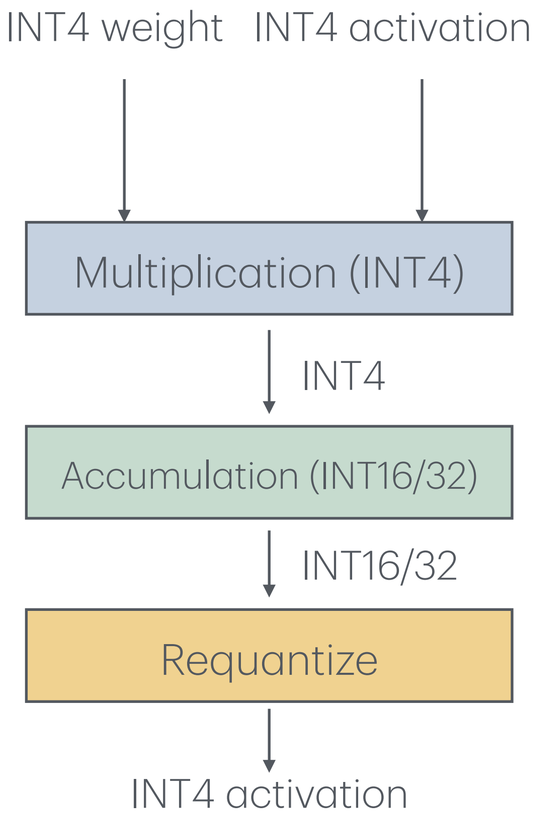
Quantized inference



FP16, No quantization



INT4 simulated quantization (fake quantization)



Integer only quantization

Ref. [7]

Post-training quantization (PTQ)

Quantizes a pre-trained model requires small calibration dataset

Up to 8-bit width, performance degradation is manageable

Challenges in LLM quantization:

- Activation layer has outliers and this causes performance degradation

Methods only quantizes weights only.

Methods that quantizes weights and activations. Requires to handle activation outliers

Ref. [4]

Knowledge distillation

Challenges in using large model

Larger model requires more memory and compute

Smaller model is less costly to operation

Can we train and use smaller model?

- Small model is hard to train and it underfit large datasets

Can we use a large model as a teacher to train a small model as a student?

- Knowledge distillation

Ref. [1]

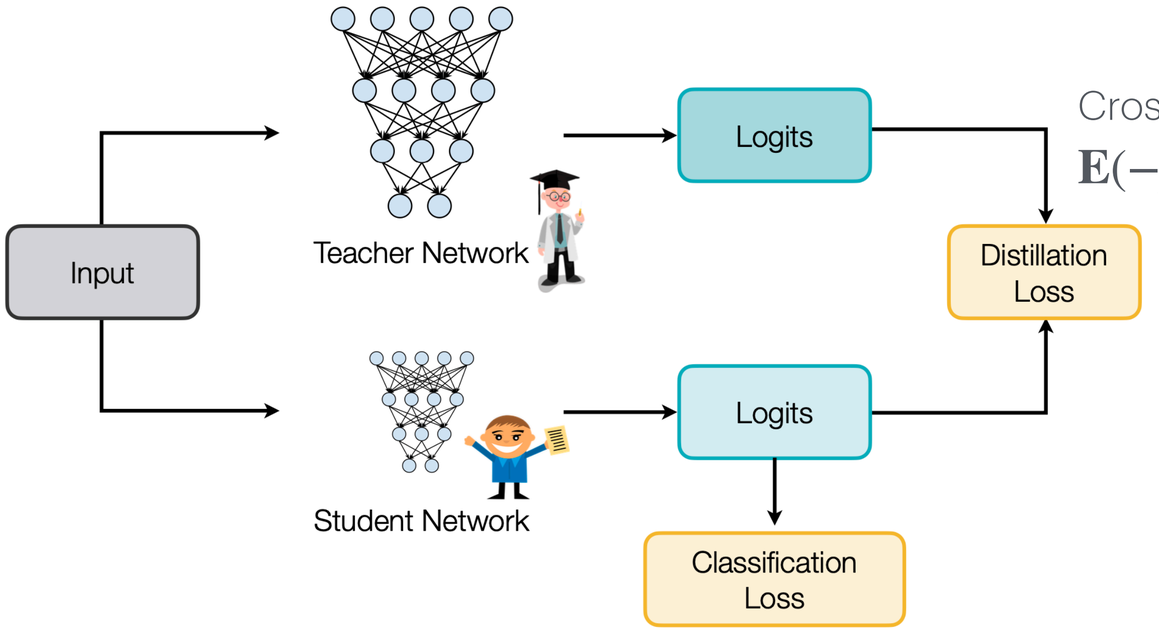
Knowledge distillation

Larger model is called teacher, and smaller model is called student

Model distillation trains the student to match the performance of the teach model using distillation loss

Output logits used to to compute cross-entropy loss

Gemini 1.5 Flash model was KD (knowledge distillation) trained with Gemini 1.5 Pro.



Cross entropy loss:

$$\mathbf{E}(-p_t \log p_s)$$

Ref. [4], image source [1]

Summary: Key technical concepts

1. Pruning: remove less important parameters to improve memory and computation requirements

2. Quantization: use low-precision numbers to improve memory and computation requirements

3. Knowledge distillation: teach smaller student model using the large teacher model

Algorithm optimization

Speculative decoding

LLM decode phase is memory bound. LLM generates token-by-token.

Use a small draft model to generate, and a large model to verify.

- Draft model: a small LLM (e.g 7B)
- Target model: a large LLM (e.g. 175B, main model that we are using)

Steps:

- The draft model generates (decode phase) K tokens autoregressively (one by one)
- The target model process K tokens in parallel, as in prefill phase, and get the prob. distribution for each location
- Verify the K token from the draft model and decide to keep K tokens or reject them

2-3x times speed up for identical output for common cases

Ref. [1]

Speculative decoding

Small draft model writes a draft -> Large target model verifies it

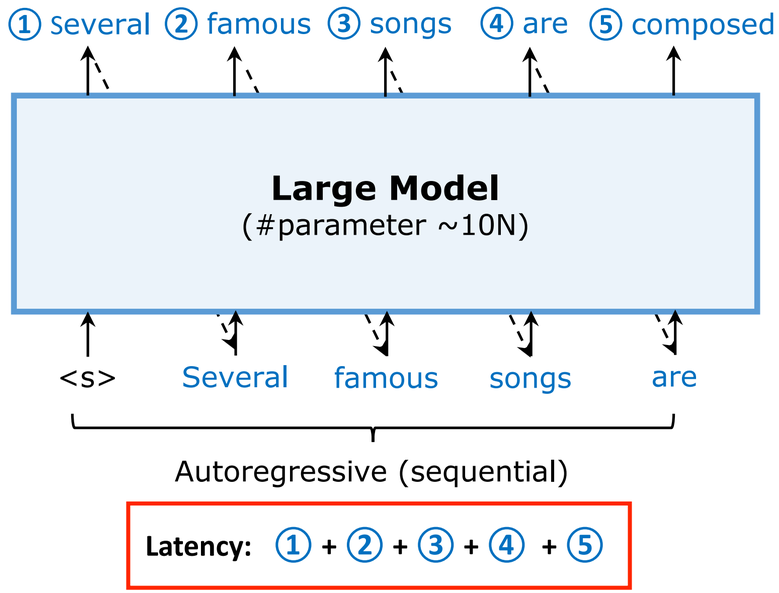


Image source: link

Speculative decoding

Small draft model writes a draft -> Large target model verifies it

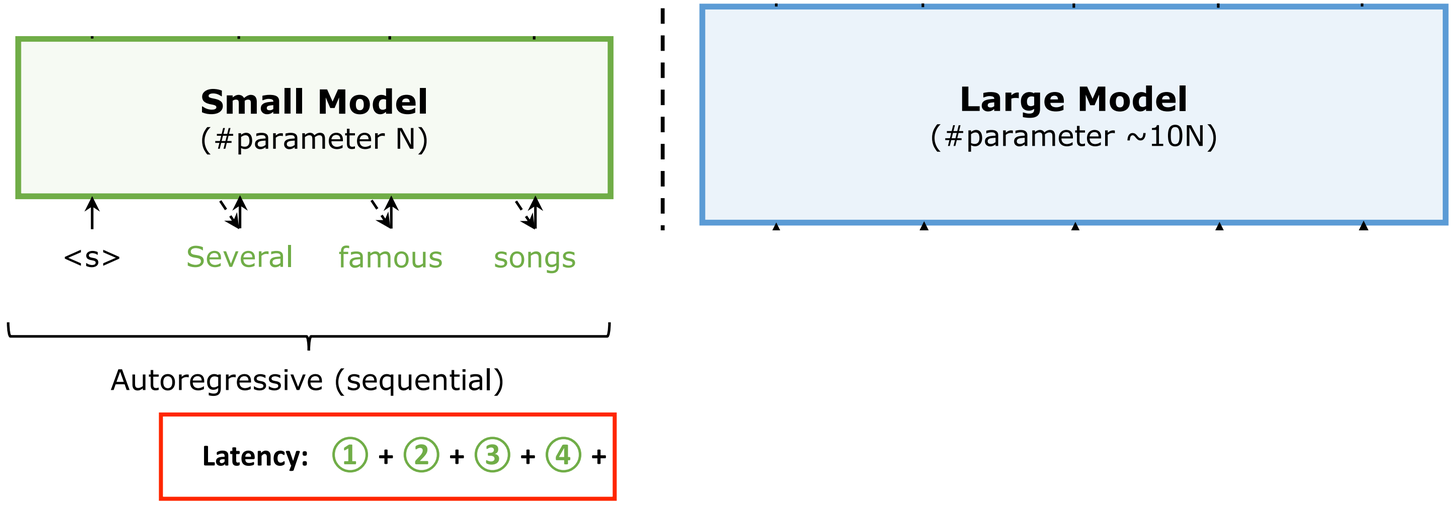


Image source: link

Speculative decoding

Small draft model writes a draft -> Large target model verifies it

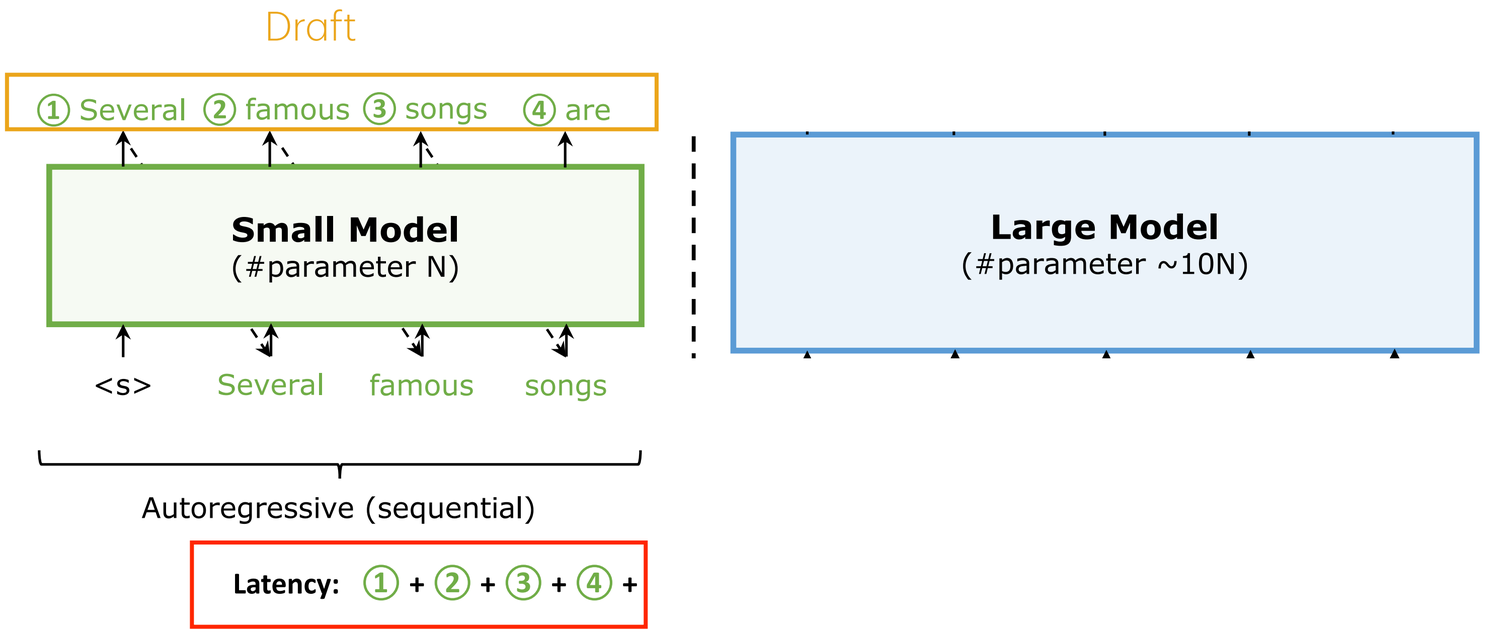


Image source: link

Speculative decoding

Small draft model writes a draft -> Large target model verifies it

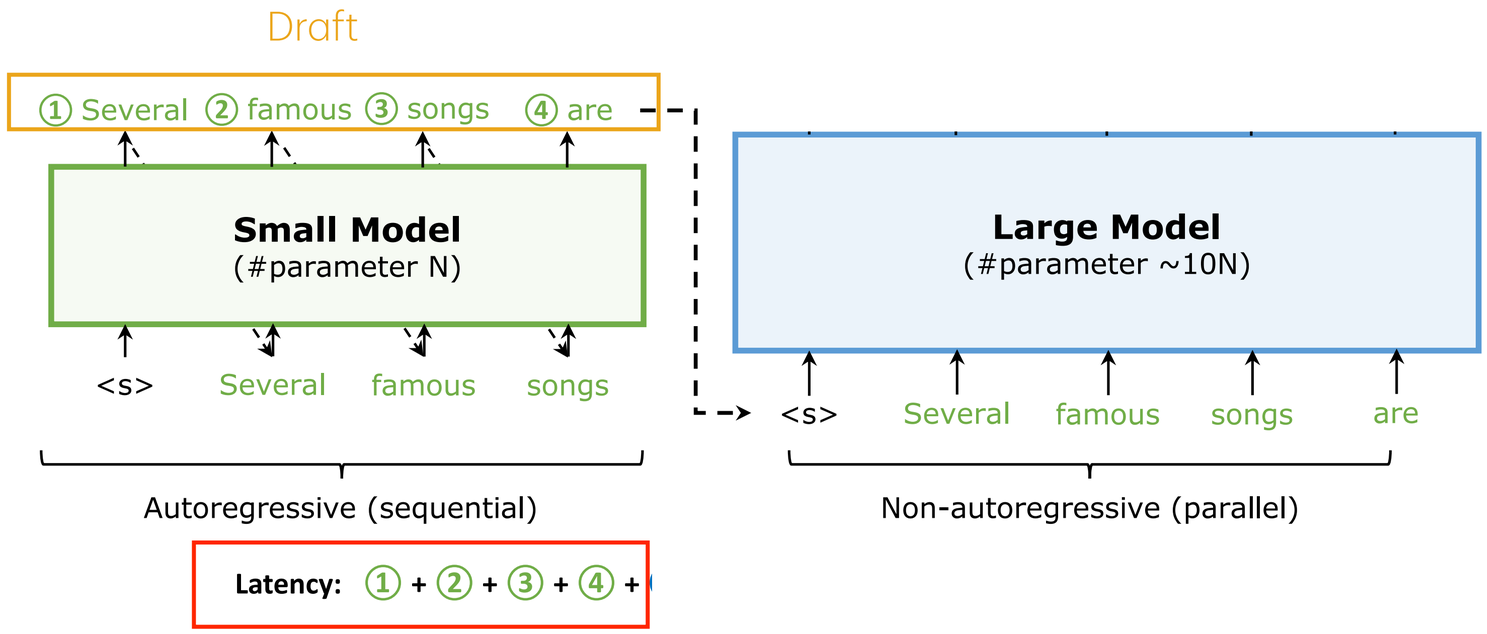


Image source: link

Speculative decoding

Small draft model writes a draft -> Large target model verifies it

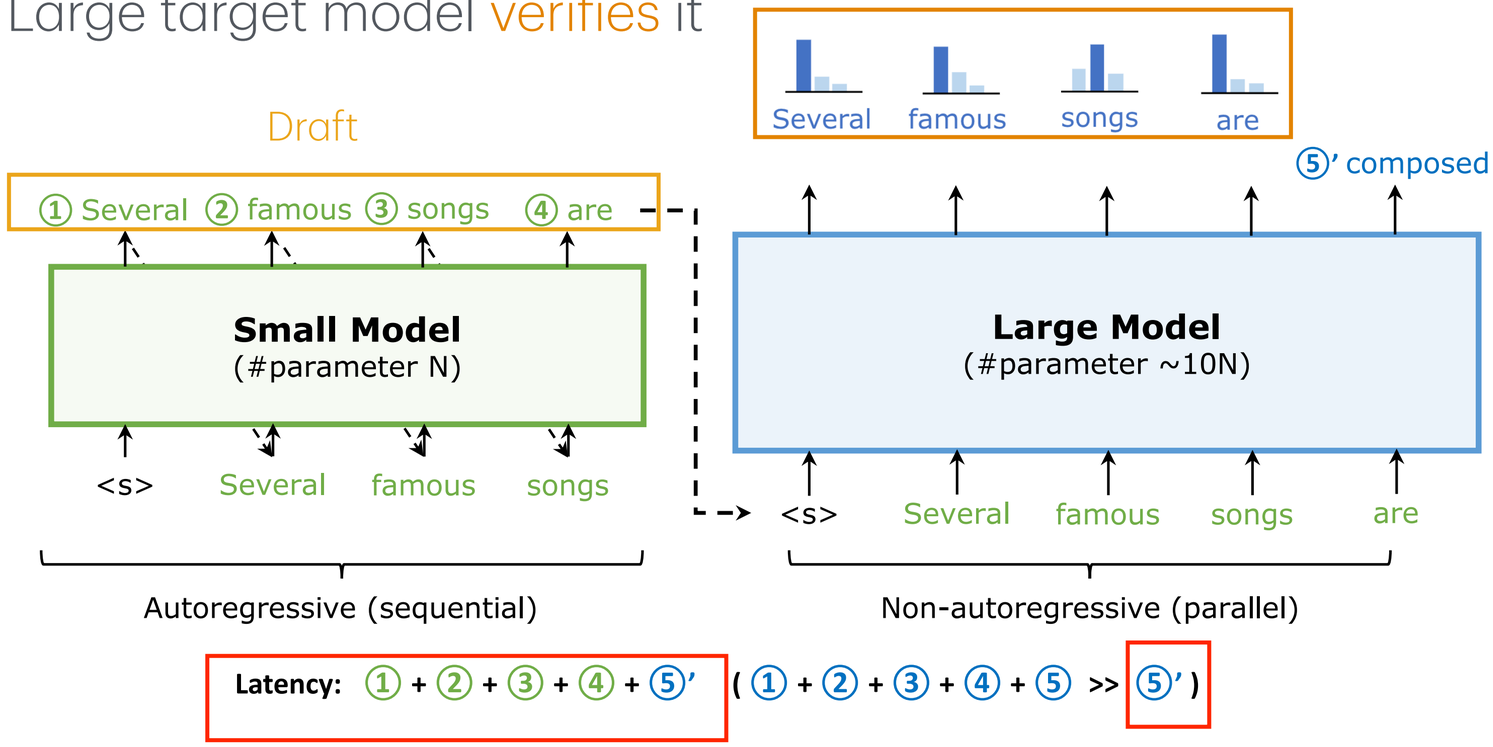
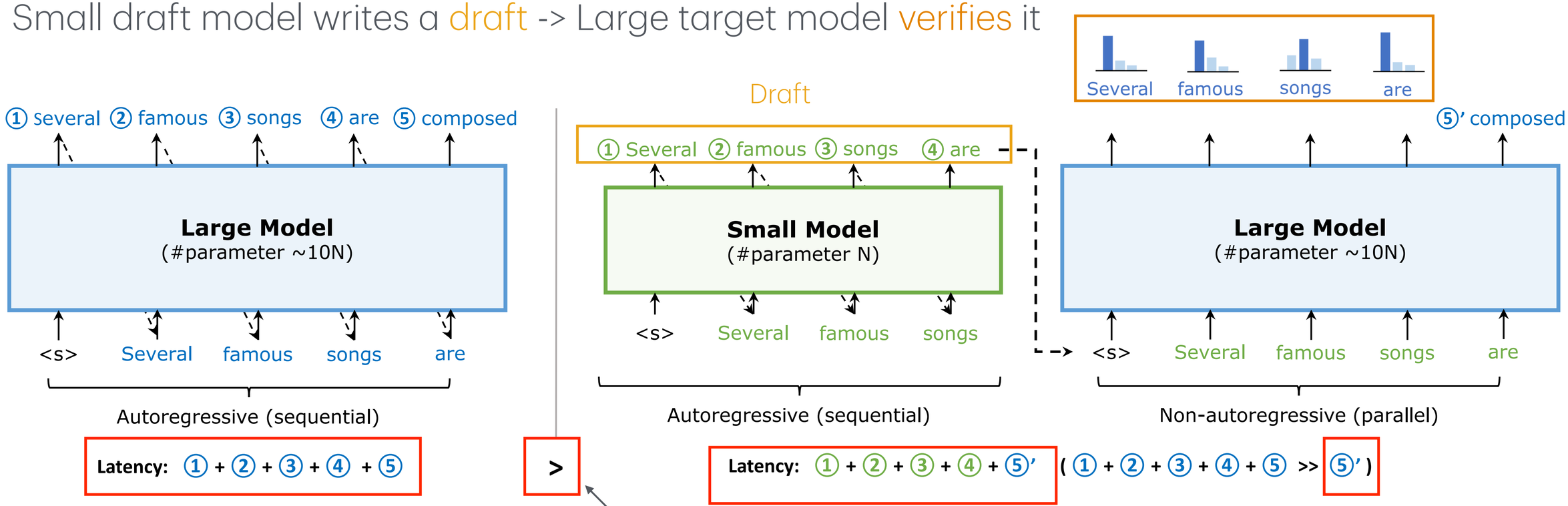


Image source: link

Speculative decoding

Small draft model writes a draft -> Large target model verifies it



Latency with the draft and the target model is smaller than the single target model

Image source: link

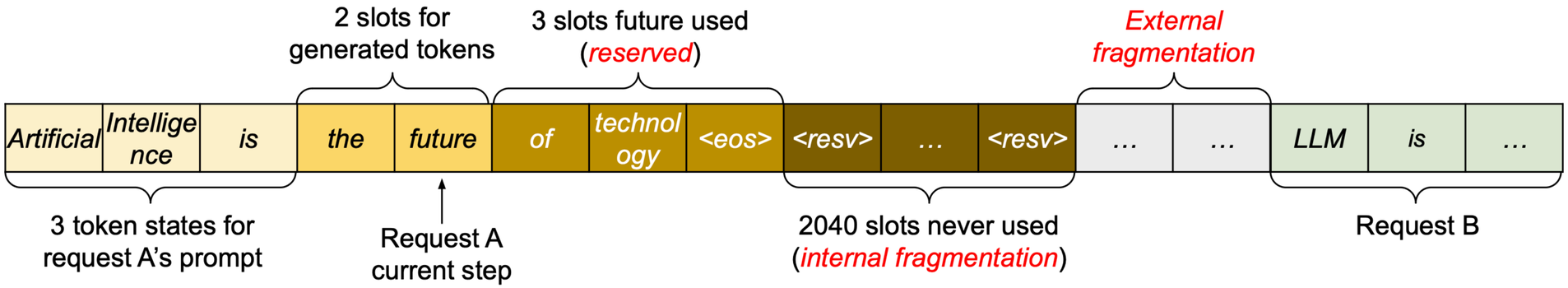
System level optimization

PagedAttention (vLLM)

The KV cache could be large with long context.
During decoding phase, Keys and Values of all previous tokens in all layers need to be cached

Memory waste in KV cache

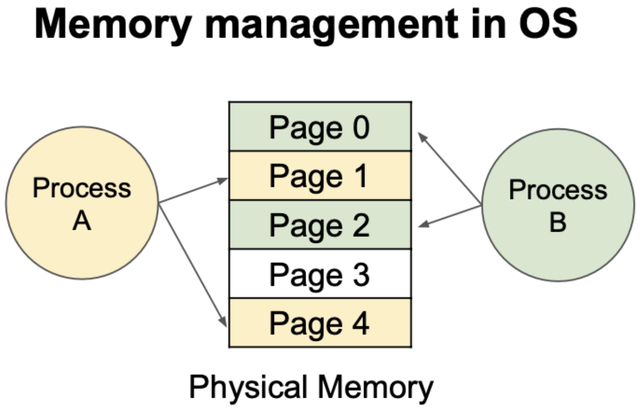
- Internal fragmentation: over-allocated due to the unknown output length.
- Reservation: not used at the current step, but used in the future.
- External fragmentation: due to different sequence lengths.



Ref. [1], image source: link

PagedAttention (vLLM)

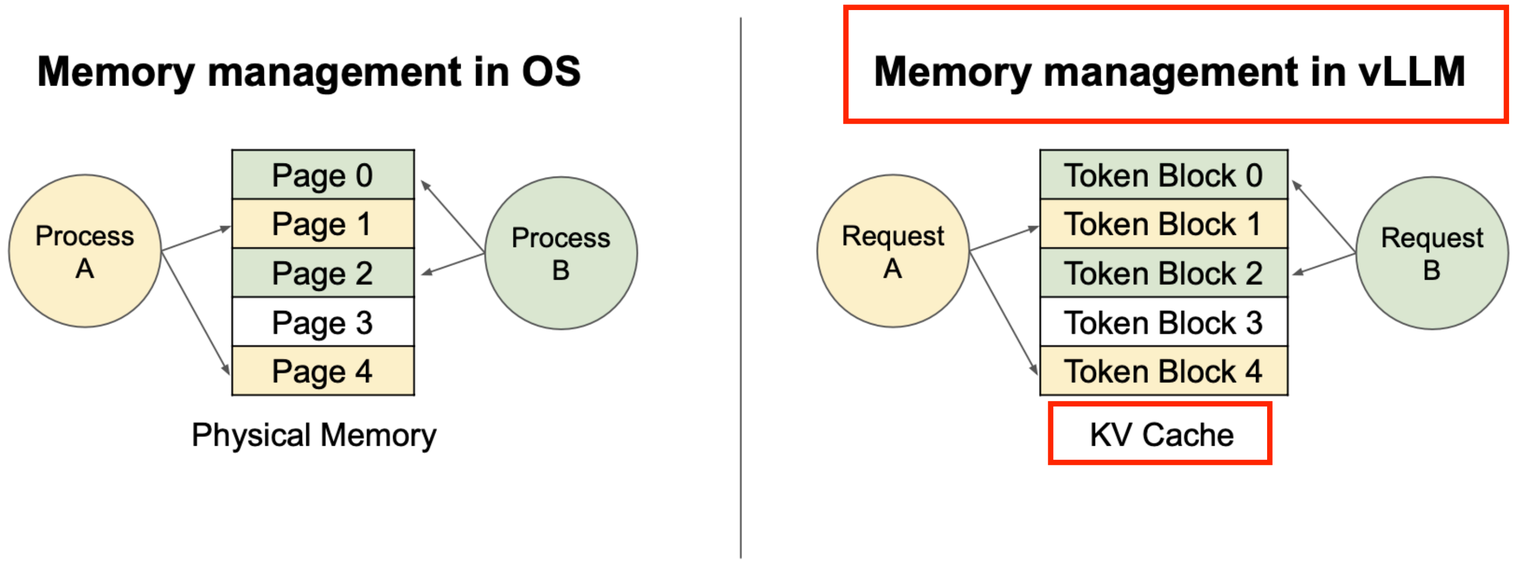
Inspired by OS virtual memory and paging



Ref. [1], image source: link

PagedAttention (vLLM)

Inspired by OS virtual memory and paging



Ref. [1], image source: link

Hardware aware algorithm design

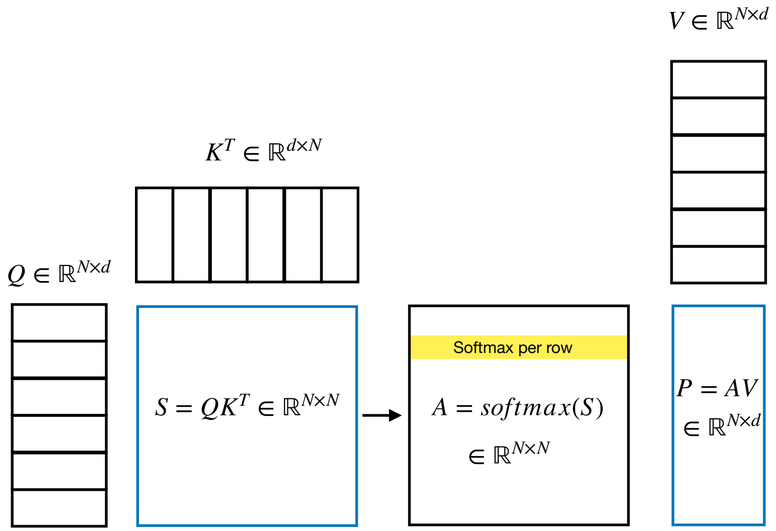
Attention scaling

With Q, K, V in Nxd dimension, attention score is NxN matrix. N is the sequence length

Attention memory and compute scales quadratically in sequence length (N)

Transformer model scales poorly for long sequences

There are many proposals to improve efficiency of the quadratic scale, but they are less effective in performance (link).



Ref. [2]

FlashAttention*

Standard attention implementations do not account for long latency global memory (HBM) access

FlashAttention's approach:

- Tiling: assign blocks of work to parallel processing units to exploit locality as much as possible. Avoid materialize the full NxN matrix during the attention computation. Then we can load all of Q and K into shared memory
- Recompute: when task is in memory-bound region, it makes sense to recompute
- Fusion: save trips to/from memory by fusing operations

FlashAttention access 9x times fewer HBM resulting 6x faster in runtime than standard attention despite that it computes 13x more times.

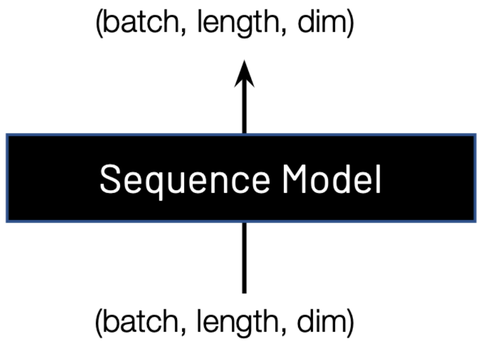
Attention	Standard	FlashAttention
GFLOPs	66.6	75.2 (↑13%)
HBM reads/writes (GB)	40.3	4.4 (↓9x)
Runtime (ms)	41.7	7.3 (↓6x)

*: link

Ref. and image source [2]

Non-transformer based models

Sequence model



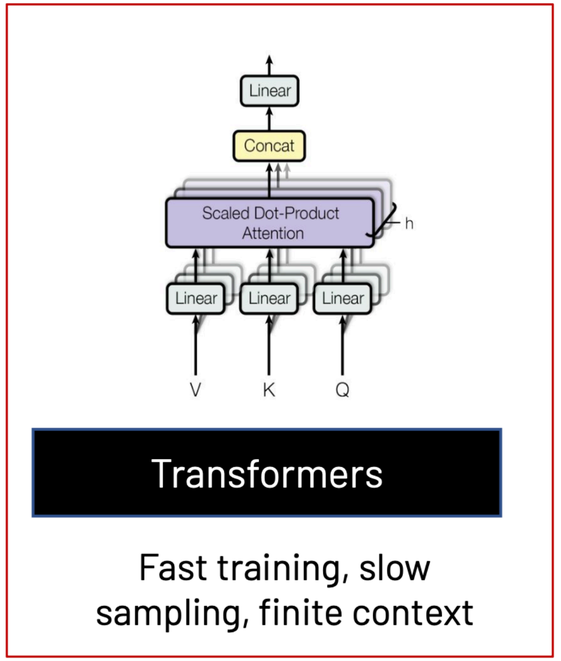


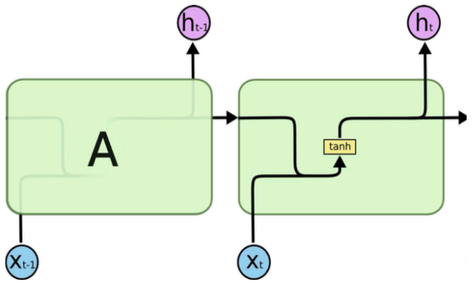
Image source [2]

Sequence model

(batch, length, dim)

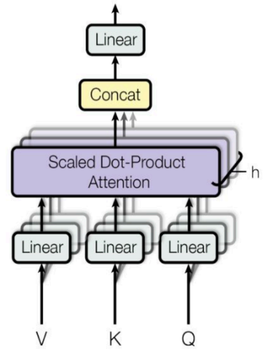
Sequence Model

(batch, length, dim)



Recurrent Neural Nets

Slow training, fast sampling, (potentially) infinite context



Transformers

Fast training, slow sampling, finite context

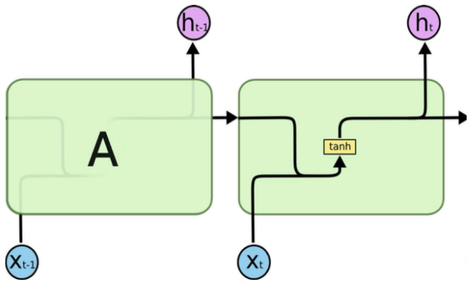
Image source [2]

Sequence model

(batch, length, dim)

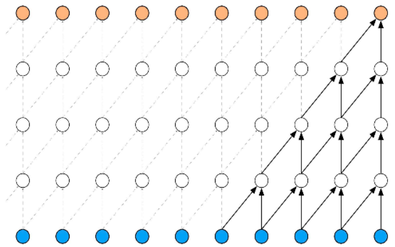
Sequence Model

(batch, length, dim)



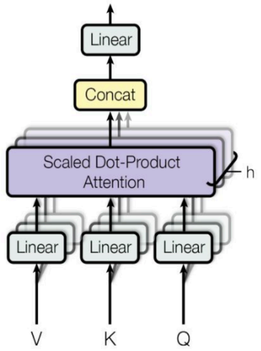
Recurrent Neural Nets

Slow training, fast sampling, (potentially) infinite context



ConvNets

Fast training, slow sampling, finite context



Transformers

Fast training, slow sampling, finite context

Image source [2]

What properties do we want from sequence model?

	Training	Inference	Patterns
Transformer	- Parallelizable - Bounded Context $O(N^2)$ scaling	- O(N) Inference with KV caching	- Local and global patterns, highly expressive!
RNN	- Sequential - Hard to train - Unbounded Context O(N) scaling	- Efficient O(1) Inference	- Local and global patterns (realistically, can't store all patterns)
CNN	- Parallelizable - Bounded Context O(N log N) scaling	- O(N log N) Inference with FFT-Convolution	- Local patterns - Interpretable patterns

Image source [2]

What properties do we want from sequence model?

	Training	Inference	Patterns
Transformer	- Parallelizable - Bounded Context $O(N^2)$ scaling	- O(N) Inference with KV caching	- Local and global patterns, highly expressive!
RNN	- Sequential - Hard to train - Unbounded Context O(N) scaling	- Efficient O(1) Inference	- Local and global patterns (realistically, can't store all patterns)
CNN	- Parallelizable - Bounded Context O(N log N) scaling	- O(N log N) Inference with FFT-Convolution	- Local patterns - Interpretable patterns

Transformer architecture is used by most of LLMs in present days, there are renewed interests in different types of sequence model to improve inference characteristics while keeping the performance of the Transformer models

Image source [2]

Summary: Key technical concepts

1. Speculative decoding: use a small draft model to predict tokens and a large target model verifies them, if matched, the resulting shorter generation time

2. PagedAttention: apply OS virtual memory management ideas to KV-cache management

3. FlashAttention: using tiling, recomputation, and fusing to attention computation and reduced global memory access to gain high gain in end-to-end run time

4. Non-transformer based models: despite that most of the model still use Transformer architecture, the inefficiency in inference triggered renew interest in other sequence models

Parting thoughts, next is LLM apps

LLMs are useful in many applications. To use them more efficiently and at a lower cost, there are various approaches for optimizing LLM inference and training.

Whether the size of the models will continue to grow or the Transformer architecture will continue to dominate is hard to predict. However, what we can do is use and understand the available models and make the best use of them for your work.

In the next and final session of the series, we will cover how to best use LLMs for different types of tasks. Many LLMs are very powerful; however, as the industry learns more about LLMs and their applications, we see many LLM-based solutions being built with numerous software components, resulting in more complex software and LLM interactions.

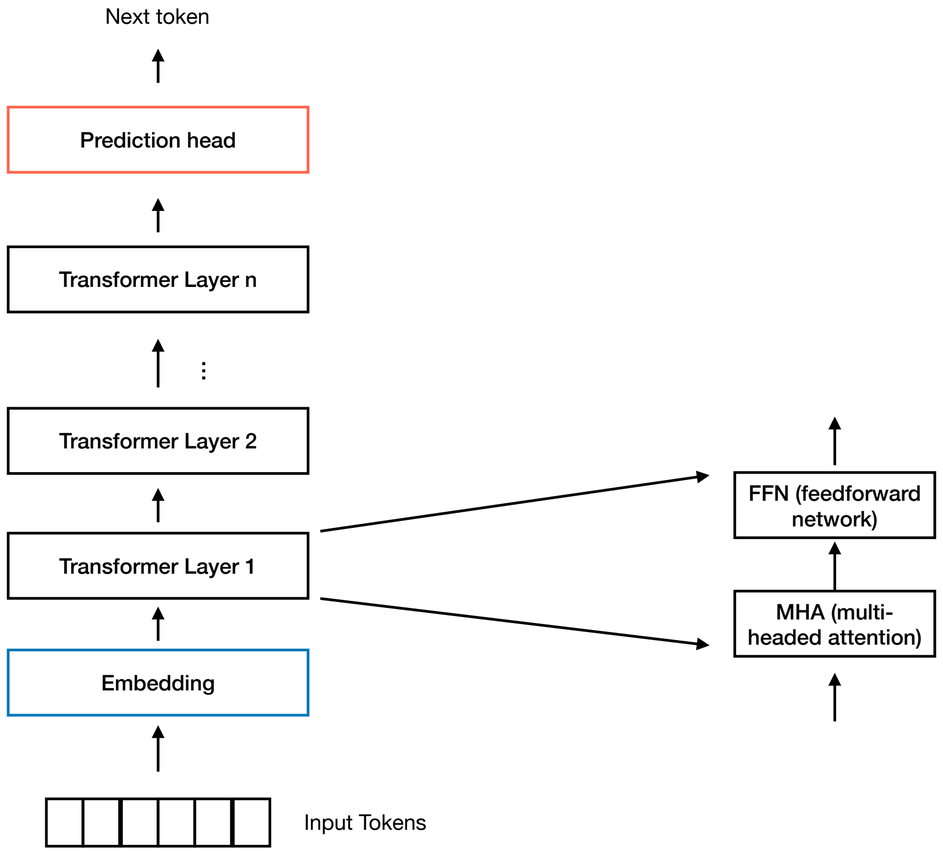
Stay tuned for the next session.

References

1. MIT 6.5940: TinyML and Efficient Deep Learning Computing
2. Systems for Machine Learning
3. A Survey of Techniques for Efficient Inference with Large Language Models, MSJAR, Aug., 2024
4. Inference Optimization for Foundation Models on AI Accelerators, KDD 2024
5. Machine Learning Systems (Spring 2022)
6. Advanced Machine Learning Systems
7. Quantization Methods for Efficient ML Inference (Hotchips 2023)

Additional slides

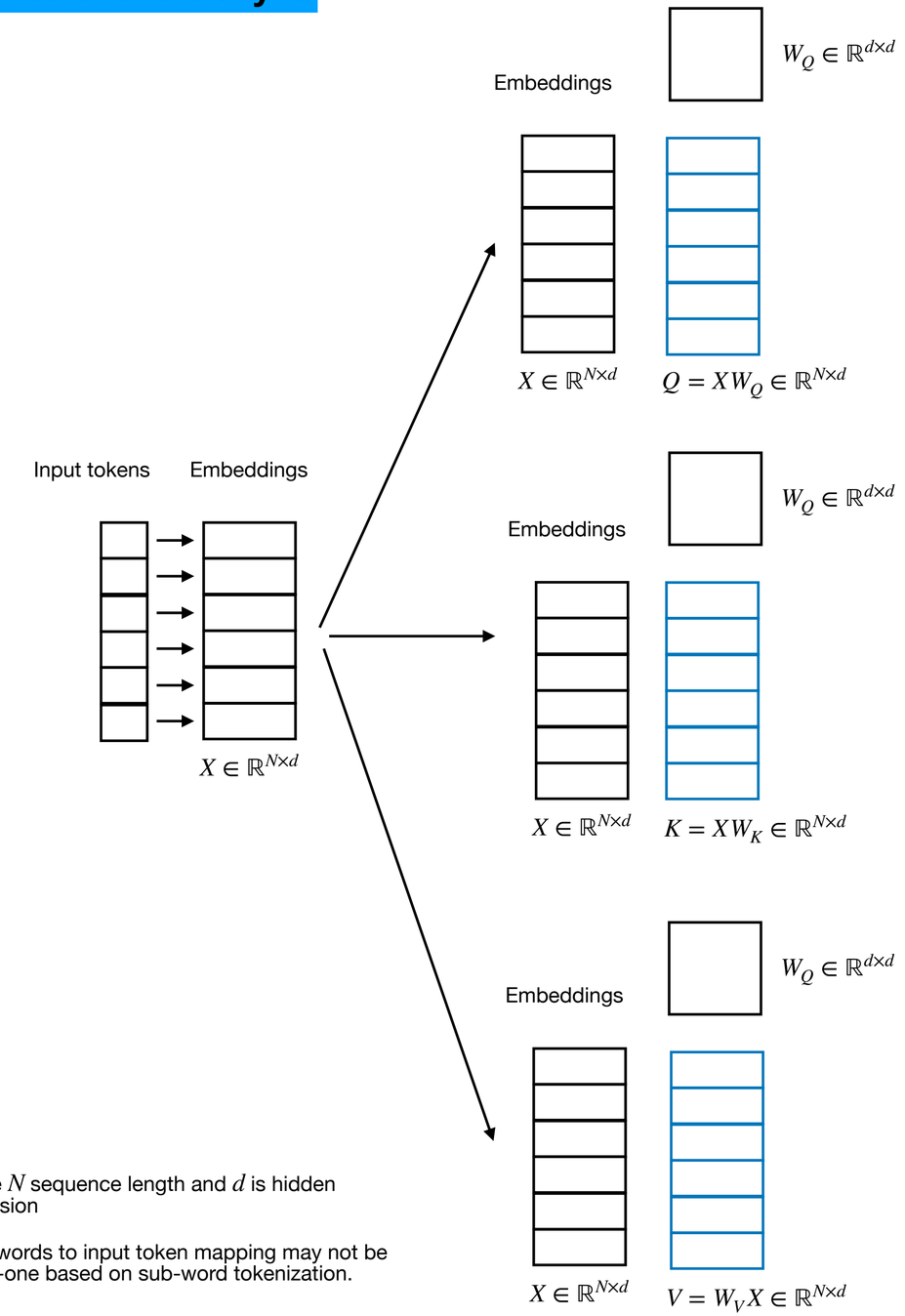
Overview of GPT (decoder) style LLM structure



* Tokenization from Input words to input tokens is omitted for simplicity

** We omit layer norm and residual addition for simplicity

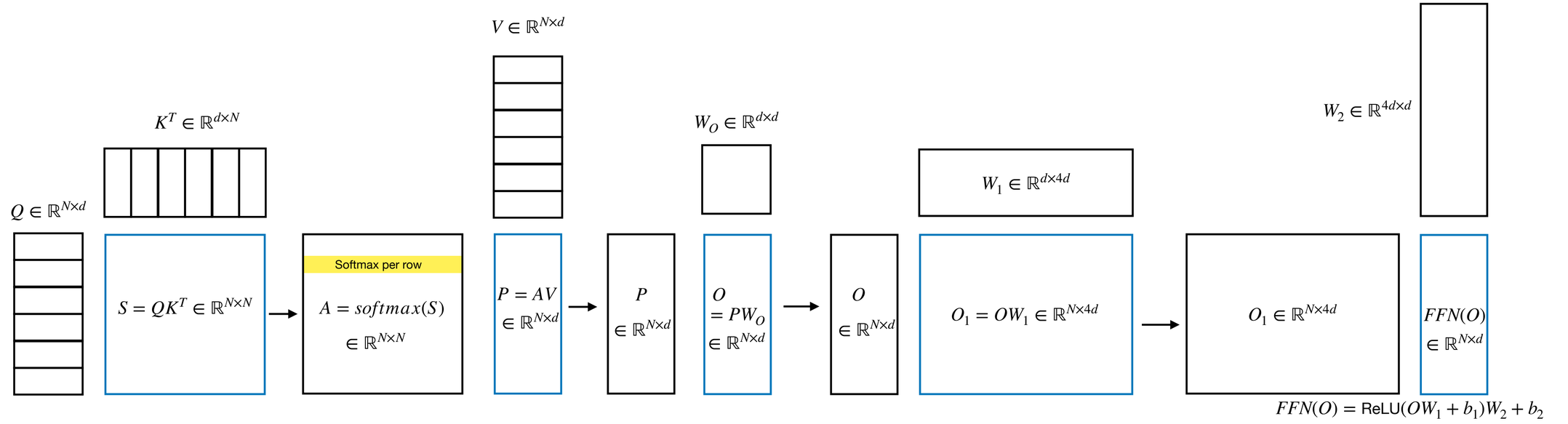
Input tokens to Q, K, V in a single Transformer layer



Where N sequence length and d is hidden dimension

*Input words to input token mapping may not be one-to-one based on sub-word tokenization.

Training: MHA (multi headed attention) and FFN (Feedforward network) in a single transformer layer



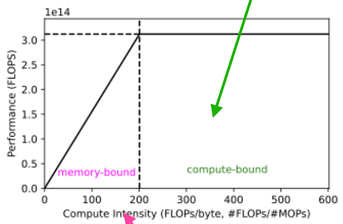
Forward pass compute	$O(N^2d)$	$O(N^2)$	$O(N^2d)$	$O(Nd^2)$	$O(Nd^2)$	$O(Nd^2)$

Where N sequence length and d is hidden dimension

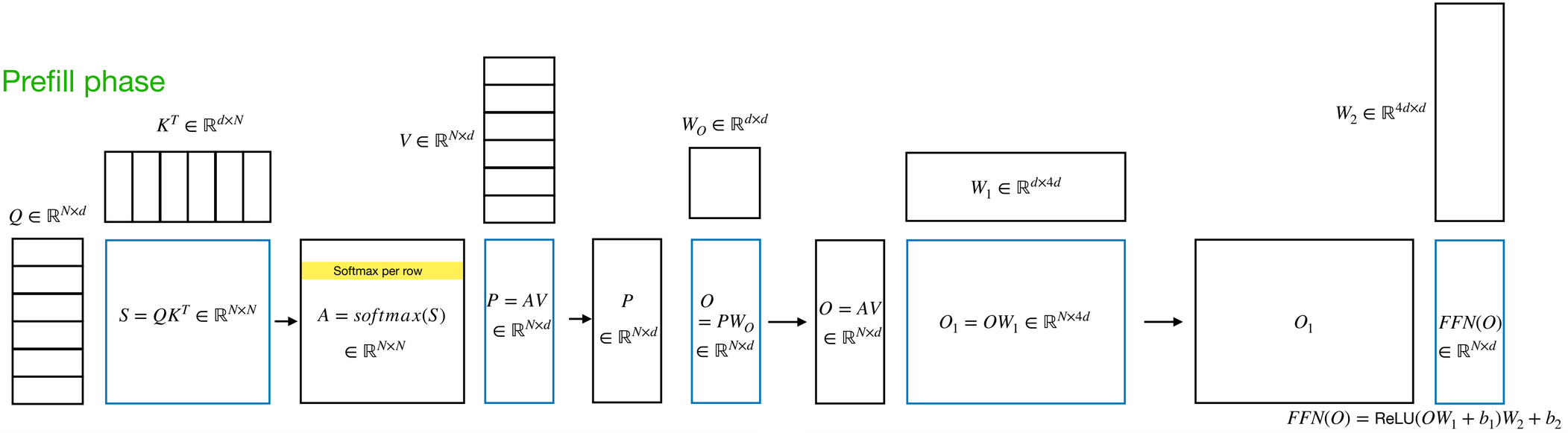
* We omit layer norm, residual addition, and causal masking for simplicity

** We use b, batch size of 1.

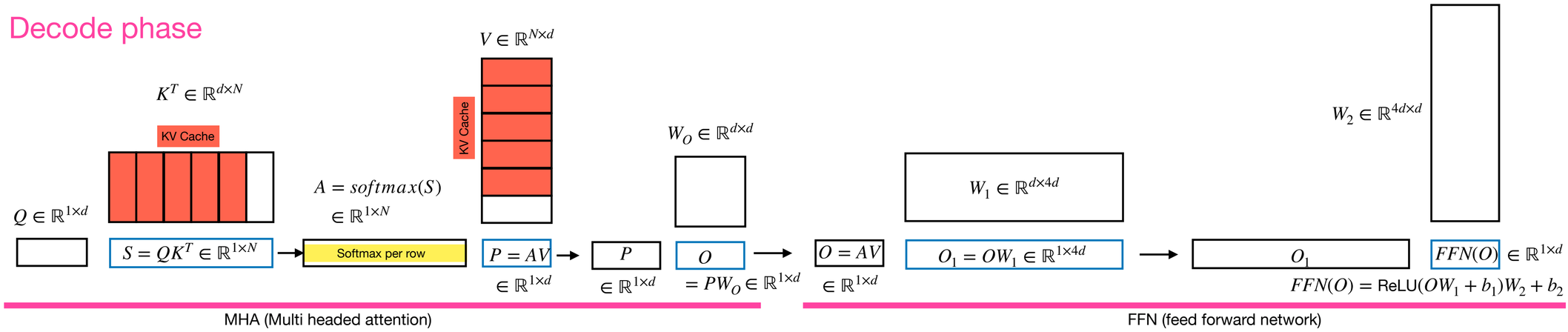
Inference: Two phases of inference: MHA (multi headed attention) and FFN



Prefill phase



Decode phase



Where N sequence length and d is hidden dimension

* We omit layer norm, residual addition, and causal masking for simplicity.

** This roofline model is based on A100 40G HBM