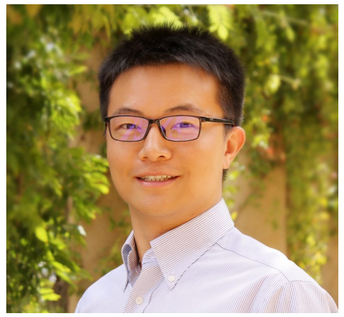
MIT HAN LAB

EfficientML.ai Lecture 13
LLM Deployment Techniques



Song Han

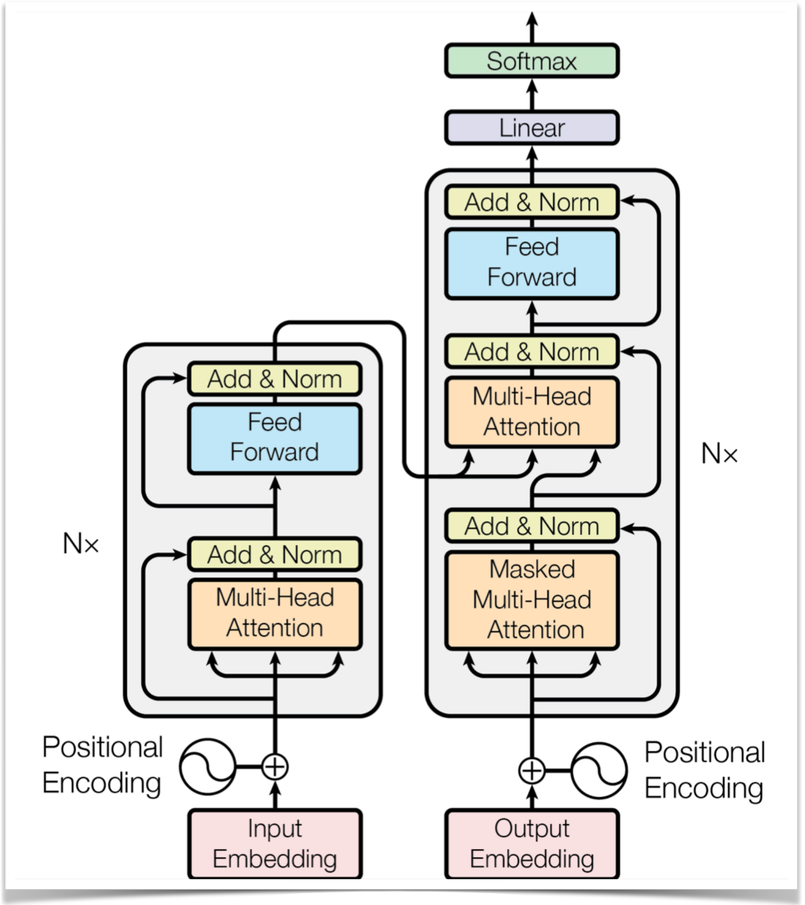
Associate Professor, MIT
Distinguished Scientist, NVIDIA

@SongHan_MIT

Lecture Plan

Today, we will cover:

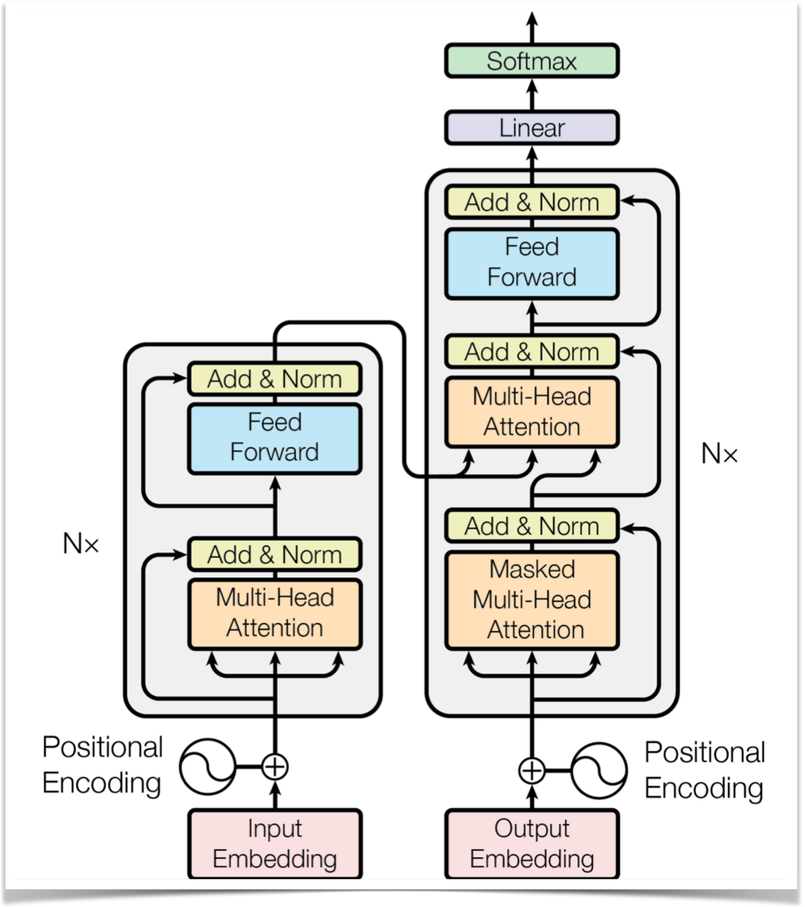
1. **Quantization**
    1. Weight-Activation Quantization: SmoothQuant
    2. Weight-Only Quantization: AWQ and TinyChat
    3. Further Practice: QServe (W4A8KV4)
2. **Pruning & Sparsity**
    1. Weight Sparsity: Wanda
    2. Contextual Sparsity: DejaVu, MoE
    3. Attention Sparsity: SpAtten, H2O
3. **LLM Serving Systems**
    1. Metrics for LLM Serving
    2. Paged Attention (vLLM)
    3. FlashAttention
    4. Speculative Decoding
    5. Batching



Lecture Plan

Today, we will cover:

1. **Quantization**
    1. Weight-Activation Quantization: SmoothQuant
    2. Weight-Only Quantization: AWQ and TinyChat
    3. Further Practice: QServe (W4A8KV4)
2. **Pruning & Sparsity**
    1. Weight Sparsity: Wanda
    2. Contextual Sparsity: DejaVu, MoE
    3. Attention Sparsity: SpAtten, H2O
3. **LLM Serving Systems**
    1. Metrics for LLM Serving
    2. Paged Attention (vLLM)
    3. FlashAttention
    4. Speculative Decoding
    5. Batching

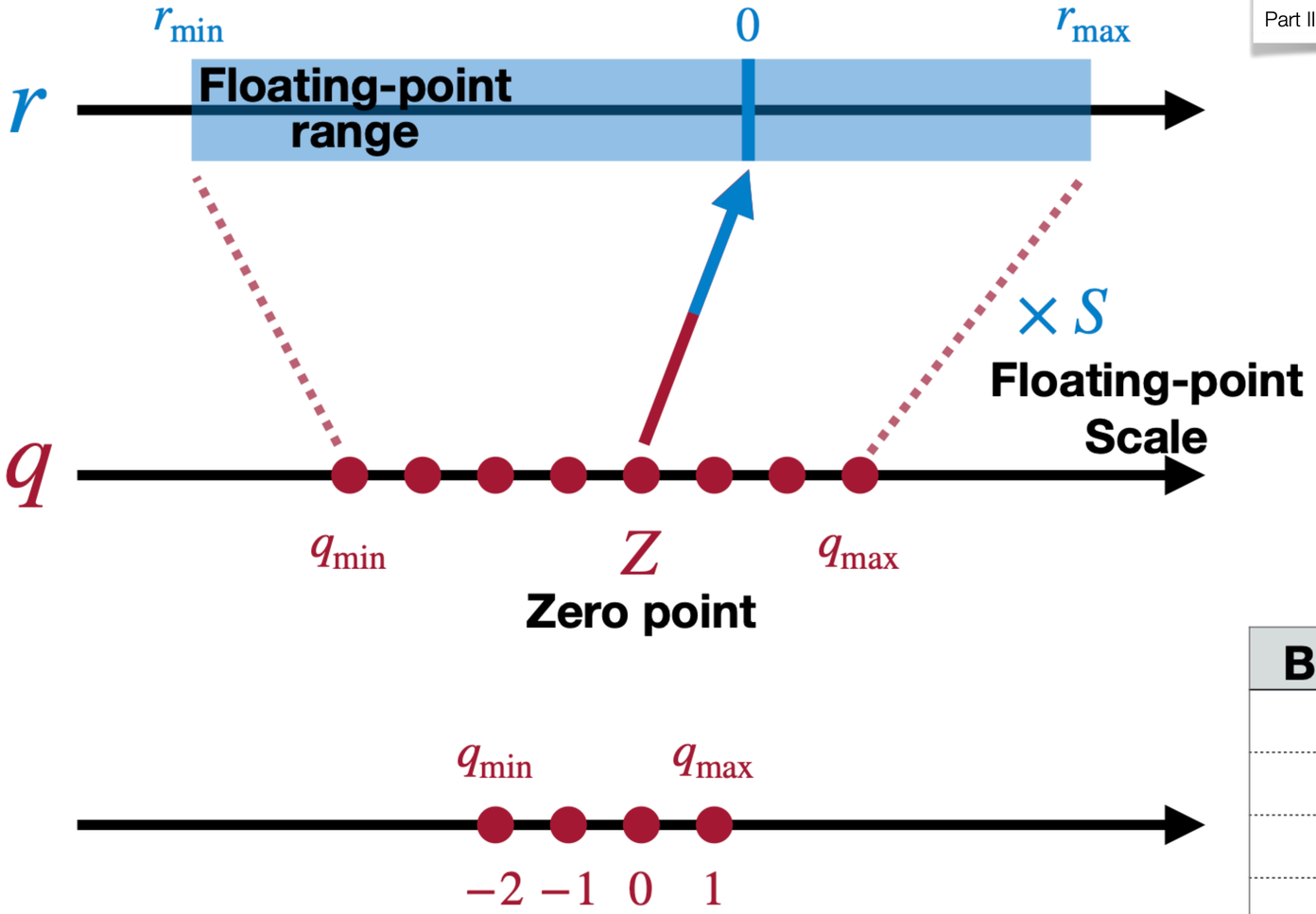


Quantization

Lowers the bit-width and improves efficiency

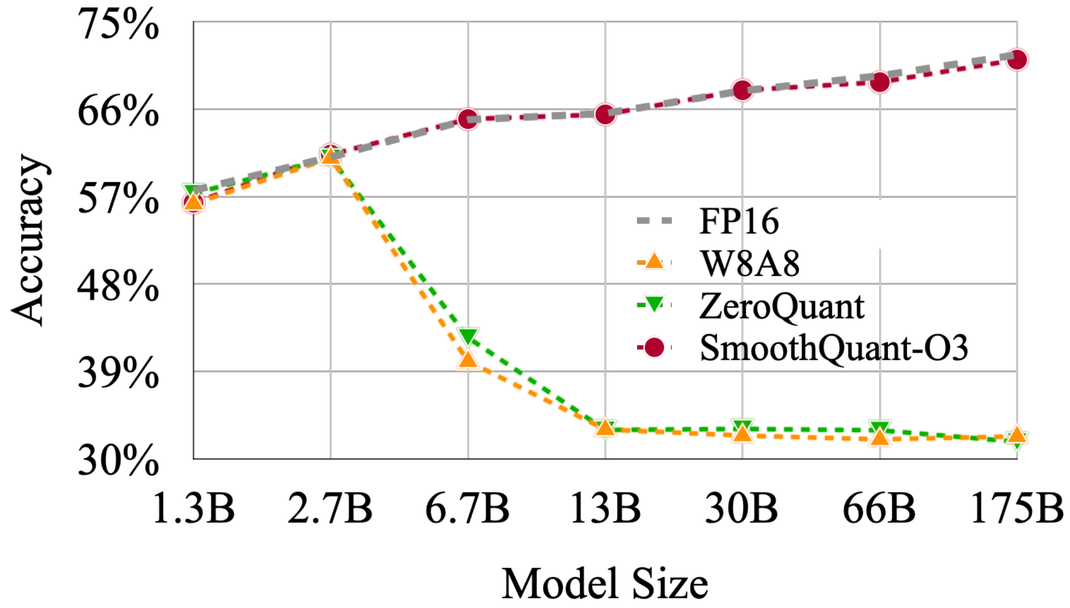
Lecture 05
Quantization
Part I

Lecture 06
Quantization
Part II



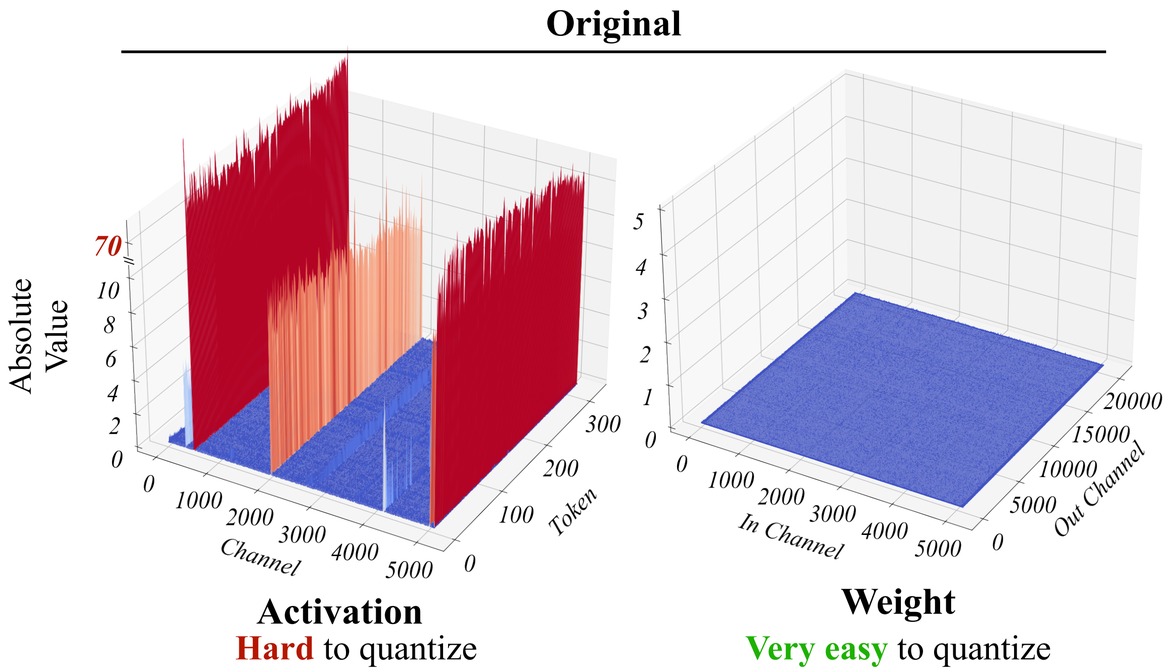
Binary	Decimal
01	1
00	0
11	-1
10	-2

Naive Quantization Methods are Inaccurate



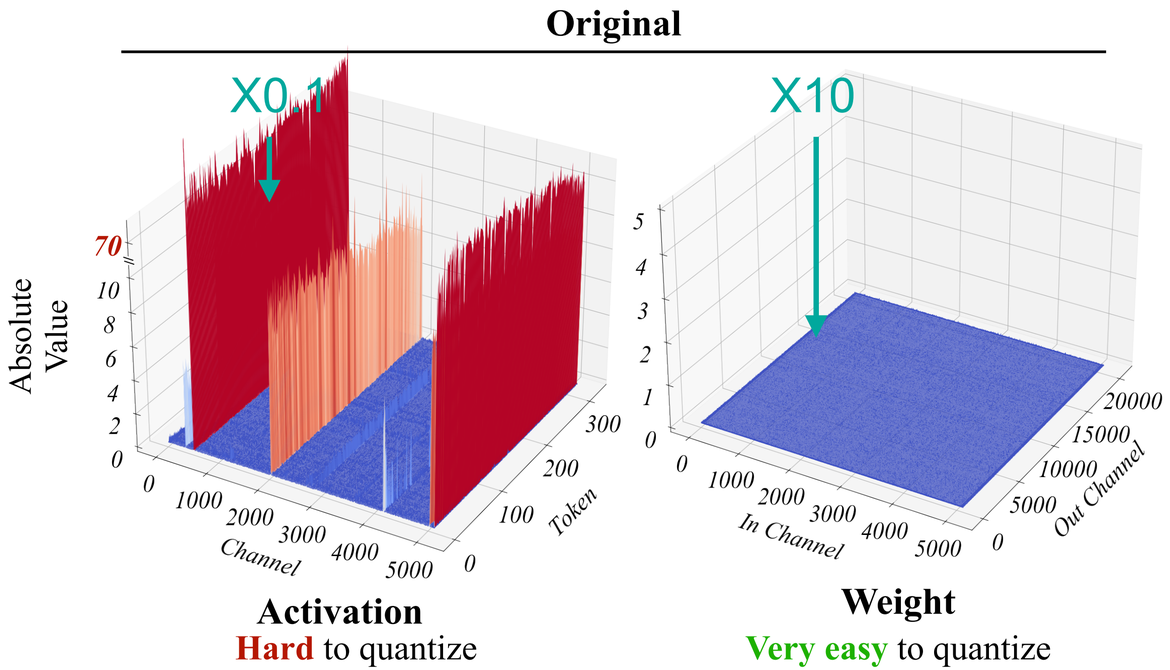
- W8A8 quantization has been an industrial standard for CNNs, but not LLM. Why?
- Systematic outliers emerge in **activations** when we scale up LLMs beyond 6.7B. Traditional CNN quantization methods will destroy the accuracy.

Understanding the Quantization Difficulty of LLMs



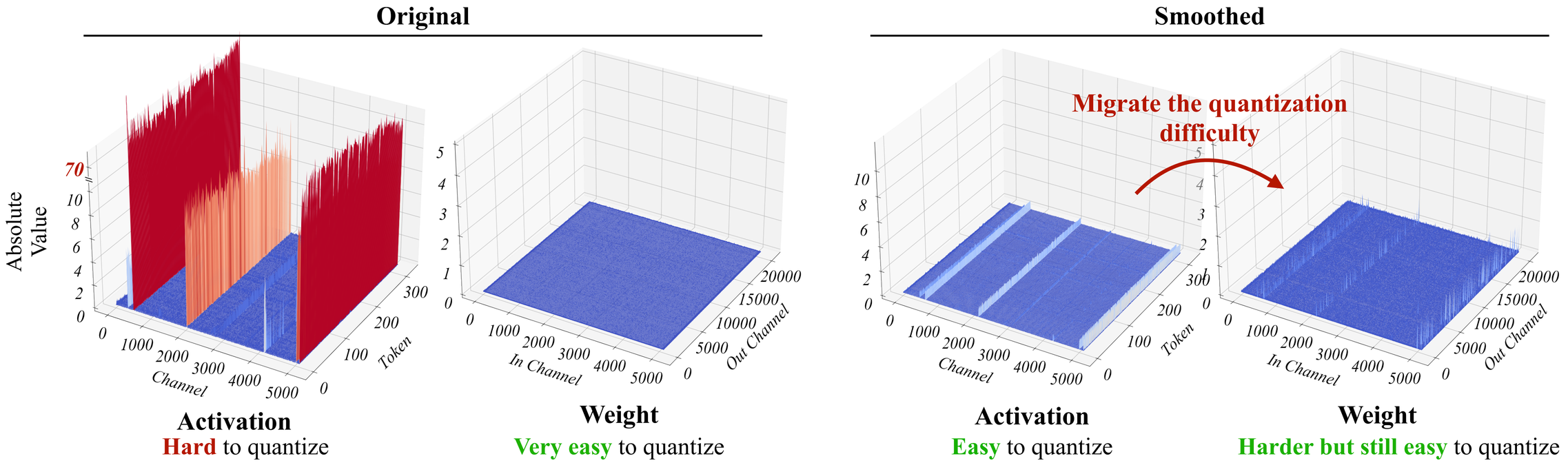
- Weights are easy to quantize, but activation is hard due to outliers

Understanding the Quantization Difficulty of LLMs



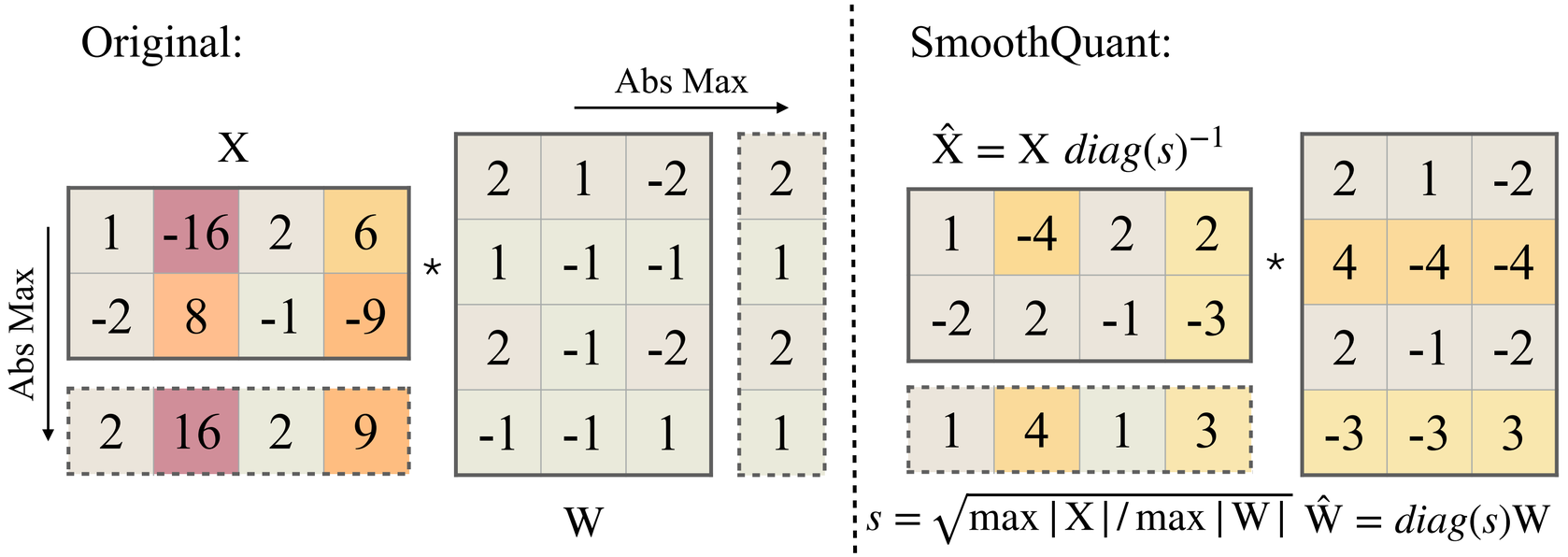
- Weights are easy to quantize, but activation is hard due to outliers
- Luckily, outliers persist in fixed channels

Understanding the Quantization Difficulty of LLMs



- Weights are easy to quantize, but activation is hard due to outliers
- Luckily, outliers persist in fixed channels
- Migrate the quantization difficulty from activation to weights, so both are easy to quantize

Activation Smoothing



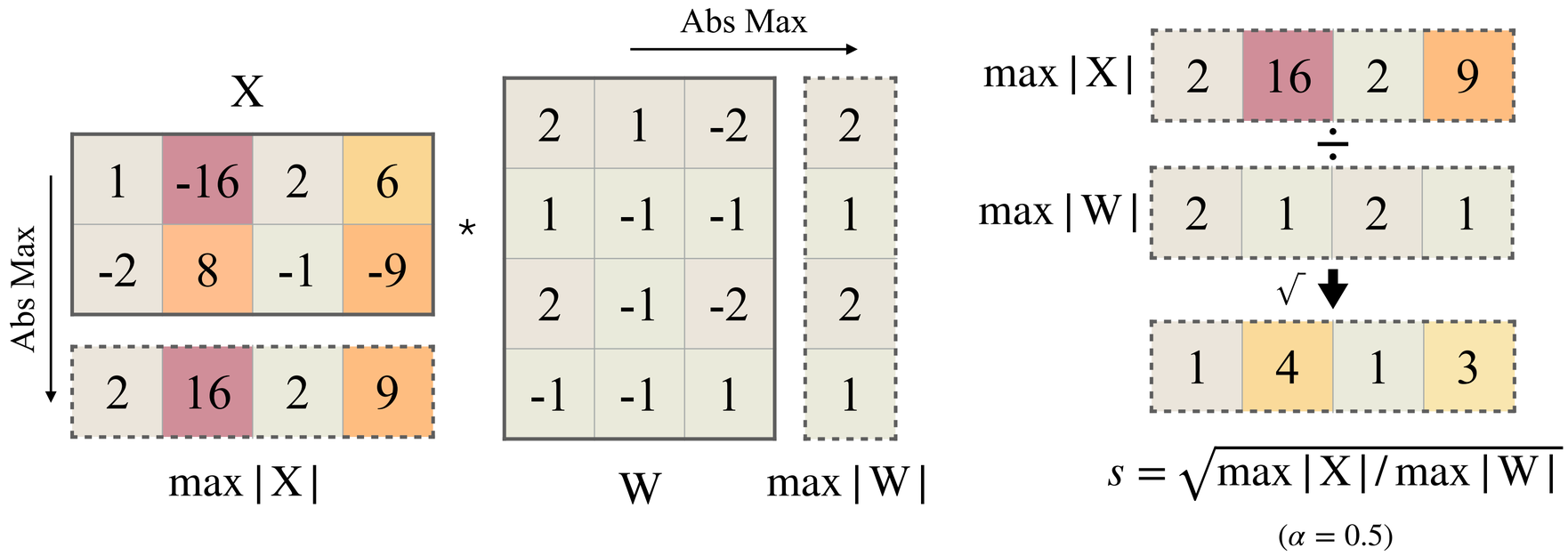
$$\mathbf{s}_j = \max(|\mathbf{X}_j|)^{\alpha} / \max(|\mathbf{W}_j|)^{1-\alpha},\ j = 1,2,\ldots,C_i$$

$$\mathbf{Y} = (\mathbf{X}diag(\mathbf{s})^{-1}) \cdot (diag(\mathbf{s})\mathbf{W}) = \hat{\mathbf{X}}\hat{\mathbf{W}}$$

α: Migration Strength

Activation Smoothing

1.Calibration Stage (Offline):

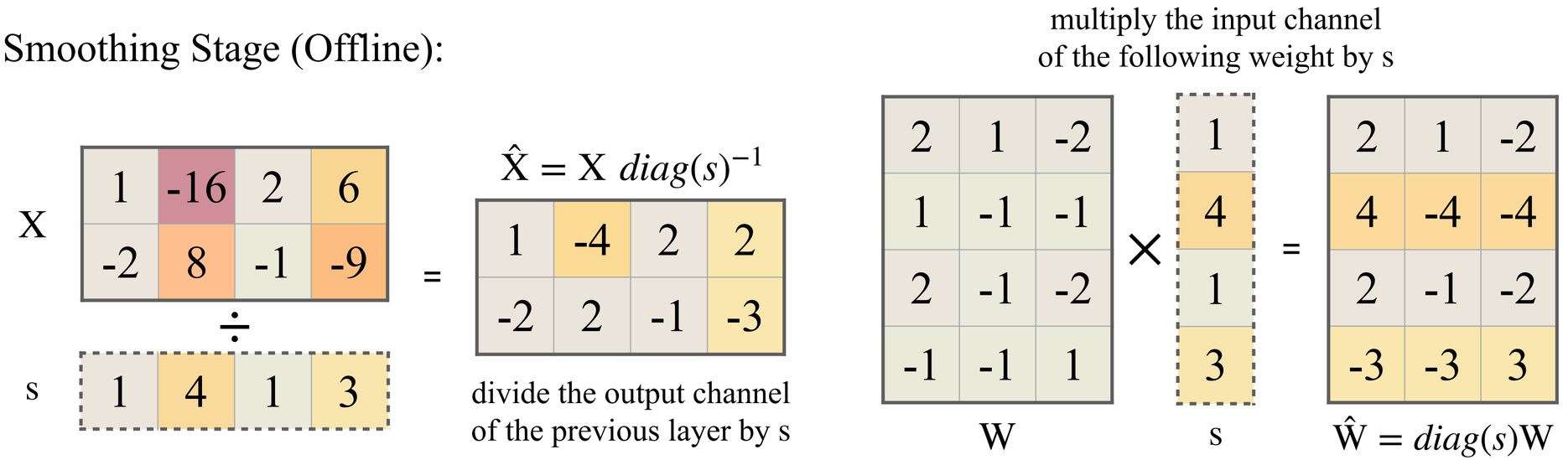


$$\mathbf{s}_j = \max(|\mathbf{X}_j|)^{\alpha} / \max(|\mathbf{W}_j|)^{1-\alpha},\ j = 1,2,\ldots,C_i$$

α: Migration Strength

Activation Smoothing

2. Smoothing Stage (Offline):



$$\mathbf{s}_j = \max(|\mathbf{X}_j|)^{\alpha} / \max(|\mathbf{W}_j|)^{1-\alpha},\ j = 1,2,\ldots,C_i$$

$$\mathbf{Y} = (\mathbf{X}diag(\mathbf{s})^{-1}) \cdot (diag(\mathbf{s})\mathbf{W}) = \hat{\mathbf{X}}\hat{\mathbf{W}}$$

α: Migration Strength

Activation Smoothing

3. Inference (deployed model):

$\hat{X}$

1	-4	2	2
-2	2	-1	-3

At runtime, the activations are smooth and easy to quantize

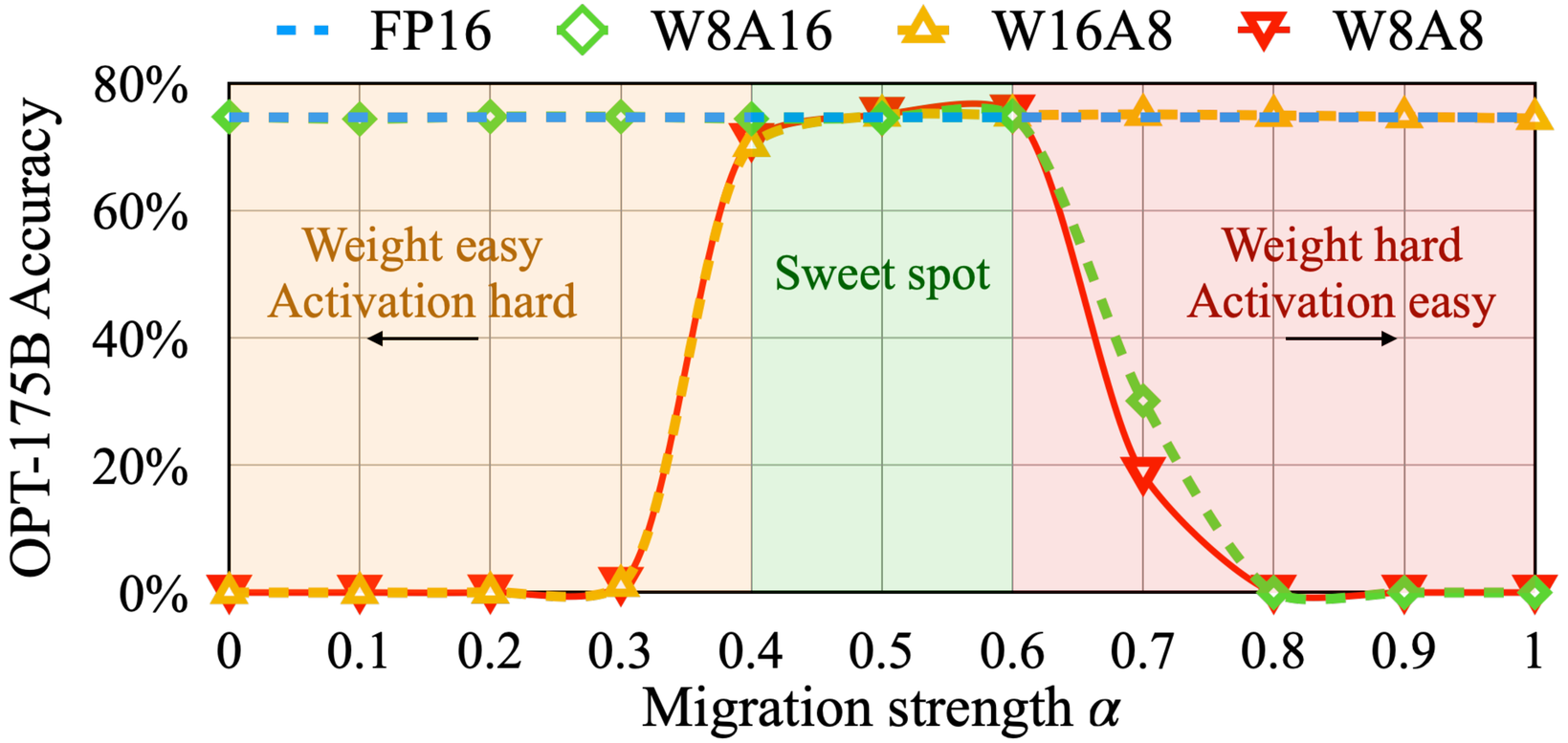
*

2	1	-2
4	-4	-4
2	-1	-2
-3	-3	3

$\hat{W}$

$$\mathbf{Y} = \hat{\mathbf{X}}\hat{\mathbf{W}}$$

Ablation Study on the Migration Strength α

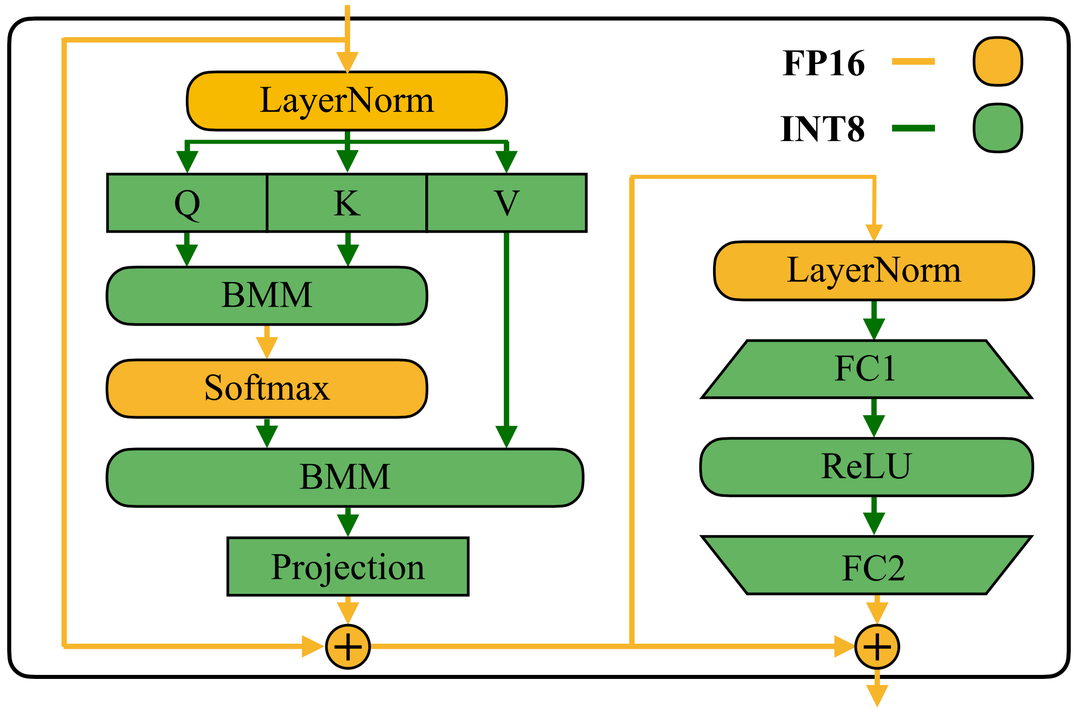


$$\mathbf{s}_j = \max(|\mathbf{X}_j|)^{\alpha} / \max(|\mathbf{W}_j|)^{1-\alpha},\ j = 1,2,\ldots,C_i \qquad \mathbf{Y} = (\mathbf{X}diag(\mathbf{s})^{-1}) \cdot (diag(\mathbf{s})\mathbf{W}) = \hat{\mathbf{X}}\hat{\mathbf{W}}$$

- Migration strength α controls the amount of quantization difficulty migrated from activations to weights.
- A suitable migration strength α (sweet spot) makes both activations and weights easy to quantize.
- If the α is too large, weights will be hard to quantize; if too small, activations will be hard to quantize.

System Implementation

- We integrate SmoothQuant into FasterTransformer
- All compute-intensive operators (Linear, BMM) are quantized

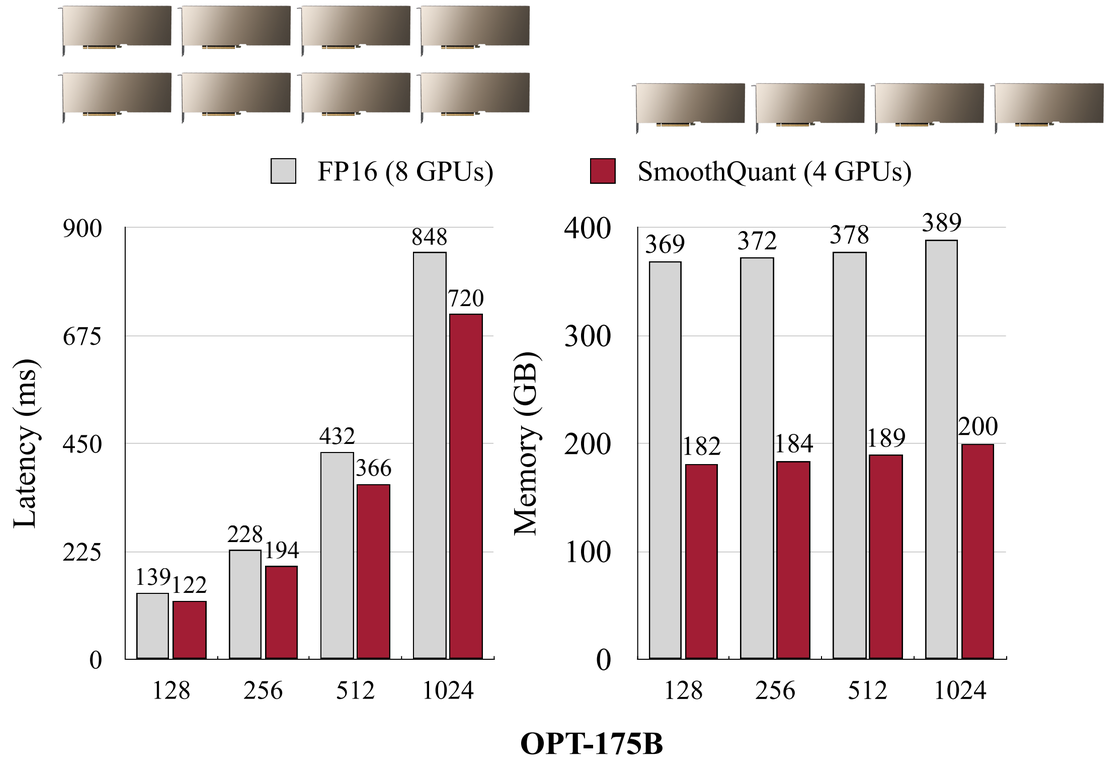


SmoothQuant is Accurate and Efficient

- SmoothQuant well maintains the accuracy without fine-tuning.
- SmoothQuant can both accelerate inference and halve the memory footprint.

LAMBADA Accuracy

	OPT-175B	BLOOM-176B	GLM-130B
FP16	71.6%	68.2%	73.8%
SmoothQuant	71.2%	68.3%	73.7%



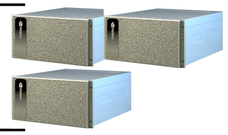
Scaling Up: 530B Model Within a Single Node

MT-NLG 530B Accuracy

	LAMBADA	HellaSwag	PIQA	WinoGrande	Average
FP16	76.6%	62.1%	81.0%	72.9%	73.1%
INT8	77.2%	60.4%	80.7%	74.1%	73.1%

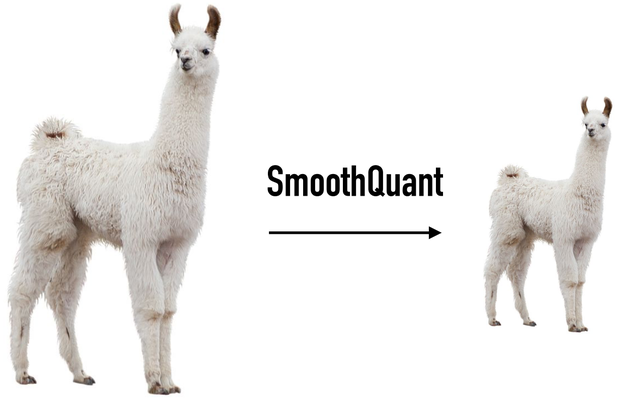
MT-NLG 530B Efficiency

SeqLen	Prec.	#GPUs	Latency	Memory
512	FP16	16	838ms	1068GB
	INT8	8	839ms	545GB
1024	FP16	16	1707ms	1095GB
	INT8	8	1689ms	570GB



- SmoothQuant can accurately quantize MT-NLG 530B model and reduce the serving GPU numbers by half at a similar latency, which allows serving the 530B model within a single node.

SmoothQuant



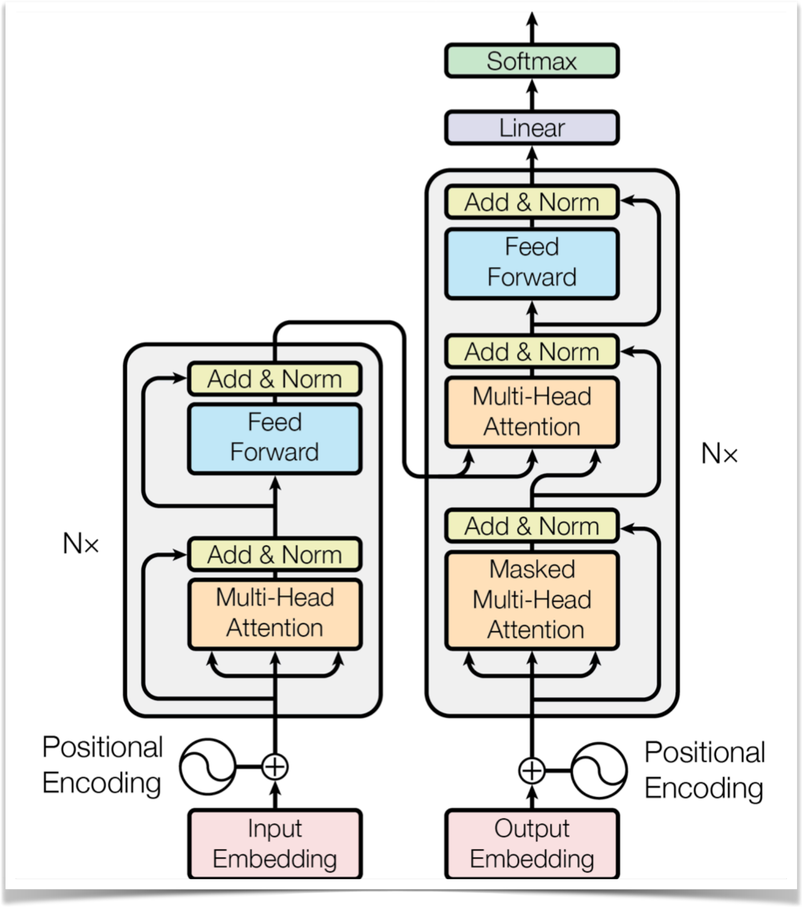
- **LLaMA** (and its variants like Alpaca) are popular open-source LLMs, which introduced SwishGLU, making activation quantization even harder
- SmoothQuant can losslessly quantize LLaMA families, further lowering the hardware barrier

Wikitext↓	LLaMA 7B	LLaMA 13B	LLaMA 30B	LLaMA 65B
FP16	11.51	10.05	7.53	6.17
SmoothQuant	11.56	10.08	7.56	6.20

Lecture Plan

Today, we will cover:

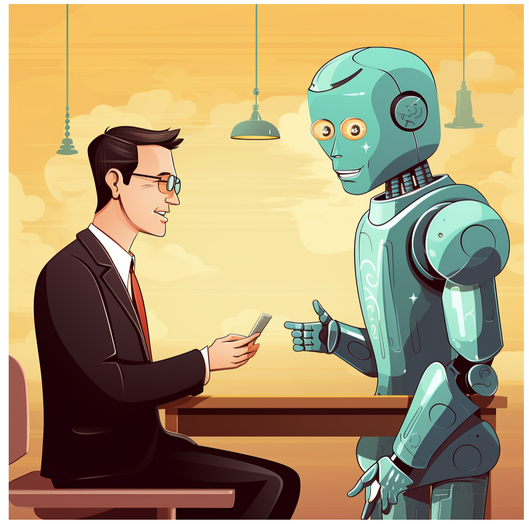
1. **Quantization**
    1. Weight-Activation Quantization: SmoothQuant
    2. Weight-Only Quantization: AWQ and TinyChat
    3. Further Practice: QServe (W4A8KV4)
2. **Pruning & Sparsity**
    1. Weight Sparsity: Wanda
    2. Contextual Sparsity: DejaVu, MoE
    3. Attention Sparsity: SpAtten, H_2O
3. **LLM Serving Systems**
    1. Metrics for LLM Serving
    2. Paged Attention (vLLM)
    3. FlashAttention
    4. Speculative Decoding
    5. Batching

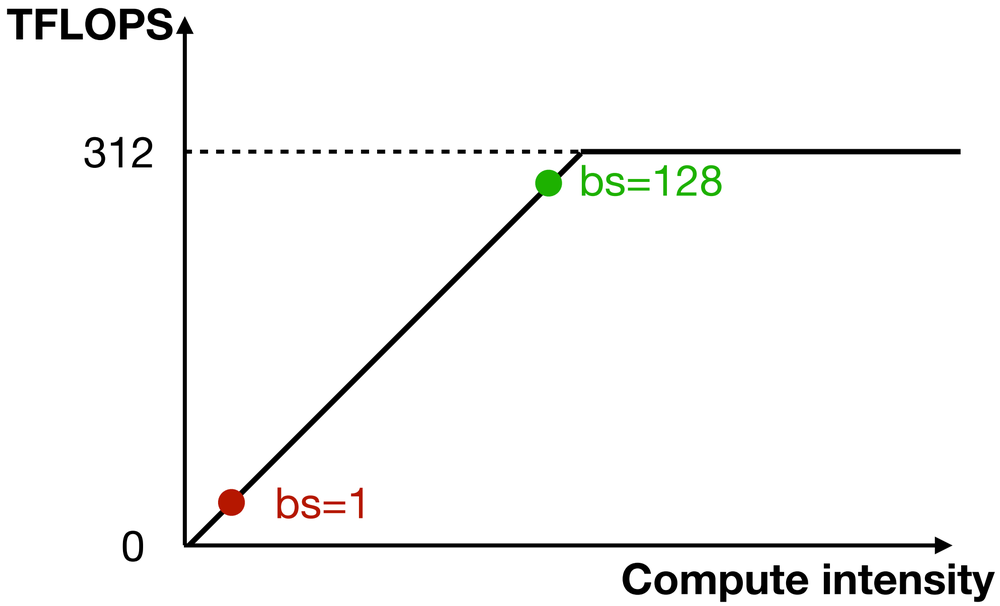


W4A16 for Single-batch serving

LLM decoding is highly memory-bounded; W8A8 is not enough

- W8A8 quantization is good for batch serving (e.g., batch size 128)
- But single-query LLM inference (e.g., local) is still highly memory-bounded
- We need **low-bit weight-only** quantization (e.g., W4A16) for this setting

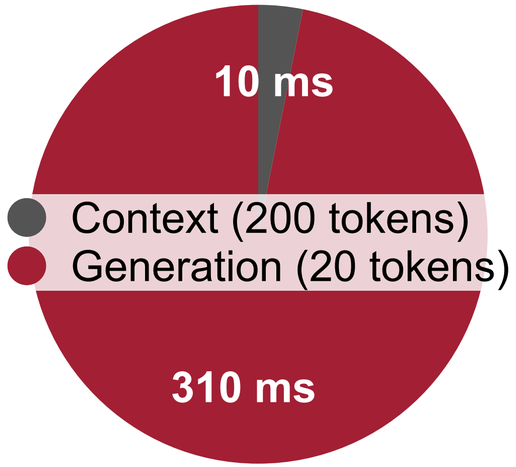




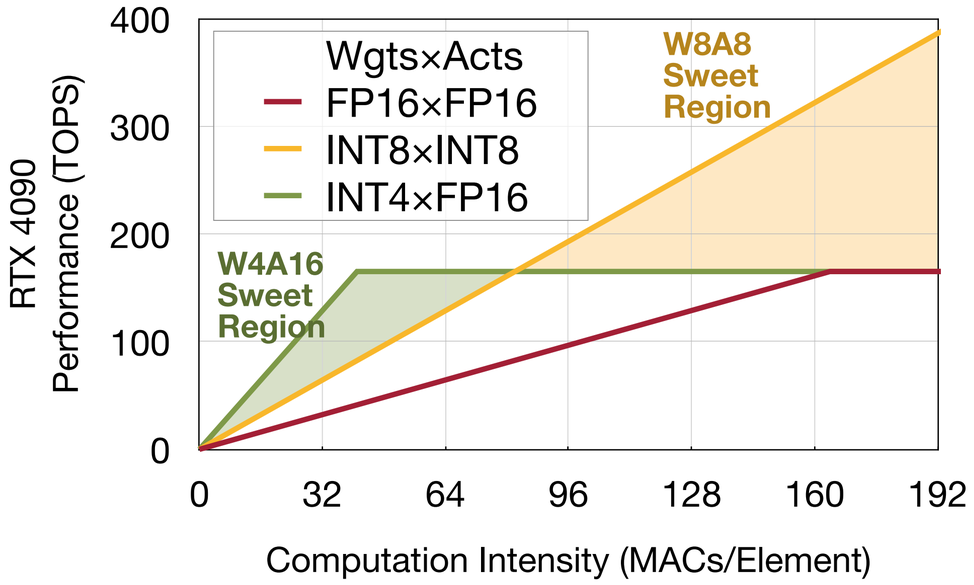
- A100 GPU
- LLaMA-65B decoding

Low-bit weight quantization brings speedup

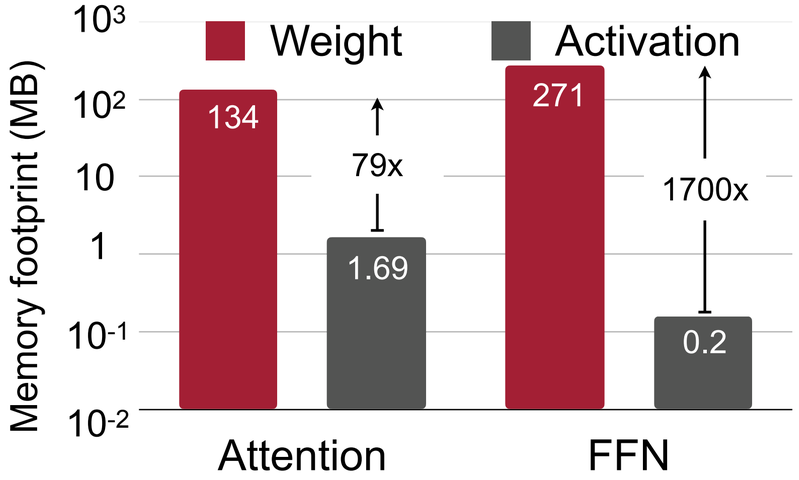
Weight loading is the efficiency bottleneck for edge LLM inference



(a) Generation stage is slower in low batch size settings



(b) Generation stage is bounded by memory bandwidth



(c) Weight loading is more expensive

AWQ: Activation-aware Weight Quantization

Targeting group-wise low-bit weight-only quantization (W4A16)

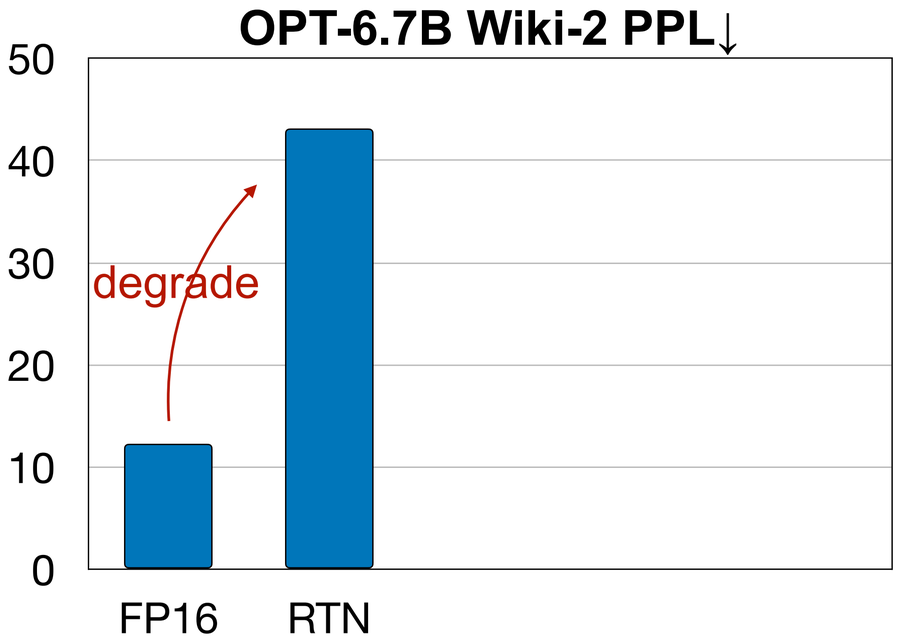
$\mathbf{W}_{\text{FP16}}$

+1.2	−0.2	−2.4	−3.4
−2.5	−3.5	+1.9	+1.4
−0.9	+1.6	−2.5	−1.9
−3.5	+1.5	+0.5	−0.1
+1.8	−1.6	−3.2	−3.4
+2.4	−3.5	−2.8	−3.9
+0.1	−3.8	+2.4	+3.4
+0.9	+3.3	−1.9	−2.3

RTN →

$Q(\mathbf{W})_{\text{INT3}}$

+1	+0	−2	−3
−3	−4	+2	+1
−1	+2	−3	−2
−4	+2	+1	+0
+2	−2	−3	−3
+2	−4	−3	−4
+0	−4	+2	+3
+1	+3	−2	−2



- Weight-only quantization reduces the memory requirement, and accelerates token generation by alleviating the memory bottleneck.
- Group-wise/block-wise quantization (e.g., 64/128/256) offers a better accuracy-model size trade-off.
- But there is still a performance gap with round-to-nearest (RTN) quantization (INT3-g128)

AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration [Lin et al., MLSys 2024]

AWQ: Activation-aware Weight Quantization

Observation: Weights are not equally important; 1% salient weights

$\mathbf{W}_{\text{FP16}}$

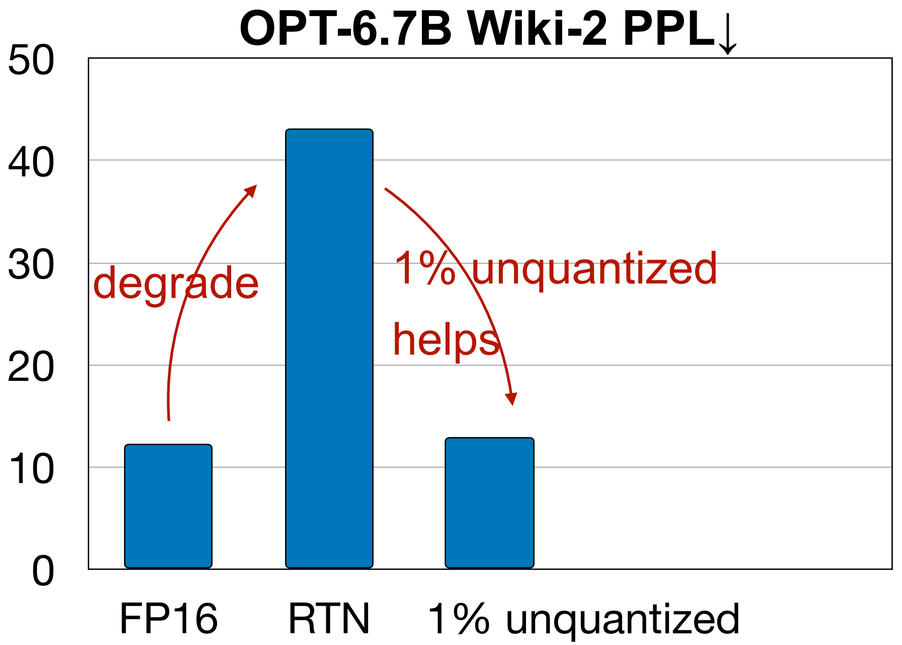
+1.2	−0.2	−2.4	−3.4
−2.5	−3.5	+1.9	+1.4
−0.9	+1.6	−2.5	−1.9
−3.5	+1.5	+0.5	−0.1
+1.8	−1.6	−3.2	−3.4
+2.4	−3.5	−2.8	−3.9
+0.1	−3.8	+2.4	+3.4
+0.9	+3.3	−1.9	−2.3

RTN →

$Q(\mathbf{W})_{\text{MixPrec}}$

+1	+0	−2	−3
−2.5	−3.5	+1.9	+1.4
−1	+2	−3	−2
−4	+2	+1	+0
+2	−2	−3	−3
+2	−4	−3	−4
+0	−4	+2	+3
+1	+3	−2	−2

Unquantized channel



- We find that weights are not equally important, keeping **only 1%** of salient weight channels unquantized can greatly improve perplexity

AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration [Lin et al., MLSys 2024]

AWQ: Activation-aware Weight Quantization

Observation: Weights are not equally important; 1% salient weights

$\mathbf{W}_{\text{FP16}}$

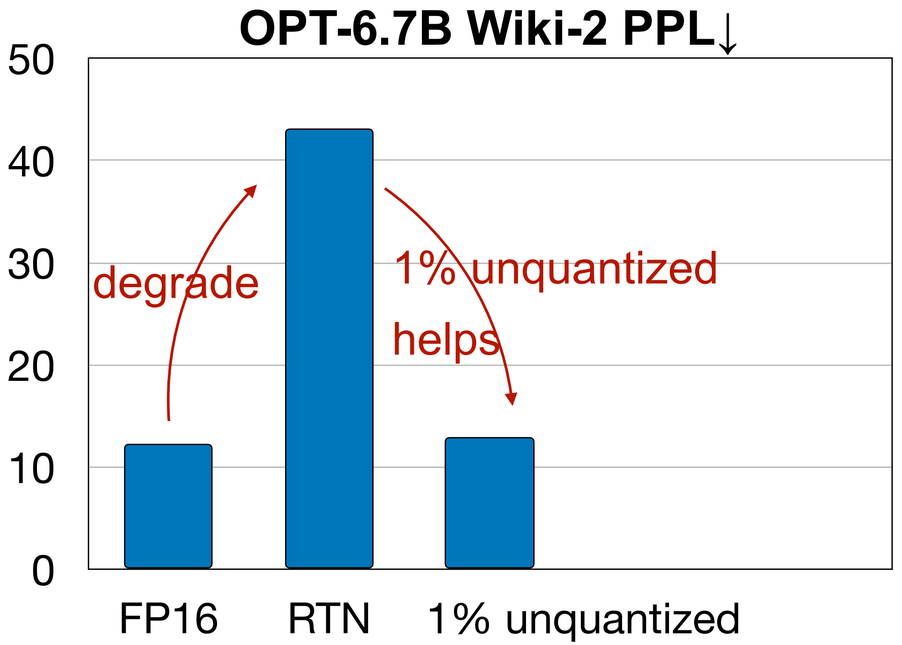
+1.2	−0.2	−2.4	−3.4
−2.5	−3.5	+1.9	+1.4
−0.9	+1.6	−2.5	−1.9
−3.5	+1.5	+0.5	−0.1
+1.8	−1.6	−3.2	−3.4
+2.4	−3.5	−2.8	−3.9
+0.1	−3.8	+2.4	+3.4
+0.9	+3.3	−1.9	−2.3

RTN →

$Q(\mathbf{W})_{\text{MixPrec}}$

+1	+0	−2	−3
−2.5	−3.5	+1.9	+1.4
−1	+2	−3	−2
−4	+2	+1	+0
+2	−2	−3	−3
+2	−4	−3	−4
+0	−4	+2	+3
+1	+3	−2	−2

Unquantized channel

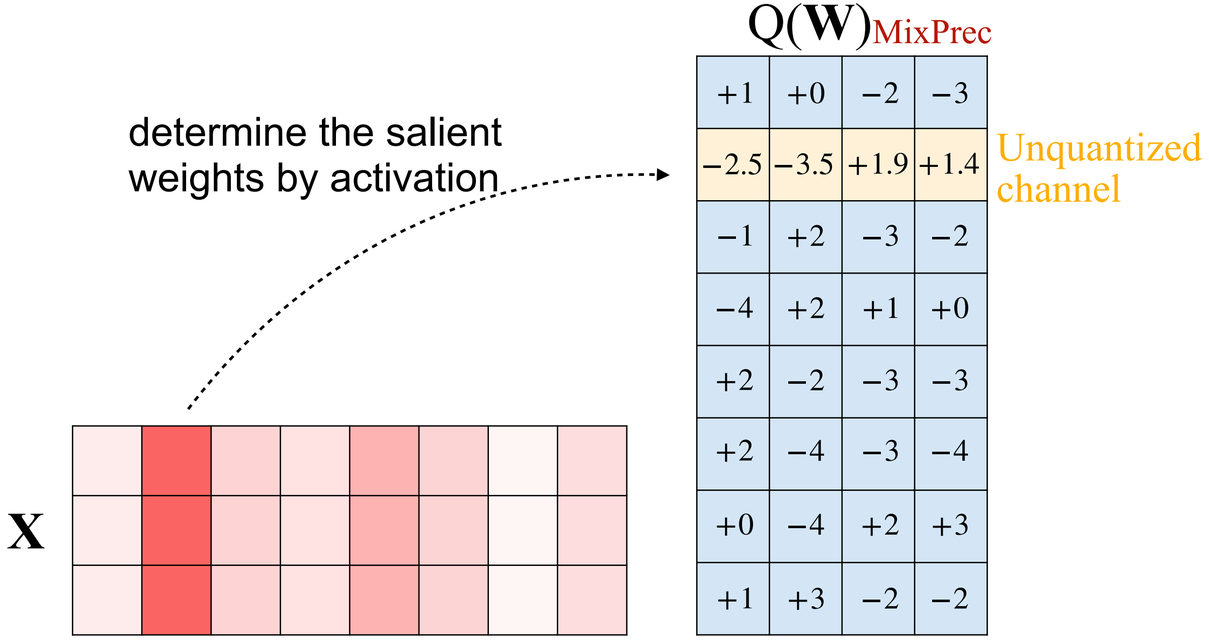


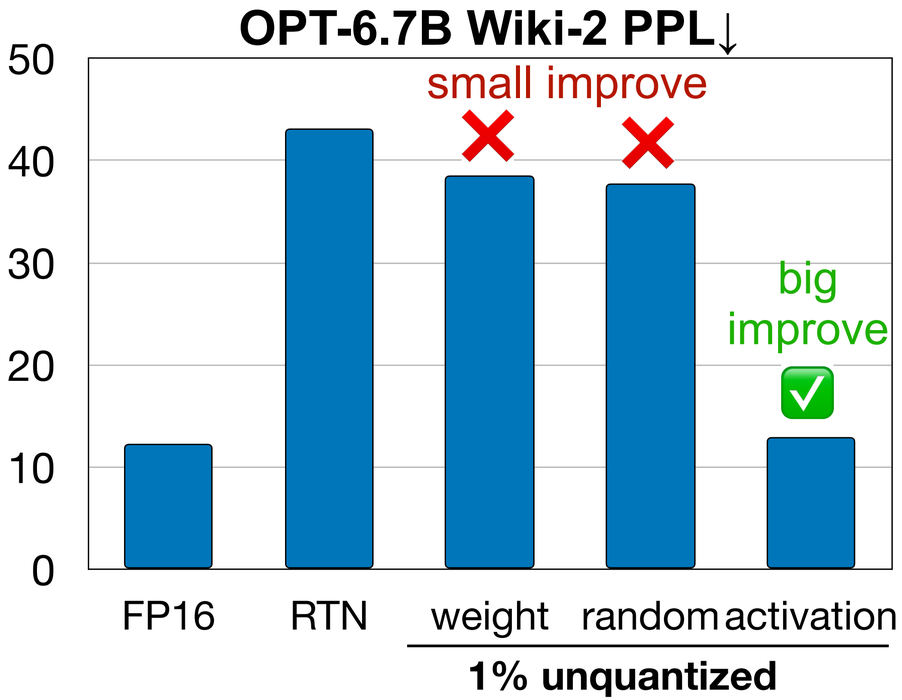
- We find that weights are not equally important, keeping **only 1%** of salient weight channels unquantized can greatly improve perplexity
- But how do we select salient channels? Should we select based on weight magnitude?

AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration [Lin et al., MLSys 2024]

AWQ for low-bit weight-only quantization

Salient weights are determined by activation distribution, not weight





- But how do we select salient channels? Should we select based on weight magnitude?
- No! We should look for **activation distribution, but not weight! (Activation has outliers!)**
- **However, is it really necessary to introduce mixed precision?**

AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration [Lin et al., MLSys 2024]

AWQ for low-bit weight-only quantization

Protecting salient weights by scaling (without mixed precision)

Quantization scaler, Absmax within the group

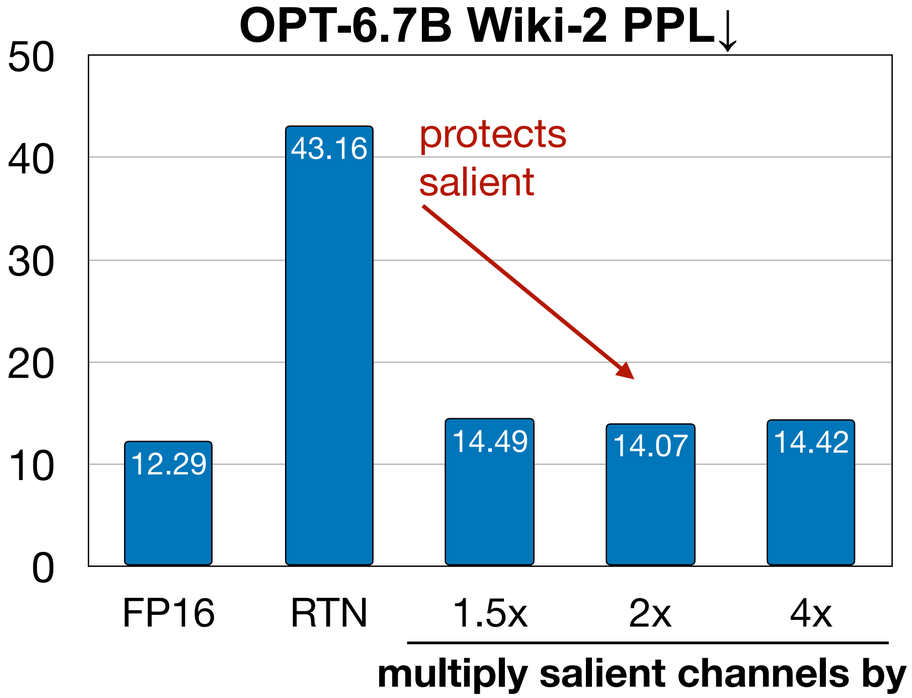
$$\text{Let} \quad Q(\mathbf{w}) = \Delta \cdot \text{round}(\frac{\mathbf{w}}{\Delta})$$

$$\Delta = \frac{\max(|\mathbf{w}|)}{2^{N-1}}$$

fuse to previous op

Quantize

$$\mathbf{WX} \longrightarrow Q(\mathbf{W} \cdot \mathbf{s})(\mathbf{s}^{-1} \cdot \mathbf{X})$$



- We need to consider **activation-awareness** for salient channels.

AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration [Lin et al., MLSys 2024]

Why AWQ reduces quantization error?

When scale s is not too large, quantization error is inversely proportional to s.

$$Q(w) \cdot x = \Delta \cdot \text{Round}(\frac{w}{\Delta}) \cdot x$$

$$Q(w \cdot s) \cdot \frac{x}{s} = \Delta' \cdot \text{Round}(\frac{w \cdot s}{\Delta'}) \cdot x \cdot \frac{1}{s}$$

$\rightarrow$

Quantization scaler, Absmax within the group

$$\text{Err}(Q(w) \cdot x) = \Delta \cdot \text{Err}(\text{Round}(\frac{w}{\Delta})) \cdot x$$

$$\approx$$

$$\text{Err}(Q(w \cdot s) \cdot \frac{x}{s}) = \Delta' \cdot \text{Err}(\text{Round}(\frac{w \cdot s}{\Delta'})) \cdot x \cdot \frac{1}{s}$$

A scalar with an expectation of 0.25

- We assume weights are quantized to signed INT4 numbers in this analysis.
- As long as AWQ scales **s** is not too large (maintaining $\Delta \approx \Delta'$, since scaling up a single channel will usually not change the global absmax), quantization error is inversely proportional to **s**.

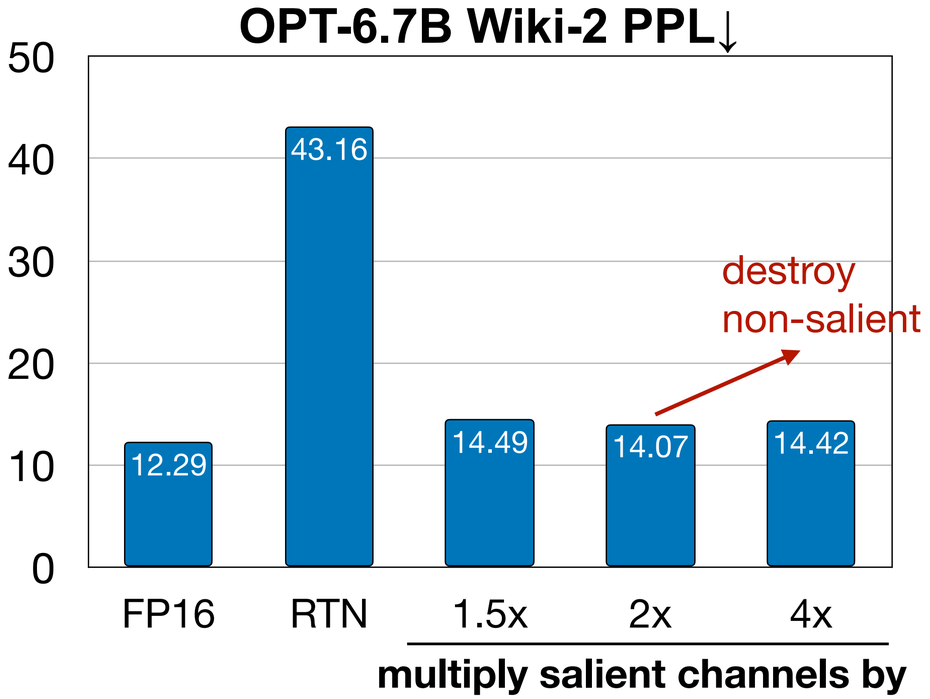
AWQ for low-bit weight-only quantization

Protecting salient weights by scaling (no mixed precision)

fuse to previous op

Quantize

$$\mathbf{WX} \longrightarrow Q(\mathbf{W} \cdot \mathbf{s})(\mathbf{s}^{-1} \cdot \mathbf{X})$$



- We need to consider **activation-awareness** for salient channels.
- Scaling up salient channels does not always bring about better performance.
- We need to search for the best scales.

AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration [Lin et al., MLSys 2024]

AWQ for low-bit weight-only quantization

Activation-aware optimal scaling searching

fuse to previous op

Quantize

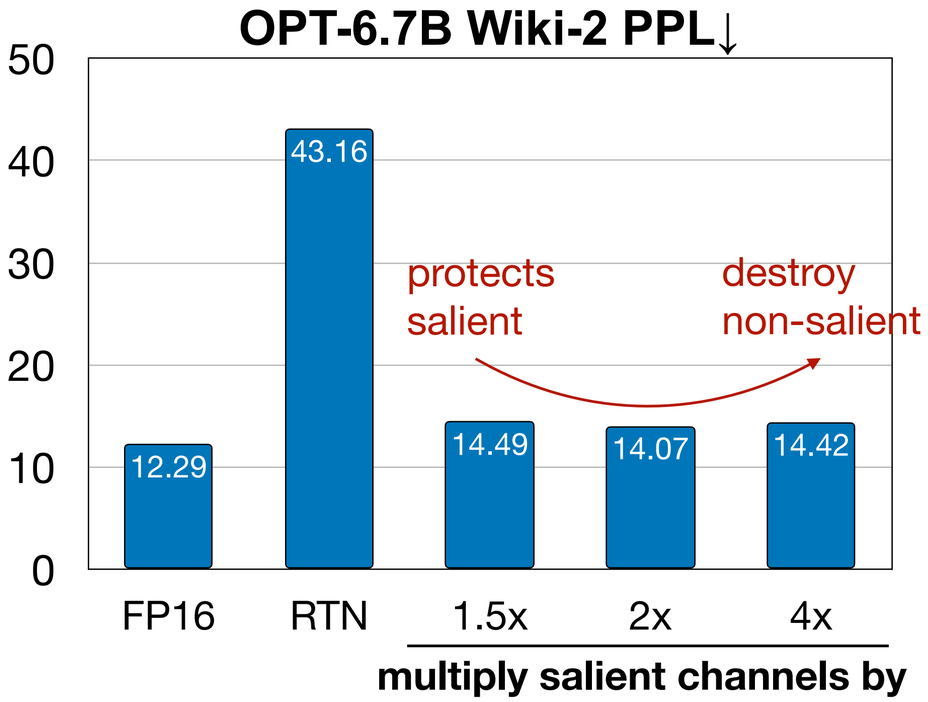
$$\mathbf{WX} \longrightarrow Q(\mathbf{W} \cdot \mathbf{s})(\mathbf{s}^{-1} \cdot \mathbf{X})$$

Activation diff, not weight diff

$$\mathscr{L}(\mathbf{s}) = \|Q(\mathbf{W} \cdot \mathbf{s})(\mathbf{s}^{-1} \cdot \mathbf{X}) - \mathbf{WX}\|$$

$$\mathbf{s} = \mathbf{s}_{\mathbf{X}}^{\alpha}, \alpha^* = \text{argmin}_{\alpha} \mathscr{L}(\mathbf{s}_{\mathbf{X}}^{\alpha})$$

avg. activation magnitude

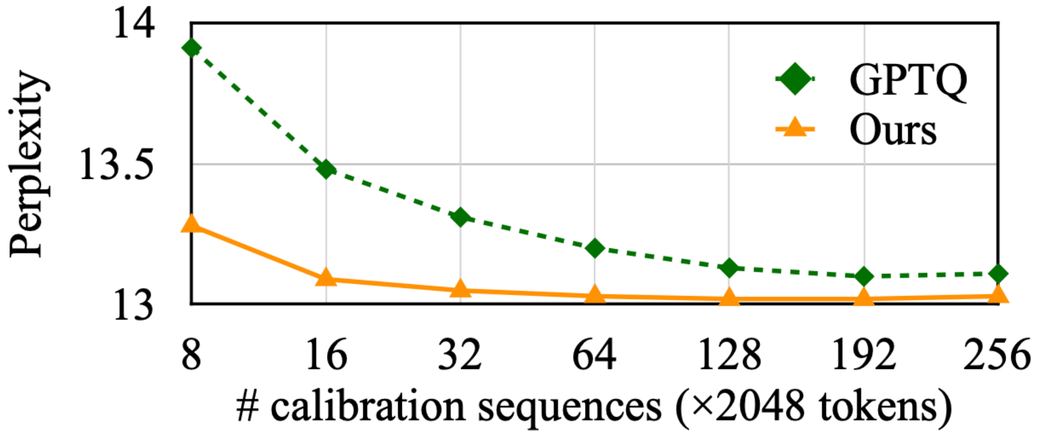


- Scale s is only determined by the average activation magnitude (**activation awareness**).
- The search objective is mean square root error for the **activation**, not the **weights** themselves (as in GPTQ).

AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration [Lin et al., MLSys 2024]

Advantages of AWQ

- Accurate, simple and easy to implement
- High hardware efficiency
- Less dependency on calibration set compared to regression-based method
  - Better data efficiency and distribution robustness
  - Generalize to instruction-tuned model and multi-modal LMs (different distributions)



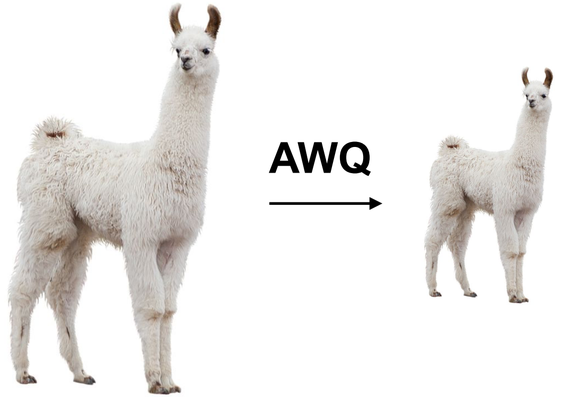
(a) Our method needs a smaller calibration set

Calib \ Eval	GPTQ		Ours	
	PubMed	Enron	PubMed	Enron
PubMed	32.48	50.41 +4.89	32.56	45.07 +0.50
Enron	+2.33 34.81	45.52	+0.60 33.16	44.57

(b) Our method is more robust to calibration set distribution

AWQ results

Improving general LLM quantization

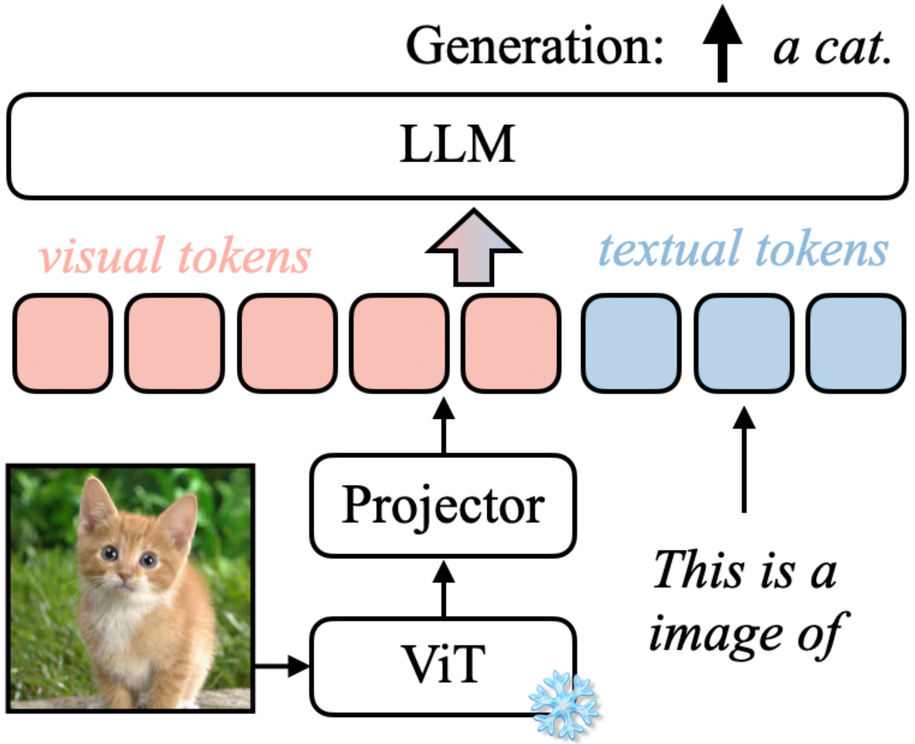


PPL↓		Llama-2			LLaMA			
		7B	13B	70B	7B	13B	30B	65B
FP16	-	5.47	4.88	3.32	5.68	5.09	4.10	3.53
INT3 g128	RTN	6.66	5.52	3.98	7.01	5.88	4.88	4.24
	GPTQ	6.43	5.48	3.88	8.81	5.66	4.88	4.17
	GPTQ-R	6.42	5.41	3.86	6.53	5.64	4.74	4.21
	AWQ	**6.24**	**5.32**	**3.74**	**6.35**	**5.52**	**4.61**	**3.95**
INT4 g128	RTN	5.73	4.98	3.46	5.96	5.25	4.23	3.67
	GPTQ	5.69	4.98	3.42	6.22	5.23	4.24	3.66
	GPTQ-R	5.63	4.99	3.43	5.83	5.20	4.22	3.66
	AWQ	**5.60**	**4.97**	**3.41**	**5.78**	**5.19**	**4.21**	**3.62**

AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration [Lin et al., MLSys 2024]

AWQ for visual language model

- Goal: multi-modal LLM with excellent *visual-language* performance and *text-only* performance
- We study PaLM-E alike architecture due to its flexibility for multi-modal input/output
- Augment LLM with tokenized visual input



VILA: On Pre-training for Visual Language Models [Lin et al., CVPR 2024]

AWQ for visual language model

AWQ well preserves the accuracy of GPT-4o-like multi-modal LLM

Q: Please tell me what happens in the video.

A: The video shows a soccer game where a player scores a goal. The crowd cheers as the player celebrates with his teammates.

Demo: https://vila.mit.edu

VILA: On Pre-training for Visual Language Models [Lin et al., CVPR 2024]

AWQ results for visual language models

Quantization of multi-modal LMs (VILA)

Model	Prec.	VQAv2	GQA	VizWiz	SQA-I	VQA-T	POPE	MME	MMB	MMB-CN	SEED	llava-bench	MMVet	Average
VILA-7B	FP16	80.3	63.1	59.6	68.0	62.6	86.3	1489.4	69.8	61.0	61.7	75.2	35.1	65.7
VILA-7B-AWQ	INT4	80.1	63.0	57.8	68.0	61.9	85.3	1486.3	68.8	59.0	61.3	75.8	35.9	65.2
VILA-13B	FP16	80.5	63.6	63.1	70.5	64.0	86.3	1553.6	73.8	66.7	62.8	78.3	42.6	68.4
VILA-13B-AWQ	INT4	80.4	63.6	63.0	71.2	63.5	87.0	1552.9	73.6	66.3	62.2	77.6	42.0	68.2

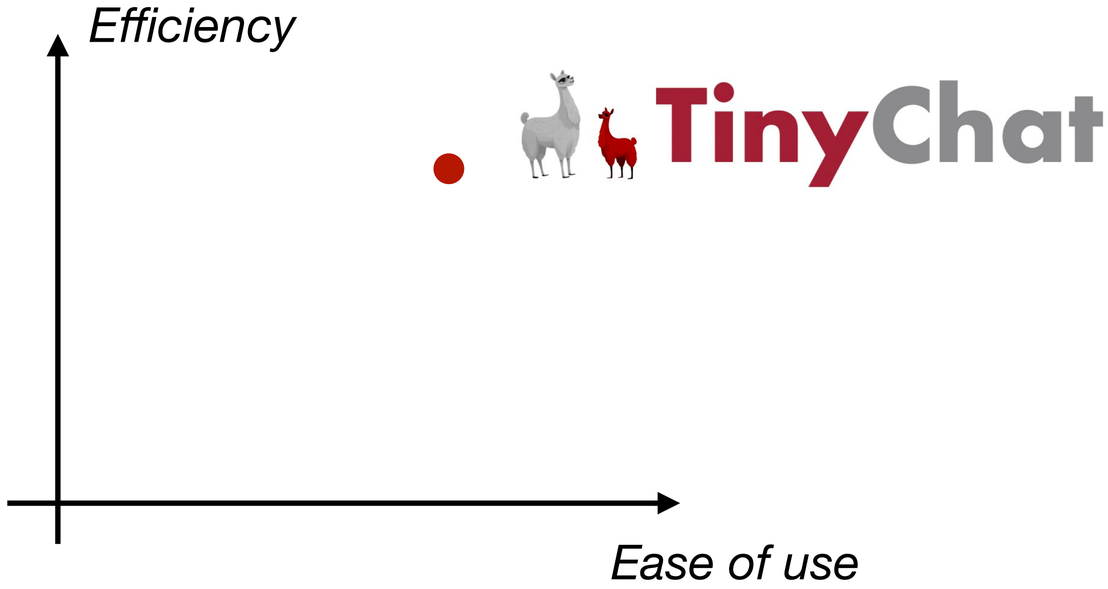
VILA: On Pre-training for Visual Language Models [Lin et al., CVPR 2024]

TinyChat : LLM Inference Engine on Edge

TinyChat: a lightweight LLM inference engine

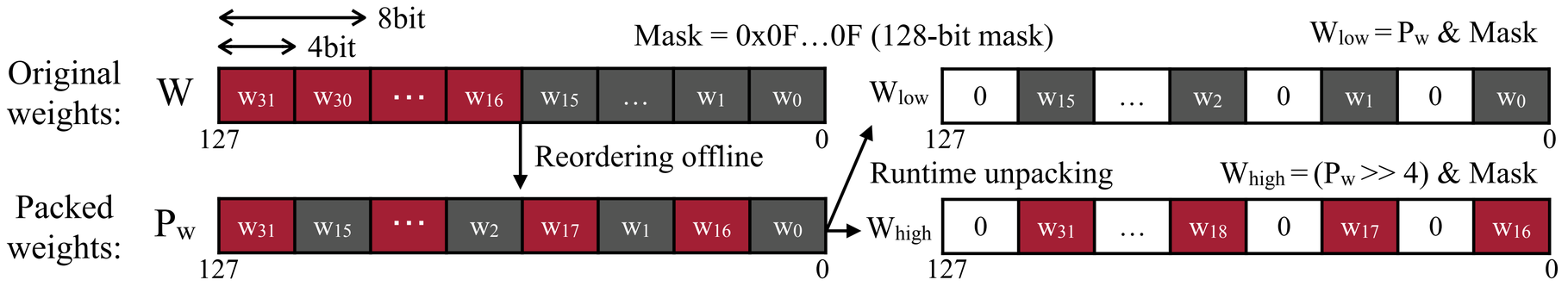
Pythonic, lightweight, efficient, multi-platform

- We need a framework to serve the quantized model to achieve low latency
- **TinyChat**: efficient, lightweight, Python-native (composable with other stacks like vLLM), multi-platform (cloud, desktop/laptop, edge GPUs; desktop/laptop, mobile CPUs)

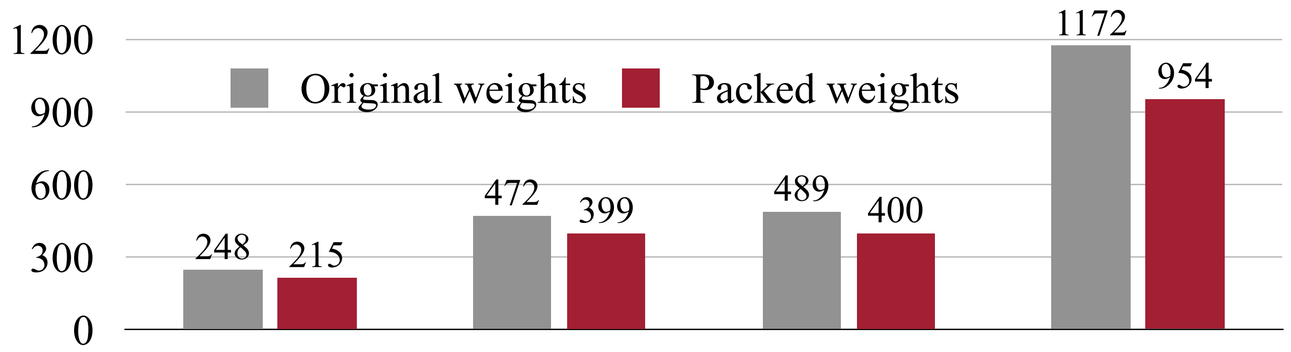


Hardware-aware packing

Reducing number of logical instructions to decode weights



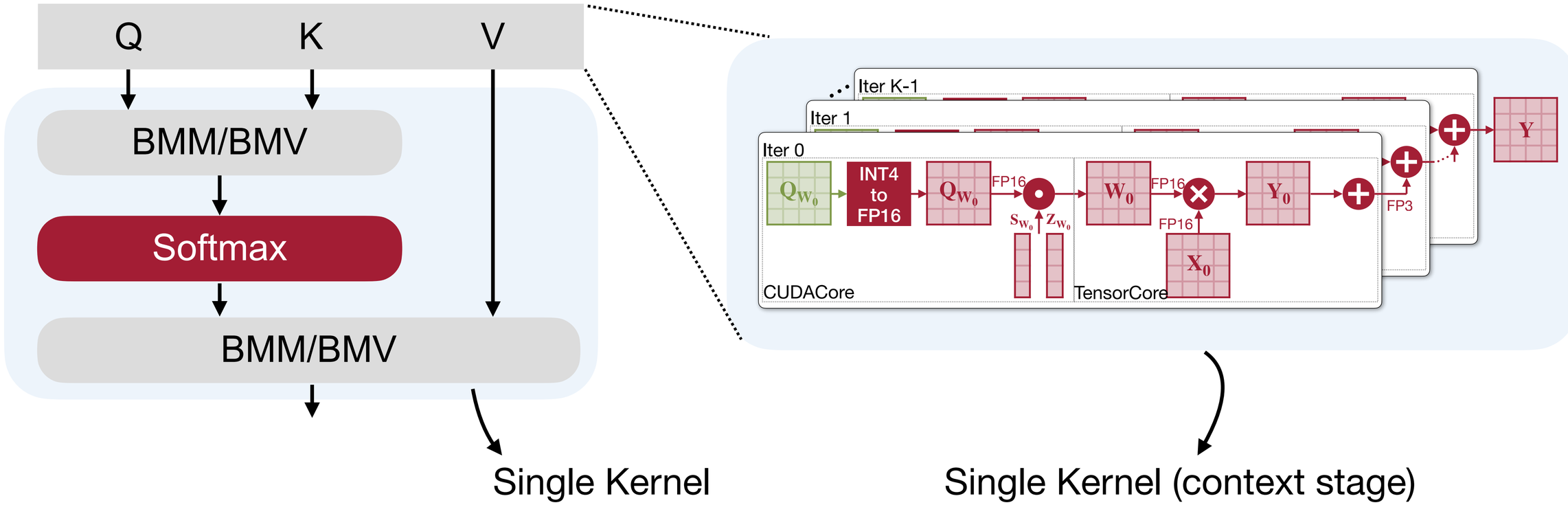
Decode 32 weights with only 3 logical operations



1.2x speedup on ARM NEON

Kernel fusion

Reduce intermediate DRAM access in memory bound ops



Single Kernel

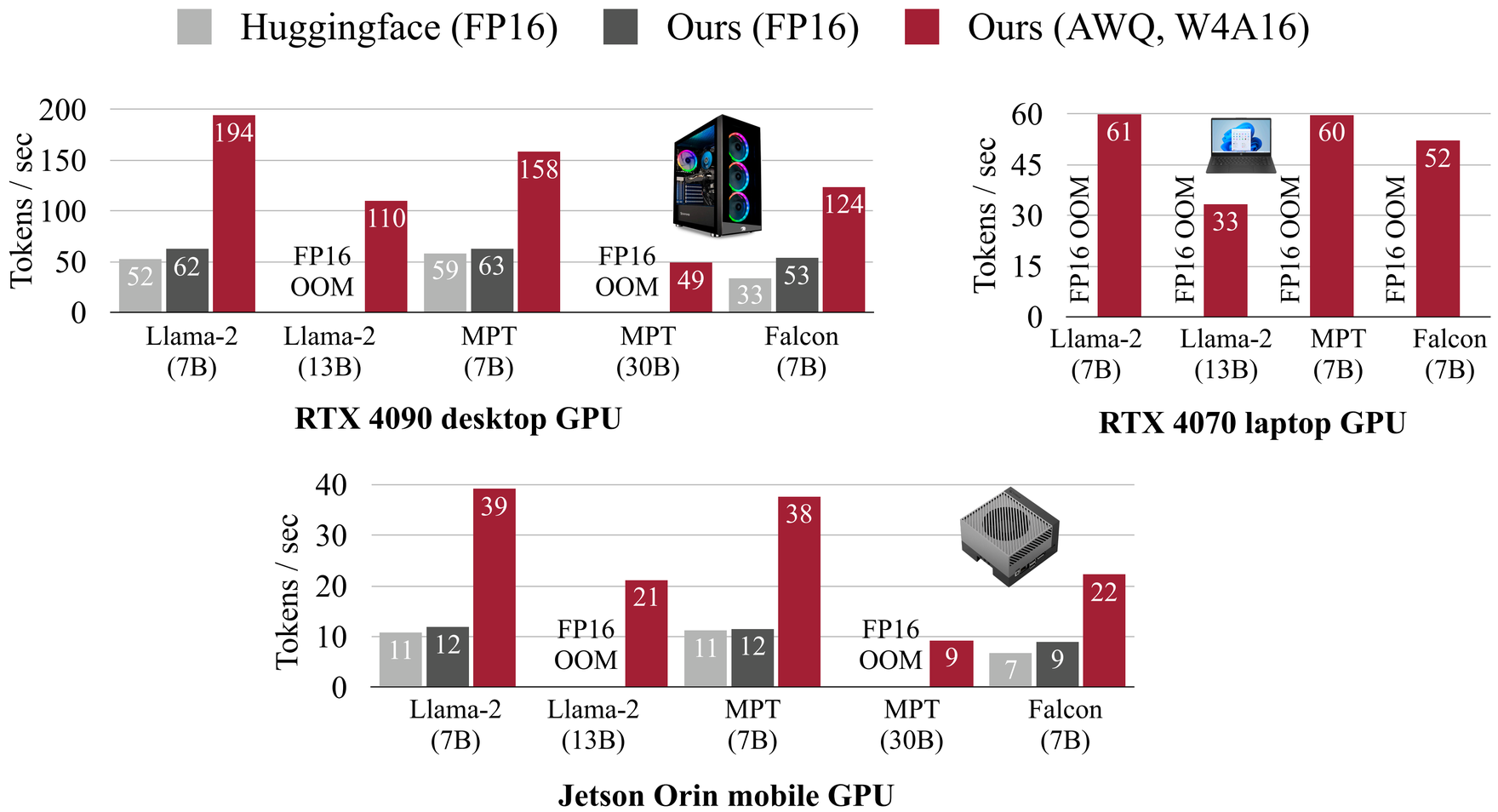
Fused kernel for attention

Single Kernel (context stage)

Fused kernel for GEMM

TinyChat significantly accelerates edge LLMs

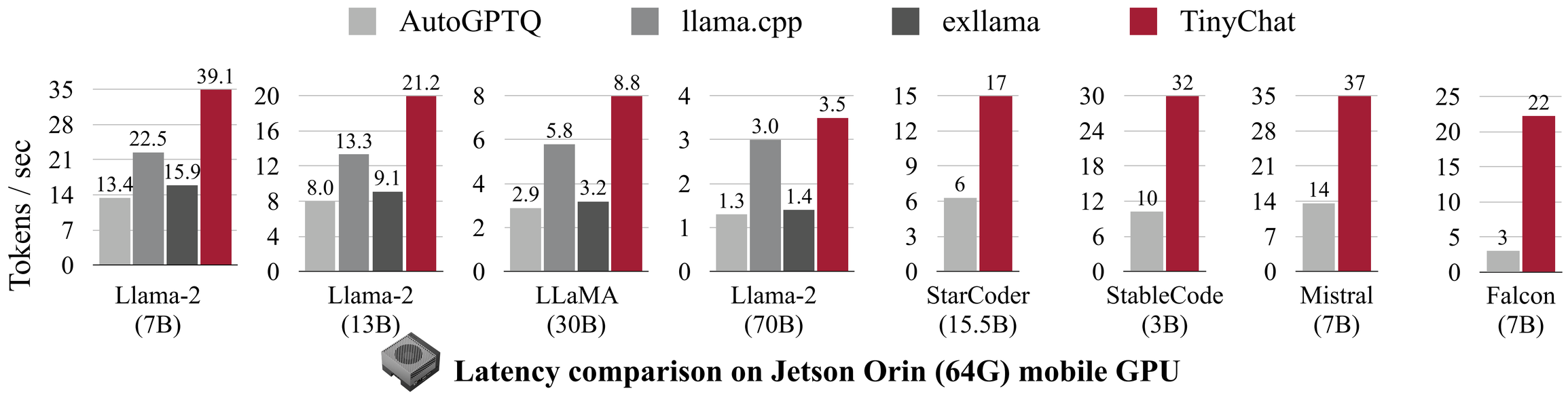
More than 3x faster than Huggingface FP16 LLM inference



AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration [Lin et al., MLSys 2024]

TinyChat outperforms other systems

At least 2.6x faster than AutoGPTQ and more flexible than llama.cpp, exllama



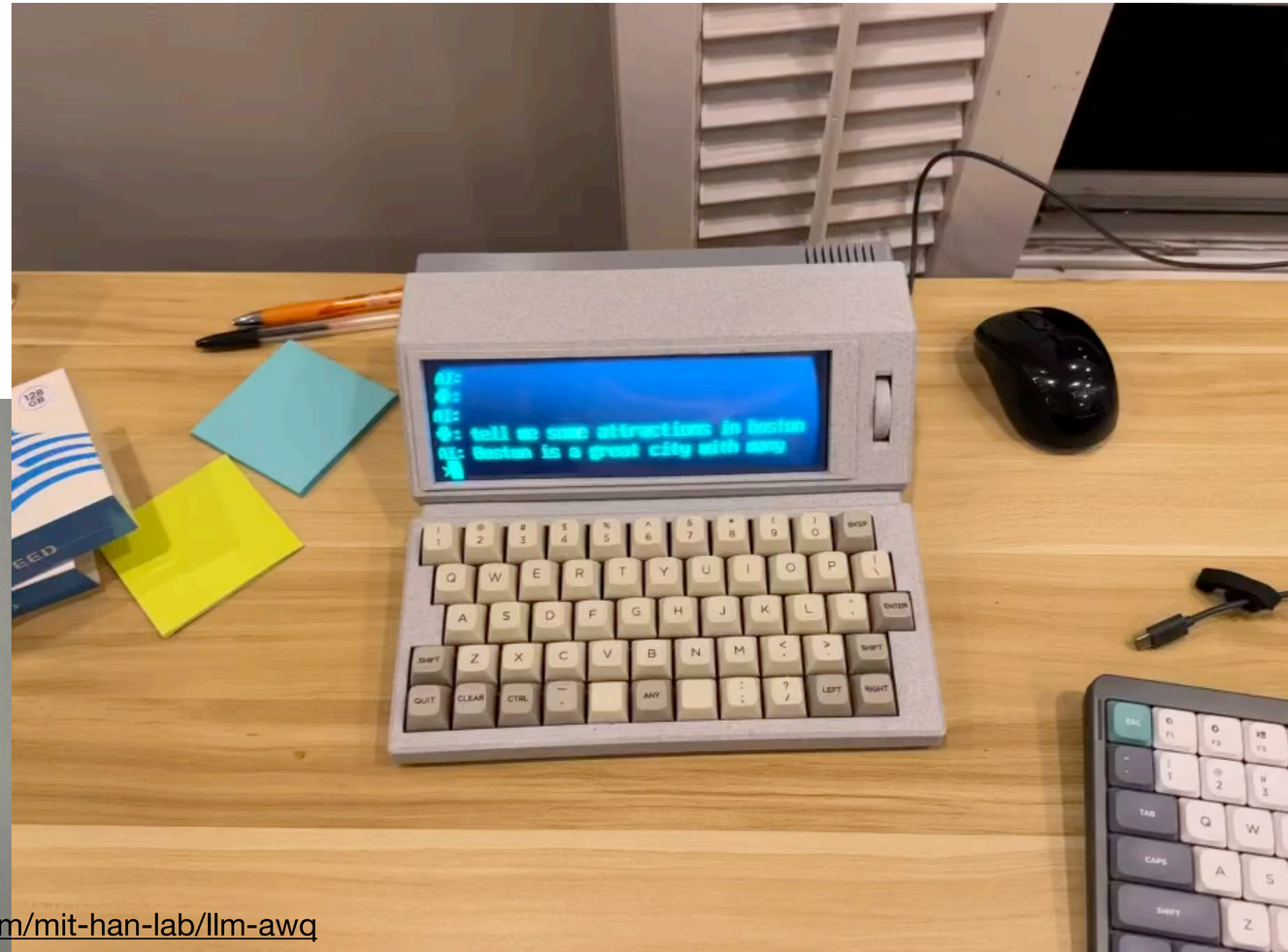
AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration [Lin et al., MLSys 2024]

TinyChat: a lightweight LLM inference engine

Demo on TinyChatComputer, powered by NVIDIA Jetson Orin Nano

- On a GPU board with just ~7G available memory, TinyChat enables efficient deployment of 7B large language models, thanks to AWQ quantization.

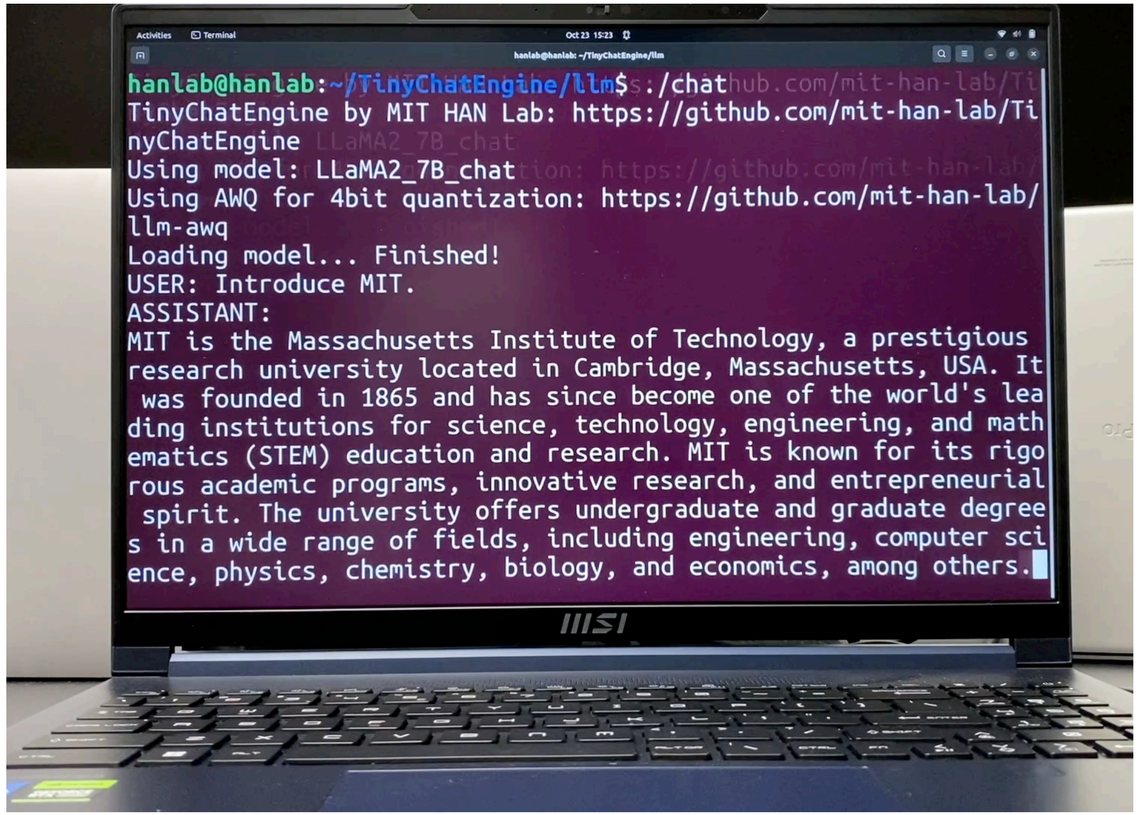
- Importance of edge LLM deployment: **data privacy and security**, **continued operation in disconnected environment**, **customization and personalization**.



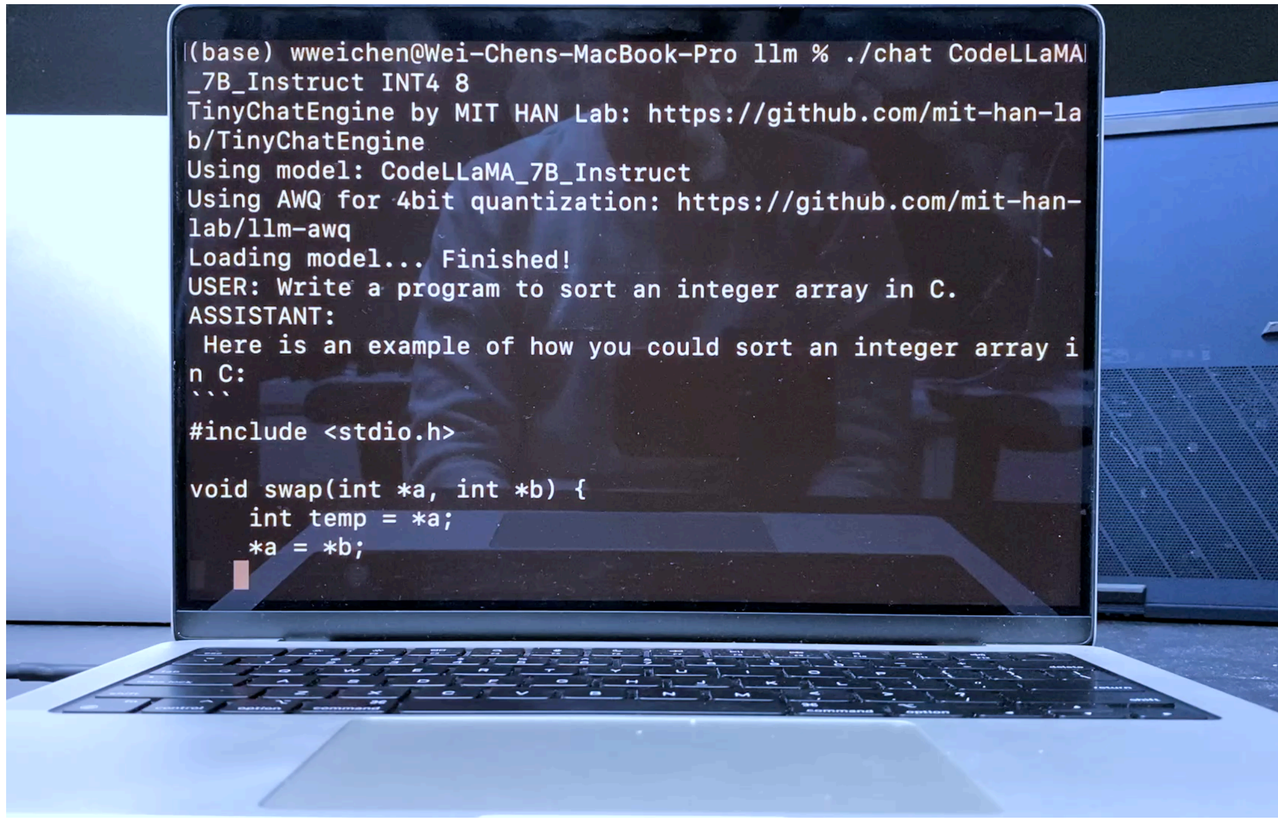
https://github.com/mit-han-lab/llm-awq

TinyChat: a lightweight LLM inference engine

TinyChat seamlessly supports personal laptops with Intel / ARM CPUs



MSI Laptop (RTX 4070 GPU)



Macbook (ARM CPU)

AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration [Lin et al., MLSys 2024]

TinyChat seamlessly supports VLMs

Accelerating visual-language models across different GPU platforms

- TinyChat also seamlessly supports VILA, delivering ~3x speedup over FP16 on Orin and allows interactive VLM deployment on the edge (laptops and AIoT).

Model	Precision	A100 Tok/sec	4090 Tok / sec	Orin Tok / sec
Llama-3-VILA1.5-8B	FP16	74.9	57.4	10.2
Llama-3-VILA1.5-8B-AWQ	INT4	**168.9**	**150.2**	**28.7**
VILA1.5-13B	FP16	50.0	OOM	6.1
VILA1.5-13B-AWQ	INT4	**115.9**	**105.7**	**20.6**

AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration [Lin et al., MLSys 2024]

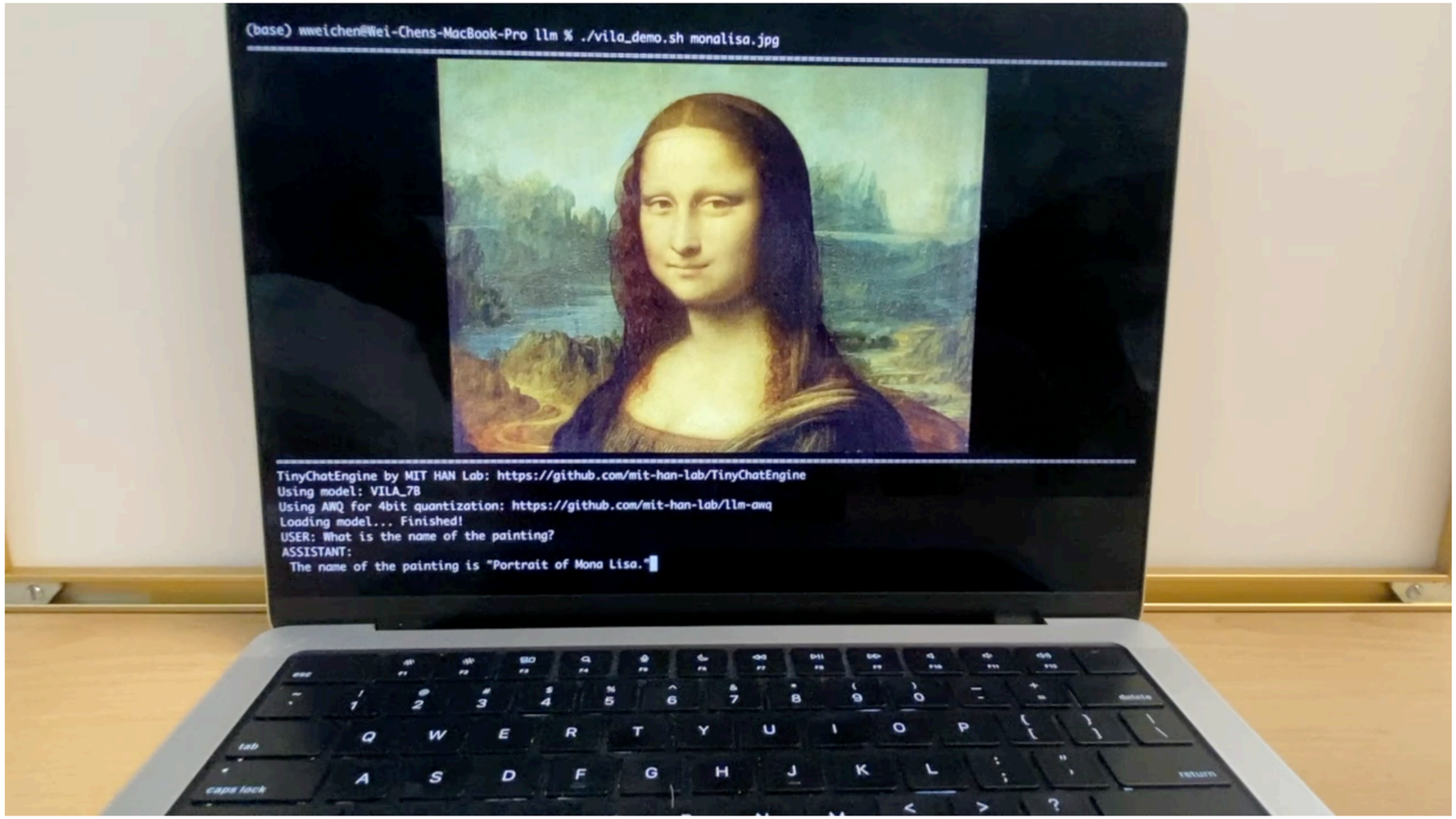
TinyChat for visual language model

Efficient image reasoning on Jetson Orin: TinyChat w/ VILA model family

VILA: On Pre-training for Visual Language Models [Lin et al., CVPR 2024]

TinyChat : a lightweight LLM inference engine

Run visual language models on personal laptops



VILA-7B + AWQ: Running on MacBook Arm CPU

VILA: On Pre-training for Visual Language Models [Lin et al., CVPR 2024]

Community impact of AWQ

AWQ quantized models are downloaded 1 million times on HuggingFace

TensorRT-LLM

https://github.com/NVIDIA/TensorRT-LLM#key-features

Transformer Quantization API

https://huggingface.co/docs/transformers/main_classes/quantization

Granite

IBM's internal code model, Granite, utilizes AWQ for quantization.

Lmdeploy

https://github.com/InternLM/lmdeploy/blob/main/lmdeploy/lite/quantization/awq.py

vLLM

https://github.com/vllm-project/vllm/blob/main/vllm/model_executor/layers/quantization/awq.py

Vertex AI

https://console.cloud.google.com/vertex-ai/publishers/google/model-garden

lm-sys/FastChat

https://github.com/lm-sys/FastChat/blob/main/docs/awq.md

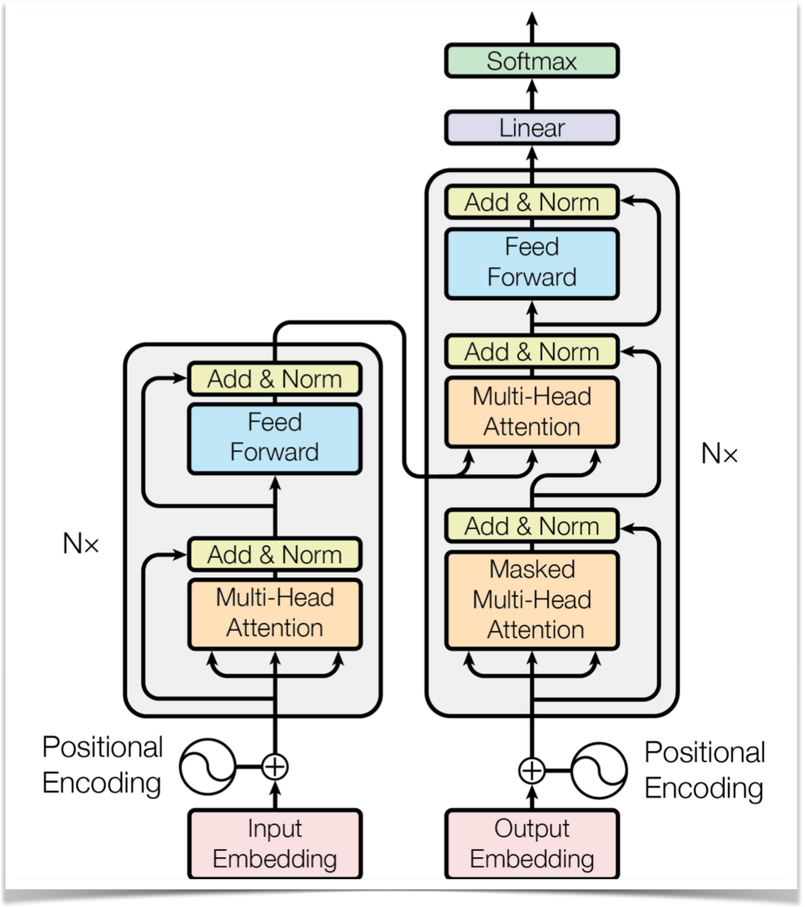
https://github.com/intel/neural-compressor/blob/master/docs/source/smooth_quant.md

AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration [Lin et al., MLSys 2024]

Lecture Plan

Today, we will cover:

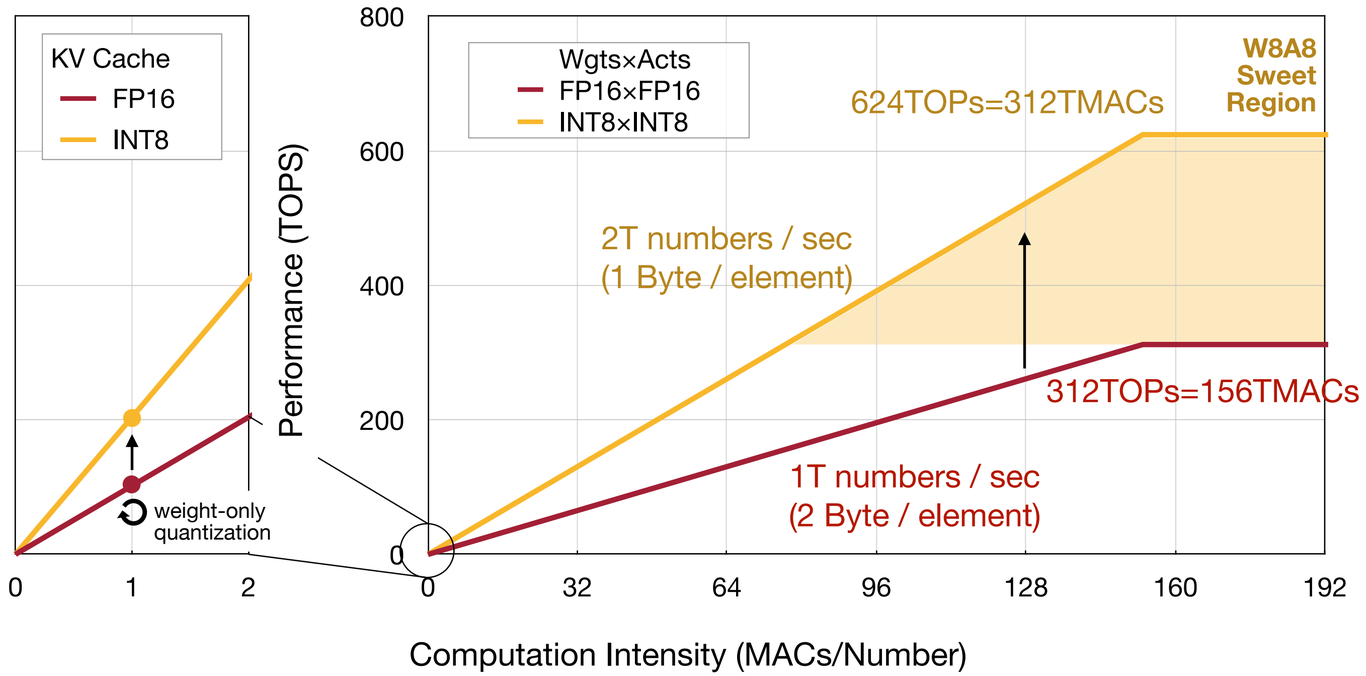
1. **Quantization**
    1. Weight-Activation Quantization: SmoothQuant
    2. Weight-Only Quantization: AWQ and TinyChat
    3. Further Practice: QServe (W4A8KV4)
2. **Pruning & Sparsity**
    1. Weight Sparsity: Wanda
    2. Contextual Sparsity: DejaVu, MoE
    3. Attention Sparsity: SpAtten, H_2O
3. **LLM Serving Systems**
    1. Metrics for LLM Serving
    2. Paged Attention (vLLM)
    3. FlashAttention
    4. Speculative Decoding
    5. Batching



Inconsistency b/w Cloud and Edge Inference

W8-A8-KV8 for cloud serving while W4-A16-KV16 for edge inference.

- For cloud serving, the most popular quantization setting is W8-A8-KV8
  - Weights: 8-bit Per-Channel Quantization
  - Activations: 8-bit Per-Token Quantization
  - KV: 8-bit Per-Tensor Quantization

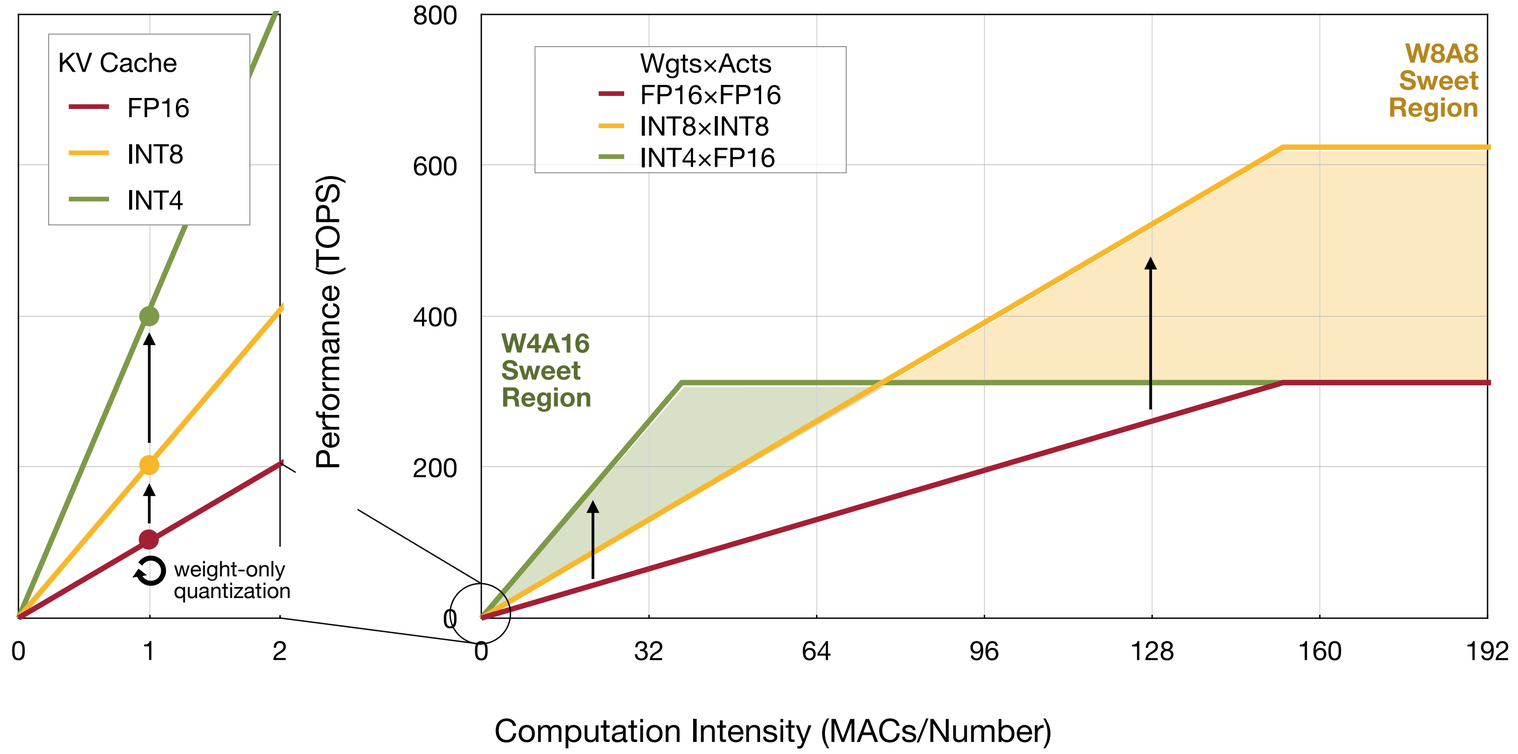


(i.e., batch size for GEMM, 1 for ATTN in decoding, sequence length for ATTN in prefilling)

Inconsistency b/w Cloud and Edge Inference

W8-A8-KV8 for cloud serving while W4-A16-KV16 for edge inference.

- For edge inference, the most popular quantization setting is W4-A16-KV16
  - Weights: 4-bit Per-Group Quantization ($C_g = 128$)
  - Activations: keeps FP16
  - KV: keeps FP16

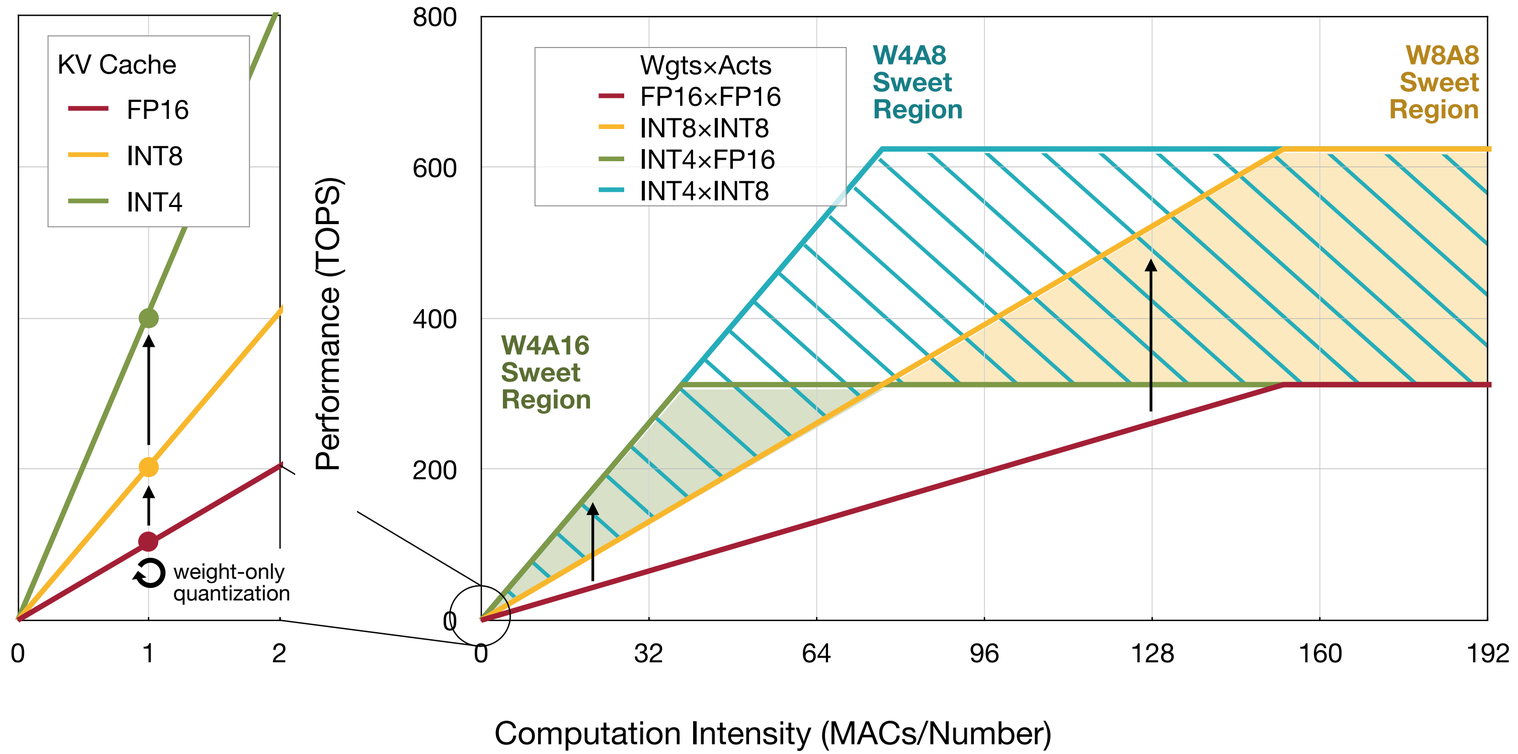


(i.e., batch size for GEMM, 1 for ATTN in decoding, sequence length for ATTN in prefilling)

Inconsistency b/w Cloud and Edge Inference

W8-A8-KV8 for cloud serving while W4-A16-KV16 for edge inference.

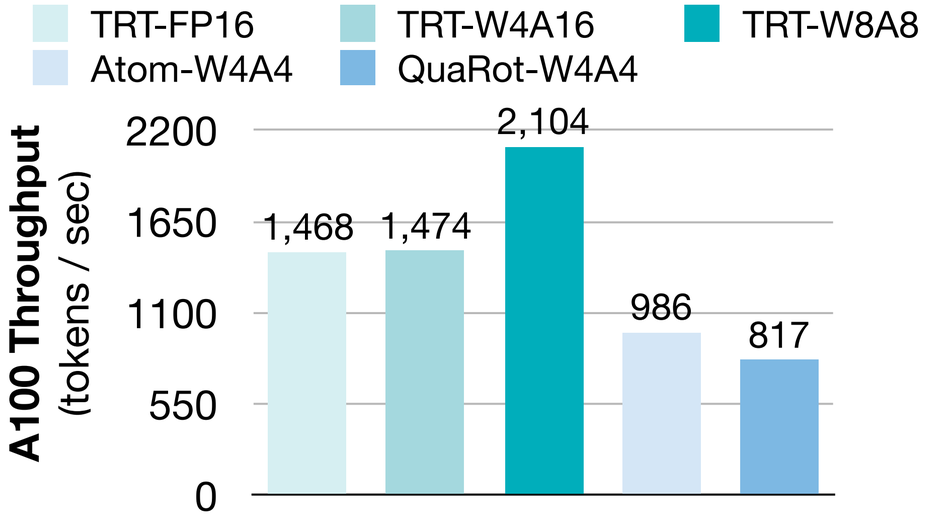
- Can we combine the advantage of both worlds?
  - Using 4-bit weights to save memory bandwidth
  - Using 8-bit activations to improve peak performance



(i.e., batch size for GEMM, 1 for ATTN in decoding, sequence length for ATTN in prefilling)

Current status of LLM serving systems

W8A8 provides the highest throughput for cloud serving.



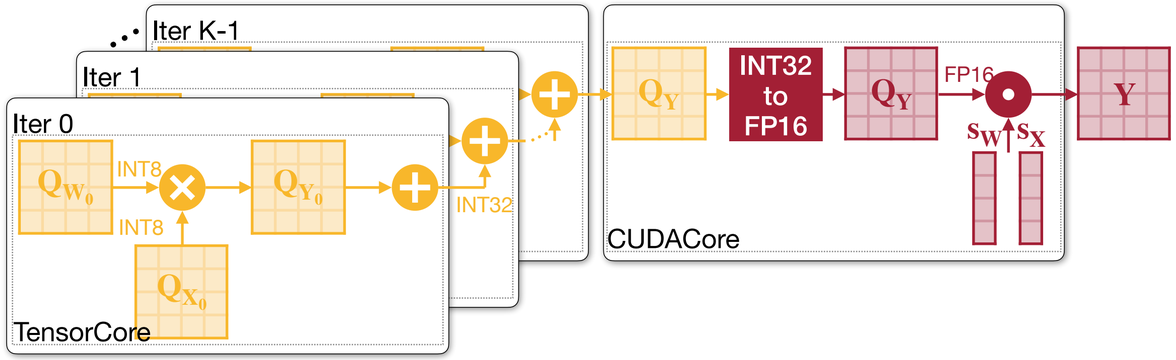
(b) Llama-2-7B system throughput on A100

Despite the existence of aggressive W4A4 quantization algorithms, they:

1. Bring about **significant accuracy loss**;
2. **Cannot run efficiently** on current GPUs.

Understanding the overhead in Quantized GEMM on GPUs

CUDA core operations in the main loop are expensive

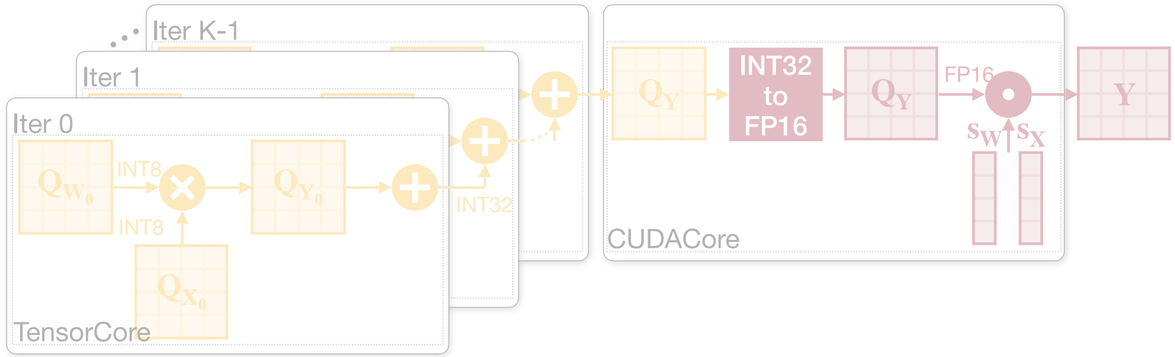


(a) TensorRT-LLM (INT8 Weights and INT8 Activations)

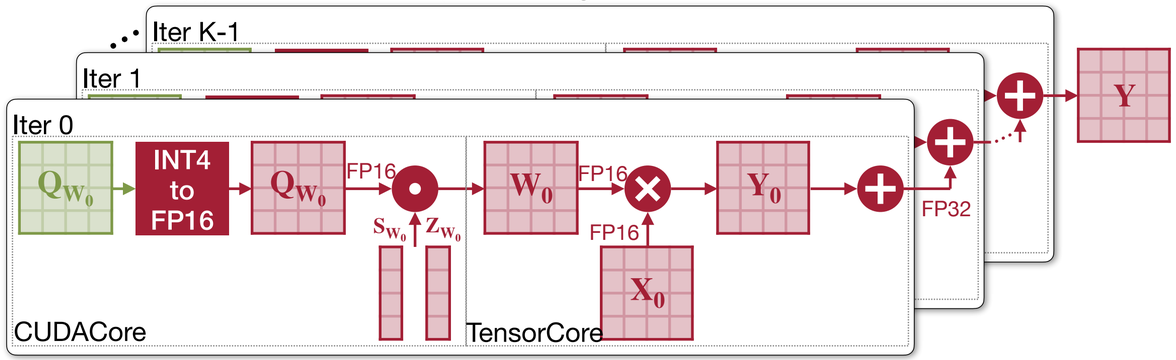
W8A8 is hardware efficient since the CUDA core operations are in the GEMM epilogue.

Understanding the overhead in Quantized GEMM on GPUs

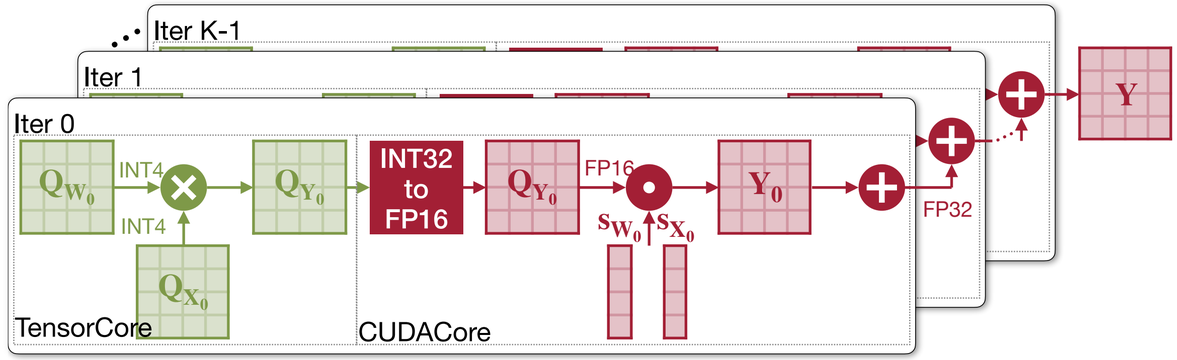
CUDA core operations in the main loop are expensive



(a) TensorRT-LLM (INT8 Weights and INT8 Activations)



(b) TensorRT-LLM (INT4 Weights and FP16 Activations)



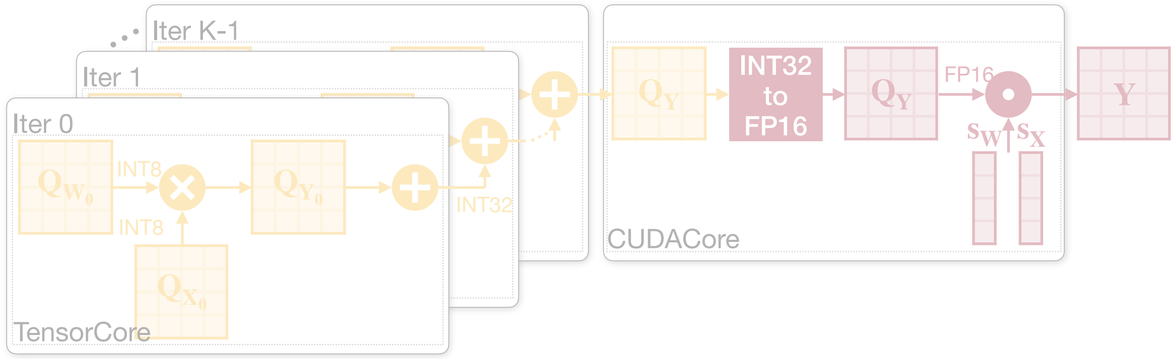
(c) ATOM (INT4 Weights and INT4 Activations)

- **Partial sums** in GEMM consumes much more registers compared with **inputs**.
- Consequently, **partial sum dequantization** requires more computation.
- Partial sum dequantization requires doubling the partial sum registers for software pipelining => large **register pressure**

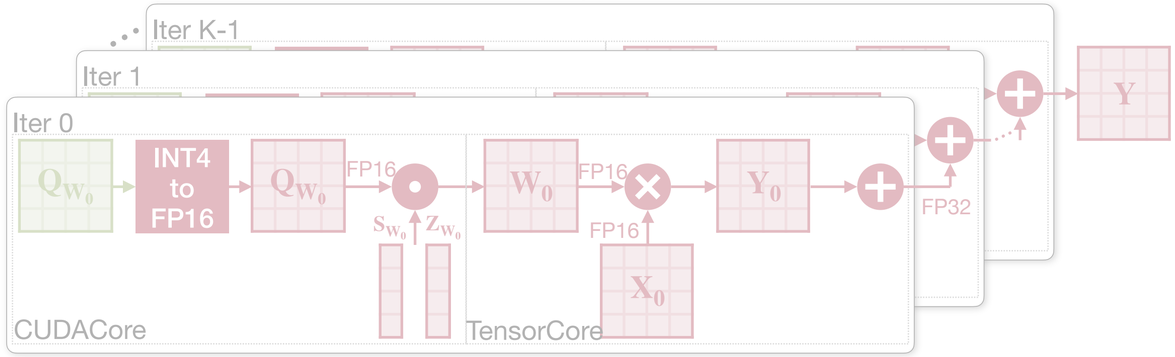
W4A16 (TRT-LLM) and W4A4 (Atom) both introduced CUDA core operations (dequantization) in the main loop. Partial sum dequantization in Atom is less hardware efficient in terms of both computation and register usage.

Understanding the overhead in Quantized GEMM on GPUs

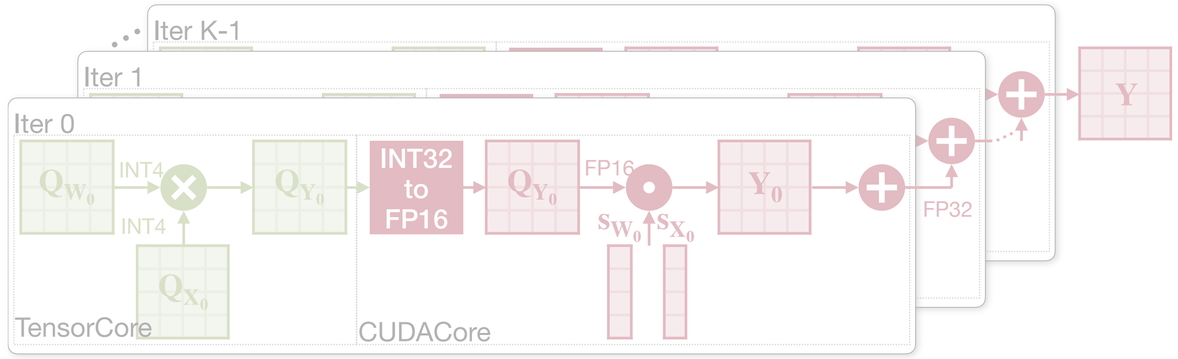
CUDA core operations in the main loop are expensive



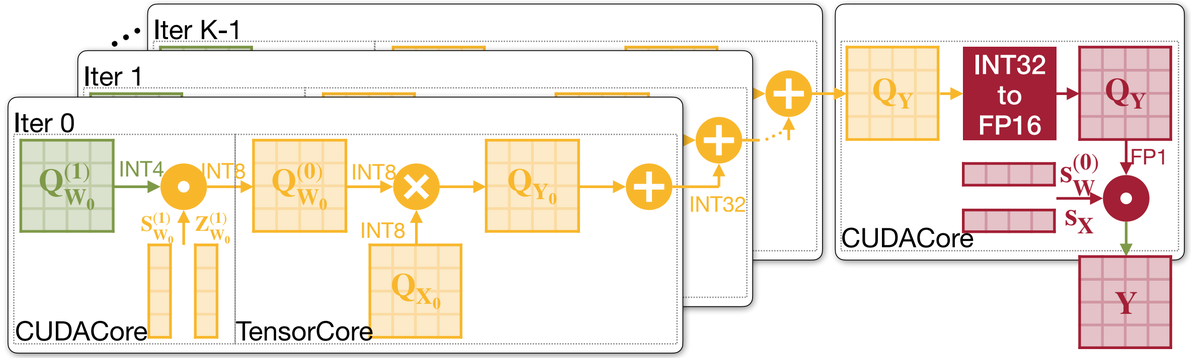
(a) TensorRT-LLM (INT8 Weights and INT8 Activations)



(b) TensorRT-LLM (INT4 Weights and FP16 Activations)



(c) ATOM (INT4 Weights and INT4 Activations)

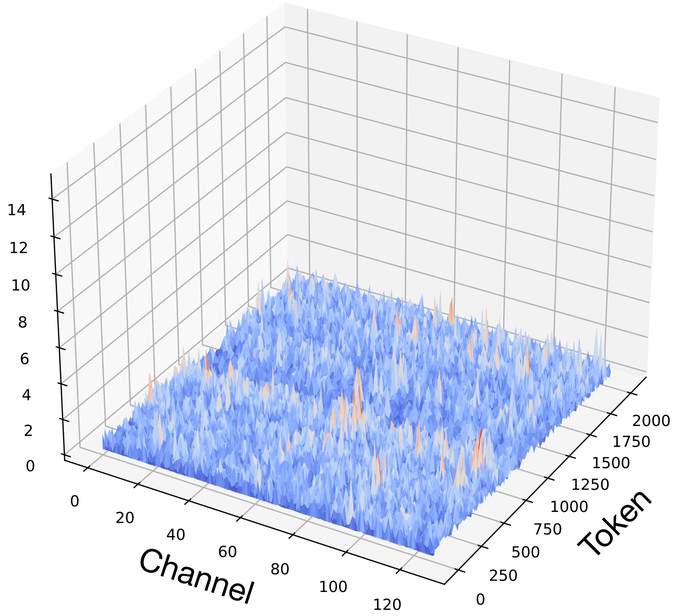


(d) Ours (INT4 Weights and INT8 Activations)

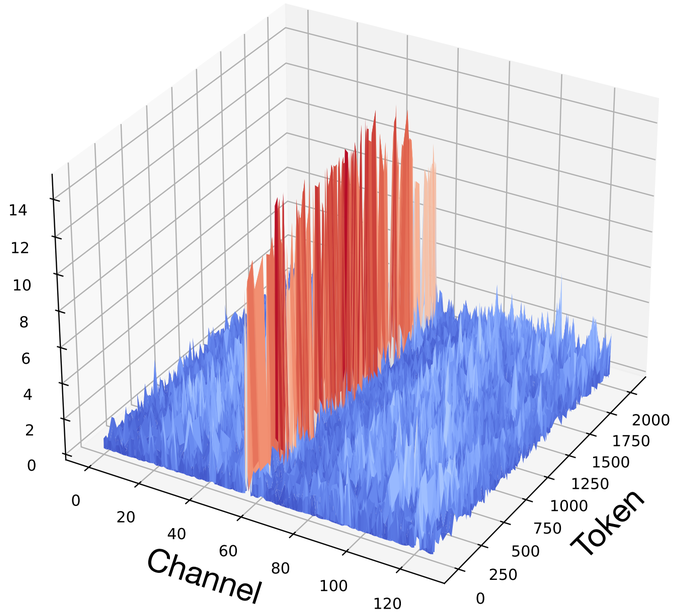
QServe (W4A8) reduces dequantization overhead in main loop using register level parallelism and avoids partial sum dequantization.

SmoothAttention

Migrate quantization difficulty from K cache to Q matrix



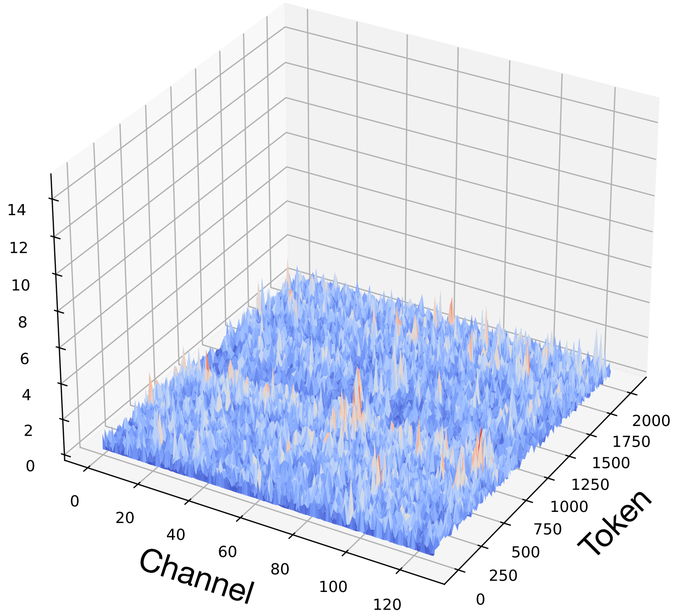
Layer 24 Values



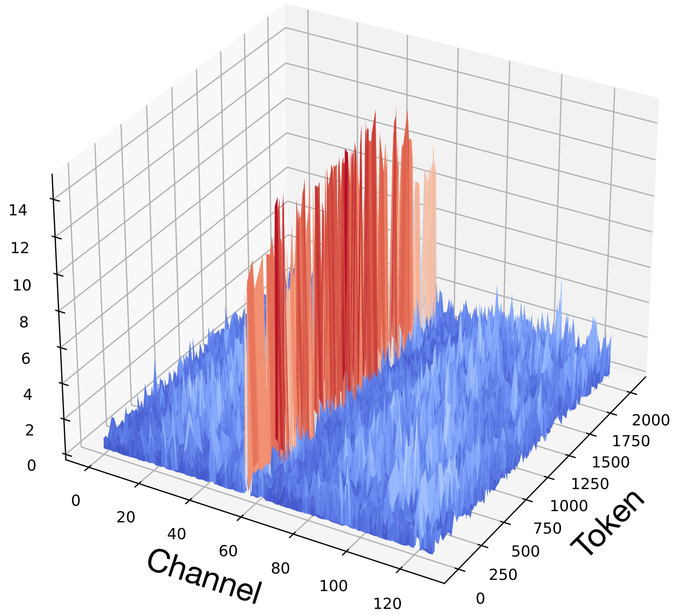
Layer 24 post-RoPE Keys
(Original)

SmoothAttention

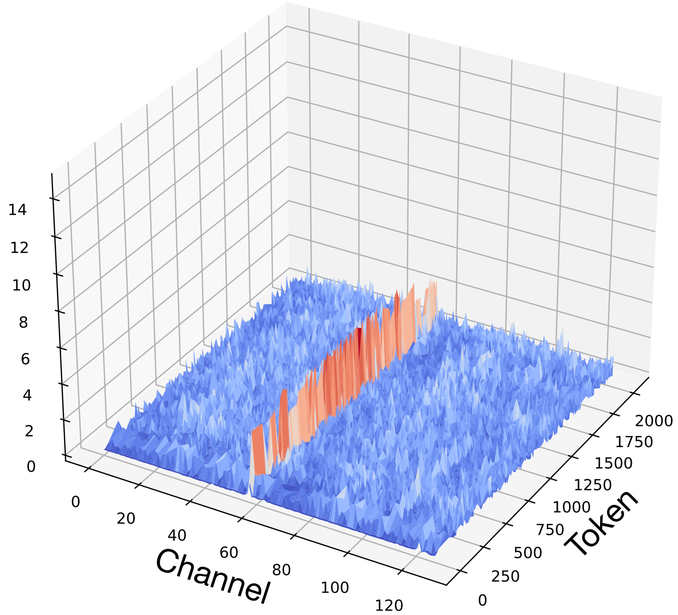
Migrate quantization difficulty from K cache to Q matrix



Layer 24 Values



Layer 24 post-RoPE Keys
(Original)



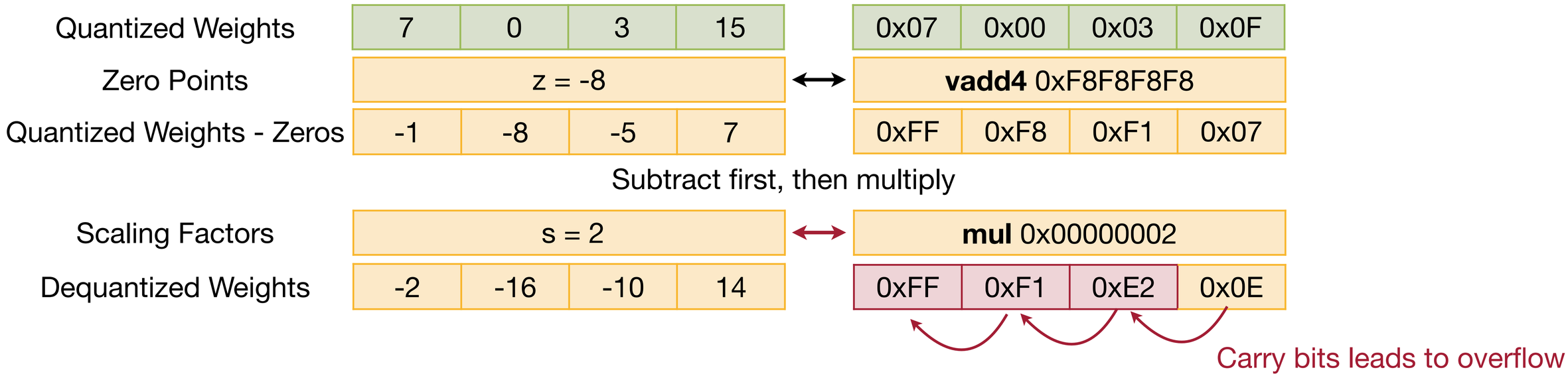
Layer 24 post-RoPE Keys
(SmoothAttention)

$$\mathbf{Z} = \left(\mathbf{Q}\boldsymbol{\Lambda}\right) \cdot \left(\mathbf{K}\boldsymbol{\Lambda}^{-1}\right)^{T}, \quad \boldsymbol{\Lambda} = \mathrm{diag}\left(\lambda\right)$$

$$\lambda_i = \max\left(\left|\mathbf{K}_i\right|\right)^{\alpha}$$

Efficient Dequantization with Reg-Level Parallelism in per-group QServe

UINT4 => UINT8 (similar to TinyChat), then UINT8 => SINT8 (apply zeros, scales)

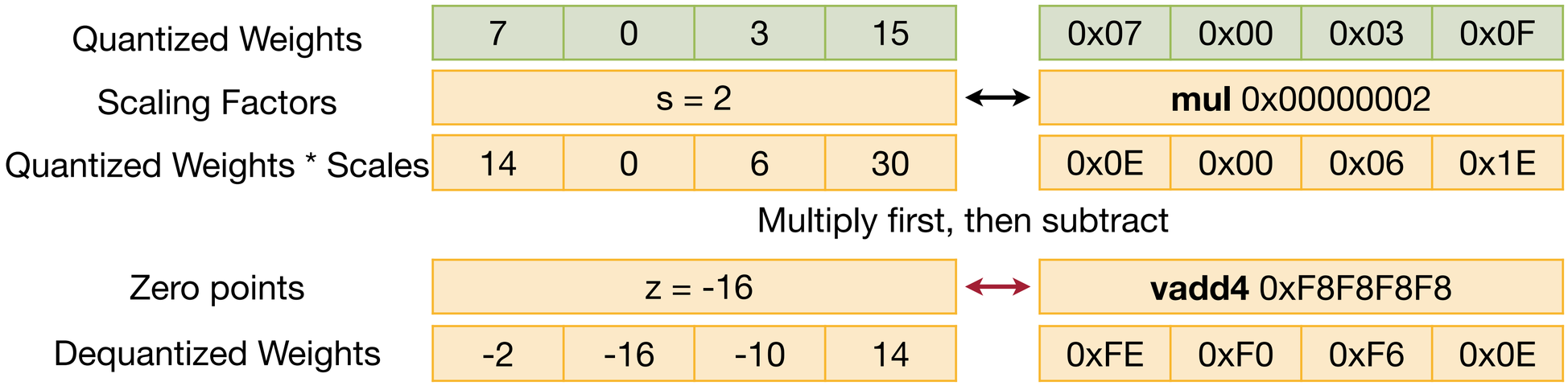


(a) **Overflow** fails register level parallelism

Register level parallelism **is not mathematically equal** in “Subtraction before multiplication” dequantization because of overflow.

Efficient Dequantization with Reg-Level Parallelism in per-group QServe

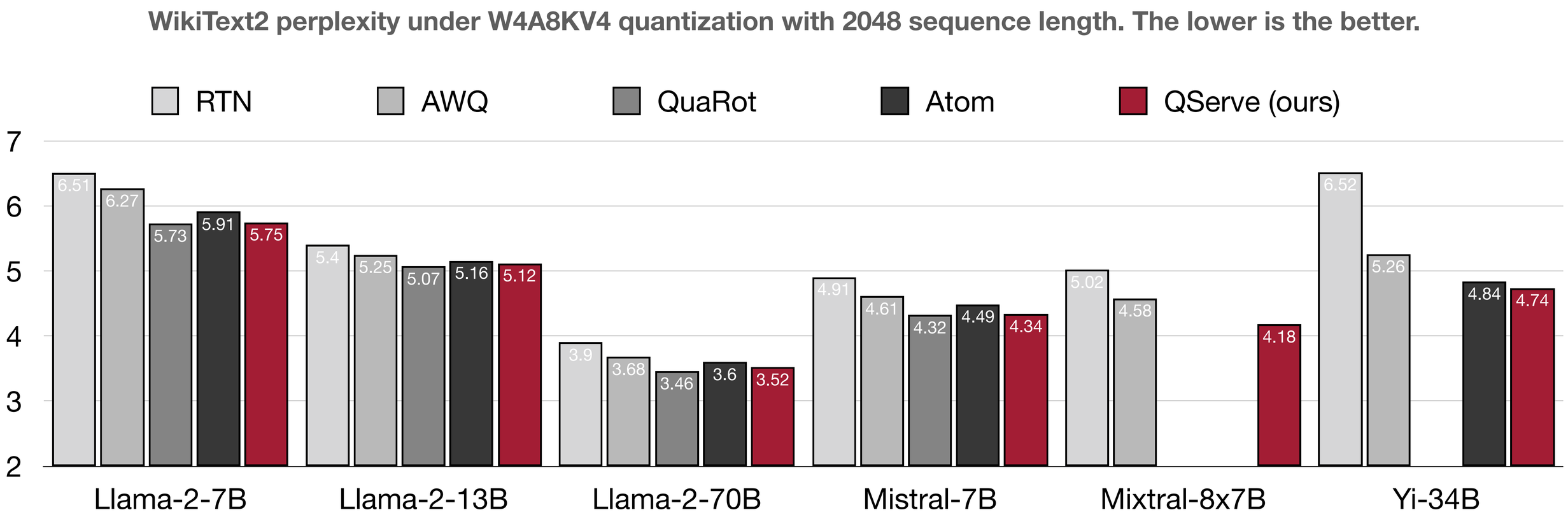
UINT4 => UINT8 (similar to TinyChat), then UINT8 => SINT8 (apply zeros, scales)



(b) **Progressive Quantization** guarantees register-level parallelism

Register level parallelism works in “multiplication before subtraction” dequantization.

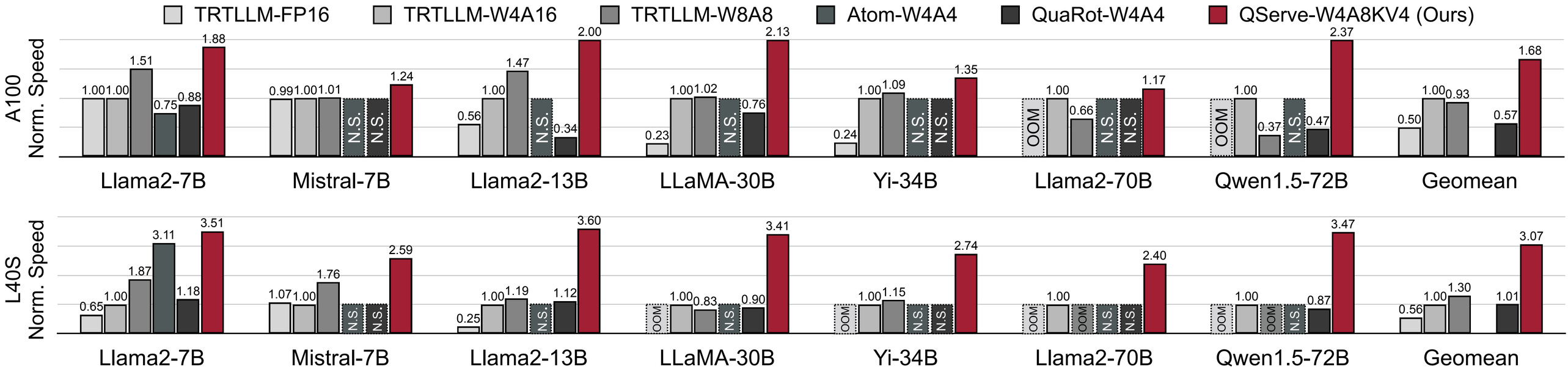
QServe outperforms the state-of-the-art



QServe offers state-of-the-art quantization accuracy, and more importantly it offers superior hardware efficiency compared with QuaRot.

End-to-end Throughput Evaluation Results

Up to 2.4x-3.5x faster than TensorRT-LLM on A100, L40S

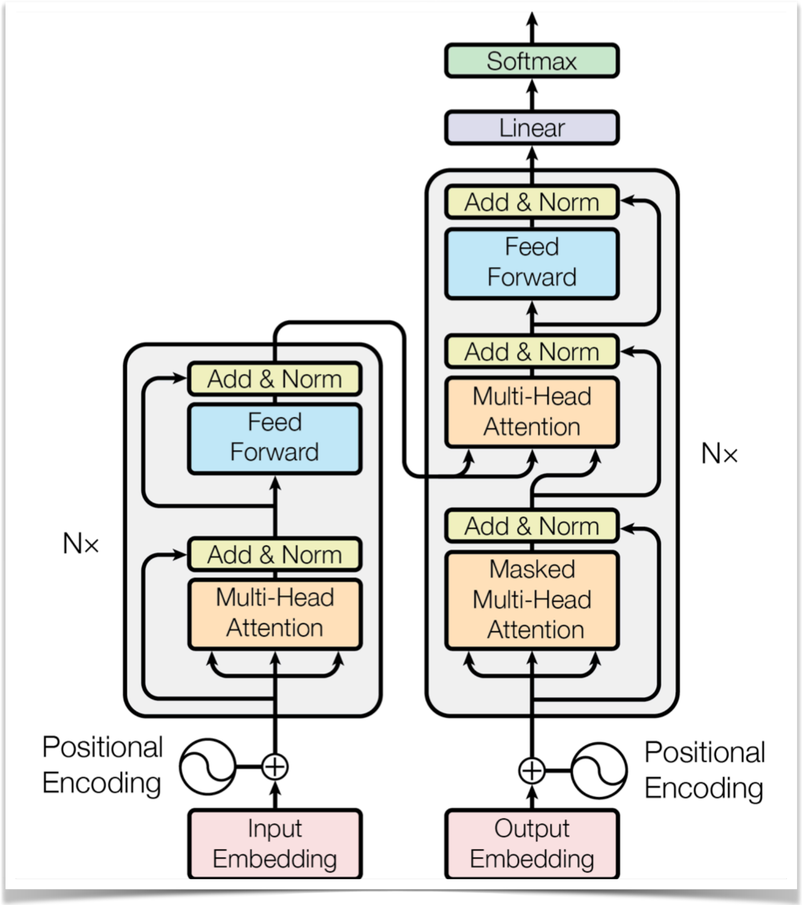


QServe outperforms TensorRT-LLM (W4A16) by 1.7-3.1x on average.

Lecture Plan

Today, we will cover:

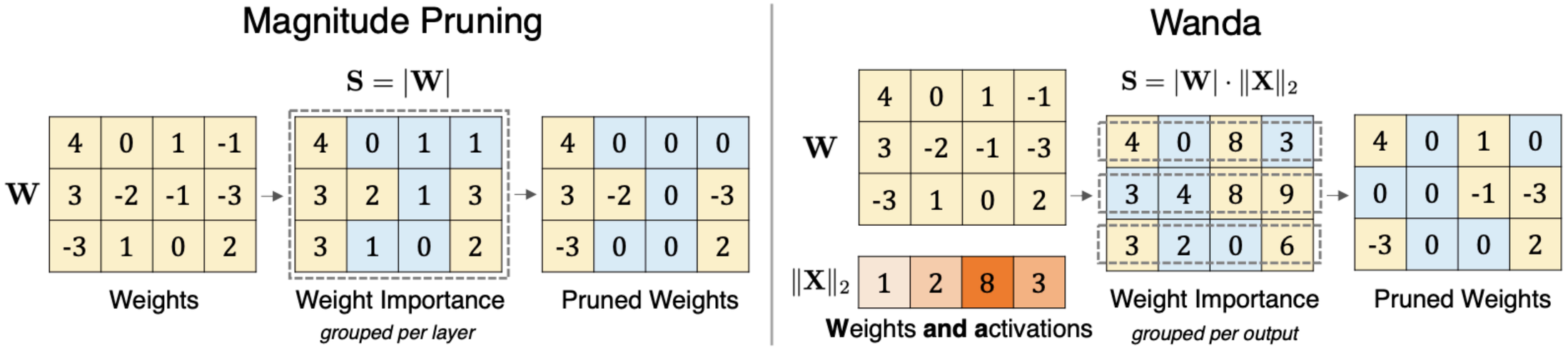
1. Quantization
    1. Weight-Activation Quantization: SmoothQuant
    2. Weight-Only Quantization: AWQ and TinyChat
    3. Further Practice: QServe (W4A8KV4)
2. **Pruning & Sparsity**
    1. Weight Sparsity: Wanda
    2. Contextual Sparsity: DejaVu, MoE
    3. Attention Sparsity: SpAtten, H_2O
3. LLM Serving Systems
    1. Metrics for LLM Serving
    2. Paged Attention (vLLM)
    3. FlashAttention
    4. Speculative Decoding
    5. Batching



Weight Sparsity (Pruning)

Wanda: pruning by considering weights and activations

- Similar idea compared to AWQ: we should also consider activation distribution when pruning weights!
- Use |weight| * ||activation|| as the criteria for pruning



A Simple and Effective Pruning Approach for Large Language Models (Sun et al., 2023)

Weight Sparsity (Pruning)

Wanda: pruning by considering weights and activations

- Similar idea compared to AWQ: we should also consider activation distribution when pruning weights!
- Use |weight| * ||activation|| as the criteria for pruning
- Consistently outperforms magnitude-based pruning

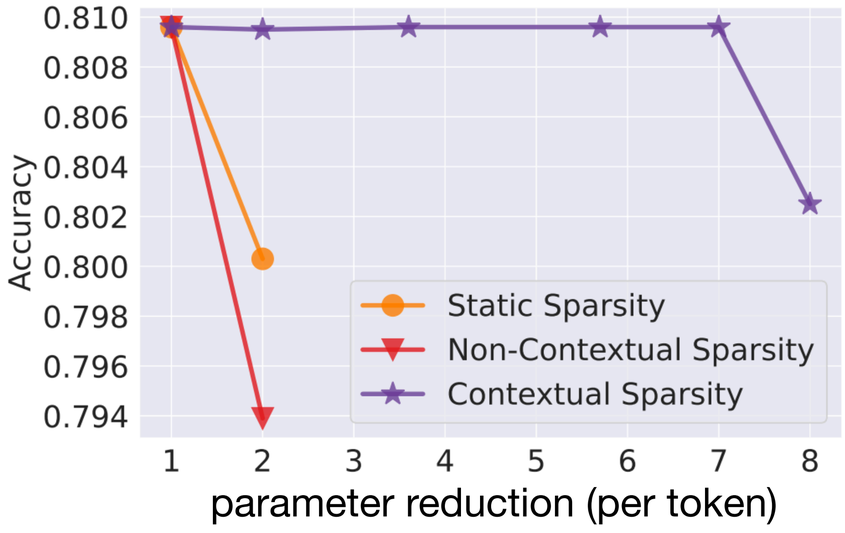
			LLaMA				LLaMA-2		
Method	Weight Update	Sparsity	7B	13B	30B	65B	7B	13B	70B
Dense	-	0%	59.99	62.59	65.38	66.97	59.71	63.03	67.08
Magnitude	✗	50%	46.94	47.61	53.83	62.74	51.14	52.85	60.93
SparseGPT	✓	50%	**54.94**	58.61	63.09	66.30	**56.24**	60.72	**67.28**
Wanda	✗	50%	54.21	**59.33**	**63.60**	**66.67**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

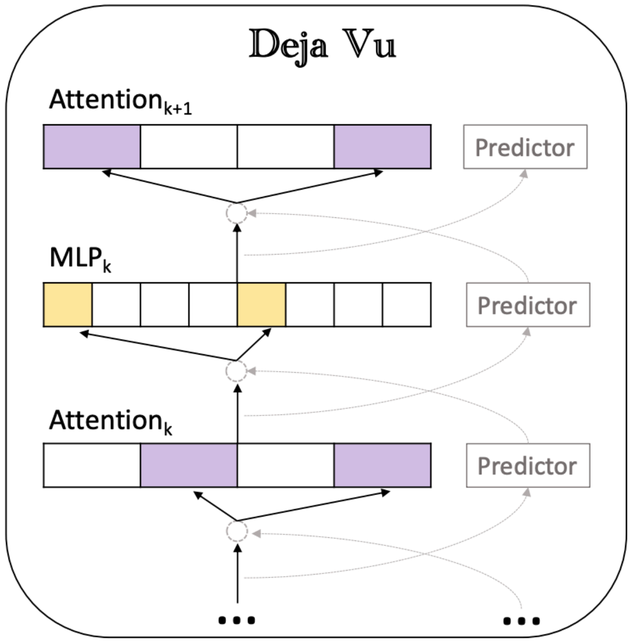
A Simple and Effective Pruning Approach for Large Language Models (Sun et al., 2023)

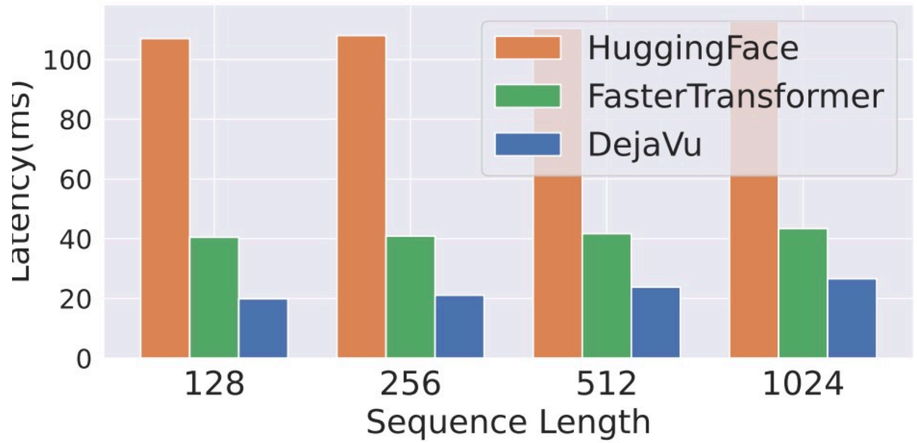
Contextual Sparsity

DejaVu (input dependent sparsity)

- *Static* sparsity: hurts the accuracy with a medium-high sparsity
- *Contextual* sparsity: small, input-dependent sets of redundant heads and features
- Contextual sparsity exists and can be predicted (using an async predictor head)
- Accelerate inference without hurting model quality





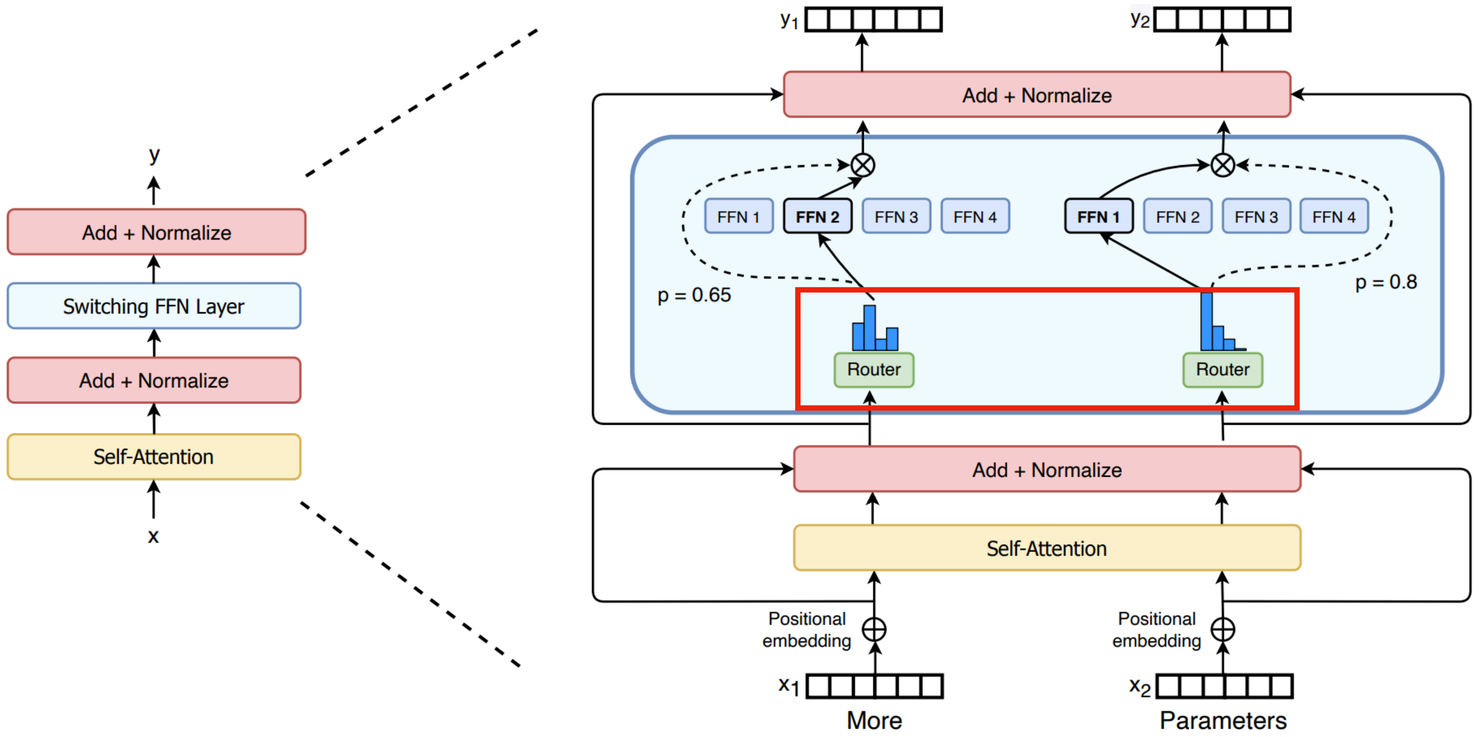


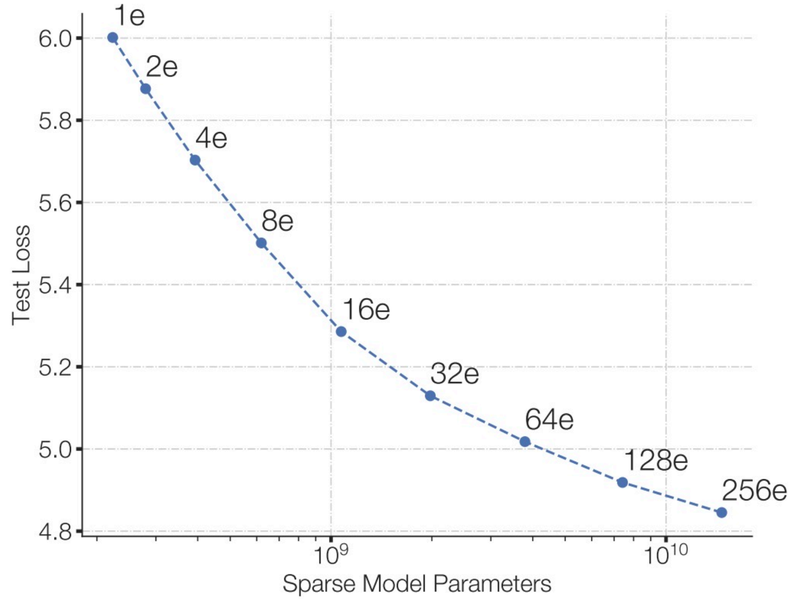
Deja Vu: Contextual Sparsity for Efficient LLMs at Inference Time (Liu et al., 2023)

Mixture-of-Experts (MoE)

Sparsely activated for each token

- MoE allows us to sparsely use part of the parameters for each token during inference
- It can increase the **total** amount of parameters without increasing inference costs **per token**
- A router will distribute the workload among different experts
- More experts -> larger total model size -> lower loss/better perplexity



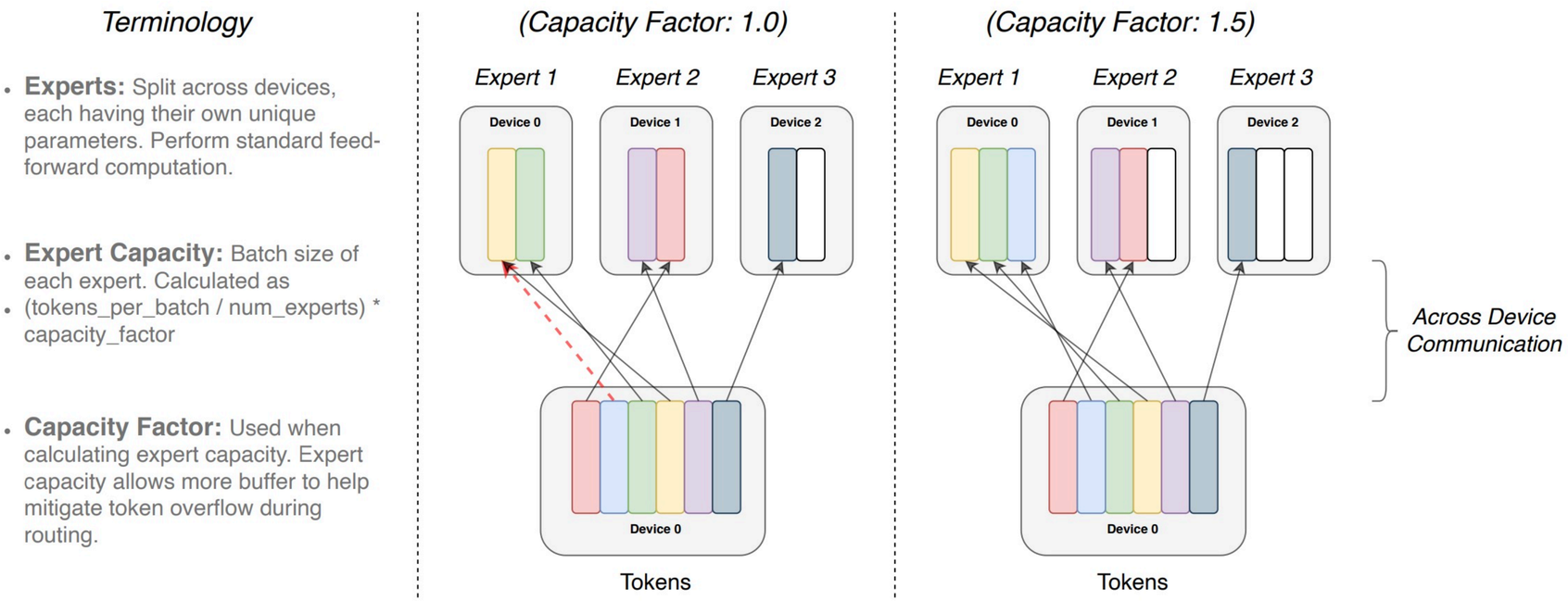


Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity [Fedus et al., 2022]

Mixture-of-Experts (MoE)

Sparsely activated for each token

- A key component of MoE is the routing function
- Capacity factor $C = \left(\frac{\text{tokens per batch}}{\text{number of experts}}\right) \times \text{capacity factor}.$
- $C = 1$, every expert can process at most 6/3*1=2 tokens; one token is skipped
- $C = 1.5$, every expert can process at most 6/3*1.5=3 tokens; slack capacity for expert 2&3

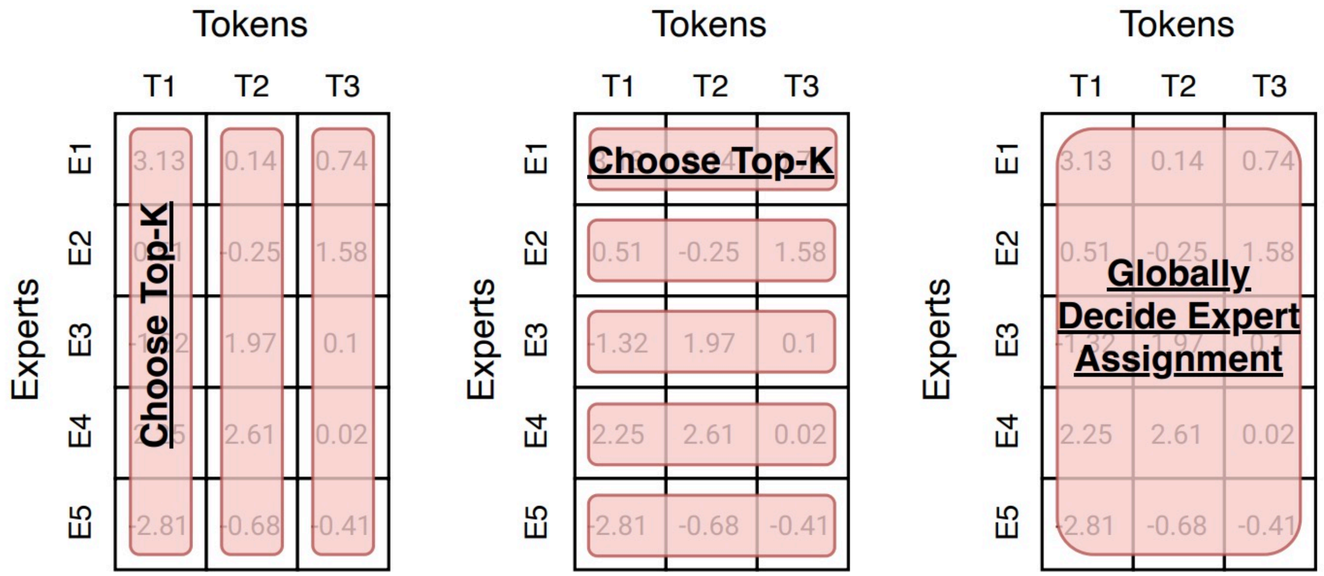


Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity [Fedus et al., 2022]

Mixture-of-Experts (MoE)

Sparsely activated for each token

- Different routing mechnisms

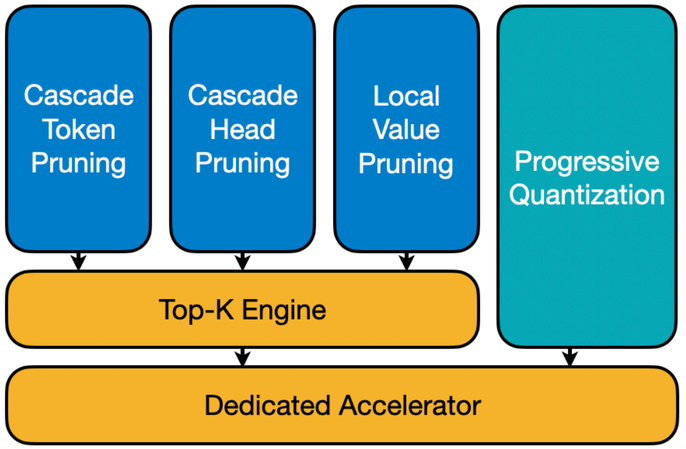


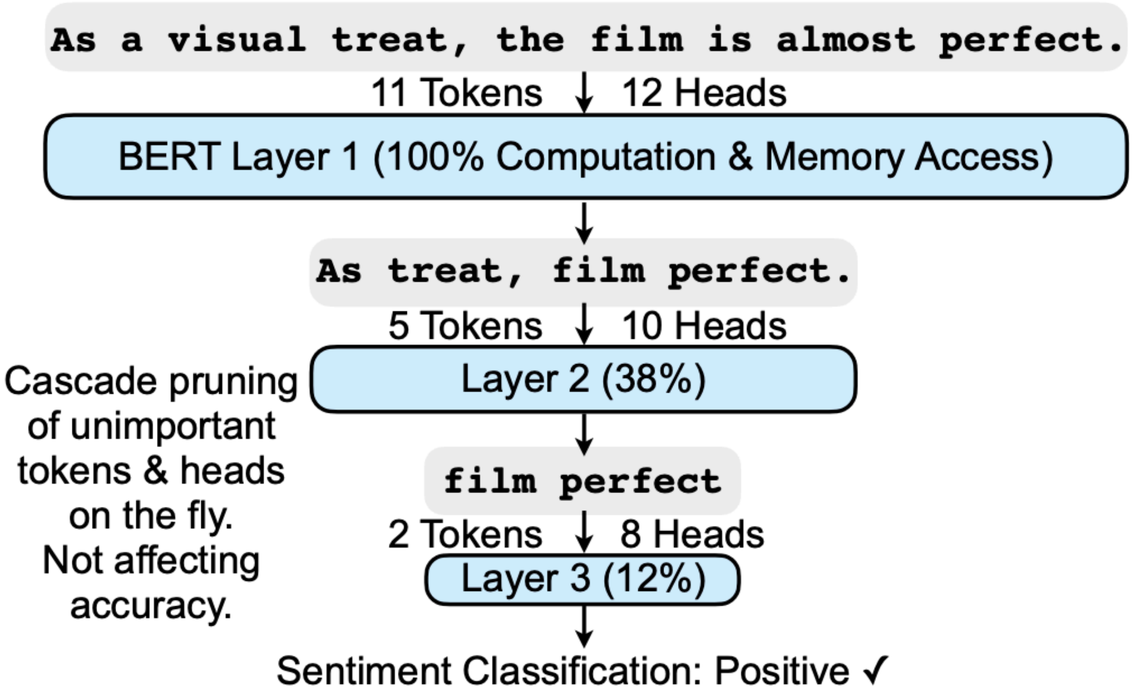
A Review of Sparse Expert Models in Deep Learning [Fedus et al., 2022]

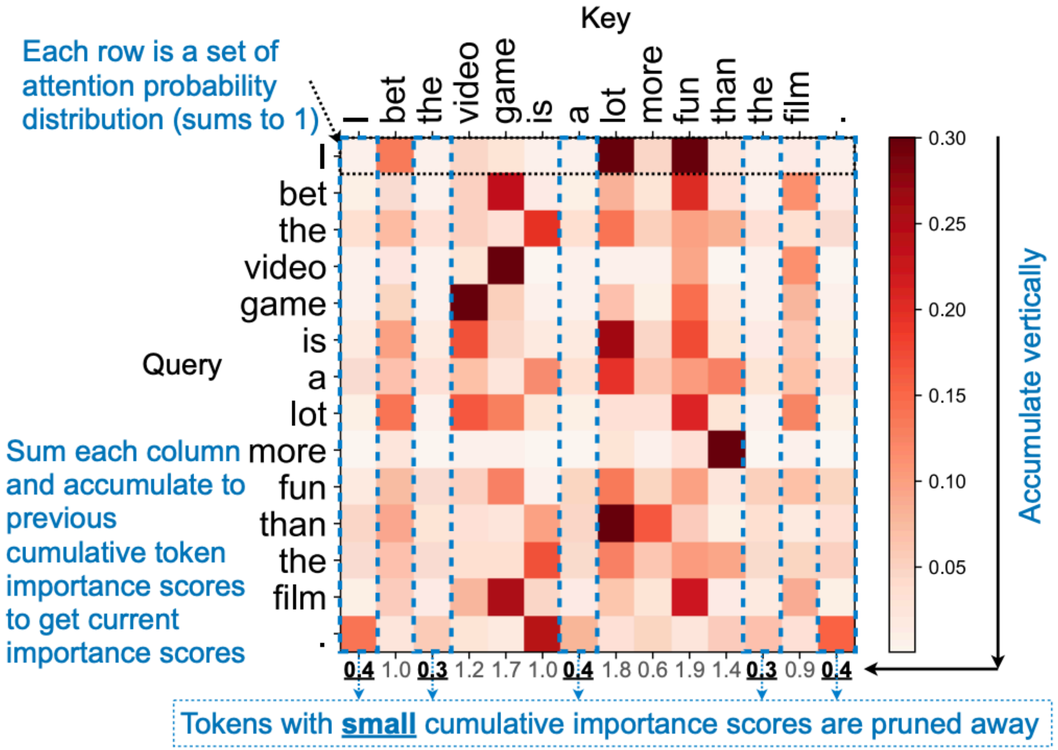
Attention Sparsity

SpAtten: token pruning & head pruning

- Cascade pruning of unimportant tokens and heads.
- Tokens with a small cumulative attention are pruned away.
- V pruning: don't fetch V if QK is small.
- Progressive quantization: low precision first, if not confident => high precision.





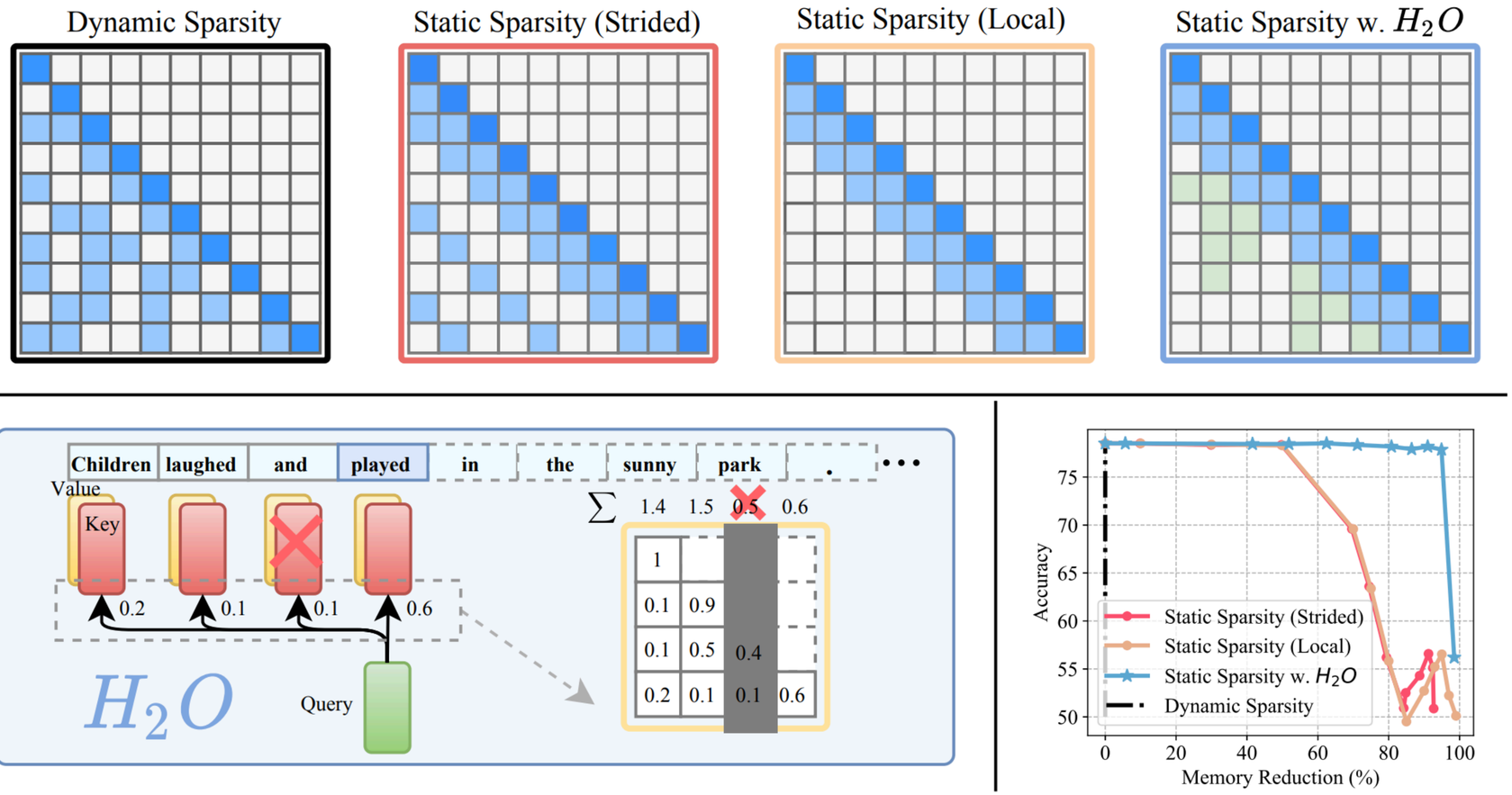


SpAtten: Efficient Sparse Attention Architecture with Cascade Token and Head Pruning (Wang et al., 2020)

Attention Sparsity

H_2O: token pruning in KV cache

- Keep the local tokens and heavy Hitter Tokens (H_2) in the cache

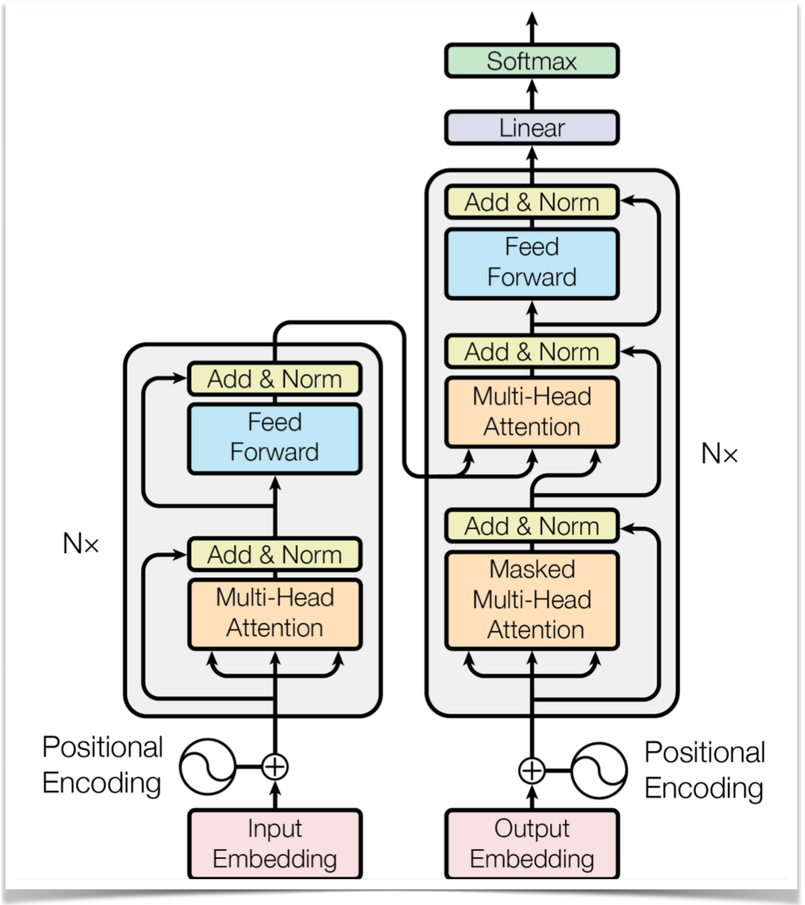


H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models (Zhang et al., 2023)

Lecture Plan

Today, we will cover:

1. **Quantization**
    1. Weight-Activation Quantization: SmoothQuant
    2. Weight-Only Quantization: AWQ and TinyChat
    3. Further Practice: QServe (W4A8KV4)
2. **Pruning & Sparsity**
    1. Weight Sparsity: Wanda
    2. Contextual Sparsity: DejaVu, MoE
    3. Attention Sparsity: SpAtten, H_2O
3. **LLM Serving Systems**
    1. Metrics for LLM Serving
    2. Paged Attention (vLLM)
    3. FlashAttention
    4. Speculative Decoding
    5. Batching



Important Metrics for LLM Serving

- **Time To First Token (TTFT):** Measures how quickly users begin to see model output after submitting a query.
  - Crucial for real-time interactions.
  - Driven by prompt processing time and the generation of the first token.
- **Time Per Output Token (TPOT):** Time taken to generate each output token.
  - Impacts user perception of speed *(e.g., 100 ms/token = 10 tokens/second, ~450 words/minute).*
- **Latency = (TTFT) + (TPOT * the number of tokens to be generated).**
  - Total time to generate the complete response.
- **Throughput:** Number of tokens generated per second across all requests by the inference server.

https://www.databricks.com/blog/llm-inference-performance-engineering-best-practices

Optimizing LLM Serving: Goals and Tradeoffs

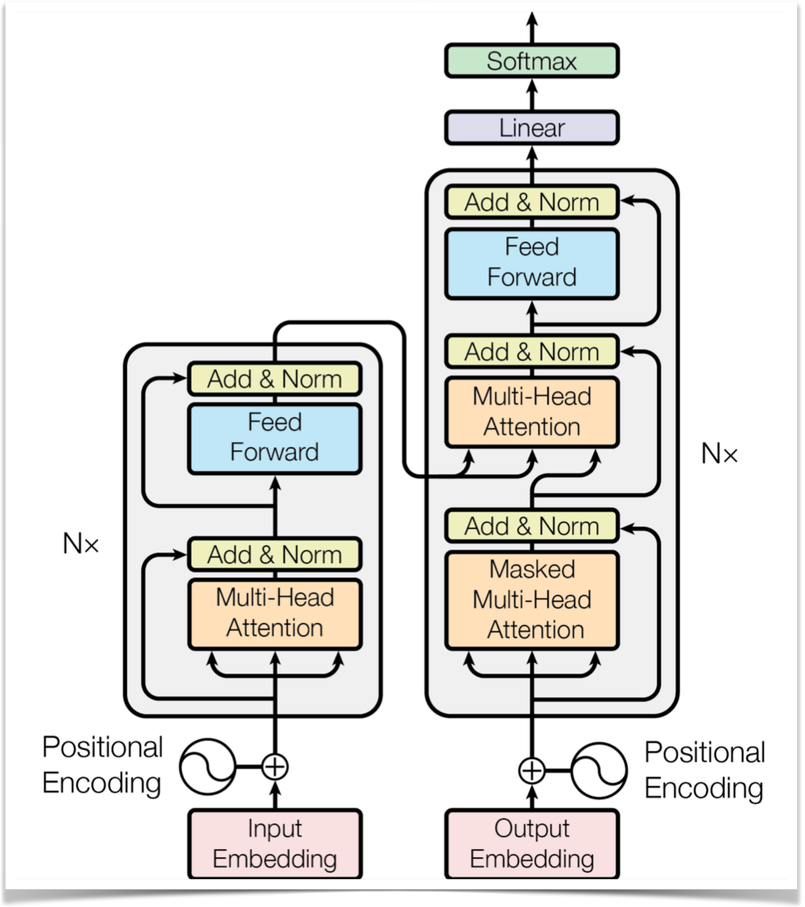
- **Optimization Goals & Tradeoffs:**
  - **Goal:** *Minimize* TTFT, *maximize* throughput, and *reduce* TPOT.
  - **Throughput vs. TPOT Tradeoff:** *Processing multiple queries concurrently increases throughput but extends TPOT for each user.*
- **Key Heuristics for Model Evaluation:**
  - **Output Length:** Dominates latency.
  - **Input Length:** Minimal impact on performance but significant for hardware.
  - **Model Size:** Larger models have higher latency, but not proportional to their size.
    - Example: Llama-70B is 2x Llama-13B.

https://www.databricks.com/blog/llm-inference-performance-engineering-best-practices

Lecture Plan

Today, we will cover:

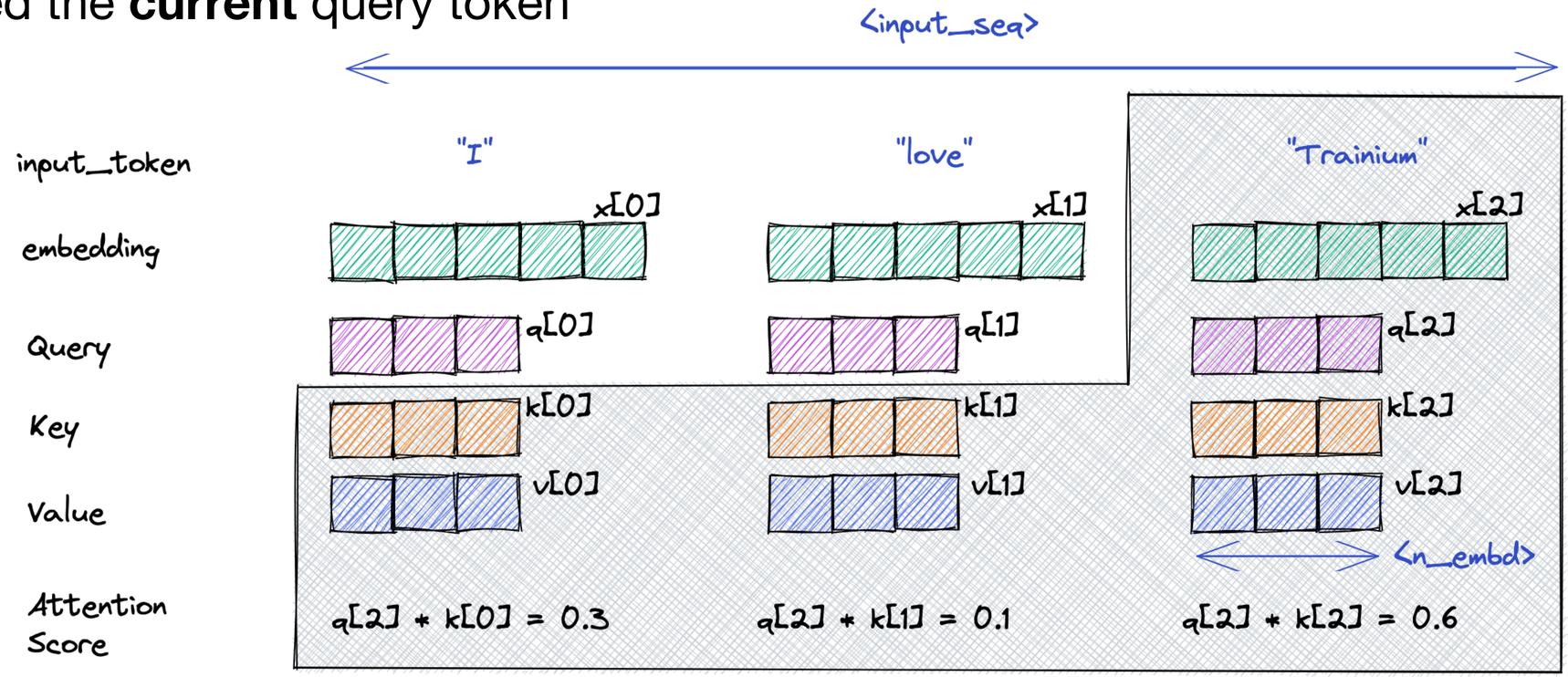
1. **Quantization**
    1. Weight-Activation Quantization: SmoothQuant
    2. Weight-Only Quantization: AWQ and TinyChat
    3. Further Practice: QServe (W4A8KV4)
2. **Pruning & Sparsity**
    1. Weight Sparsity: Wanda
    2. Contextual Sparsity: DejaVu, MoE
    3. Attention Sparsity: SpAtten, H2O
3. **LLM Serving Systems**
    1. Metrics for LLM Serving
    2. Paged Attention (vLLM)
    3. FlashAttention
    4. Speculative Decoding
    5. Batching



Recap: KV Cache

The KV cache could be large with long context

- During Transformer decoding (GPT-style), we need to store the **K**eys and **V**alues of **all previous** tokens so that we can perform the attention computation, namely the **KV cache**
  - Only need the **current** query token



$$a_{ij} = \frac{\exp(q_i^\top k_j/\sqrt{d})}{\sum_{t=1}^{i}\exp(q_i^\top k_t/\sqrt{d})},\ o_i = \sum_{j=1}^{i} a_{ij} v_j$$

Image credit: https://awsdocs-neuron.readthedocs-hosted.com/en/latest/general/appnotes/transformers-neuronx/generative-llm-inference-with-neuron.html

Recap: the Challenge of Large KV Cache

The KV cache could be large with long context

- We can calculate the memory required to store the KV cache
- Consider Llama-2-70B (if using MHA), KV cache requires

$$\underbrace{BS}_{minibatch} * \underbrace{80}_{layers} * \underbrace{64}_{kv-heads} * \underbrace{128}_{n_{emd}} * \underbrace{N}_{length} * \underbrace{2}_{K\&V} * \underbrace{2\text{bytes}}_{FP16} = 2.5\text{MB} \times BS \times N$$

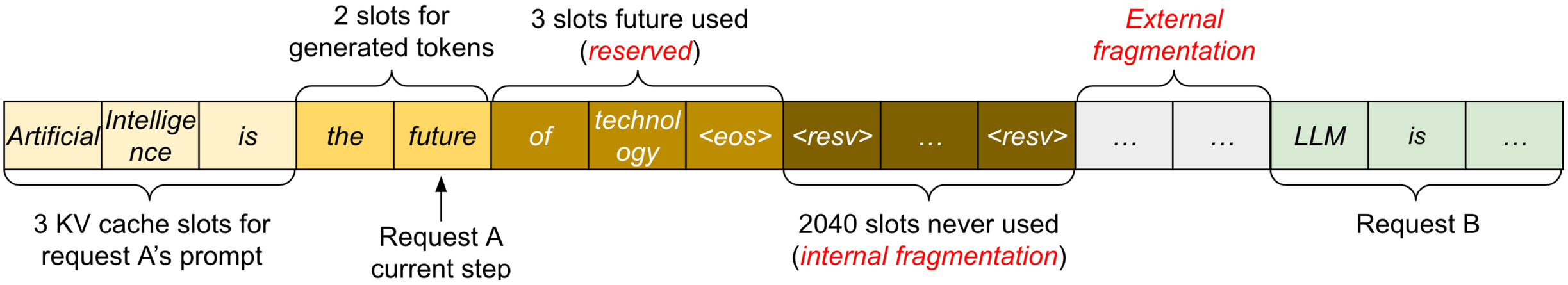
- bs=1, n_seq=512: 1.25GB
- bs=1, n_seq=4096 : 10GB (~ a paper)
- bs=16, n_seq=4096: 160GB (requires two A100 GPU!)

https://www.databricks.com/blog/llm-inference-performance-engineering-best-practices

vLLM and Paged Attention

High-throughput and memory-efficient serving

- Analyzing the waste in KV Cache usage:
  - *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.

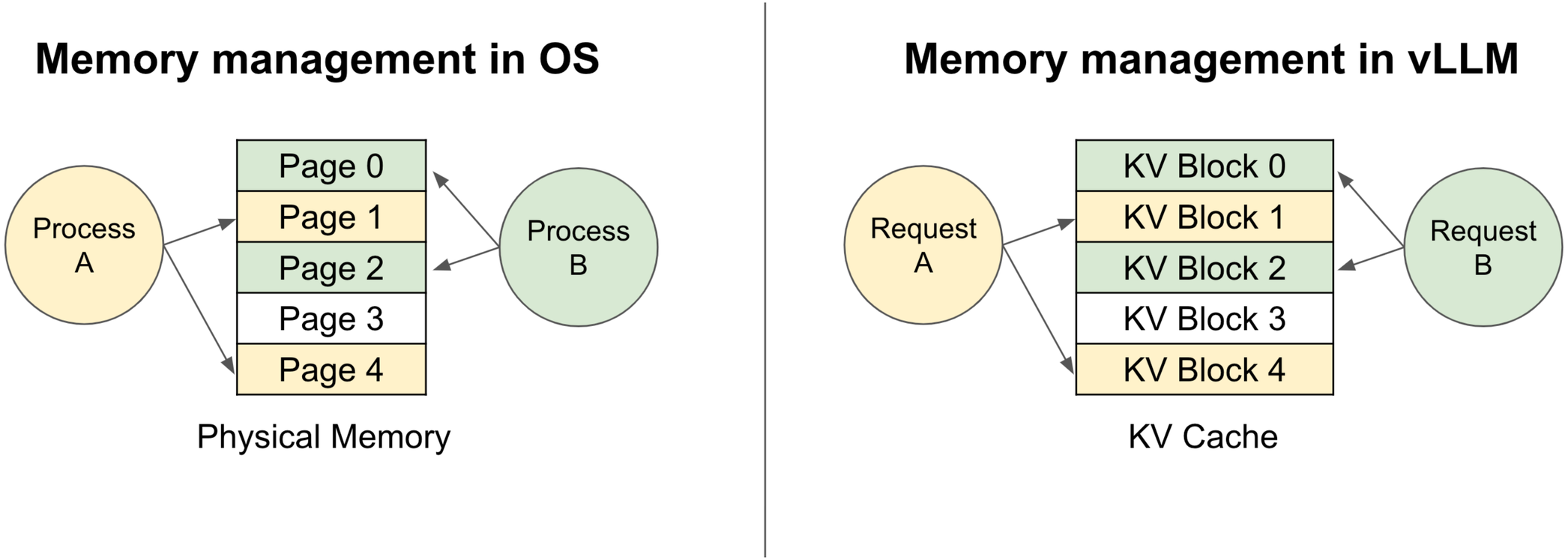


Efficient Memory Management for Large Language Model Serving with PagedAttention (Kwon et al., 2023)

vLLM and Paged Attention

High-throughput and memory-efficient serving

- Inspiration from operating systems (OS): virtual memory and paging!

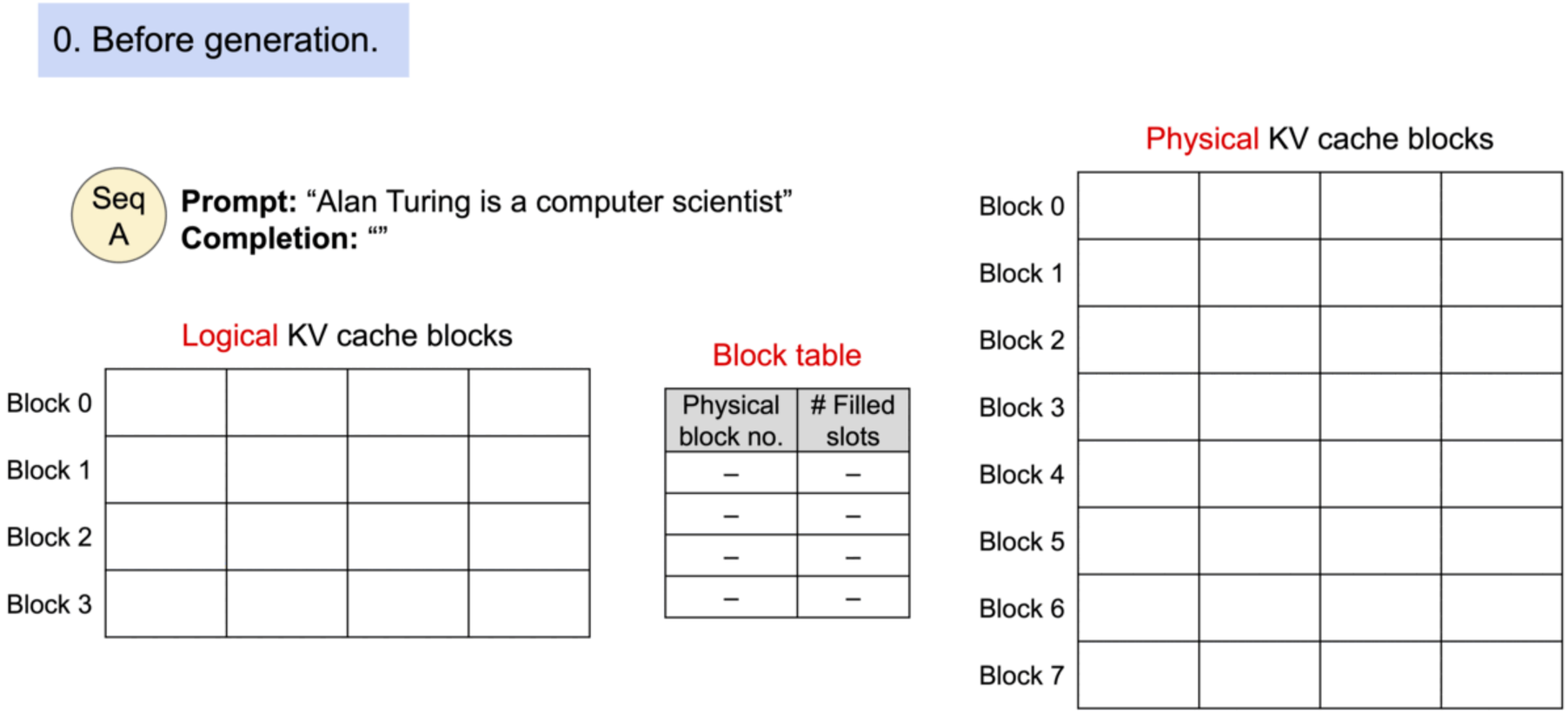


Efficient Memory Management for Large Language Model Serving with PagedAttention (Kwon et al., 2023)

vLLM and Paged Attention

PagedAttention

- PagedAttention addresses the KV-cache memory fragmentation
- It allows for storing continuous keys and values in non-contiguous memory space

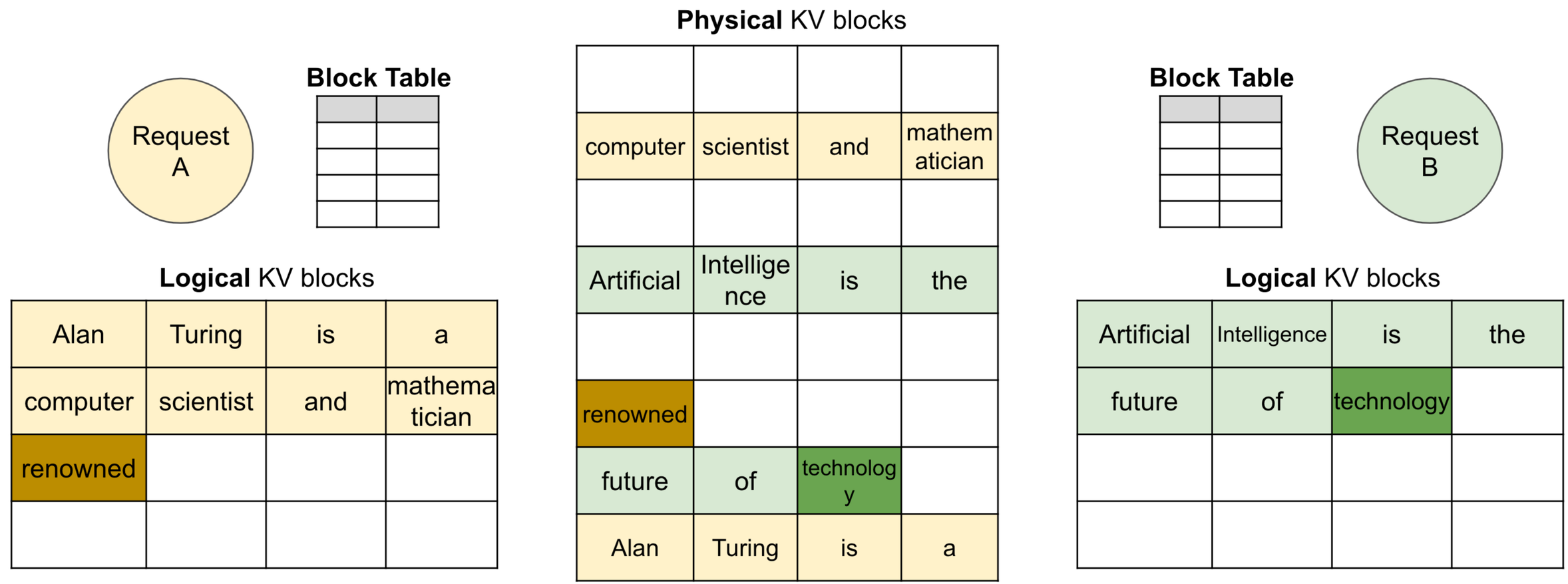


Efficient Memory Management for Large Language Model Serving with PagedAttention (Kwon et al., 2023)

vLLM and Paged Attention

PagedAttention

- The case for multiple requests

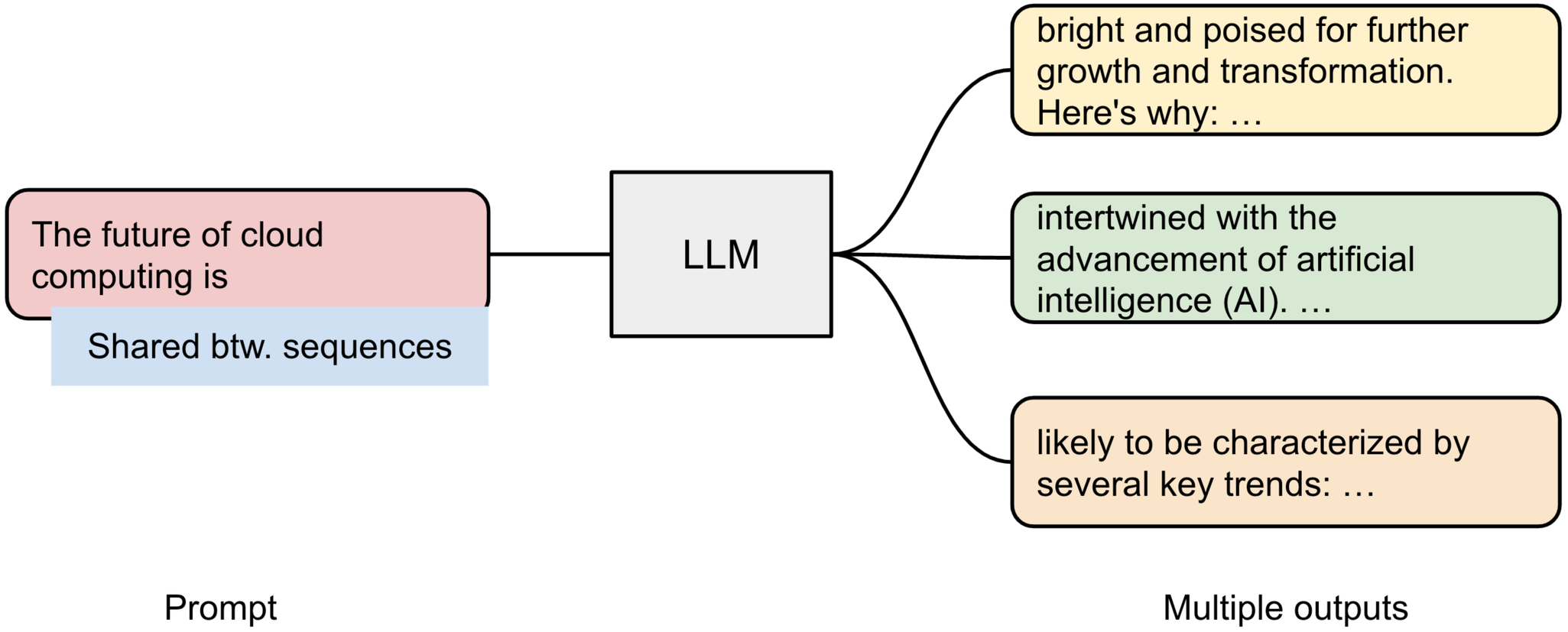


Efficient Memory Management for Large Language Model Serving with PagedAttention (Kwon et al., 2023)

vLLM and Paged Attention

PagedAttention

- Dynamic block mapping enables prompt sharing in parallel sampling:

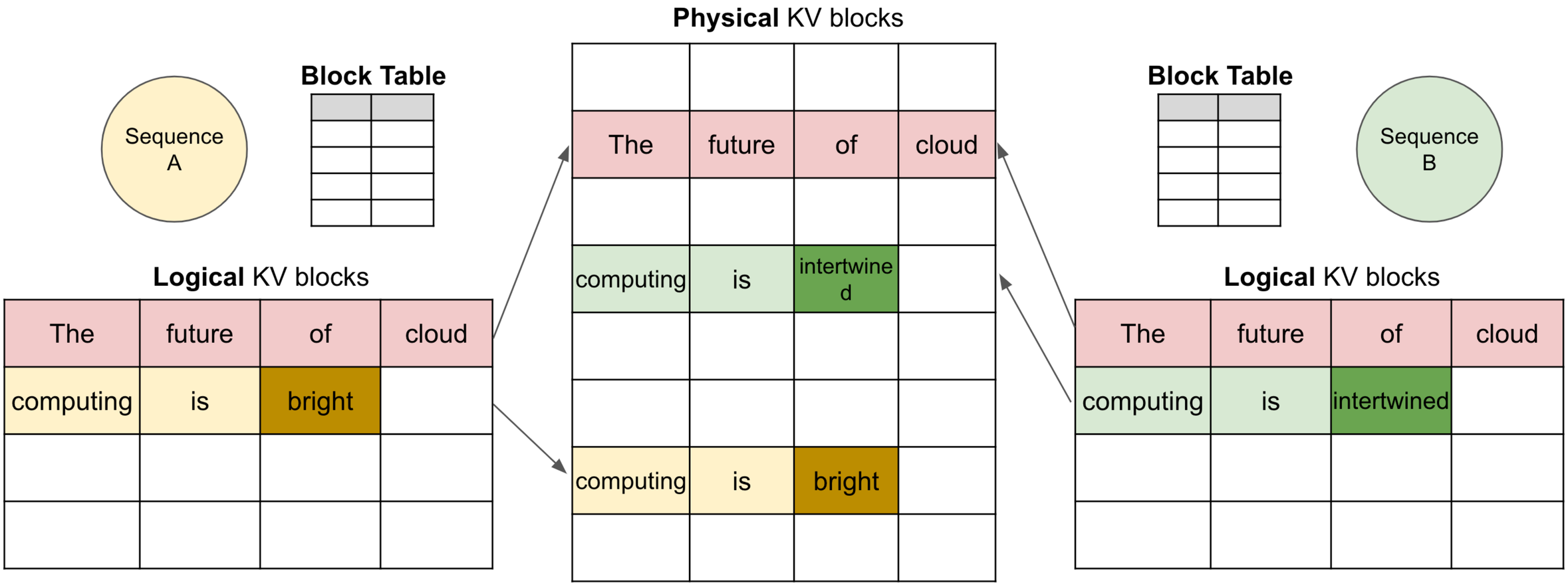


Efficient Memory Management for Large Language Model Serving with PagedAttention (Kwon et al., 2023)

vLLM and Paged Attention

PagedAttention

- Dynamic block mapping enables prompt sharing in parallel sampling

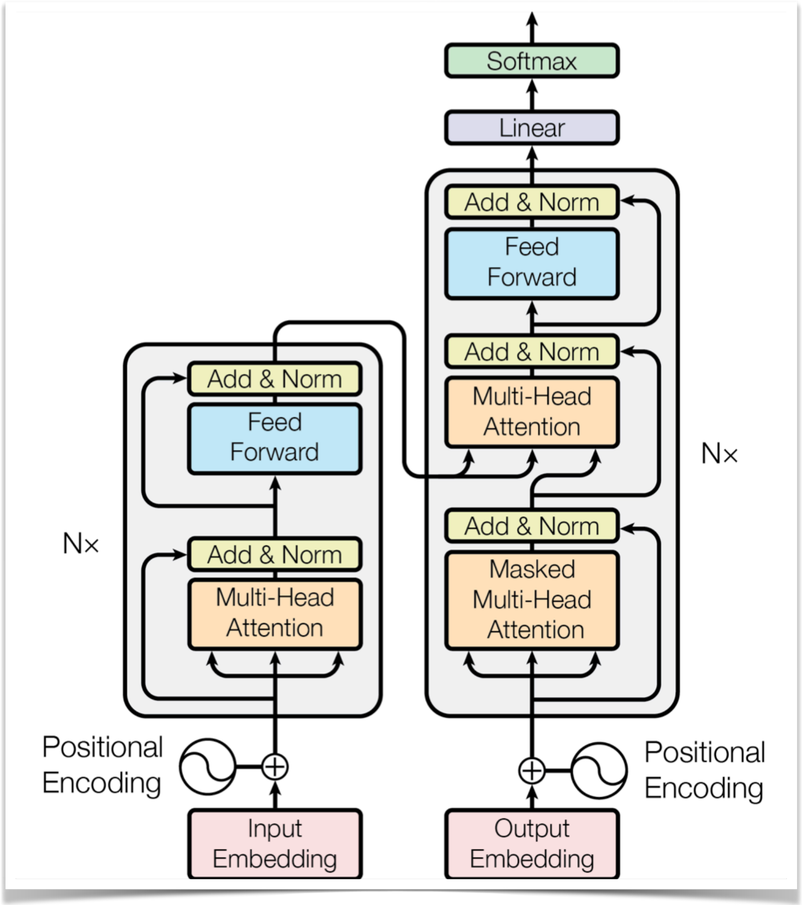


Efficient Memory Management for Large Language Model Serving with PagedAttention (Kwon et al., 2023)

Lecture Plan

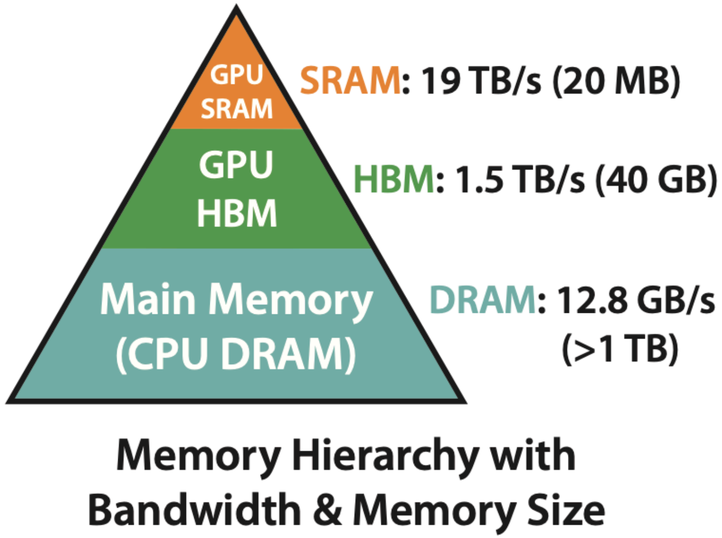
Today, we will cover:

1. **Quantization**
    1. Weight-Activation Quantization: SmoothQuant
    2. Weight-Only Quantization: AWQ and TinyChat
    3. Further Practice: QServe (W4A8KV4)
2. **Pruning & Sparsity**
    1. Weight Sparsity: Wanda
    2. Contextual Sparsity: DejaVu, MoE
    3. Attention Sparsity: SpAtten, H2O
3. **LLM Serving Systems**
    1. Metrics for LLM Serving
    2. Paged Attention (vLLM)
    3. FlashAttention
    4. Speculative Decoding
    5. Batching

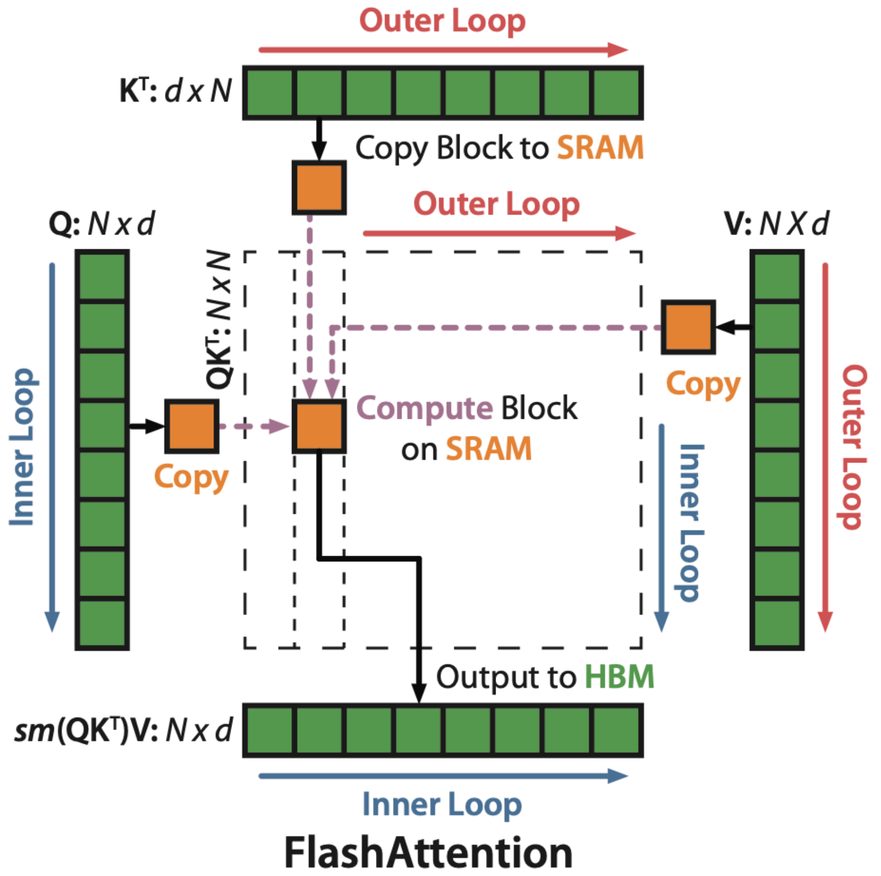


FlashAttention

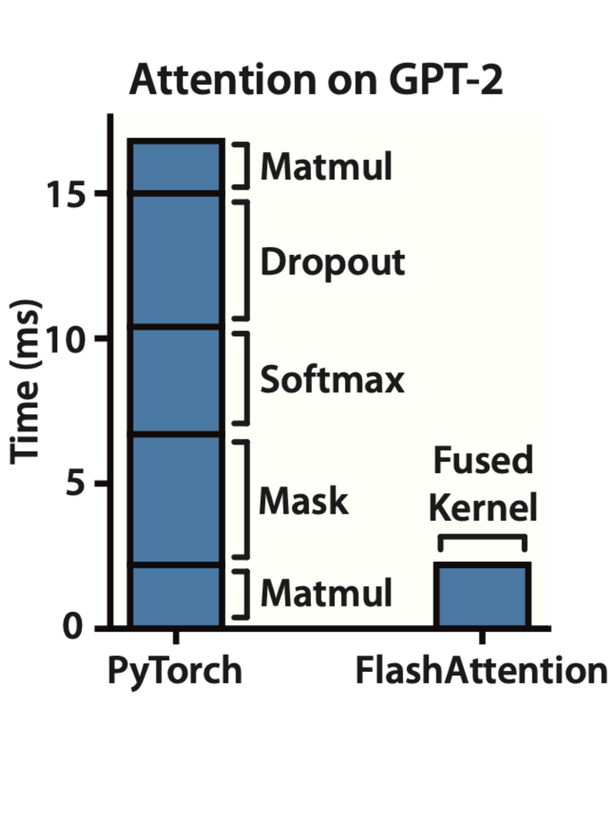
- Use tiling to prevent the materialization of the large $N \times N$ attention matrix, thus avoid using the slow HBM; kernel fusion.



Memory Hierarchy with Bandwidth & Memory Size



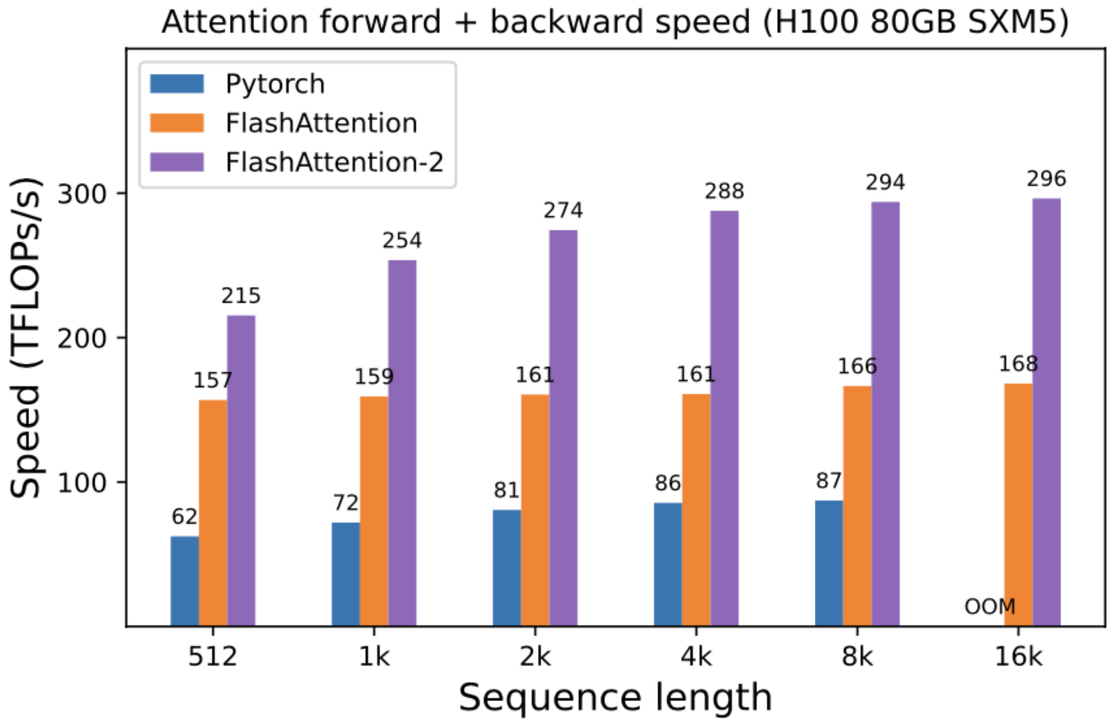
FlashAttention



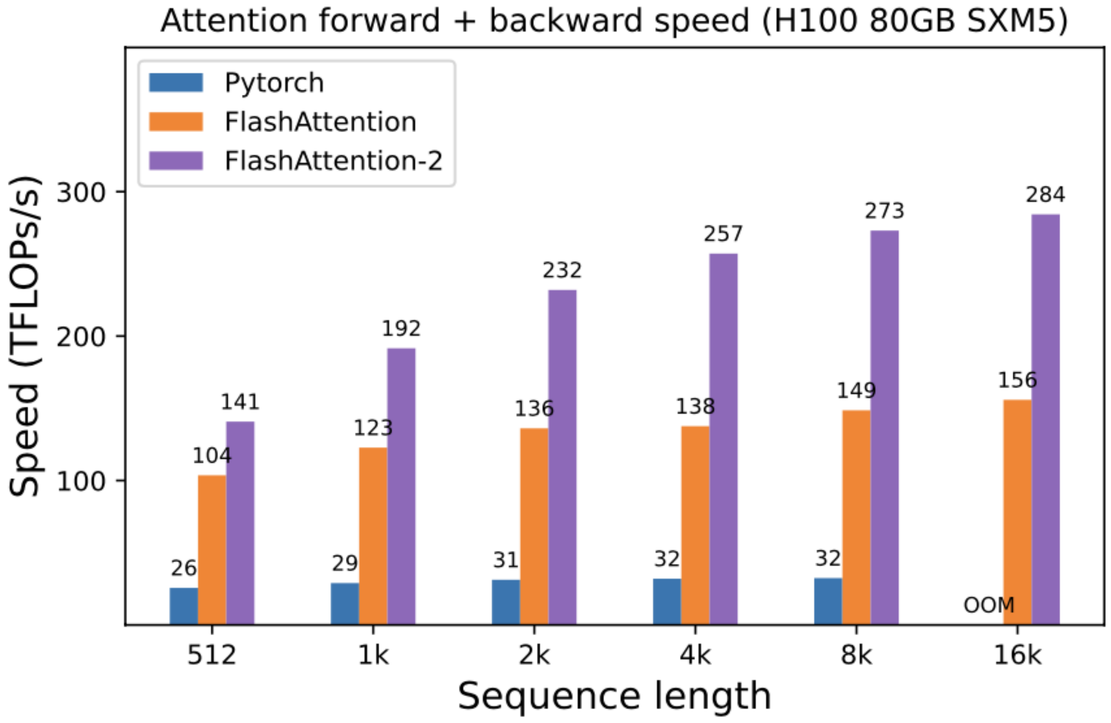
FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness [Dao *et al.*, 2022]

FlashAttention

- Training acceleration results on NVIDIA H100



(a) Without causal mask, head dimension 64



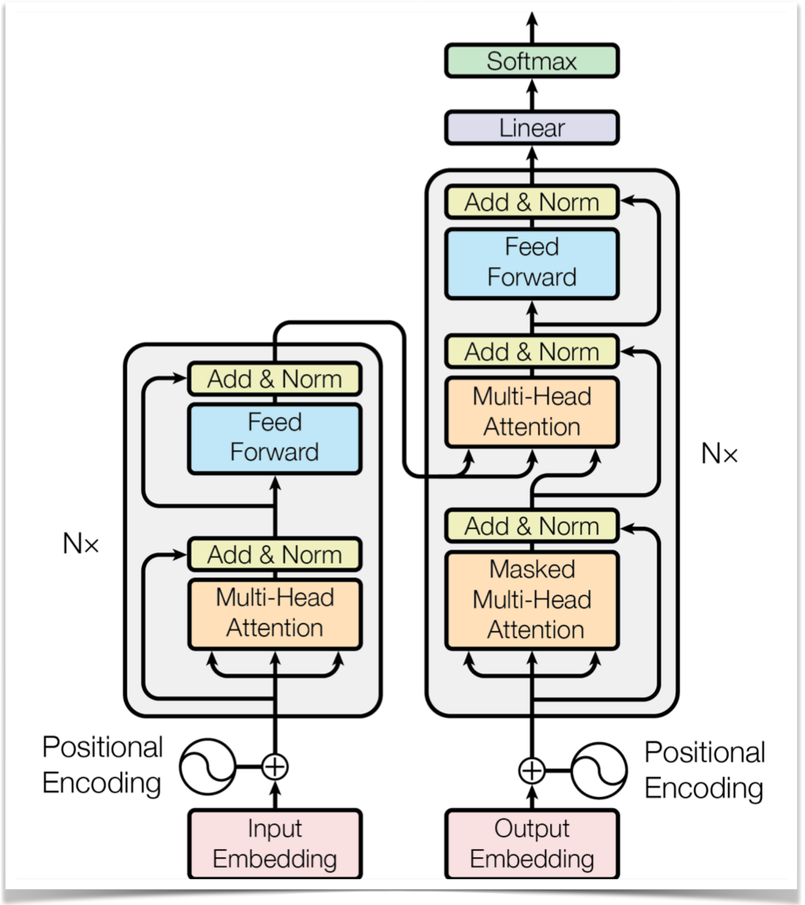
(b) With causal mask, head dimension 64

FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness [Dao *et al.*, 2022]

Lecture Plan

Today, we will cover:

1. Quantization
    1. Weight-Activation Quantization: SmoothQuant
    2. Weight-Only Quantization: AWQ and TinyChat
    3. Further Practice: QServe (W4A8KV4)
2. Pruning & Sparsity
    1. Weight Sparsity: Wanda
    2. Contextual Sparsity: DejaVu, MoE
    3. Attention Sparsity: SpAtten, H2O
3. **LLM Serving Systems**
    1. Metrics for LLM Serving
    2. Paged Attention (vLLM)
    3. FlashAttention
    4. Speculative Decoding
    5. Batching



Speculative Decoding

Accelerating memory-bounded generation

- The decoding phase of LLM generates outputs token by token, which is highly memory-bounded (especially at a small batch size)
- There are two models in speculative decoding:
  - *Draft model*: a small LLM (e.g., 7B)
  - *Target model*: a large LLM (e.g., 175B, the one we are trying to accelerate)
- Procedure:
  - The draft model decodes K tokens **autoregressively**
  - Feed the K generated tokens **in parallel** into the target model and get the predicted probabilities on each location
  - Decide if we want to keep the K tokens or reject them

Leviathan et al., Fast Inference from Transformers via Speculative Decoding, 2023

Speculative Decoding

Accelerating memory-bounded generation

- From generator to verifier

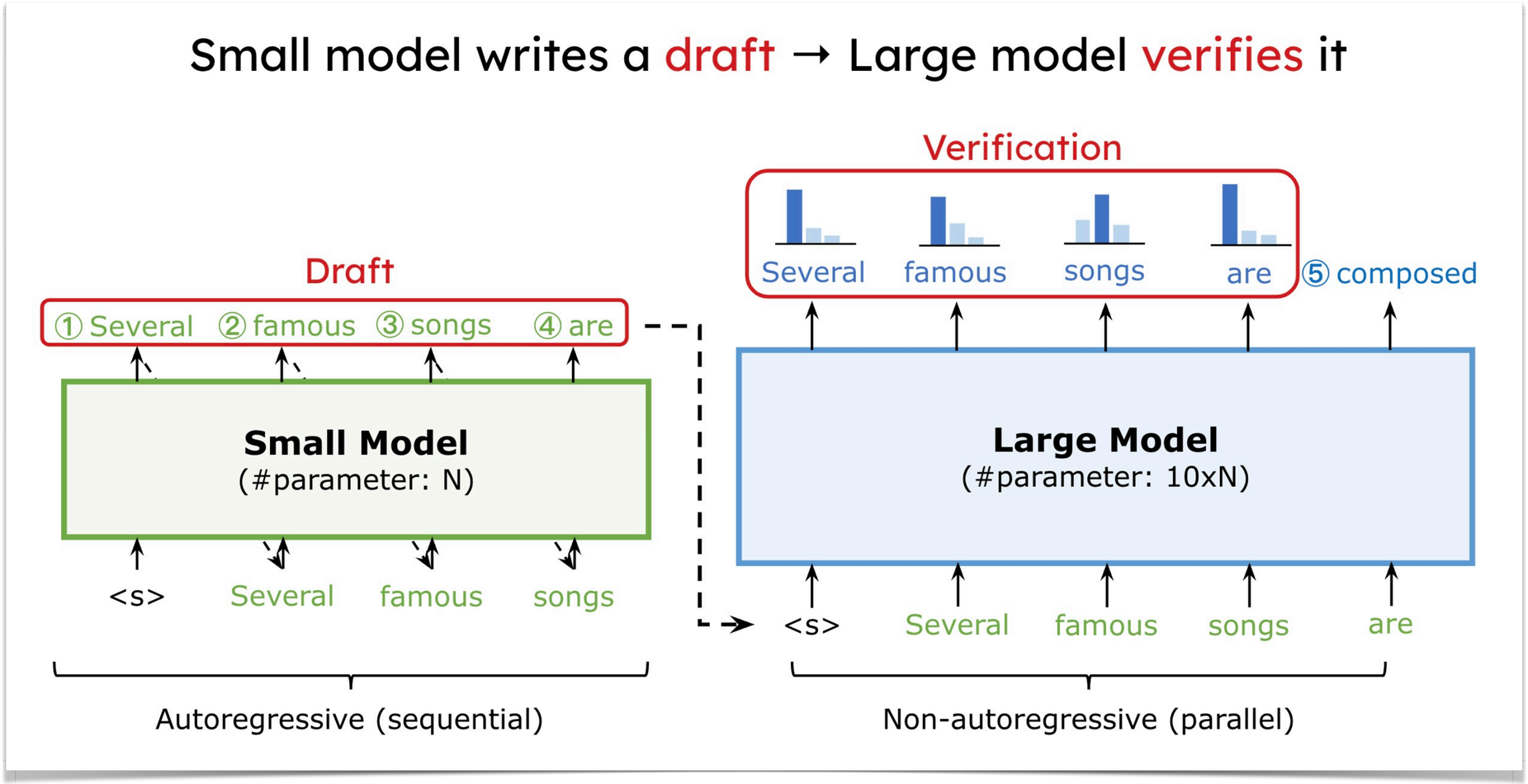
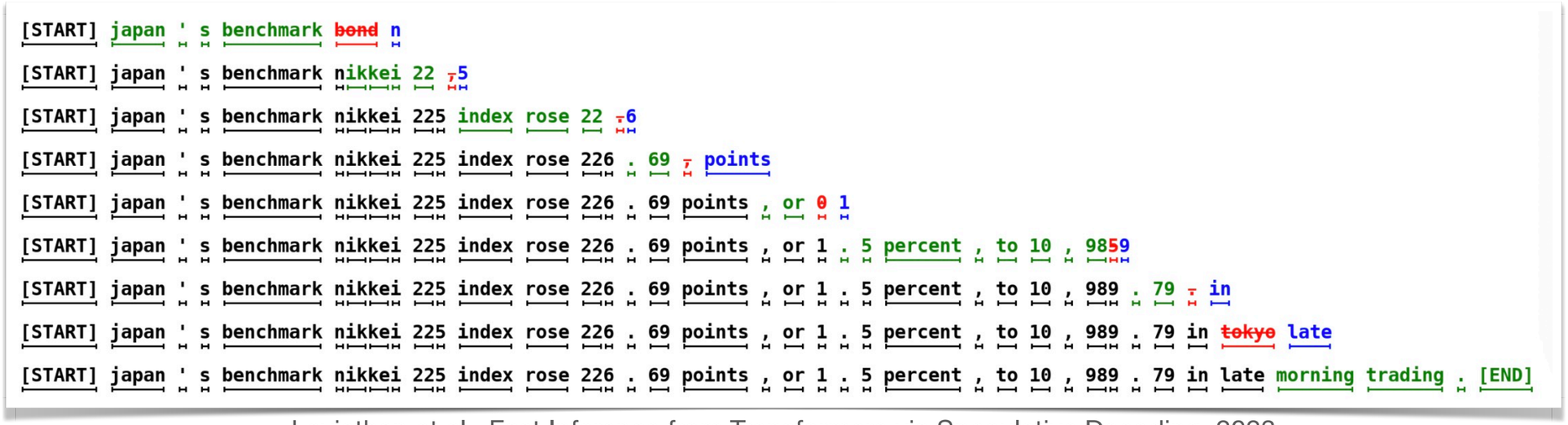


Image credit: vLLM First SF Meetup Slides

Speculative Decoding

Accelerating memory-bounded generation

- The decoding phase of LLM generates outputs token by token, which is highly memory-bounded (especially at a small batch size)
- Since multiple tokens are fed to the target model in parallel, it lifts the memory bottleneck
- An example from speculative decoding
  - green: accepted, red: rejected, blue: correction
- 2-3x speed up with **identical** output

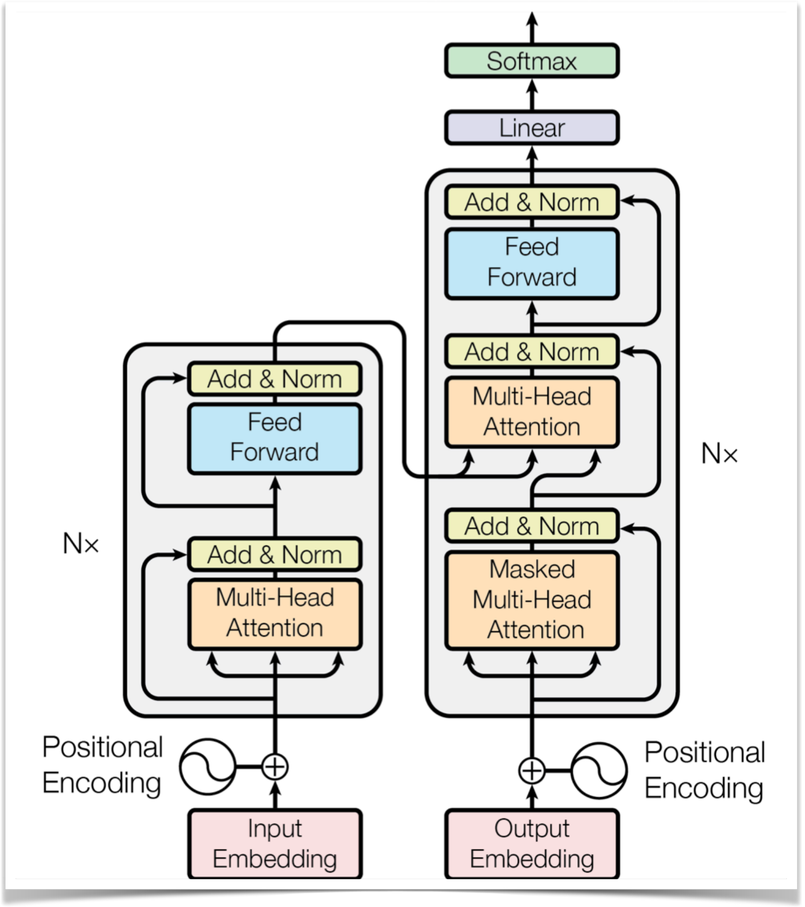


Leviathan et al., Fast Inference from Transformers via Speculative Decoding, 2023

Lecture Plan

Today, we will cover:

1. Quantization
    1. Weight-Activation Quantization: SmoothQuant
    2. Weight-Only Quantization: AWQ and TinyChat
    3. Further Practice: QServe (W4A8KV4)
2. Pruning & Sparsity
    1. Weight Sparsity: Wanda
    2. Contextual Sparsity: DejaVu, MoE
    3. Attention Sparsity: SpAtten, H_2O
3. **LLM Serving Systems**
    1. Metrics for LLM Serving
    2. Paged Attention (vLLM)
    3. FlashAttention
    4. Speculative Decoding
    5. Batching



Batching

Maximizing Throughput of LLM Serving

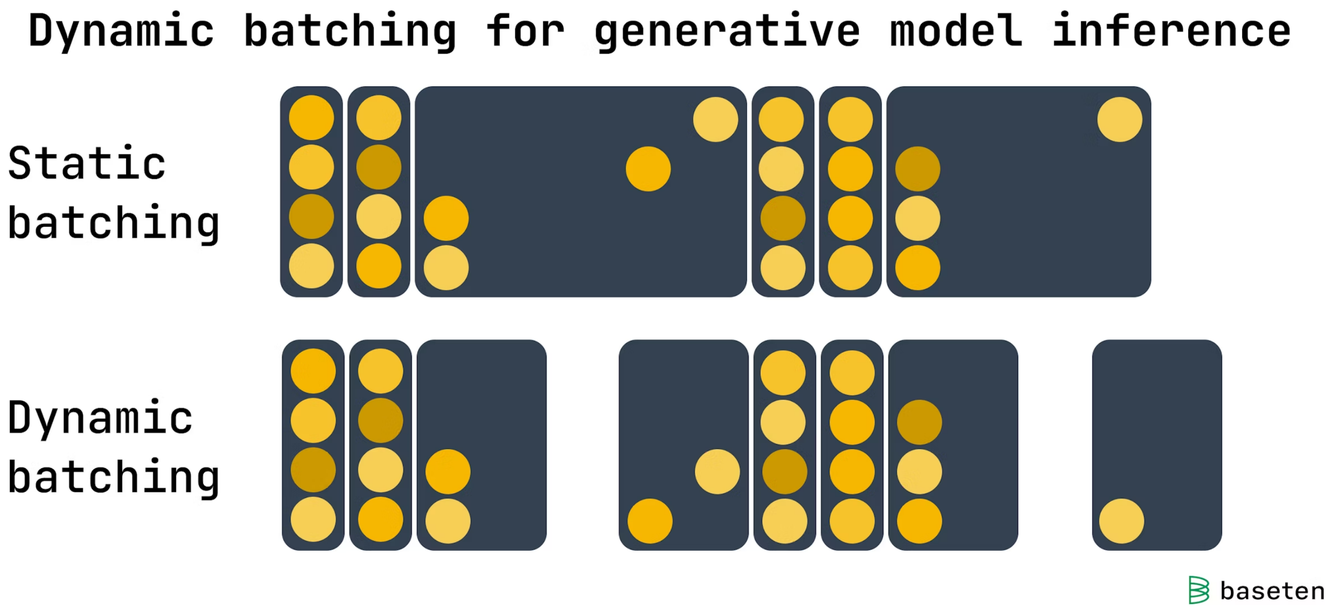
- **Why Batching?:** Running multiple inputs simultaneously maximizes throughput by fully utilizing the GPU resources, avoiding idle time.
- **Batching Methods:**
  - **No Batching:** Each request processed individually, leading to underutilization of GPU resources.
  - **Static Batching:** Waits for a full batch of requests before processing. Good for scheduled tasks (can be processed offline); increases the latency for online tasks.
  - **Dynamic Batching:** Batches are processed when full or after a set time delay to balance throughput and latency.
  - **Continuous Batching (a.k.a., In-Flight Batching):** Processes requests token-by-token, ideal for LLMs, improving GPU utilization by eliminating idle time waiting for the longest response.

https://www.baseten.co/blog/continuous-vs-dynamic-batching-for-ai-inference/

Dynamic Batching

Flexibility in High Traffic

- **Requests are collected and processed once the batch is full or maximum time has elapsed.**
- **Suitable for models like Stable Diffusion where latency matters.**
- **Analogy:** A bus that leaves when full or after a timer expires, ensuring minimal wait time.
- **Ideal for:** Generative models with uniform inference latency, balancing throughput and latency.

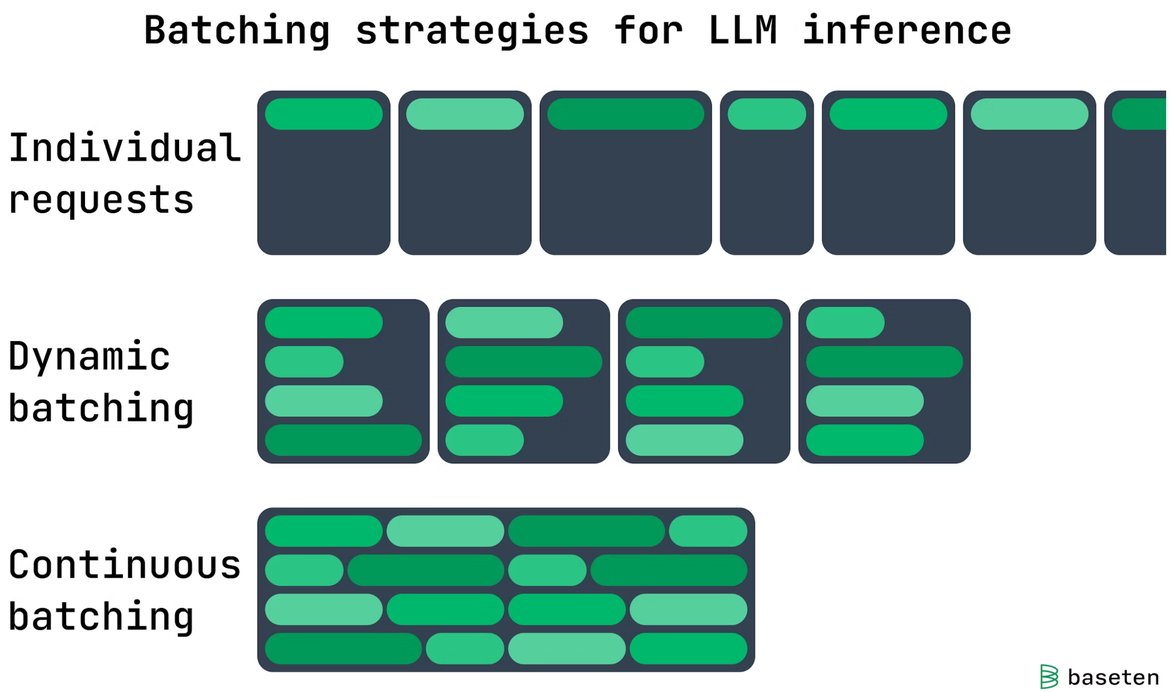


https://www.baseten.co/blog/continuous-vs-dynamic-batching-for-ai-inference/

Continuous Batching (In-Flight Batching)

Optimized for LLM (Token-by-Token Generative Model) Serving

- **Works token-by-token, processing new requests as previous ones finish.**
- **Maximizes GPU efficiency by avoiding idle time while waiting for the longest response.**
- **Analogy:** A bus where passengers get off at different stops, making space for new riders.
- **Ideal for:** LLMs with varying output lengths, optimizing next-token generation.



https://www.baseten.co/blog/continuous-vs-dynamic-batching-for-ai-inference/

TensorRT-LLM Public Release

State-of-the-art LLM serving infra from NVIDIA

Key Features

TensorRT-LLM contains examples that implement the following features.

- Multi-head Attention(MHA)
- Multi-query Attention (MQA)
- Group-query Attention(GQA)
- In-flight Batching
- Paged KV Cache for the Attention
- Tensor Parallelism
- Pipeline Parallelism
- INT4/INT8 Weight-Only Quantization (W4A16 & W8A16)
- SmoothQuant
- GPTQ
- AWQ
- FP8
- Greedy-search
- Beam-search
- RoPE

TensorRT-LLM

A TensorRT Toolbox for Large Language Models

docs latest | python 3.10.12 | cuda 12.2 | TRT 9.1 | release 0.5.0 | license Apache 2

Architecture | Results | Examples | Documentations

covered in the lecture

will introduce in future lectures

https://github.com/NVIDIA/TensorRT-LLM

Summary of Today's Lecture

1. **Quantization**
    1. Weight-Activation Quantization: SmoothQuant
    2. Weight-Only Quantization: AWQ and TinyChat
    3. Further Practice: QServe (W4A8KV4)
2. **Pruning & Sparsity**
    1. Weight Sparsity: Wanda
    2. Contextual Sparsity: DejaVu, MoE
    3. Attention Sparsity: SpAtten, H_2O
3. **LLM Serving Systems**
    1. Metrics for LLM Serving
    2. Paged Attention (vLLM)
    3. FlashAttention
    4. Speculative Decoding
    5. Batching