SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models
ICML 2023

Authors: Guangxuan Xiao (Massachusetts Institute of Technology), Ji Lin (Massachusetts Institute of Technology), Mickael Seznec (NVIDIA), Hao Wu (NVIDIA), Julien Demouth (NVIDIA), Song Han (Massachusetts Institute of Technology)

Presented by: Aditi Tiwari (aditit5)

tldr

The core of SmoothQuant is a mathematical transformation that redistributes the difficulty of quantization between **activations** and **weights** in large language models (LLMs). The motivation behind this approach is that activations, especially in LLMs, often contain **outliers**—values that can be up to 100 times larger than typical values—making them challenging to quantize directly without losing accuracy. In contrast, weights are usually more evenly distributed and easier to quantize. SmoothQuant works by shifting some of this quantization "difficulty" from the activations to the weights, allowing for a more balanced and effective quantization process

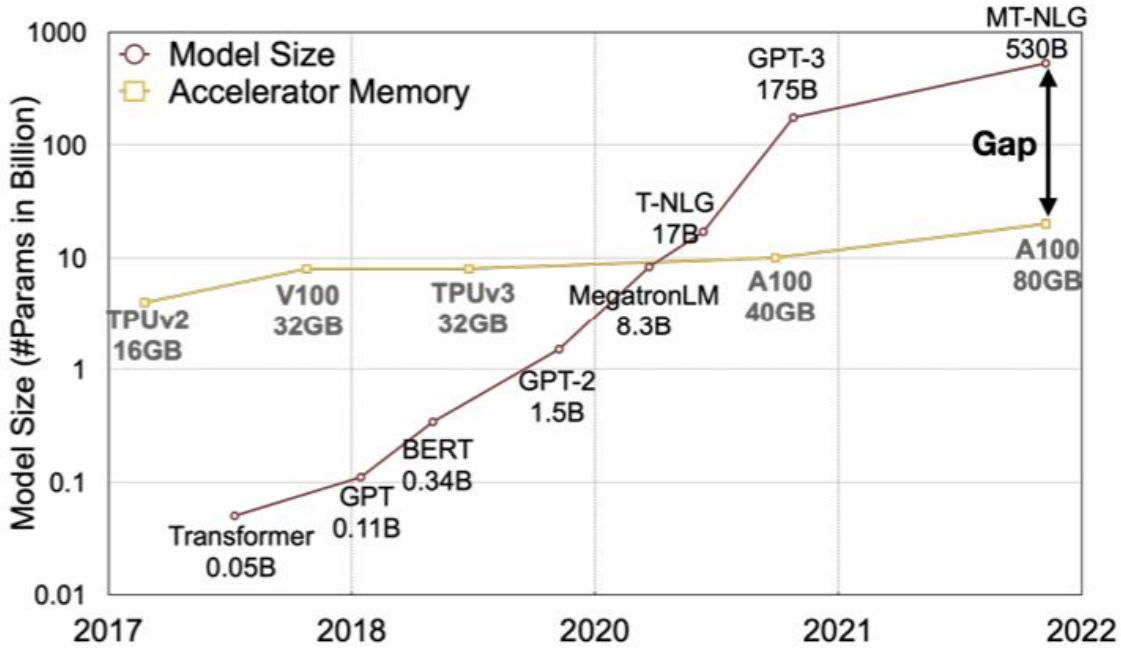
Introduction

- Large language models (LLMs) are taking over every field.

- As the models get larger, serving such models for inference becomes **expensive** and **challenging**!

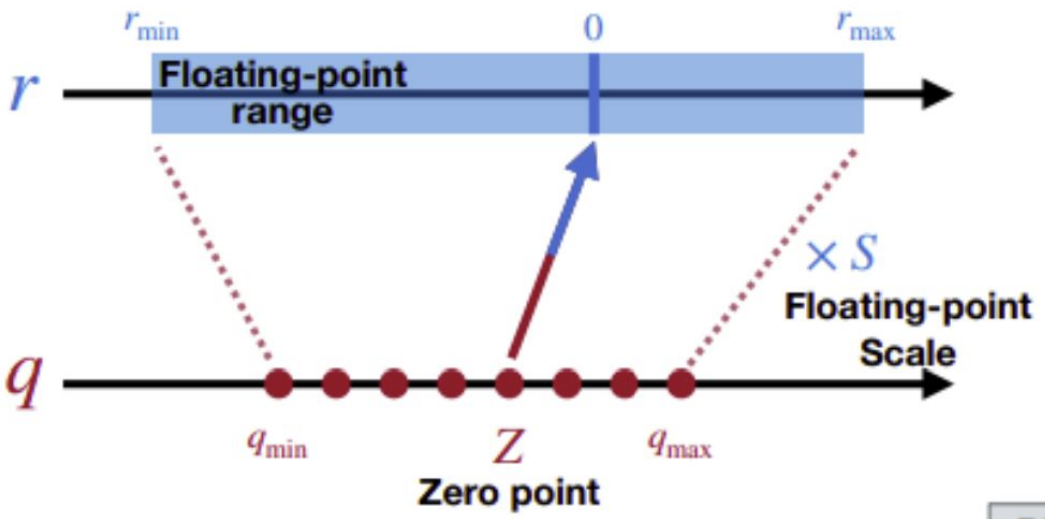
Introduction

- Size of LLMs is developing faster than GPU memory -> creates a big bag between supply and demand
- Thereby creating a need for quantization and model compression techniques
- LLMs often come in high precision formats such as FP16
  - Significant GPU memory requirements (size and bandwidth)
  - Slow matrix multiplication operations



Quantization

- **Lowers the bit width** and improves the **efficiency**
- Its the process of converting a **high-precision** value (e.g., FP16 or FP32) into a **lower-precision** representation (e.g., INT8).
- Reduces **memory usage** and **computation costs** while maintaining performance, especially during inference.



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

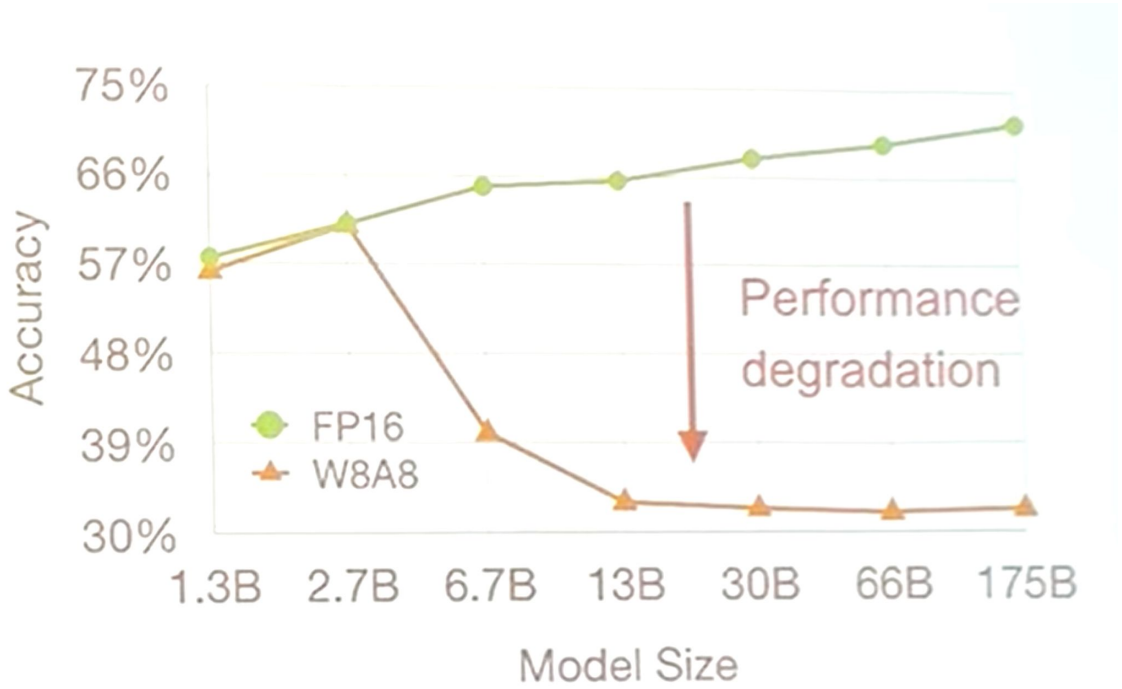
Post-Training Quantization (PTQ)

- Technique applied **after the model is fully trained**.
- Doesn’t require retraining, making it efficient for deployment.
- No modifications to training -> easy to implement and wide applicability
- Reduces the cost of LLMs.
- Mitigates memory consumption and reduce computational overhead => higher performance

Trade-offs: Traditional PTQ often leads to **accuracy loss**, particularly in large models where activation outliers are prominent.

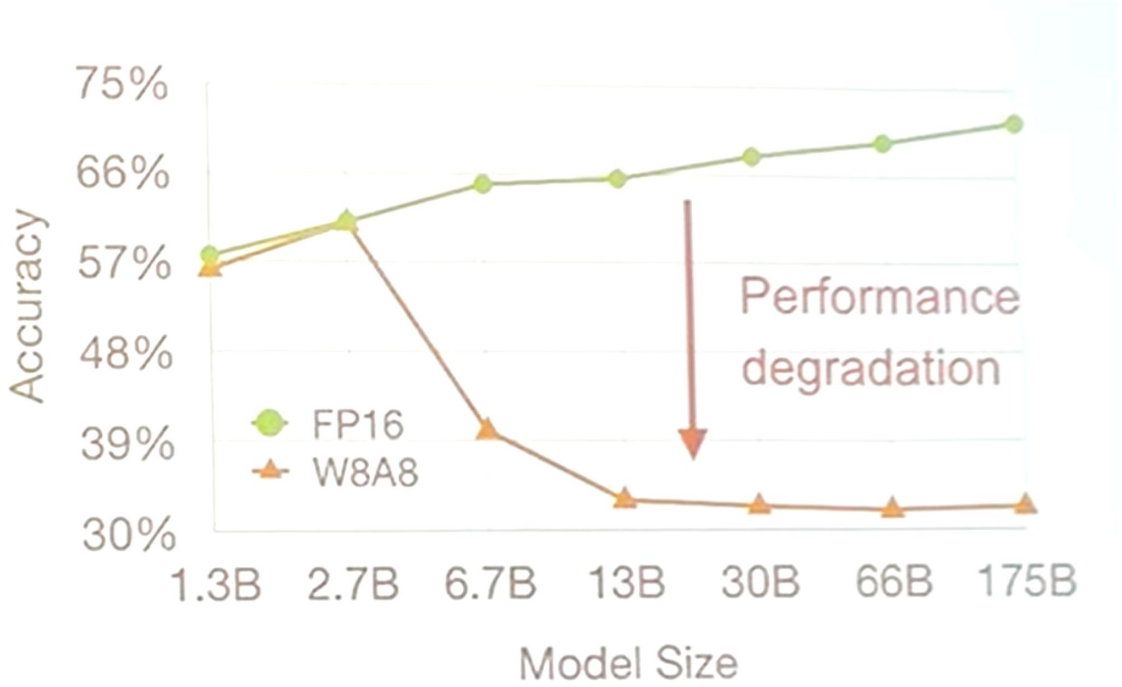
Can we directly apply the techniques we have learnt to LLMs?

- When model gets larger than **6B parameters**, performance **degradation** is pretty severe.
- This is because LLMs have **outliers** in the **activation -> difficult to quantize**



W8A8 quantization has been an industrial standard for CNNs, but not LLM. Why?

- W8A8 quantization format where both weights and activations are represented using 8-bit integers
- Systematic outliers emerge in activations when we scale up LLMs beyond 6.7B. Traditional CNN quantization methods will **destroy the accuracy.**



Motivation

1. **High Memory and Compute Costs for LLMs**:
2. **Increased Costs as Models Scale**:
3. **Need for Efficiency Without Accuracy Loss**:

Motivation

1. **High Memory and Compute Costs for LLMs**:
   - LLMs like GPT-3, BLOOM, and MT-NLG are highly accurate but resource-intensive.
   - Example: **GPT-3** with **175B** parameters requires at least **350GB** memory in **FP16**, demanding expensive hardware setups (e.g., 8x48GB A6000 GPUs or 5x80GB A100 GPUs for inference).
2. **Increased Costs as Models Scale**:
3. **Need for Efficiency Without Accuracy Loss**:

Motivation

1. **High Memory and Compute Costs for LLMs**:
2. **Increased Costs as Models Scale**:
    - Scaling up model size (e.g., GPT-3 to MT-NLG 530B) leads to an exponential increase in **memory usage** and **inference time**.
    - **Bottleneck**: High costs prevent large models from being used widely in real-time applications or edge deployments.
3. **Need for Efficiency Without Accuracy Loss**:

Motivation

1. **High Memory and Compute Costs for LLMs**:
2. **Increased Costs as Models Scale**:
3. **Need for Efficiency Without Accuracy Loss**:
    - **Quantization** can reduce memory and computation, but current methods (e.g., W8A8, ZeroQuant) degrade accuracy.
    - For models larger than 6.7B parameters, traditional quantization methods struggle due to activation outliers.

Motivation

1. **High Memory and Compute Costs for LLMs**:
2. **Increased Costs as Models Scale**:
3. **Need for Efficiency Without Accuracy Loss**:
    - **Quantization** can reduce memory and computation, but current methods (e.g., W8A8, ZeroQuant) degrade accuracy.
    - For models larger than 6.7B parameters, traditional quantization methods struggle due to activation outliers.

SmoothQuant Goal: Achieve efficient 8-bit quantization without compromising the accuracy, making LLMs more accessible for deployment.

Existing Quantization Methods

- **W8A8 (Weight and Activation Quantization)**:
  - Both weights and activations are quantized to 8-bit (INT8).
  - As model size increases (e.g., beyond 6.7B parameters), accuracy degrades significantly because it cannot handle **activation outliers** effectively.

- **ZeroQuant**:
  - **Dynamic quantization** method that adjusts precision for activations during inference, improving accuracy.
  - It **works for smaller models** (delivers good accuracy for **GPT-3-350M** and **GPT-J-6B**)
  - ZeroQuant struggles with larger LLMs (can **not maintain the accuracy for the large OPT model** with 175 billion parameters) due to its inability to handle extreme activation outliers.
  - Uses layer-by-layer knowledge distillation without the original training data

- **LLM.int8()**:
  - **Mixed-precision technique** that keeps activations in **FP16** while quantizing weights to **INT8**.
  - **Increases accuracy** by keeping outliers in FP16 and uses INT8 for the other activations
  - This approach is **inefficient** in terms of **hardware utilization**, as it requires complex data precision switching during inference.

Challenges with Current Methods

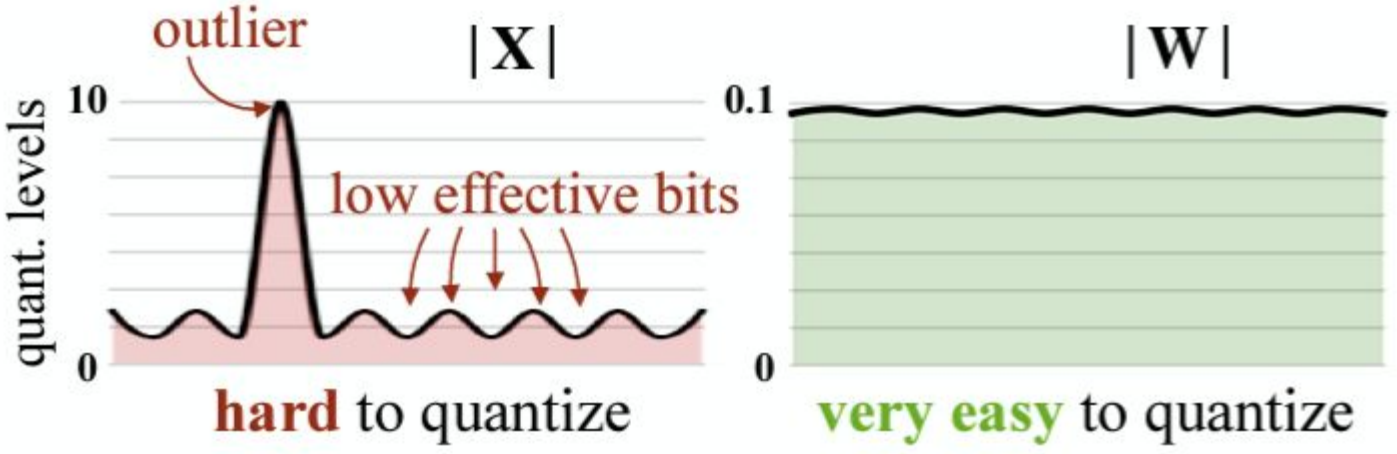
- **Difficulty in Activation Quantization**:
  - Weights -> easier to quantize -> relatively uniform distribution
  - Activations -> contain outliers -> harder to reduce to INT8 w/o significant accuracy loss.

- **Existing Methods' Performance on Large Models**:
  - Current methods (W8A8 and ZeroQuant) **degrade accuracy** as model size increases, particularly for models > **6.7B parameters**.

Weights and Activations

Weights (W) :

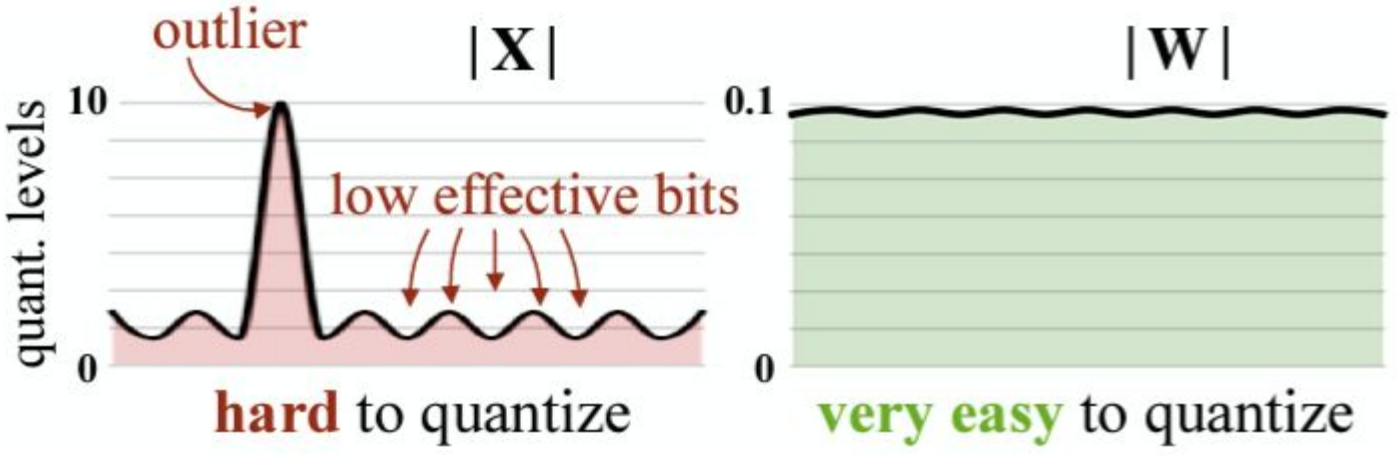
- Learned parameters of the model, fixed during inference
- Easier to quantize -> more evenly distributed without extreme outliers.



Weights and Activations

Activations (X):

- Dynamic outputs of each layer during inference
- More challenging to quantize -> outliers.
- Outliers significantly stretch the range of activation values, reducing the effectiveness of quantization and leading to quantization errors if not handled properly.



Quantization Process

$$\bar{\mathbf{X}}^{\text{INT8}} = \left\lceil \frac{\mathbf{X}^{\text{FP16}}}{\Delta} \right\rfloor, \quad \Delta = \frac{\max(|\mathbf{X}|)}{2^{N-1} - 1}$$

Where:

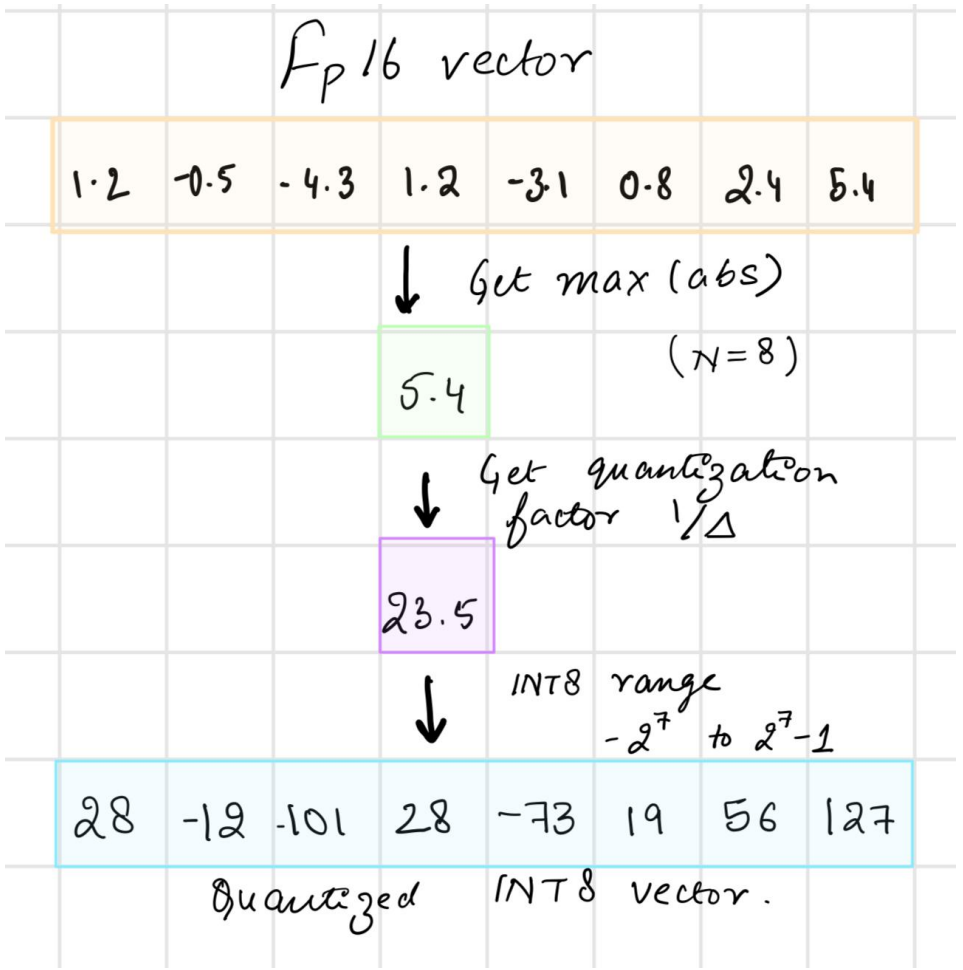
X -> floating-point tensor

$\bar{X}$ -> quantized counterpart

Δ -> quantization step size

$\lceil \cdot \rfloor$-> rounding function

N-> Number Of Bits(8 in our case)



Quantization Process

$$\bar{\mathbf{X}}^{\mathrm{INT8}} = \left\lceil \frac{\mathbf{X}^{\mathrm{FP16}}}{\Delta} \right\rfloor, \quad \Delta = \frac{\max(|\mathbf{X}|)}{2^{N-1} - 1}$$

Where:

X -> floating-point tensor

$\bar{X}$ -> quantized counterpart

Δ -> quantization step size

$\lceil \cdot \rfloor$-> rounding function

N-> Number Of Bits(8inourcase)

How to get Δ?

- Dynamic Range Quantization
  - At runtime
  - Use the runtime statistics of activations to get Δ
- Static Quantization
  - Before runtime
  - Calculate Δ offline with the activations of some calibration samples

SmoothQuant uses a **static approach**, where it collects activation statistics from **512 random sentences** sampled from the **pre-training dataset (Pile)**. These statistics are then used to calculate the **per-channel scaling factors** that redistribute the quantization difficulty from activations to weights.

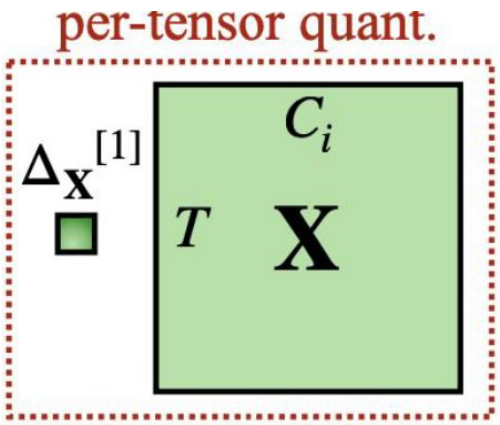
Quantization granularity

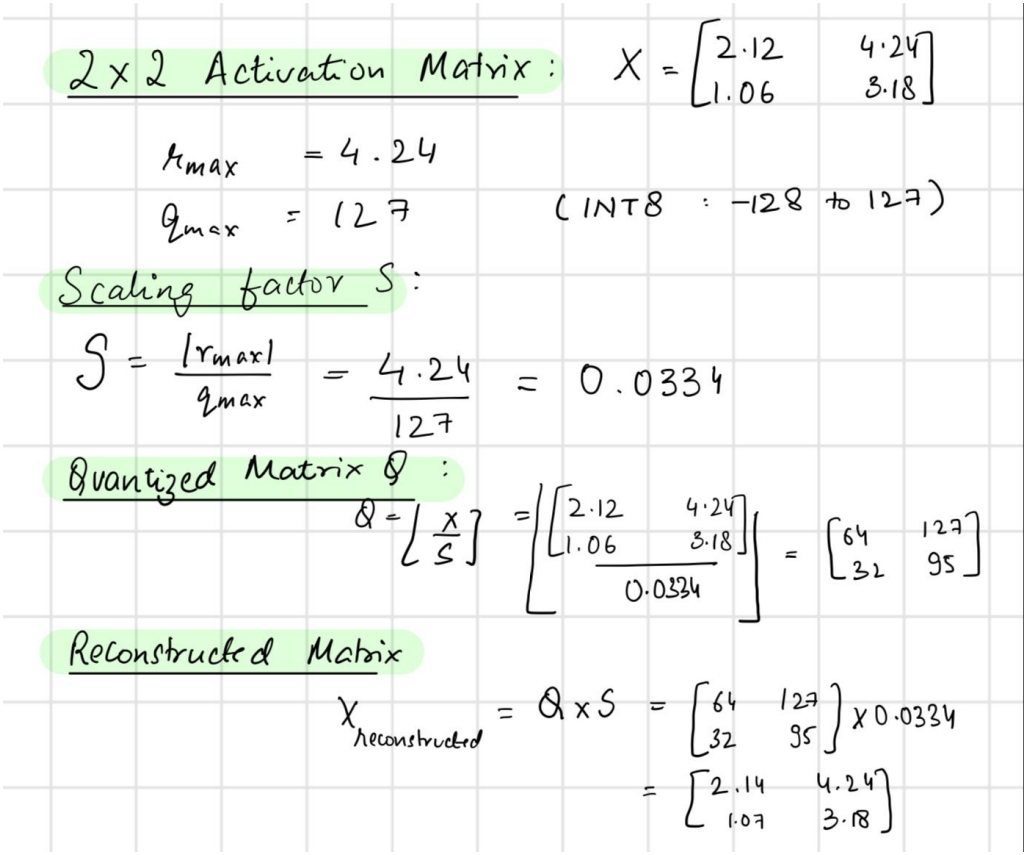
- Refers to the **level of detail** at which quantization is applied to a model's weights or activations.

Granularity	Scaling Factor Applied To	Benefits
Per-tensor	Entire matrix	Simple and efficient
Per-channel	Each column (channel)	More precise across channels
Per-token	Each row (token)	Adapts to variability across tokens
Group-wise	Groups of rows or columns	Balances precision and efficiency

Quantization granularity

- Per-Tensor Quantization
  - single scaling factor is applied to the entire matrix, meaning all elements share the same quantization scale.
- Per-Token Quantization
- Per-Channel Quantization
- Group-Wise Quantization

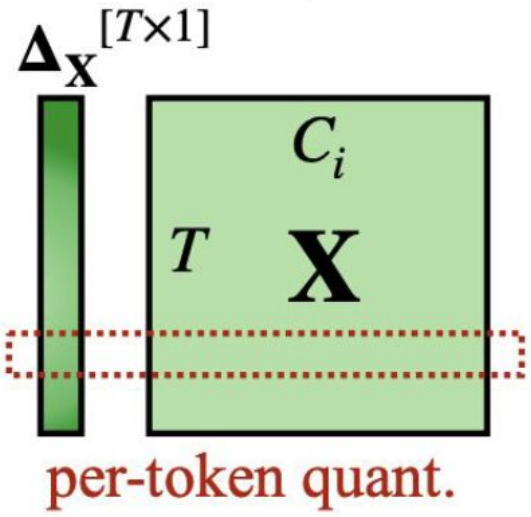




close to the original matrix but may have small errors due to the precision loss in quantization.

Quantization granularity

- Per-Tensor Quantization
- Per-Token Quantization
  - Each row (token) gets its own scaling factor.
- Per-Channel Quantization
- Group-Wise Quantization



Activation :

$$X = \begin{bmatrix} 2.12 & 4.24 \\ 1.06 & 3.18 \end{bmatrix}$$

Row wise Scaling Factor :

$$S_{row_1} = \frac{4.24}{127} = 0.0334$$

$$S_{row_2} = \frac{3.18}{127} = 0.0250$$

Quantized Matrix Q :

$$Q = \left\lfloor \frac{X}{S_{row}} \right\rceil = \begin{bmatrix} \frac{2.12}{S_{row1}} & \frac{4.24}{S_{row1}} \\ \frac{1.06}{S_{row2}} & \frac{3.18}{S_{row2}} \end{bmatrix}$$

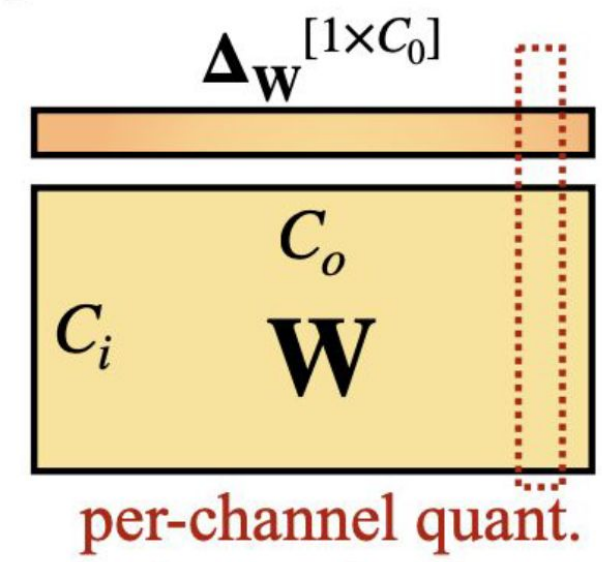
$$Q = \begin{bmatrix} 64 & 127 \\ 42 & 127 \end{bmatrix}$$

Reconstructed Matrix :

$$X_{reconstructed} = Q \times S_{row} = \begin{bmatrix} 2.14 & 4.24 \\ 1.05 & 3.18 \end{bmatrix}$$

Quantization granularity

- Per-Tensor Quantization
- Per-Token Quantization
- Per-Channel Quantization
  - Each column (channel) of the matrix gets its own scaling factor
- Group-Wise Quantization



captures variability between columns more effectively, leading to more accurate reconstructions.

Activation :

$$X = \begin{bmatrix} 2.12 & 4.24 \\ 1.06 & 3.18 \end{bmatrix}$$

Column 1 Scaling Factor :

$$S_{c_1} = \frac{2.12}{127} = 0.0167$$

Column 2 Scaling Factor :

$$S_{c_2} = \frac{4.24}{127} = 0.0334$$

Quantized Matrix Q :

$$Q = \left\lfloor \frac{X}{S_{column}} \right\rfloor = \begin{bmatrix} \frac{2.12}{0.0167} & \frac{4.24}{0.0334} \\ \frac{1.06}{0.0167} & \frac{3.18}{0.0334} \end{bmatrix}$$

$$Q = \begin{bmatrix} 127 & 127 \\ 63 & 95 \end{bmatrix}$$

Reconstructed Matrix :

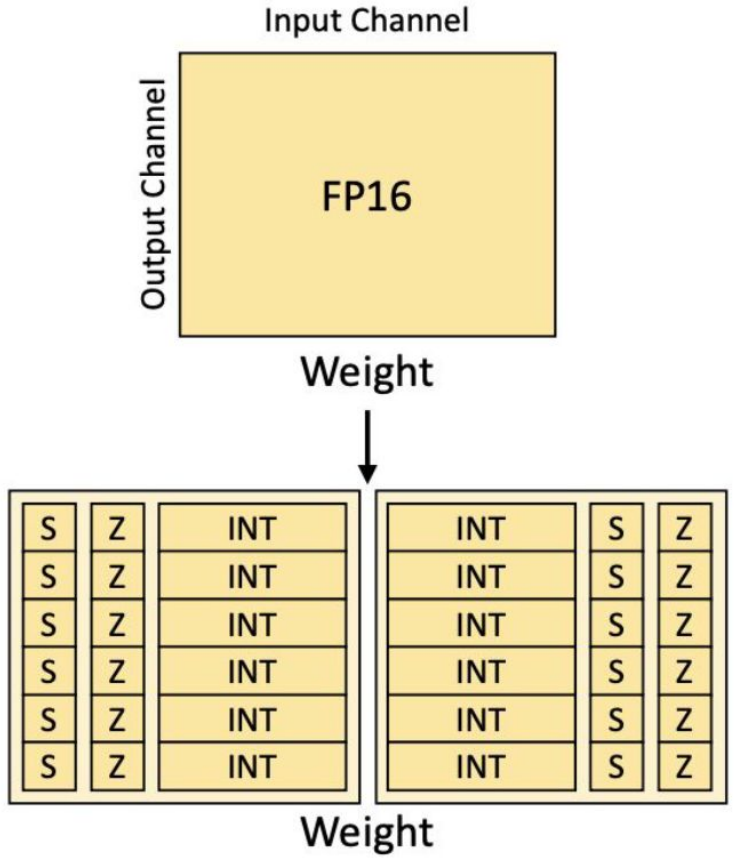
$$X_{reconstructed} = Q \times S_{column}$$

$$= \begin{bmatrix} 127 \times S_{c_1} & 127 \times S_{c_2} \\ 63 \times S_{c_1} & 95 \times S_{c_2} \end{bmatrix}$$

$$= \begin{bmatrix} 2.12 & 4.24 \\ 1.05 & 318 \end{bmatrix}$$

Quantization granularity

- Per-Tensor Quantization
- Per-Token Quantization
- Per-Channel Quantization
- Group-Wise Quantization
  - Groups of values (e.g., multiple rows or columns) share a scaling factor.



Per-Channel Quantization is Infeasible

Observations:

- Outliers lead to low effective quantization bits
- Outliers exist in a small fraction of channels
- Per-channel quantization seems like a **potential solution** - to **reduce quantization errors** as each channel would have its own scaling factor

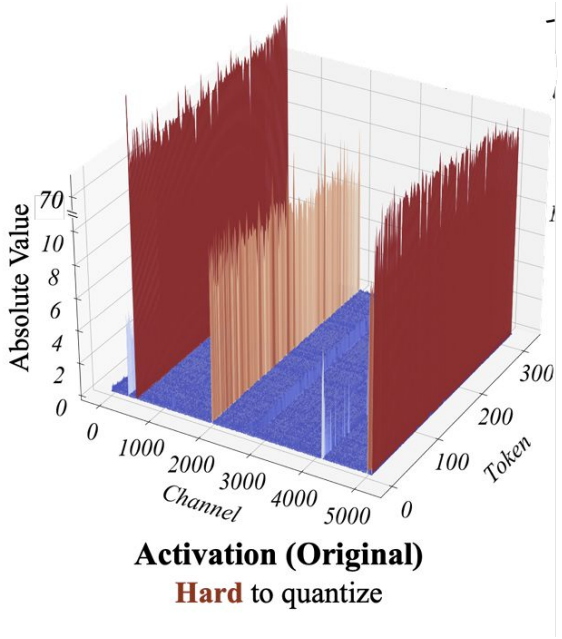
Issue with Per-Channel Quantization:

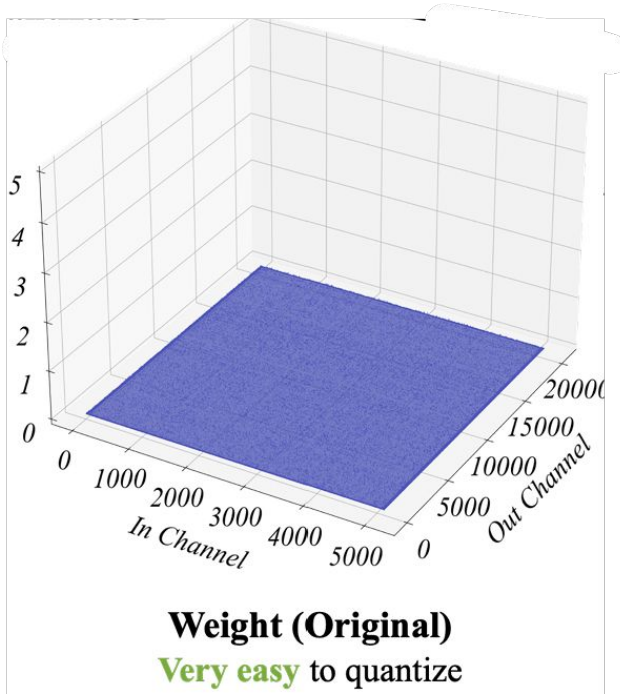
- Hardware-accelerated **GEMM kernels** are optimized for **high-throughput, parallel operations** using **INT8 data** with a **single scaling factor** for the entire tensor (per-tensor quantization).
- **Per-channel quantization** -> **additional instructions** to apply different scaling factors for each channel -> **disrupts the vectorized operations** of GEMM kernels -> significant **drop in performance**.
- GEMM kernels -> do not tolerate the insertion of instructions with **lower throughput,** making per-channel quantization **infeasible** due to its negative impact on inference speed and overall efficiency.

SmoothQuant's Solution:

- Avoids use of per-channel quantization for activations **(Key Idea #1)**
- instead use **mathematically equivalent transformation** to redistribute the quantization difficulty to weights, where it can be handled more effectively by existing hardware.

SmoothQuant

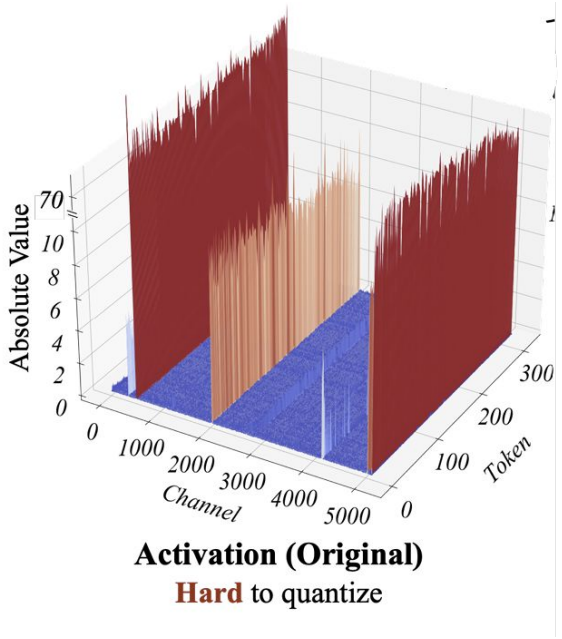


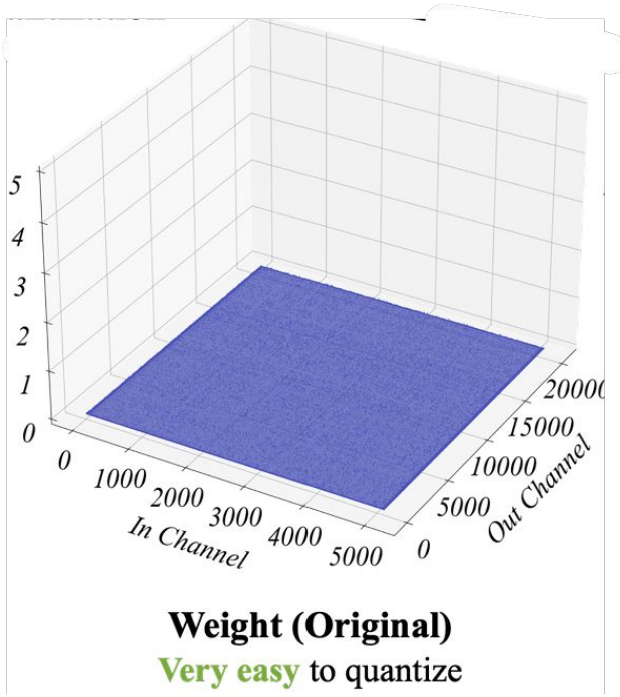


What is the outlier here?

- Activation
- Outliers persist in fixed channels

SmoothQuant

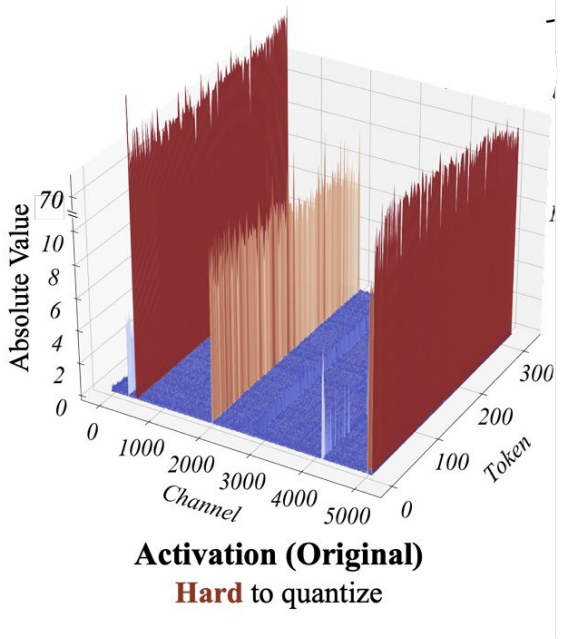


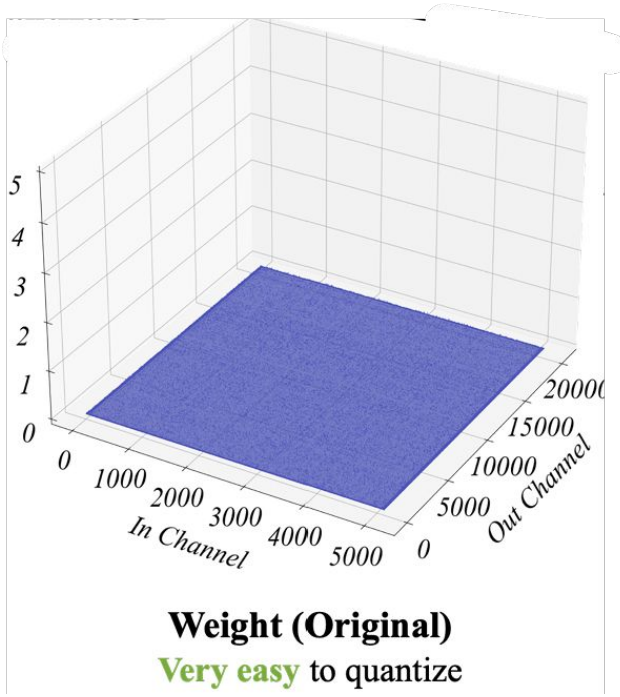


Goal?

- Smoothing activation to reduce quantization error

SmoothQuant

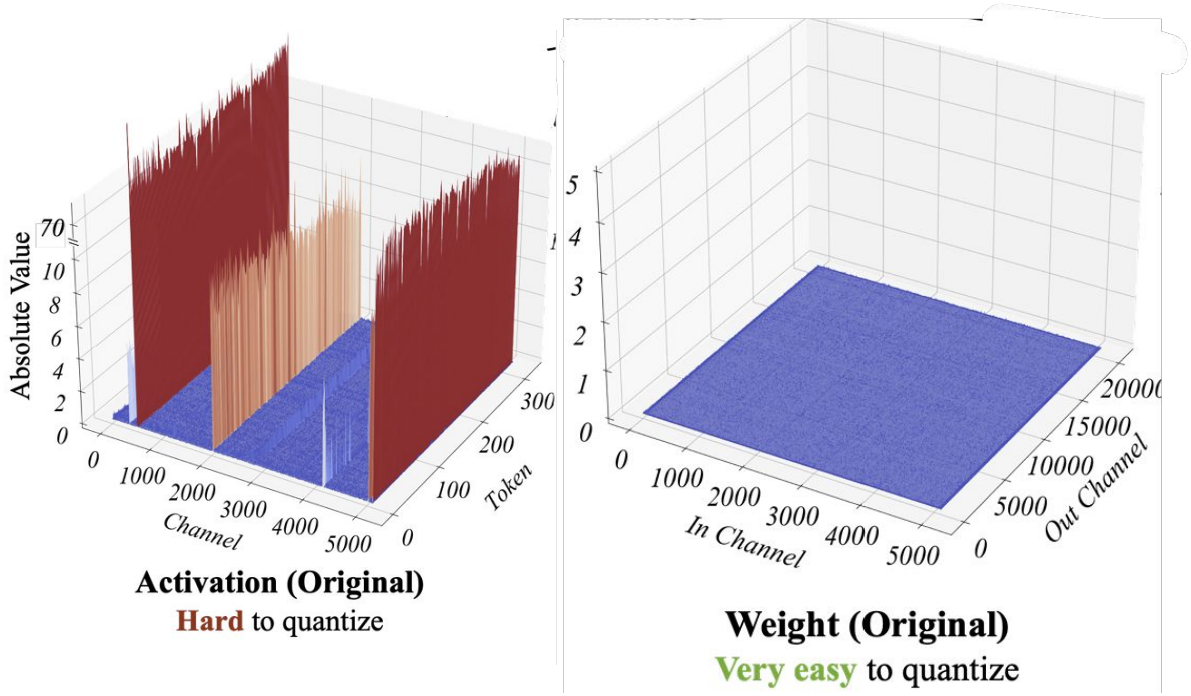




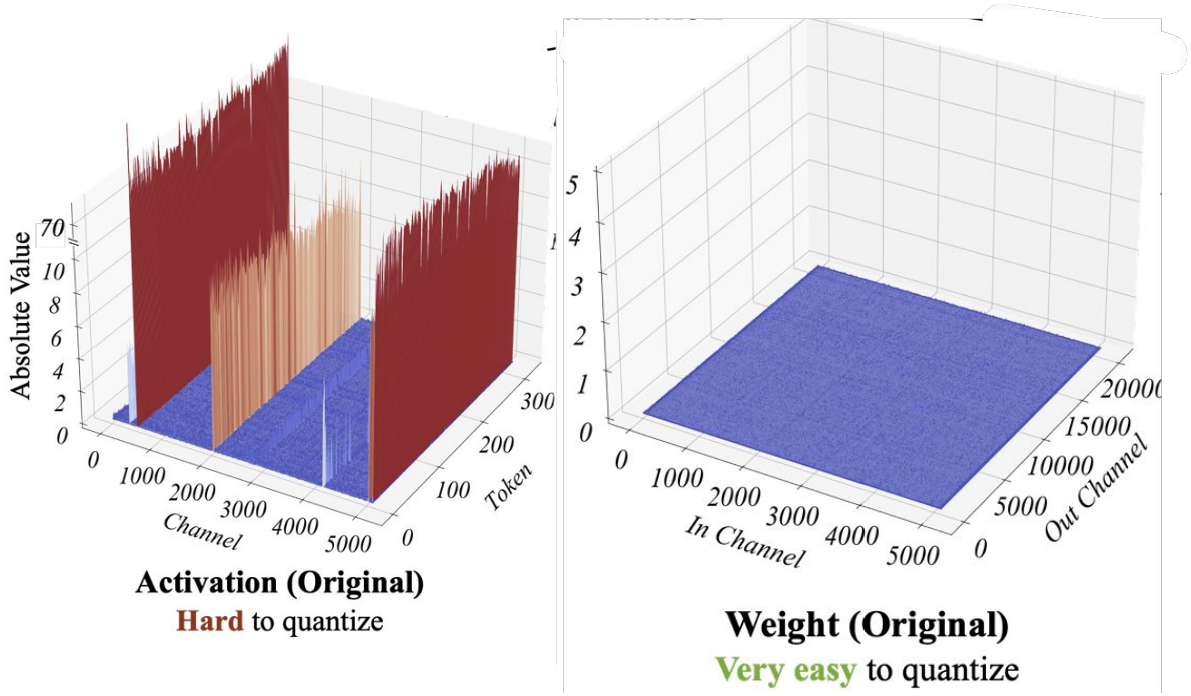
What is the difference b/w the 2 images?

- 1 has lots of outliers.
- 3 channels are much higher in value than the surrounding channels.
- Range is 0-70
- At the same time, 2 is pretty flat

SmoothQuant

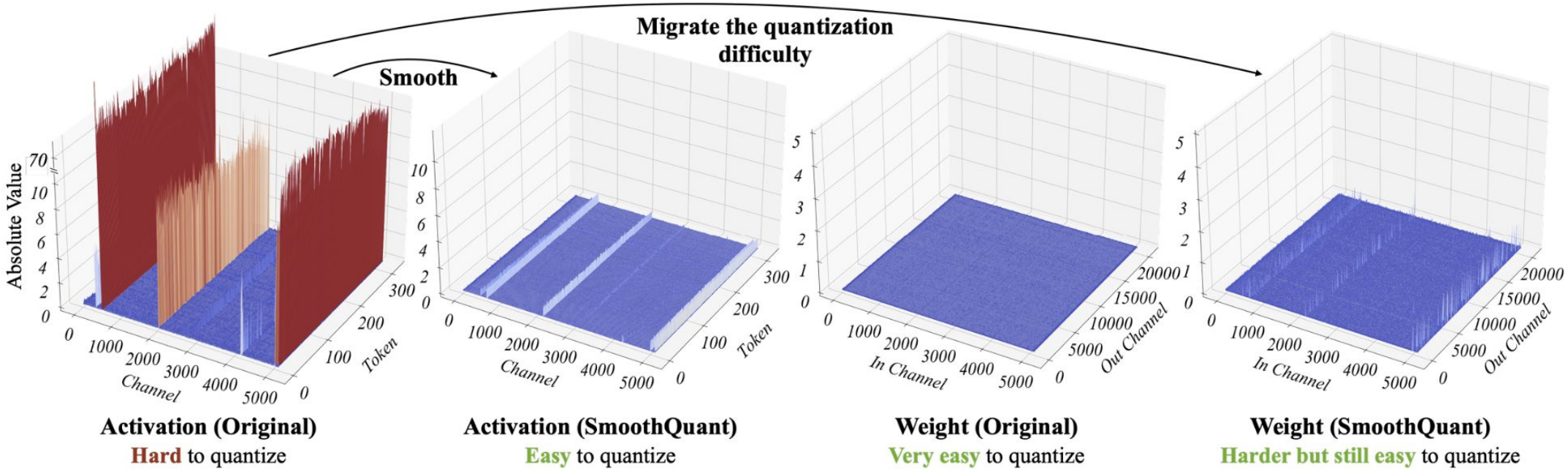


SmoothQuant



$$\mathbf{Y} = \mathbf{XW} = (\mathbf{0.01X})(\mathbf{100W})$$

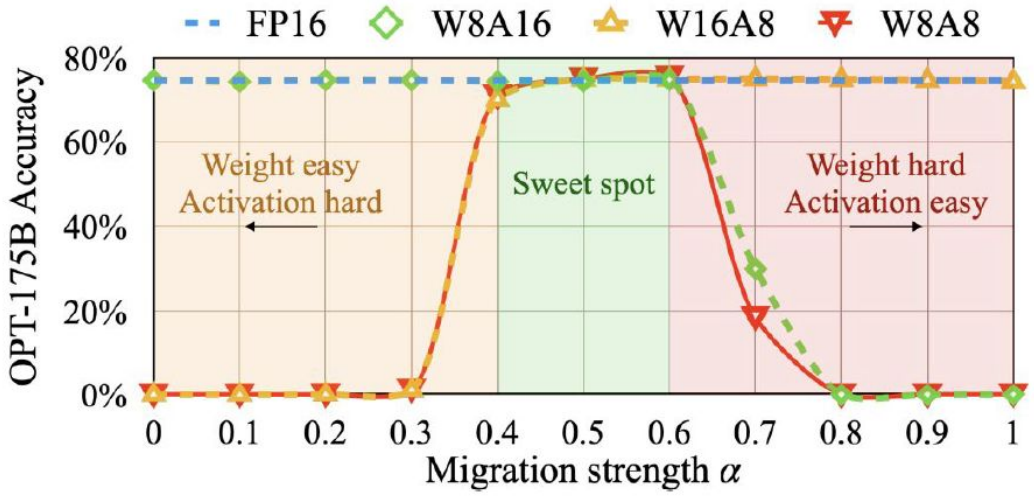
SmoothQuant



Key Idea #2: Migrating the quantization difficulty
Weights are easy to quantize, but activations are hard due to outliers

Alpha (α)

- Hyperparameter which controls the **extent to which quantization difficulty is shifted from activations to weights**.
- α is b/w 0.4 to 0.6, though larger models or models with more significant activation outliers may require higher values.

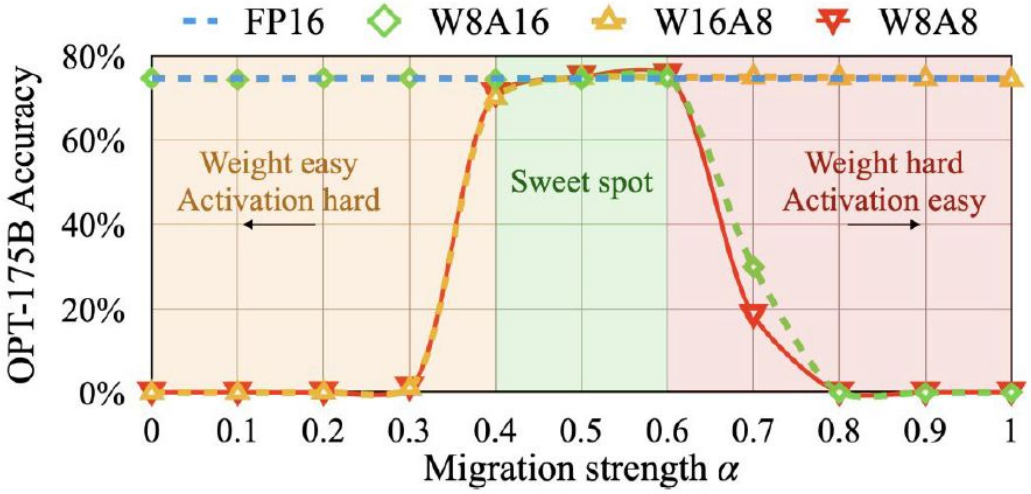


Choosing α

Case-by-case decision

If the α is too **large**, **weights will be hard to quantize**. If too **small**, **activations will be hard to quantize**.

Goal: make activations and weights both easy to quantize.



SmoothQuant Scaling Factor (s)

s is calculated based on the **maximum values** in the activation and weight channels

$$\mathbf{s}_j = \max(|\mathbf{X}_j|)^{\alpha} / \max(|\mathbf{W}_j|)^{1-\alpha}$$

Where:
X -> activation values
W -> weight values
α -> hyperparameter (controls **extent to which quantization difficulty is shifted from activations to weights**)

SmoothQuant Example

X

1	-16	2	6
-2	8	-1	-9

*

2	1	-2
1	-1	-1
2	-1	-2
-1	-1	1

W

X-> 2 outlier channels

Obtain s -> divide square root (α = 0.5) of max of X by max of W

W is flat

X-> -16 and 8 ; 6 and -9

absolute max(-16,8) -> 16 -> square root of 16 = 4

corresponding value of s -> 4

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

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

SmoothQuant Example

X

1	-16	2	6
-2	8	-1	-9

*

2	1	-2
1	-1	-1
2	-1	-2
-1	-1	1

W

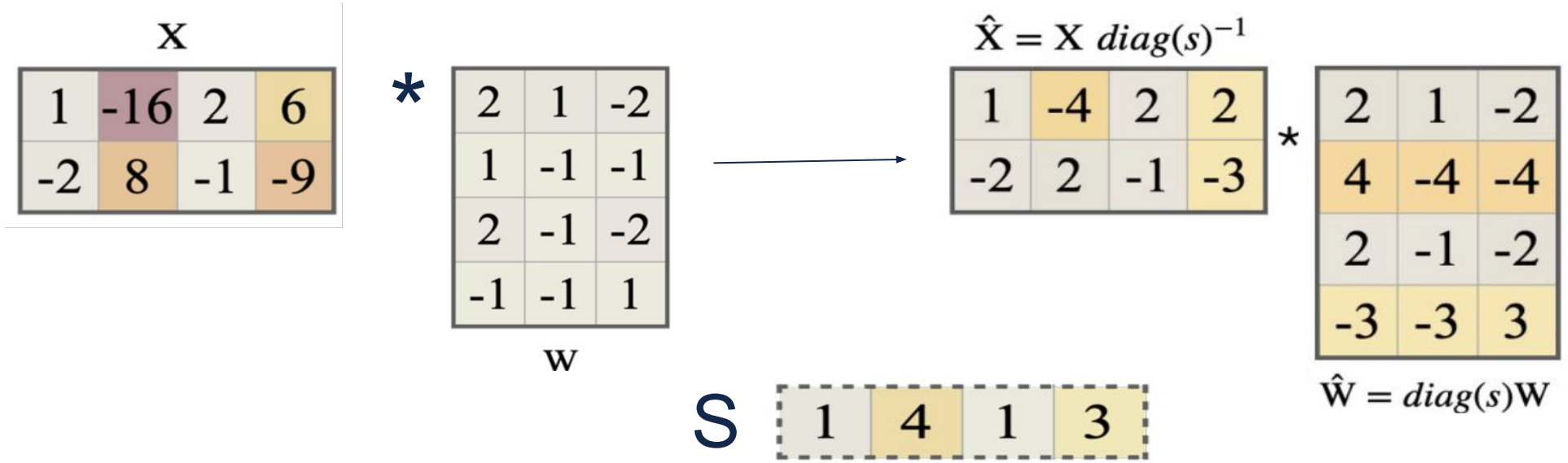
S

1	4	1	3

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

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

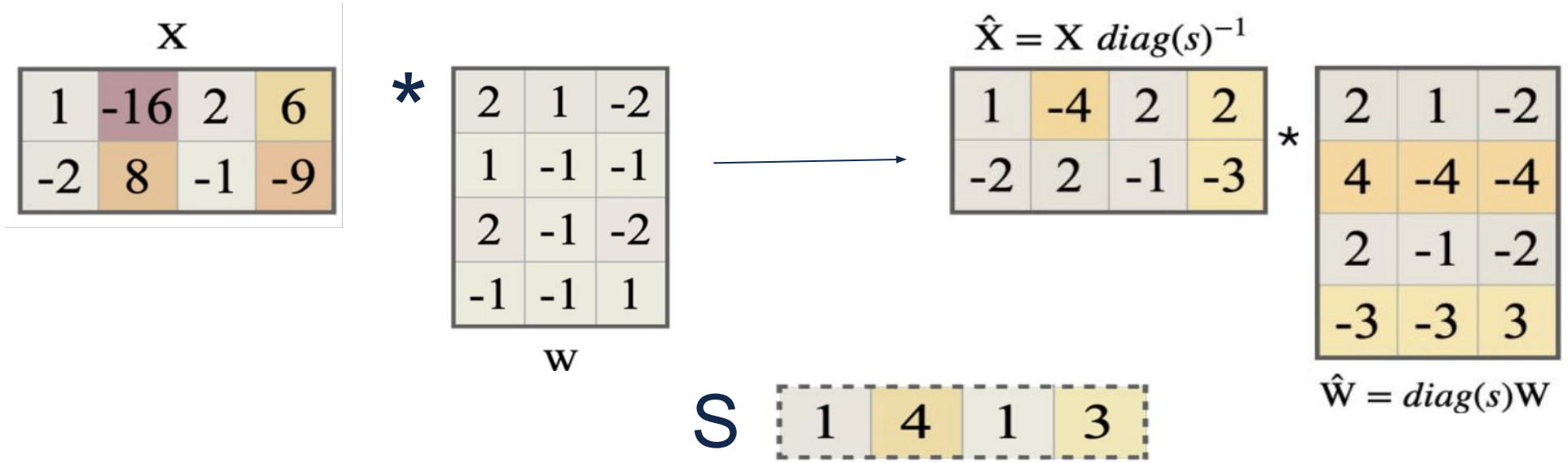
SmoothQuant Example



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

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

SmoothQuant Example



No Free Lunch! What is the new overhead here?

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

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

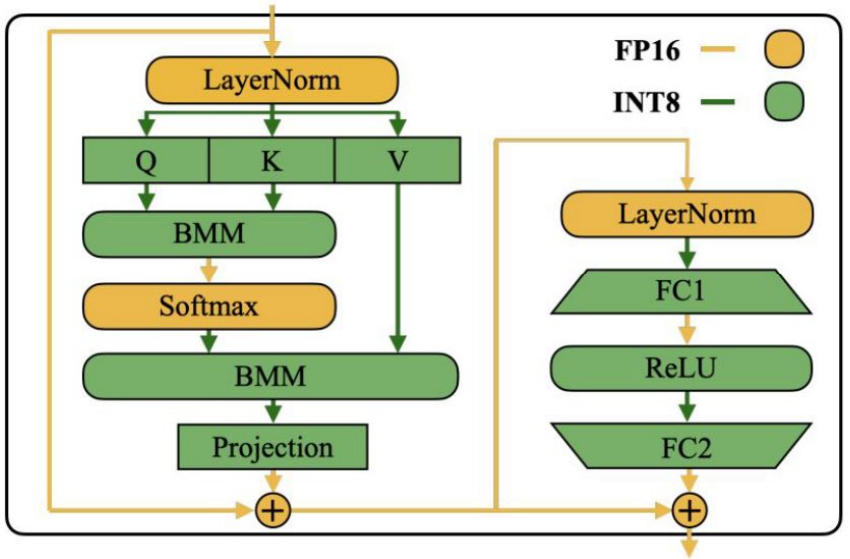
SmoothQuant

```python
import numpy as np

def smooth_quant(X, W, alpha=0.5):

  """
  The author find that α = 0.5 is a well-balanced point to evenly split the quantization difficulty

  """

    # Calculate per-channel smoothing factor s
    max_X = np.max(np.abs(X), axis=0)
    max_W = np.max(np.abs(W), axis=1)

    # The formula ensures that the W and X share a similar maximum value and quantization difficulty.
    s = np.power(max_X, alpha) / np.power(max_W, 1 - alpha)

    # Smooth the input activation
    X_hat = np.power(X * np.diag(s), -1)

    # Scale the weights accordingly to keep the mathematical equivalence of a linear layer
    W_hat = np.diag(s) * W

    Y = X_hat @ W_hat

    return X_hat, W_hat
```

SmoothQuant Hardware Efficiency

Applying SmoothQuant to transformer blocks

- Linear layers take up most of the parameters and computation
- All compute intensive operators (Linears, BMMs) are qunatized
- Smoothing factor can be fused into previous layers' parameters offline
- All linear layers are quantized with W8A8, as well as BMM operators in Attention computation



Four Baselines

LLM.int8 keeps outliers in FP16 (large latency overhead). W8A8 is the naive implementation. Outlier suppression uses token-wise clipping

Method	Weight	Activation
W8A8	per-tensor	per-tensor dynamic
ZeroQuant	group-wise	per-token dynamic
`LLM.int8()`	per-channel	per-token dynamic+FP16
Outlier Suppression	per-tensor	per-tensor static

SmoothQuant O1 to O3

Gradually aggressive and efficient (lower latency) quantization levels

Method	Weight	Activation
SmoothQuant-O1	per-tensor	per-token dynamic
SmoothQuant-O2	per-tensor	per-tensor dynamic
SmoothQuant-O3	per-tensor	per-tensor static

Evaluation

Three families of LLMs

- OPT ($\alpha = 0.5$)
- BLOOM ($\alpha = 0.5$)
- GLM-130B (α is set to 0.75 since its activations are more difficult to quantize)

- Seven zero-shot evaluation tasks e.g. LAMBADA, WikiText
- Focus on relative performance chance before/after quantization

OPT-175B Results

OPT-175B	LAMBADA	HellaSwag	PIQA	WinoGrande	OpenBookQA	RTE	COPA	**Average↑**	**WikiText↓**
FP16	74.7%	59.3%	79.7%	72.6%	34.0%	59.9%	88.0%	66.9%	10.99
W8A8	0.0%	25.6%	53.4%	50.3%	14.0%	49.5%	56.0%	35.5%	93080
ZeroQuant	0.0%*	26.0%	51.7%	49.3%	17.8%	50.9%	55.0%	35.8%	84648
`LLM.int8()`	74.7%	59.2%	79.7%	72.1%	34.2%	60.3%	87.0%	66.7%	11.10
Outlier Suppression	0.00%	25.8%	52.5%	48.6%	16.6%	53.4%	55.0%	36.0%	96151
SmoothQuant-O1	74.7%	59.2%	79.7%	71.2%	33.4%	58.1%	89.0%	66.5%	11.11
SmoothQuant-O2	75.0%	59.0%	79.2%	71.2%	33.0%	59.6%	88.0%	66.4%	11.14
SmoothQuant-O3	74.6%	58.9%	79.7%	71.2%	33.4%	59.9%	90.0%	66.8%	11.17

arrow pointing **up** (↑) means that **higher is better** for this metric

arrow pointing **down** (↓) indicates that for the **WikiText perplexity score**, a **lower value is better**

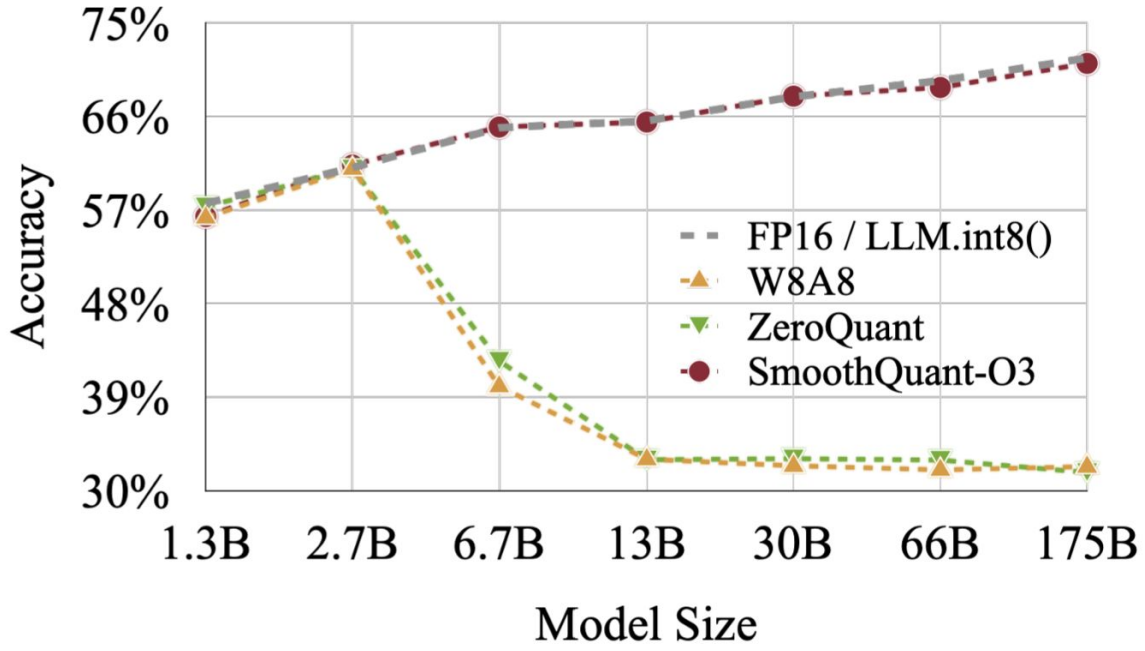
Results On Different LLMs

Method	OPT-175B	BLOOM-176B	GLM-130B*
FP16	71.6%	68.2%	73.8%
W8A8	32.3%	64.2%	26.9%
ZeroQuant	31.7%	67.4%	26.7%
`LLM.int8()`	71.4%	68.0%	73.8%
Outlier Suppression	31.7%	54.1%	63.5%
SmoothQuant-O1	**71.2%**	68.3%	**73.7%**
SmoothQuant-O2	71.1%	**68.4%**	72.5%
SmoothQuant-O3	71.1%	67.4%	72.8%

* accuracy is not column wise comparable due to different datasets

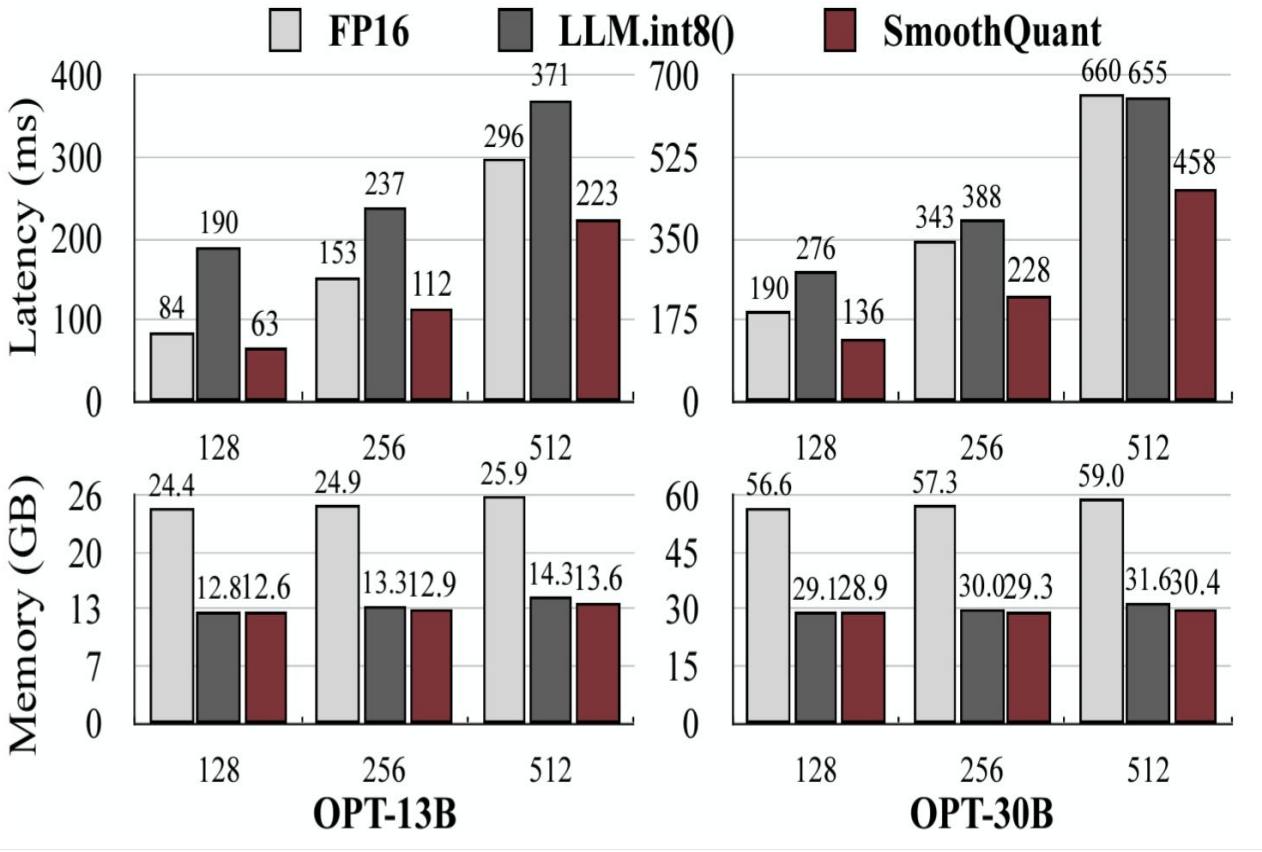
SmoothQuant

SmoothQuant well maintains the accuracy w/o fine-tuning



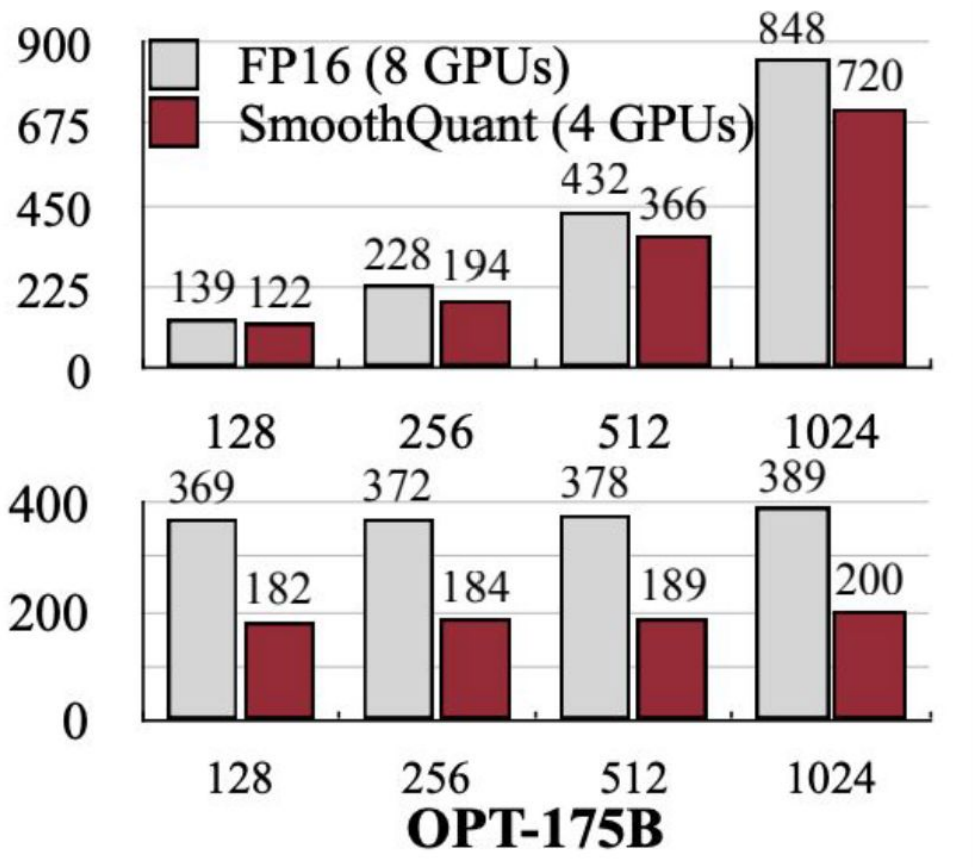
Memory/Latency Savings

- SmoothQuant accelerates the inference and halves the memory footprint
- Memory reduced to half
- 8 A100 GPUs reduced to 4 GPUs (Massive cost reduction)
- Everything is quantized to INT8 rather than FP16 so less communication and latency is even faster
- Only compares with LLM.int8() because it is the only method that maintains accuracy.



Hardware Efficiency

Similar or faster latency with half # GPUs



Newer LLaMA Models:

LLaMA (and its successors like Alpaca) are popular open source LLMs, which introduced SwishGLU, RoPE. Will that impact quantization?
SmoothQuant can losslessly quantize LLaMA families, further lowering hardware barrier

Wiki PPL↓	7B	13B	30B	65B
FP16	11.51	10.05	7.53	6.17
W8A8 SmoothQuant	11.56	10.08	7.56	6.20

Similar!

Lower perplexity is better

Key Results

- **OPT-175B**: SmoothQuant-O3 achieved **66.8% accuracy** across benchmarks compared to FP16's **66.9%**, with **1.56x speedup** and **2x memory reduction**.
- **GLM-130B**: SmoothQuant maintains accuracy similar to FP16 while reducing memory and speeding up inference.
- **MT-NLG 530B**: SmoothQuant allows deployment of the massive MT-NLG model on a **single 8-GPU node**, achieving **73.1% average accuracy** across benchmarks, which was previously infeasible with standard hardware configurations.

SmoothQuant vs Other Methods

Method:	Contributions
GPTQ (2022)	Post-training quantization using Hessian information. Push down to 4 bits per weight. Pioneering work; led to lots of weight-only post-training quantization works.
OWQ (Outlier-Aware Weight Quantization, 2023)	Mixed-precision quantization scheme considering activation outliers.
SpQR (Sparse-Quantized Representation, 2023)	Isolate outlier weights and keep them in high precision.
SqueezeLLM (2023)	Sensitivity-based non-uniform quantization, outlier extraction.
SmoothQuant (2022)	Quantize both weights and activations. Utilize faster acceleration units such as INT8 TensorCore on a NVIDIA A100 GPU.
AWQ (Activation-aware Weight Quantization, 2023)	Search for optimal per-channel scaling by observing activation.

Overall Results

- SmoothQuant is faster than FP16 baseline under all settings
- LLM.int8() is usually slower than SmoothQuant
- Additionally: SmoothQuant can serve a >500B model within a single node (8×A100 80GB GPUs) at a negligible accuracy loss

My Thoughts

Strengths:

- **Innovative outlier handling**: SmoothQuant's method of shifting quantization difficulty from activations to weights is novel and effective.
- **Accuracy & efficiency balance**: Maintains accuracy while achieving 1.56x speedup and 2x memory reduction.
- **Broad applicability**: Tested on diverse models like OPT and MT-NLG, showing flexibility across architectures.
- **Practical deployment**: Requires no retraining, making it quick to deploy in production environments.

Weaknesses:

- **Hyperparameter sensitivity**: Requires careful tuning of α\alphaα for different models.
- **Pre-computation overhead**: Gathering activation statistics adds complexity, especially for dynamic inputs.
- **Hardware dependence**: Relies on GPUs with INT8 GEMM support, limiting broad hardware applicability.

My Thoughts

Future Directions:

- **Lower bit-width quantization**: Exploring 4-bit quantization could further reduce memory and computation costs.
- **FP4 exploration**: Mixed precision or FP4 formats could offer better trade-offs between efficiency and precision.
- **Cross-layer Quantization Adjustments:** SmoothQuant applies the same general principle to all layers, but different layers in large models may have very different **activation distributions** and **weight characteristics**. In future work, we can investigate **layer-specific quantization strategies** that adjust the scaling factor α dynamically across layers, based on each layer's specific activation patterns and sensitivities.
- **Task-specific Quantization**: Different tasks (e.g., translation, summarization, question-answering) may have different **activation patterns** and **outlier characteristics**. A one-size-fits-all approach like SmoothQuant may not be optimal across all tasks.Potential future work could develop **task-specific quantization strategies** that dynamically adjust α and other parameters depending on the task being performed. For example, quantization could be fine-tuned for tasks that are **more sensitive to outliers** or require high precision.


ILLINOIS