Efficient LM Methods and Infrastructure

Chenxin An & Yazheng Yang

2023/10

Useful resources:
https://zhuanlan.zhihu.com/p/639228219
https://github.com/Dao-AILab/flash-attention
https://huggingface.co/docs/accelerate/usage_guides/megatron_lm

Motivation of this talk

The Power of Large Language Models

* LLMs can generate human-like text, translate languages, answer questions, and more.

LLMs require massive amounts of data and computing power to train.

* High computational cost
* Need for efficient memory management

Solutions:

1. **Quantizing** and **Partitioning** model parameters and optimizer states to save GPU memory
2. Using **efficient attention** when the input sequence is long

LLM.8bit() , flashAttention, CPU offload, Magtron-LM and Deepspeed

All of them are very **popular but complex** methods and **widely used** in famous github codebase

Every LLM researchers know them but only a few know the details and principles

1. help your interview
2. Help you debug
3. develop your own better frameworks… 😅

Popular solutions

1. Google: TPU + TensorFlow + XLA
2. **Open-source: Nvidia GPU + PyTorch + Megatron-LM + DeepSpeed**

The open-source solutions:

1. **Megatron-LM**: Training Multi-Billion Parameter Language Models Using Model Parallelism (Partition model parameters to save memory)
2. **ZeRO:** Memory Optimizations Toward Training Trillion Parameter Models (Partition optimizer states to save memory)

Megatron-LM (2020)

Data parallel (A minibatch is split across workers)

- Work well in when the number of workers is limited
- More workers means we have to use a very large batch size to distribute the data on each worker. But large batch training always results in reduced accuracy or long time

Recent advances in natural language processing have shown that training large transformer-based language models can achieve state-of-the-art results on many NLP tasks.

However, very large models with billions of parameters are difficult to train due to **memory** if the model parameter size exceed the GPU memory…

Model parallel (model parameters are split across workers)

Previous methods

- Frameworks like GPipe and Mesh-Tensorflow provide different types of model parallel frameworks. However, they require rewriting the model and rely on custom compilers.

Megatron-LM (2020)

Lets enumerate the model parallel with 2 workers. More workers is the same.

The first part of this MLP block is a GEMM (General Matrix Multiplication) followed by a GeLU nonlinear layer

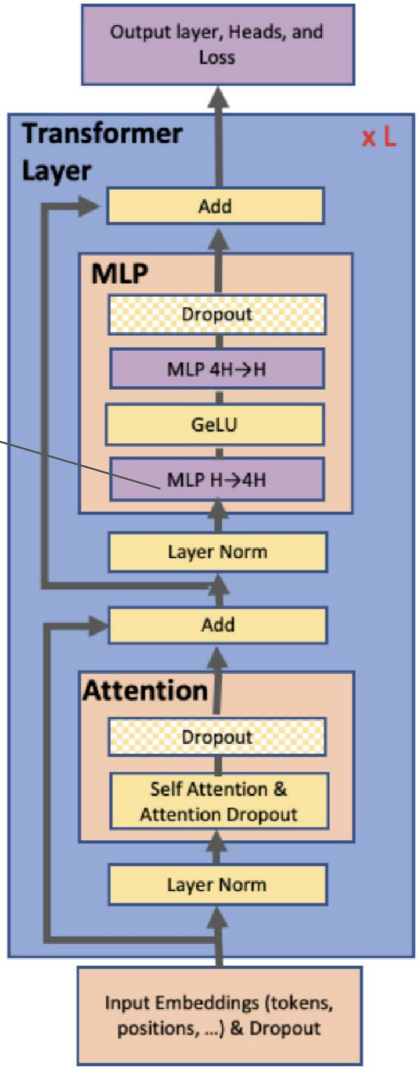
One option to parallelize the GEMM is to split the weight matrix A along its rows and input X along its columns as:

$$X = [X_1, X_2], \; A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}. \qquad (2)$$

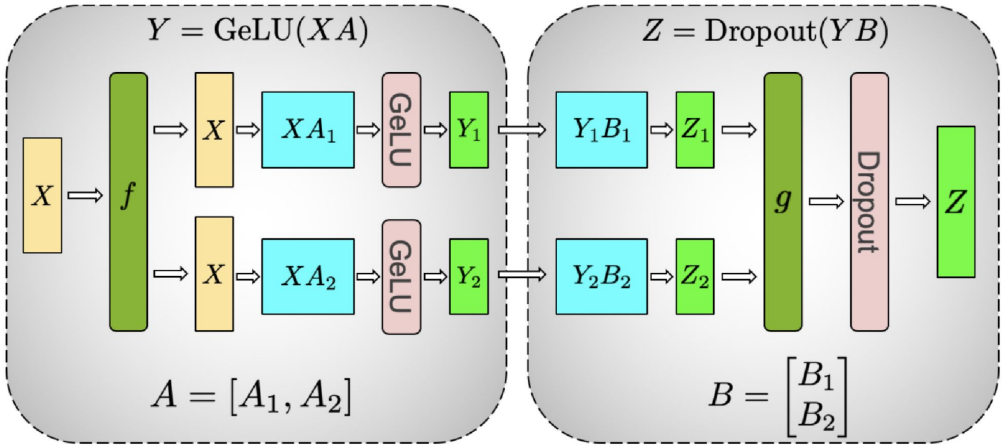
This results in Y = GeLU(X_1A_1 + X_2A_2) != GeLU(X_1A_1) +GeLU(X_2A_2)

Another option is to split A along its columns $A = [A_1, A_2]$. This partitioning allows the GeLU nonlinearity to be independently applied to the output of each partitioned GEMM:

$$[Y_1, Y_2] = [\mathrm{GeLU}(XA_1), \mathrm{GeLU}(XA_2)] \qquad (3)$$



Megatron-LM (2020)



(a) MLP

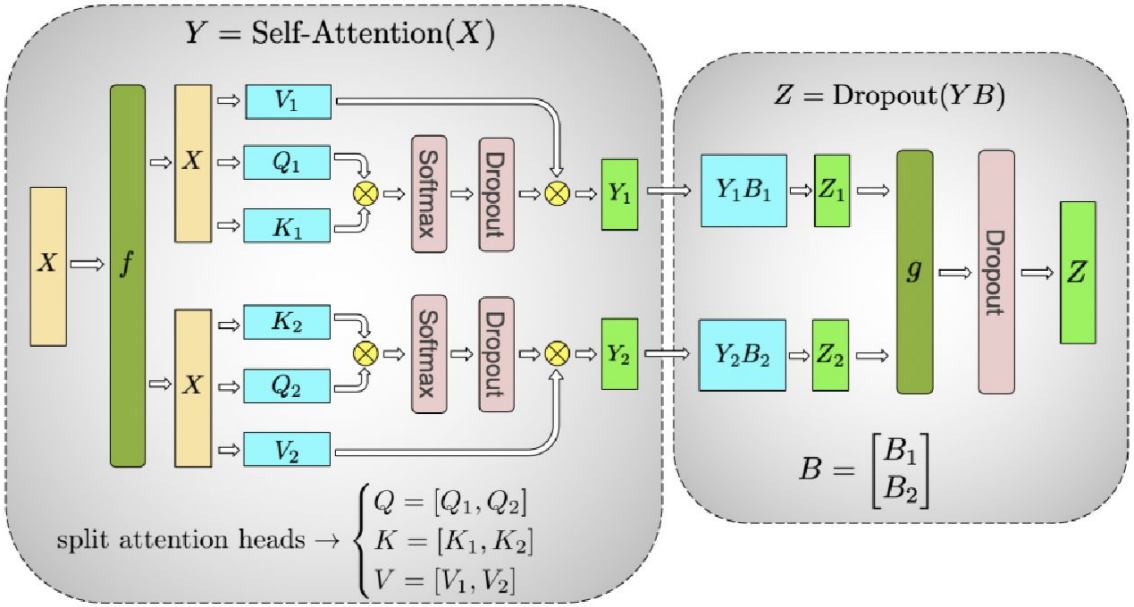
we partition the first GEMM in this column parallel fashion and split the second GEMM along its rows

```
class f(torch.autograd.Function):
    def forward(ctx, x):
        return x

    def backward(ctx, gradient):
        all_reduce(gradient)
        return gradient
```

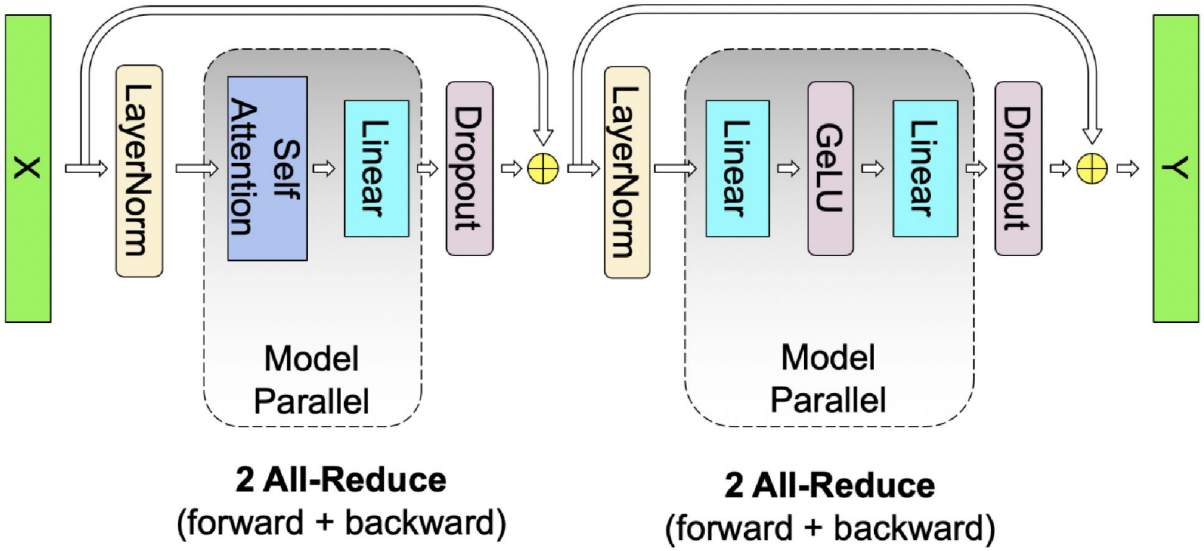
- This approach splits both GEMMs in the MLP block across GPUs and requires only a single all-reduce operation in the forward pass (g operator) and a single all-reduce in the backward pass (f operator).

Megatron-LM (2020)



Benefits: eliminating intermediate synchronization points and achieving better scalability. This allows us to complete the calculation of all GEMM in the transformer layer using only two all reduce (one is for self attention and one for MLP) in the forward path, and also use two all reduce in the reverse path

Megatron-LM (2020)



- Communication operations in a transformer layer. There are 4 total communication operations in the forward and backward pass of a single model parallel transformer layer.

Megatron-LM (2020)

Parallel efficiency

Table 1. Parameters used for scaling studies. Hidden size per attention head is kept constant at 96.

Hidden Size	Attention heads	Number of layers	Number of parameters (billions)	Model parallel GPUs	Model +data parallel GPUs
1536	16	40	1.2	1	64
1920	20	54	2.5	2	128
2304	24	64	4.2	4	256
3072	32	72	8.3	8	512

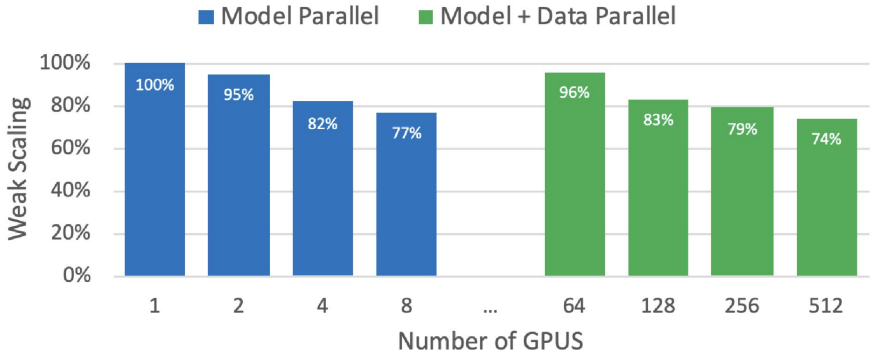


Figure 5. Model and model + data parallel weak scaling efficiency as a function of the number of GPUs.

Weak scaling is a method of measuring the performance of parallel systems
Weak Scaling Efficiency = Time_for_1_processor / (Time_for_N_processors / N)

Megatron-LM (2020)

GPT-2 experiments

Table 2. Model configurations used for GPT-2.

Parameter Count	Layers	Hidden Size	Attn Heads	Hidden Size per Head	Total GPUs	Time per Epoch (days)
355M	24	1024	16	64	64	0.86
2.5B	54	1920	20	96	128	2.27
8.3B	72	3072	24	128	512	2.10

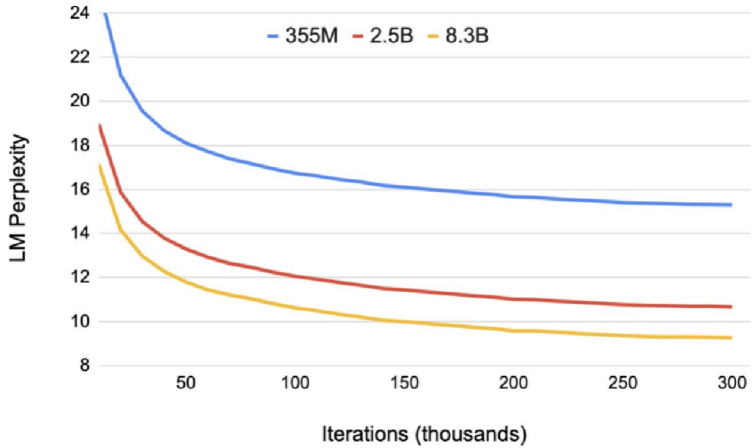


Figure 6. Validation set perplexity. All language models are trained for 300k iterations. Larger language models converge noticeably faster and converge to lower validation perplexities than their smaller counterparts.

Table 3. Zero-shot results. SOTA are from (Khandelwal et al., 2019) for Wikitext103 and (Radford et al., 2019) for LAMBADA.

Model	Wikitext103 Perplexity ↓	LAMBADA Accuracy ↑
355M	19.31	45.18%
2.5B	12.76	61.73%
8.3B	**10.81**	**66.51%**
Previous SOTA	15.79	63.24%

Megatron-LM (2020)

BERT experiments

Table 4. Model configurations used for BERT.

Parameter Count	Layers	Hidden Size	Attention Heads	Total GPUs
336M	24	1024	16	128
1.3B	24	2048	32	256
3.9B	48	2560	40	512

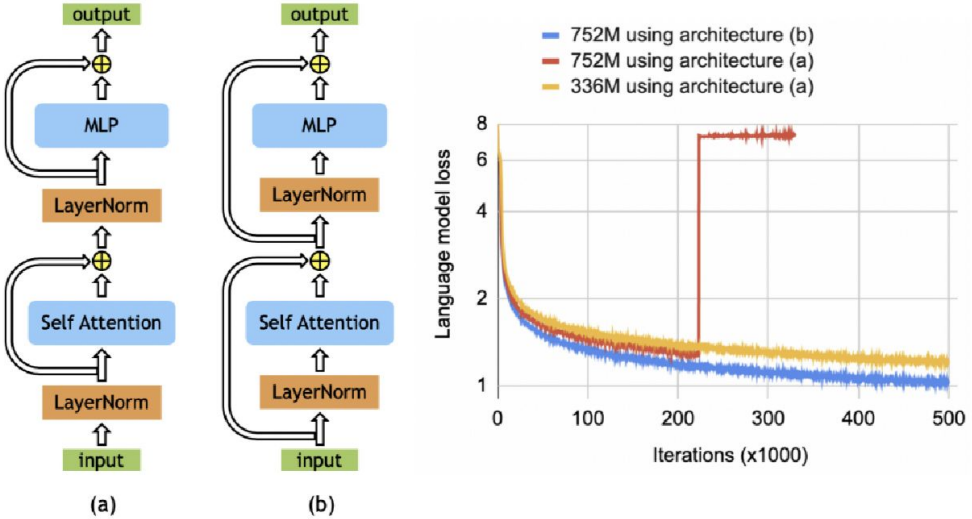


Figure 7. Training loss for BERT model using the original architecture (a) and the rearranged architecture (b). Left figure shows the training loss for 336M and 752M BERT model. While the original architecture performs well on the 336M model, the modifications in (b) enable stable training with lower training loss.

Megatron-LM (2020)

BERT experiments

Table 5. Development set results for MNLI, QQP, SQuAD 1.1 and SQuAD 2.0 and test set results for RACE. The trained tokens represents consumed tokens during model pretraining (proportional to batch size times number of iterations) normalized by consumed tokens during model pretraining for our 336M model.

Model	trained tokens ratio	MNLI m/mm accuracy (dev set)	QQP accuracy (dev set)	SQuAD 1.1 F1 / EM (dev set)	SQuAD 2.0 F1 / EM (dev set)	RACE m/h accuracy (test set)
RoBERTa (Liu et al., 2019b)	2	90.2 / 90.2	92.2	94.6 / 88.9	89.4 / 86.5	83.2 (86.5 / 81.8)
ALBERT (Lan et al., 2019)	3	90.8	92.2	94.8 / 89.3	90.2 / 87.4	86.5 (89.0 / 85.5)
XLNet (Yang et al., 2019)	2	90.8 / 90.8	92.3	95.1 / 89.7	90.6 / 87.9	85.4 (88.6 / 84.0)
Megatron-336M	1	89.7 / 90.0	92.3	94.2 / 88.0	88.1 / 84.8	83.0 (86.9 / 81.5)
Megatron-1.3B	1	90.9 / 91.0	92.6	94.9 / 89.1	90.2 / 87.1	87.3 (90.4 / 86.1)
Megatron-3.9B	1	**91.4 / 91.4**	**92.7**	**95.5 / 90.0**	**91.2 / 88.5**	**89.5 (91.8 / 88.6)**
ALBERT ensemble (Lan et al., 2019)				95.5 / 90.1	91.4 / 88.9	89.4 (91.2 / 88.6)
Megatron-3.9B ensemble				**95.8 / 90.5**	**91.7 / 89.0**	**90.9 (93.1 / 90.0)**

What's more

Gradient(activation) checkpointing

Y = sigmoid(x)

- Forward pass: implementing inplace operations and memory sharing optimizations to delete some intermediate activations that are temporarily not needed during the forward pass
- backward pass: recovering them through additional forward computation when needed during the backward pass.
- Trading FLOPs for memory

CPU offload

CPU offload swaps temporarily unused GPU memory into CPU memory for storage, and then takes it out when needed.

- The main overhead comes from copying between the CPU and GPU, which will occupy the transmission bandwidth (PCIE bandwidth)

- Trading IO for memory

Memory Efficient training and inference

Outline

1. Megatron – Using model (`split` the model) parallelism during training and inference. A single GPU leading to OOM? Let's train one model using multiple cards
2. **ZeRO: Memory Optimizations Toward Training Trillion Parameter Models**

3. FlashAttention – IO Awareness attention during training. Optimizing from CUDA kernel to accelerate attention calculation.

ZeRO (2020)

Adam Optimizer states can also be partitioned

GPU memory usage:

(1) model parameters (fp16),
(2) model gradients (fp16),
(3) Adam states (fp32 model parameter backup)
(4) fp32 1st momentum,
(5) fp32 2nd momentum.

Assuming model parameter quantity is Ψ, then a total of $2\Psi + 2\Psi + 4\Psi + 4\Psi + 4\Psi = 16\Psi$ Byte storage, as can be seen, the proportion of Adam status = 12 / (4+12) = 75%

ZeRO (2020)

ZeRO1: Partitioning optimizer states

ZeRO2: Partitioning gradients

ZeRO3: Partitioning parameters

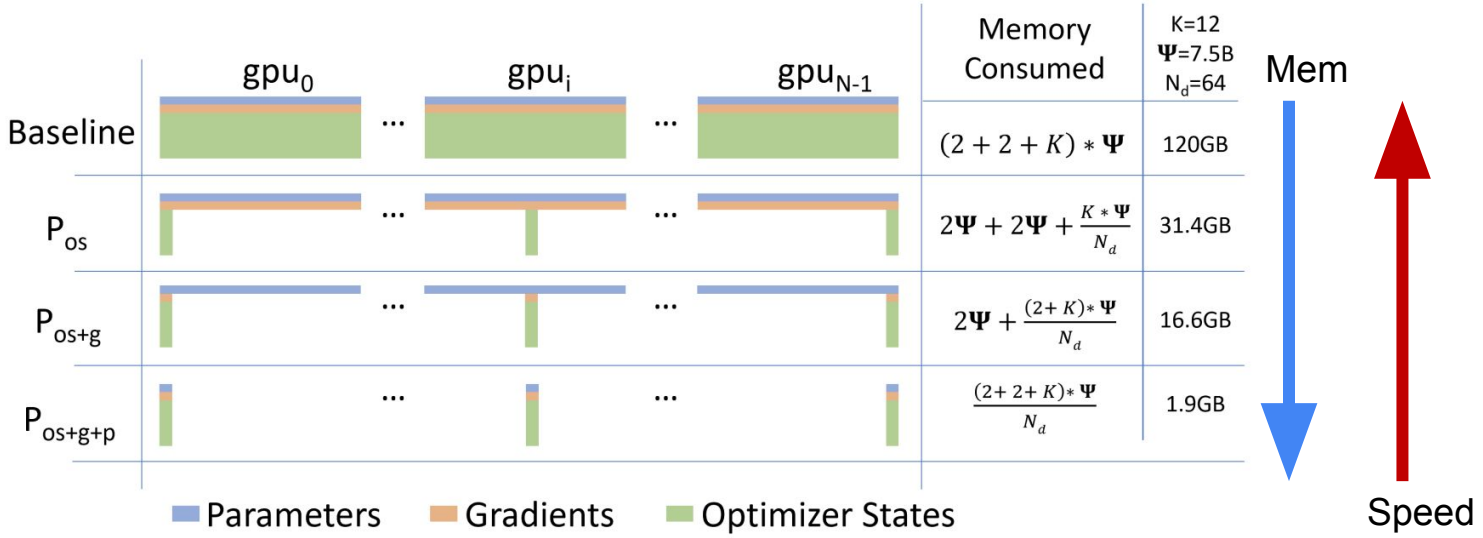


Figure 1: Comparing the per-device memory consumption of model states, with three stages of *ZeRO*-DP optimizations. Ψ denotes model size (number of parameters), K denotes the memory multiplier of optimizer states, and N_d denotes DP degree. In the example, we assume a model size of $\Psi = 7.5B$ and DP of $N_d = 64$ with $K = 12$ based on mixed-precision training with Adam optimizer.

Do you want to directly generate your presentation slides and notions for Data-8005 by just only inputting the recommended papers？

Challenge 1: Currently AI models are not `clever` enough to fully understand the paper (Maybe GPT-5)

Challenge 2: The paper is so long and we usually have to present **more than 3 papers.**
The model also has to find the **relations between them** to ensure a high quality presentation

Modeling long sequence

In Natural languge processing:
Reading very long documents is a very tedious and tiring thing for us. With long-sequence modeling ability we can use a LLM to understand very long sequence such as plays, books and papers.

In computer version:
high resolutions bring better and more robust insight.

In other areas
Modeling audio, video, medical imaging which are naturally modeled as millions of steps

Both The computational complexity and storage complexity of the transformer model increase exponentially O（N^2）with the length of the sequence

So how can we use

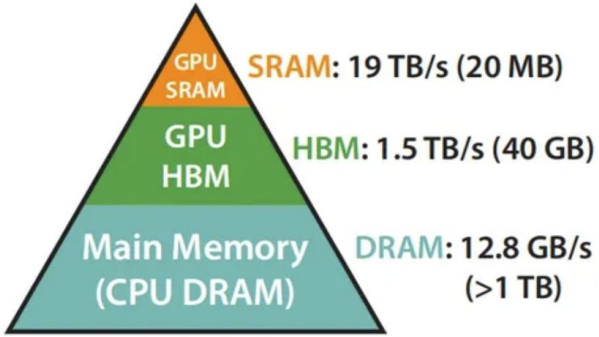
Efficient attention without trading accuracy

INPUT QKV
S=torch.matmul(Q,K)
P=torch.softmax(S)
O=torch.matmul(PV)

O = torch.flashAttn(Q, K, V) (calculation is in CUDA)

FlashAttention (2022) **background**

Fast and Memory-Efficient Exact Attention with IO-Awareness



Memory Hierarchy with Bandwidth & Memory Size

(1) each kernel loads input data from low-speed HBM to high-speed SRAM;
(2) In SRAM, perform calculations;
(3) After the calculation is completed, write the calculation results from SRAM to HBM.

Faster computation. Flash Attention does not reduce the computational load of FLOPs, but instead **reduces the number of HBM accesses** based on IO awareness, thereby reducing computational time.

FlashAttention (2022) **Motivation**

Algorithm 0 Standard Attention Implementation

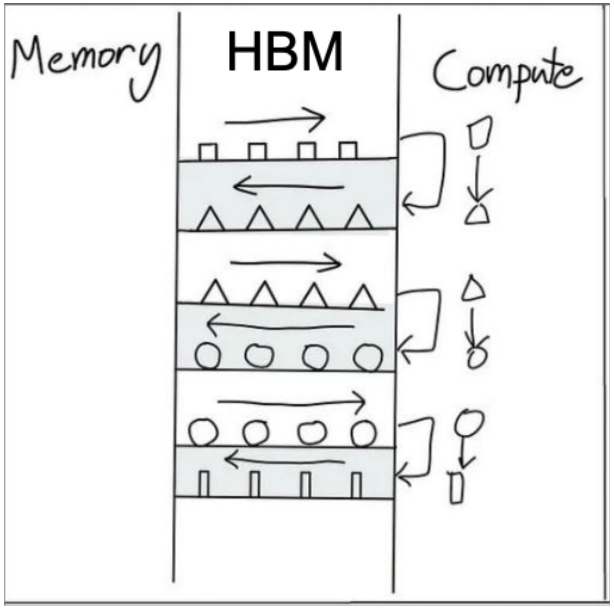
Require: Matrices $\mathbf{Q},\mathbf{K},\mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

1: Load $\mathbf{Q},\mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.

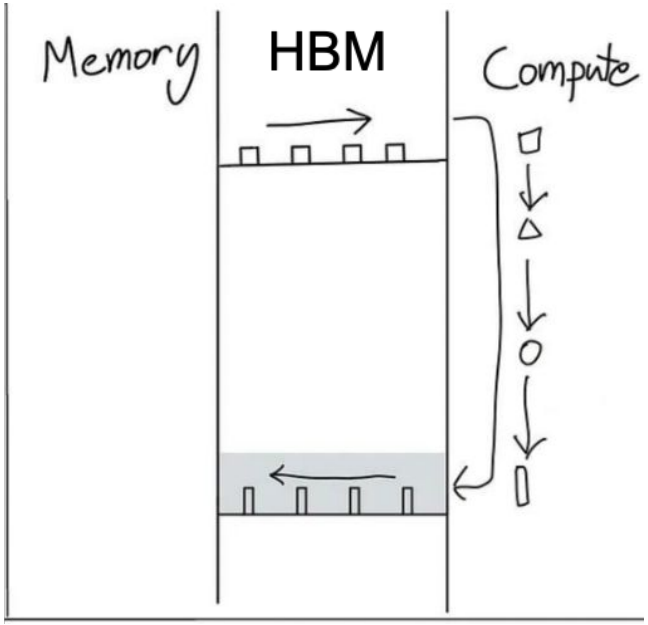
Drawbacks:

1. storing the complete attention matrix **P** and **S** **need extra memory**
2. HBM has multiple reads and writes, which **slows down the runtime** (wall clock time)

FlashAttention (2022) **Motivation**



Standard



FlashAttention

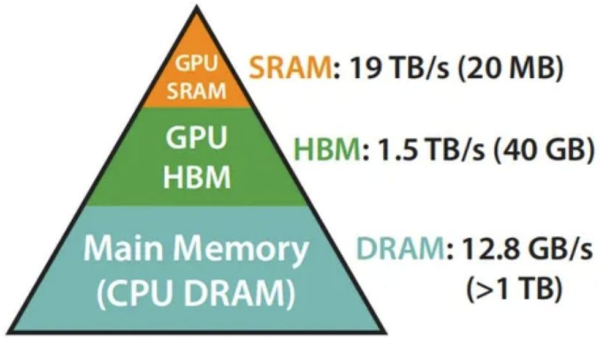
FlashAttention (2022)

How Flash attention saving the IO time:
By using kernel fusion to merge multiple operations into one operation and utilizing high-speed SRAM for computation, the number of reads and writes to HBM can be reduced.

What are the difficulties:
However, the memory size of SRAM is limited, making it impossible to calculate complete attention at once.

Therefore, it is necessary to perform block calculation so that the memory required for block calculation does not exceed the size of SRAM.

The block-wise partition allow GPUs to calculate attention on SRAM



Memory Hierarchy with Bandwidth & Memory Size

FlashAttention (2022)

Algorithm 1 FLASHATTENTION

Require: Matrices $\mathbf{Q},\mathbf{K},\mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size M.

Set block sizes $B_c = \lceil \frac{M}{4d} \rceil, B_r = \min(\lceil \frac{M}{4d} \rceil, d)$.

Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}, \ell = (0)_N \in \mathbb{R}^N, m = (-\infty)_N \in \mathbb{R}^N$ in HBM.

Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \ldots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K},\mathbf{V}$ in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_1, \ldots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \ldots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.

Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \ldots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_i, \ldots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1, \ldots, m_{T_r}$ of size B_r each.

for $1 \le j \le T_c$ **do**

  Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.

  for $1 \le i \le T_r$ **do**

    Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.

    On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.

    On chip, compute $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.

    On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.

    Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1} (\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.

    Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.

  end for

end for

Return $\mathbf{O}$.

FlashAttention (2022)

What are the difficulties of block calculation?

The difficulty lies in the block calculation of **softmax**.
Due to the need to obtain complete input data when calculating the normalization factor (denominator) of softmax

Safe softmax

To avoid the problem of **numerical overflow**, the maximum value is usually "subtracted" during calculation, known as "safe softmax".

$$m = \max_i(x_i); \quad softmax(x_i) = \frac{e^{x_i - m}}{\sum_{j=1}^{d} e^{x_j - m}}$$

FlashAttention (2022)

We introduce $m(x)$ and $l(x)$ defined as follows:

$$m(x) := \max_i \; x_i, \quad f(x) := \left[e^{x_1 - m(x)} \quad \dots \quad e^{x_B - m(x)}\right], \quad \ell(x) := \sum_i f(x)_i, \quad \text{softmax}(x) := \frac{f(x)}{\ell(x)}.$$

Given input $x = \left[x^{(1)}, x^{(2)}\right]$

$$m(x) = m(\left[x^{(1)} \; x^{(2)}\right]) = \max(m(x^{(1)}), m(x^{(2)})), \quad f(x) = \left[e^{m(x^{(1)}) - m(x)} f(x^{(1)}) \quad e^{m(x^{(2)}) - m(x)} f(x^{(2)})\right],$$

$$\ell(x) = \ell(\left[x^{(1)} \; x^{(2)}\right]) = e^{m(x^{(1)}) - m(x)} \ell(x^{(1)}) + e^{m(x^{(2)}) - m(x)} \ell(x^{(2)}), \quad \text{softmax}(x) = \frac{f(x)}{\ell(x)}.$$

By introducing $m(x)$ and $l(x)$ we can calculate **softmax** by block!

FlashAttention (2022)

$QK^T = [1,2,3,4]$ *and it cannot fit the zie of SRAM* *divide into* $Block1: [1,2]$ $Block2: [3,4]$

In Block1 we have:

$$m_1 = max([1,2]) = 2$$

$$f_1 = [e^{1-2}, e^{2-2}] = [e^{-1}, e^{0}]$$

$$l_1 = \sum f_1 = e^{-1} + e^{0}$$

$$o_1 = \frac{f_1}{l_1} = \frac{[e^{-1}, e^{0}]}{e^{-1} + e^{0}}$$

In Block2 we have:

$$m_2 = max([3,4]) = 4$$

$$f_2 = [e^{3-4}, e^{4-4}] = [e^{-1}, e^{0}]$$

$$l_2 = \sum f_2 = e^{-1} + e^{0}$$

$$o_2 = \frac{f_2}{l_2} = \frac{[e^{-1}, e^{0}]}{e^{-1} + e^{0}}$$

Merge

$$m = max(m_1, m_2) = 4$$

$$f = [e^{m_1 - m} f_1, e^{m_2 - m} f_2] = [e^{-3}, e^{-2}, e^{-1}, e^{0}]$$

$$l = e^{m_1 - m} l_1 + e^{m_2 - m} l_2 = e^{-3} + e^{-2} + e^{-1} + e^{0}$$

$$o = \frac{f}{l} = \frac{[e^{-3}, e^{-2}, e^{-1}, e^{0}]}{e^{-3} + e^{-2} + e^{-1} + e^{0}}$$

Algorithm 1 FLASHATTENTION

Require: Matrices $\mathbf{Q},\mathbf{K},\mathbf{V}\in\mathbb{R}^{N\times d}$ in HBM, on-chip SRAM of size M.

Set block sizes $B_c=\lceil\frac{M}{4d}\rceil, B_r=\min(\lceil\frac{M}{4d}\rceil,d)$. block size of Q and K, V

Initialize $\mathbf{O}=(0)_{N\times d}\in\mathbb{R}^{N\times d}, \ell=(0)_N\in\mathbb{R}^N, m=(-\infty)_N\in\mathbb{R}^N$ in HBM.

Divide $\mathbf{Q}$ into $T_r=\lceil\frac{N}{B_r}\rceil$ blocks $\mathbf{Q}_1,\dots,\mathbf{Q}_{T_r}$ of size $B_r\times d$ each, and divide $\mathbf{K},\mathbf{V}$ in to $T_c=\lceil\frac{N}{B_c}\rceil$ blocks $\mathbf{K}_1,\dots,\mathbf{K}_{T_c}$ and $\mathbf{V}_1,\dots,\mathbf{V}_{T_c}$, of size $B_c\times d$ each. init $QKVO$ m, l and

Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i,\dots,\mathbf{O}_{T_r}$ of size $B_r\times d$ each, divide ℓ into T_r blocks $\ell_i,\dots,\ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1,\dots,m_{T_r}$ of size B_r each. divide $QKVO$ m, l into blocks

for $1\le j\le T_c$ **do**

  Load $\mathbf{K}_j,\mathbf{V}_j$ from HBM to on-chip SRAM.

  for $1\le i\le T_r$ **do**

    Load $\mathbf{Q}_i,\mathbf{O}_i,\ell_i,m_i$ from HBM to on-chip SRAM.

    On chip, compute $\mathbf{S}_{ij}=\mathbf{Q}_i\mathbf{K}_j^T\in\mathbb{R}^{B_r\times B_c}$. block-wise QK^T

    On chip, compute $\tilde{m}_{ij}=\text{rowmax}(\mathbf{S}_{ij})\in\mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij}=\exp(\mathbf{S}_{ij}-\tilde{m}_{ij})\in\mathbb{R}^{B_r\times B_c}$ (pointwise), $\tilde{\ell}_{ij}=\text{rowsum}(\tilde{\mathbf{P}}_{ij})\in\mathbb{R}^{B_r}$. m(x) and l(x) in this block

    On chip, compute $m_i^{\text{new}}=\max(m_i,\tilde{m}_{ij})\in\mathbb{R}^{B_r}$, $\ell_i^{\text{new}}=e^{m_i-m_i^{\text{new}}}\ell_i+e^{\tilde{m}_{ij}-m_i^{\text{new}}}\tilde{\ell}_{ij}\in\mathbb{R}^{B_r}$.

    Write $\mathbf{O}_i\leftarrow\text{diag}(\ell_i^{\text{new}})^{-1}(\text{diag}(\ell_i)e^{m_i-m_i^{\text{new}}}\mathbf{O}_i+e^{\tilde{m}_{ij}-m_i^{\text{new}}}\tilde{\mathbf{P}}_{ij}\mathbf{V}_j)$ to HBM.

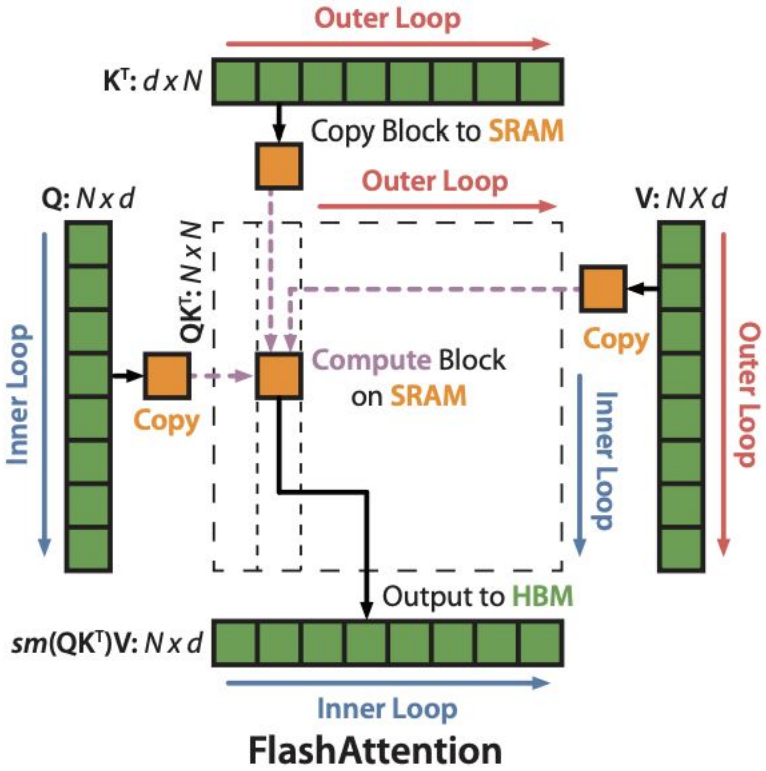
    Write $\ell_i\leftarrow\ell_i^{\text{new}}$, $m_i\leftarrow m_i^{\text{new}}$ to HBM. write the results to HBM and calculate next block

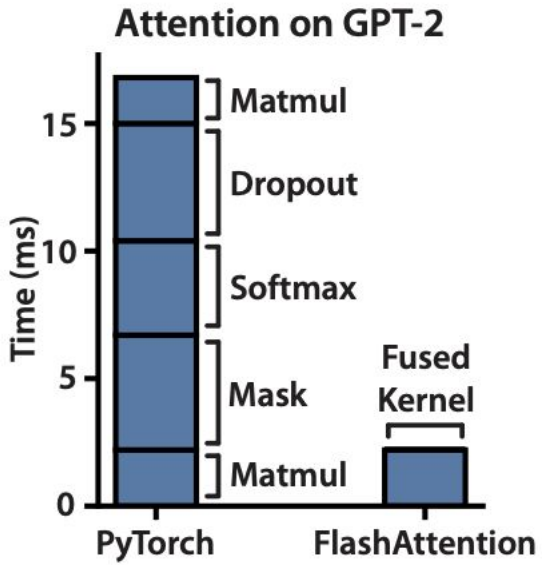
  end for

end for

Return $\mathbf{O}$.

FlashAttention (2022)





FlashAttention (2022)

Table 2: GPT-2 small and medium using FLASHATTENTION achieve up to 3× speed up compared to Huggingface implementation and up to 1.7× compared to Megatron-LM. Training time reported on 8×A100s GPUs.

Model implementations	OpenWebText (ppl)	Training time (speedup)
GPT-2 small - Huggingface [91]	18.2	9.5 days (1.0×)
GPT-2 small - Megatron-LM [80]	18.2	4.7 days (2.0×)
GPT-2 small - FLASHATTENTION	18.2	**2.7 days (3.5×)**
GPT-2 medium - Huggingface [91]	14.2	21.0 days (1.0×)
GPT-2 medium - Megatron-LM [80]	14.2	11.5 days (1.8×)
GPT-2 medium - FLASHATTENTION	14.2	**6.9 days (3.0×)**

Table 3: The performance of standard attention, FLASHATTENTION, block-sparse FLASHATTENTION, and approximate attention baselines on the Long-Range-Arena benchmarks.

Models	ListOps	Text	Retrieval	Image	Pathfinder	Avg	Speedup
Transformer	36.0	63.6	81.6	42.3	72.7	59.3	-
FLASHATTENTION	37.6	63.9	81.4	43.5	72.7	59.8	2.4×
Block-sparse FLASHATTENTION	37.0	63.0	81.3	43.6	73.3	59.6	**2.8×**
Linformer [88]	35.6	55.9	77.7	37.8	67.6	54.9	2.5×
Linear Attention [52]	38.8	63.2	80.7	42.6	72.5	59.6	2.3×
Performer [13]	36.8	63.6	82.2	42.1	69.9	58.9	1.8×
Local Attention [83]	36.1	60.2	76.7	40.6	66.6	56.0	1.7×
Reformer [53]	36.5	63.8	78.5	39.6	69.4	57.6	1.3×
Smyrf [20]	36.1	64.1	79.0	39.6	70.5	57.9	1.7×

FlashAttention (2022)

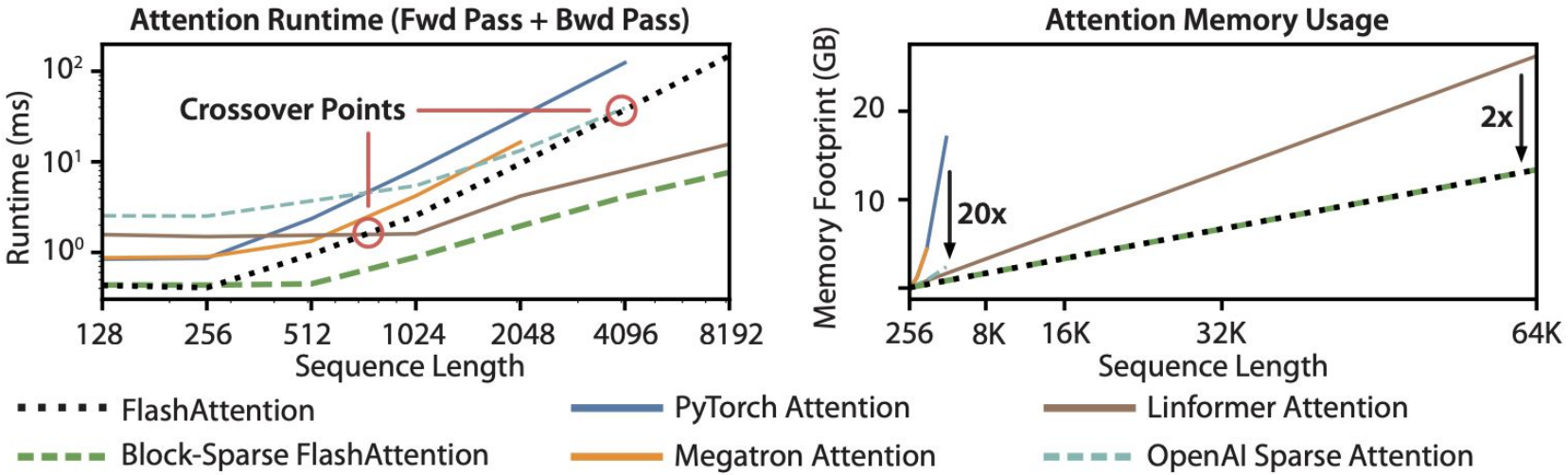
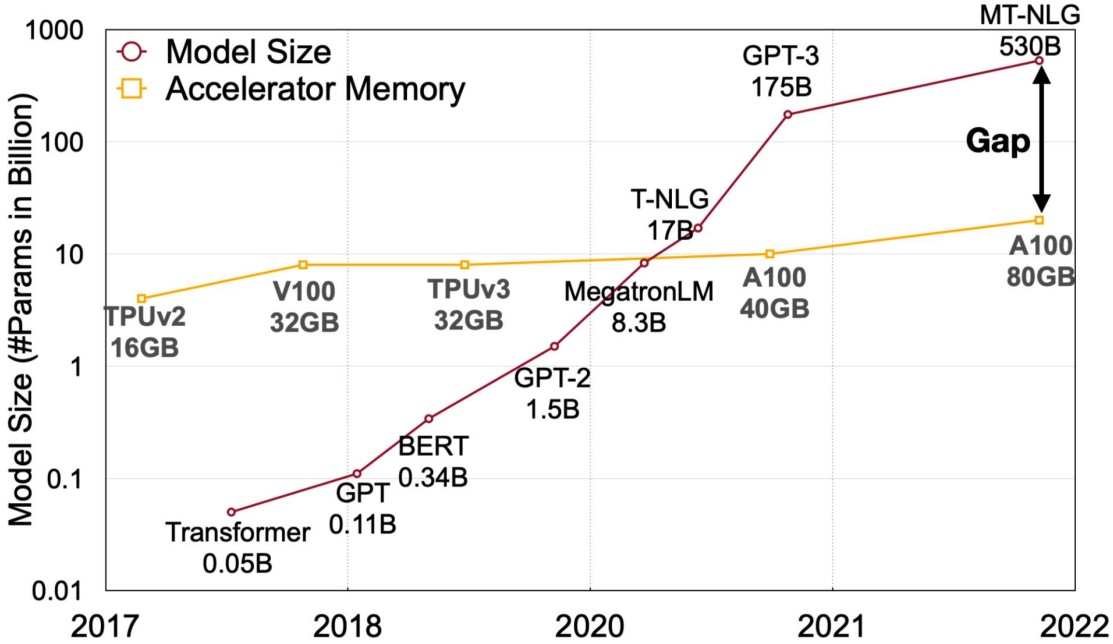


Figure 3: **Left:** runtime of forward pass + backward pass. **Right:** attention memory usage.

Quantization for Large Language Models

Increasing Model Size vs. Hardware Limitation



- Serving LLMs is budget and energy-consuming due to their gigantic model size.
- GPT-3: 175B parameters, consumes at least 350GB of memory to store and run in FP16
- Expensive and heavy computation, memory, bandwidth.

The model size of large language models is developing at a faster pace than the GPU memory in recent years, leading to a big gap between the supply and demand for memory.

Memory Efficient training and inference

Outline

1. **LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale**
2. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers
3. SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models

LLM.int8() Background

Low-bit Manipulation (Int8, Int4, Int2)

- ➢ **The majority of weights and computational consumption come from FFN and attention projecttion layers. (95% weights, 65~85% computation consumption)**
- ➢ **Reducing the parameters into low bits and using low-bit-precision matrix multiplication is appealing.**

Main Idea of low-bit Quantization

- ➢ **Scale or normalize data or matrix into discrete levels (round-to-nearest)**
- ➢ **Internal computation is carried out in lower precision. Dequantizing the result into higher precision for the output.**

LLM.int8() Background

Quantization Types

- **Maximum absolute quantization**

$$\mathbf{X}_{i8} = \left\lfloor \frac{127 \cdot \mathbf{X}_{f16}}{\max_{ij}(|\mathbf{X}_{f16_{ij}}|)} \right\rceil = \left\lfloor \frac{127}{\|\mathbf{X}_{f16}\|_\infty} \mathbf{X}_{f16} \right\rceil = \left\lfloor s_{x_{f16}} \mathbf{X}_{f16} \right\rceil$$

$\lfloor \cdot \rceil$ indicates rounding to the nearest integer

scale factor

- **Zeropoint quantization**

$$nd_{x_{f16}} = \frac{2 \cdot 127}{\max_{ij}(\mathbf{X}_{f16}^{ij}) - \min_{ij}(\mathbf{X}_{f16}^{ij})}$$

$$\mathbf{X}_{i8} = \lfloor nd_{x_{f16}} \mathbf{X}_{f16} \rceil$$

scale factor

- **Matrix Multiplication with 8-bits**

$$\begin{aligned} \mathbf{X}_{f16}\mathbf{W}_{f16} = \mathbf{C}_{f16} &\approx \frac{1}{c_{x_{f16}} c_{w_{f16}}} \mathbf{C}_{i32} = S_{f16} \cdot \mathbf{C}_{i32} \\ &\approx S_{f16} \cdot \mathbf{A}_{i8}\mathbf{B}_{i8} = S_{f16} \cdot Q(\mathbf{A}_{f16})\, Q(\mathbf{B}_{f16}) \end{aligned}$$

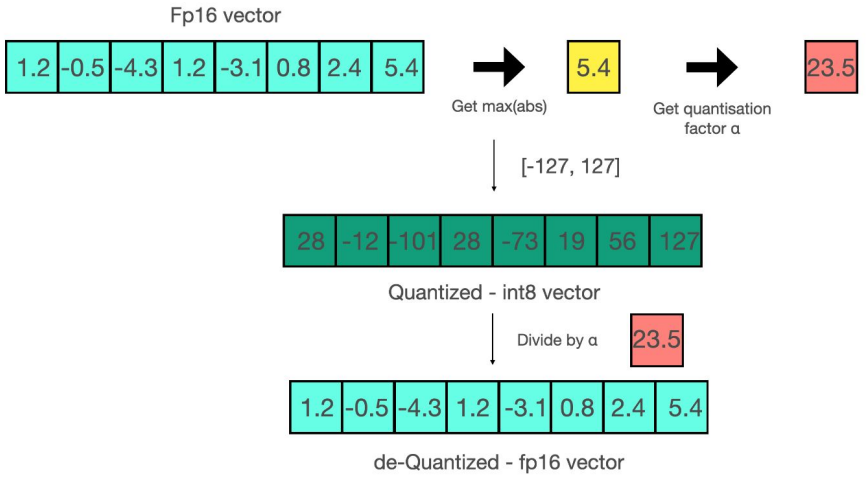
LLM.int8() Background

Quantization Types

➢ Maximum absolute quantization

$$\mathbf{X}_{i8} = \left\lfloor \frac{127 \cdot \mathbf{X}_{f16}}{\max_{ij}(|\mathbf{X}_{f16_{ij}}|)} \right\rceil = \left\lfloor \frac{127}{\|\mathbf{X}_{f16}\|_\infty} \mathbf{X}_{f16} \right\rceil$$

$\lfloor \cdot \rceil$ indicates rounding to the nearest integer



➢ Zeropoint quantization

$$nd_{x_{f16}} = \frac{2 \cdot 127}{\max_{ij}(\mathbf{X}^{ij}_{f16}) - \min_{ij}(\mathbf{X}^{ij}_{f16})}$$

$$\mathbf{X}_{i8} = \lfloor nd_{x_{f16}} \mathbf{X}_{f16} \rceil$$

scale factor

➢ Matrix Multiplication with 8-bits

$$\mathbf{X}_{f16}\mathbf{W}_{f16} = \mathbf{C}_{f16} \approx \frac{1}{c_{x_{f16}} c_{w_{f16}}} \mathbf{C}_{i32} = S_{f16} \cdot \mathbf{C}_{i32}$$

$$\approx S_{f16} \cdot \mathbf{A}_{i8}\mathbf{B}_{i8} = S_{f16} \cdot Q(\mathbf{A}_{f16})\, Q(\mathbf{B}_{f16})$$

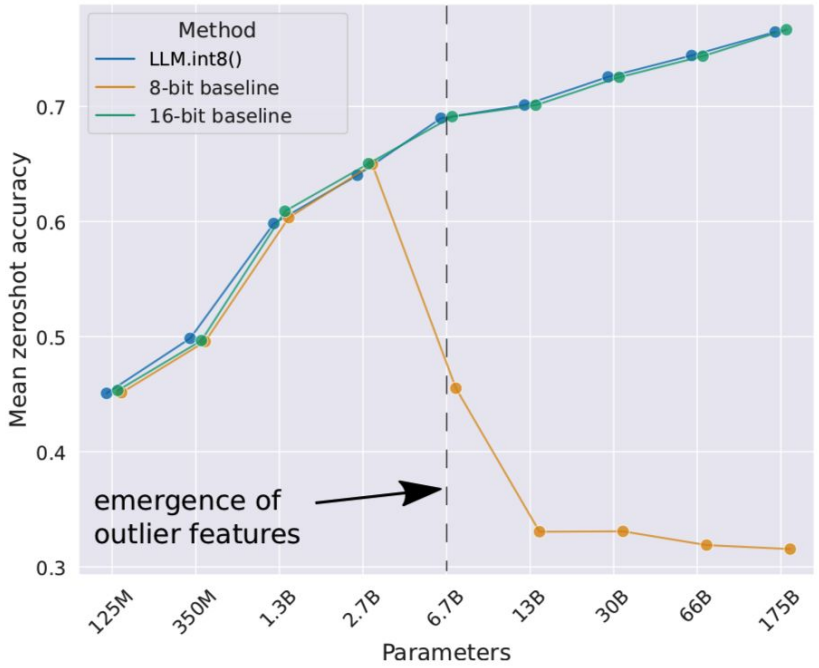
LLM.int8() Background

$$\mathbf{X}_{i8} = \left\lfloor \frac{127 \cdot \mathbf{X}_{f16}}{\max_{ij}(|\mathbf{X}_{f16_{ij}}|)} \right\rfloor = \left\lfloor \frac{127}{\|\mathbf{X}_{f16}\|_\infty} \mathbf{X}_{f16} \right\rfloor$$

Outlier Problem

Extreme magnitude features

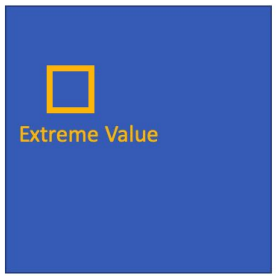
- Appear when model size > 2.7B
- ~6.7B, a phase shift occurs, affecting all transformer layers and 75% of all sequence dimensions
- systematic as the model size grows, but they are concentrated in few dimensions, e.g. 6.7B model has ~150,000 outliers concentrating in 6 dimensions.



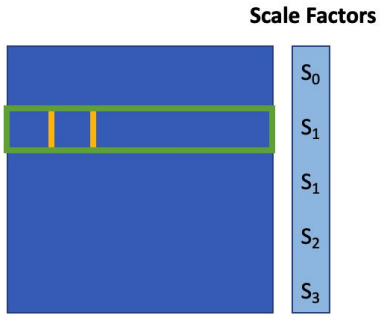
LLM.int8() Methodology

LLM.int8() Main Idea: remedy the impact of outlier

➢ **Tensor-wise ⇒ Vector-wise Quantization**



- Tensor-wise
  - Contributes to large scale factor
  - Reduce the quantization precision of all other values



- Vector-wise
  - individual scale factor for each vector
- affects values within the same vector

➢ **Mixed-precision Decomposition**

- **Decompose data into two sub-matrices: large magnitude features and outlier features**
- **Large magnitude features: quantizing into Int8**
- **Outlier features: processing in fp16**

LLM.int8() Methodology

Int8 Quantization

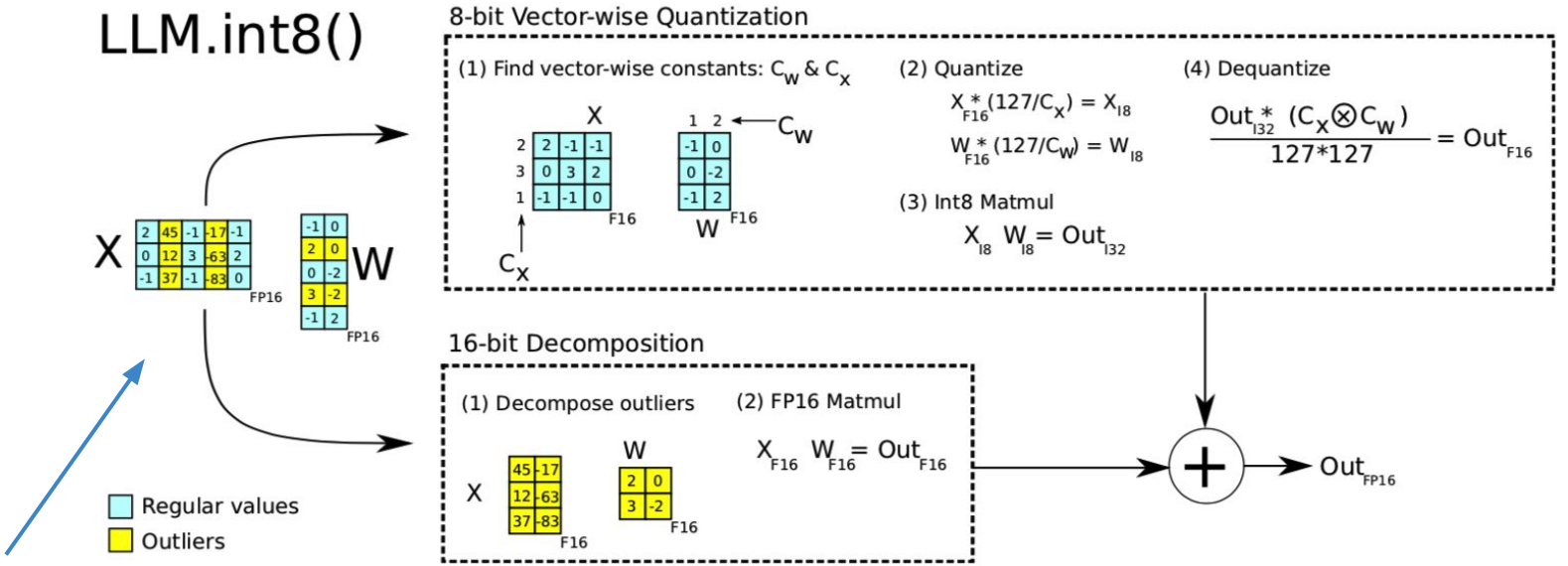
➢ **Vector-wise Quantization**

$$\mathbf{C}_{f16} \approx \frac{1}{\mathbf{c}_{x_{f16}} \otimes \mathbf{c}_{w_{f16}}} \mathbf{C}_{i32} = S \cdot \mathbf{C}_{i32} = \mathbf{S} \cdot \mathbf{A}_{i8}\mathbf{B}_{i8} = \mathbf{S} \cdot Q(\mathbf{A}_{f16})\ Q(\mathbf{B}_{f16})$$

➢ **Mixed-precision Decomposition & Dequantization**

$$\mathbf{C}_{f16} \approx \sum_{h \in O} \mathbf{X}^h_{f16}\mathbf{W}^h_{f16} + \mathbf{S}_{f16} \cdot \sum_{h \notin O} \mathbf{X}^h_{i8}\mathbf{W}^h_{i8}$$

LLM.int8() Methodology



Outliers are filtered
by a threshold value

LLM.int8() Experimental Results

Table 1: C4 validation perplexities of quantization methods for different transformer sizes ranging from 125M to 13B parameters. We see that absmax, row-wise, zeropoint, and vector-wise quantization leads to significant performance degradation as we scale, particularly at the 13B mark where 8-bit 13B perplexity is worse than 8-bit 6.7B perplexity. If we use LLM.int8(), we recover full perplexity as we scale. Zeropoint quantization shows an advantage due to asymmetric quantization but is no longer advantageous when used with mixed-precision decomposition.

Parameters	125M	1.3B	2.7B	6.7B	13B
32-bit Float	25.65	15.91	14.43	13.30	12.45
Int8 absmax	87.76	16.55	15.11	14.59	19.08
Int8 zeropoint	56.66	16.24	14.76	13.49	13.94
Int8 absmax row-wise	30.93	17.08	15.24	14.13	16.49
Int8 absmax vector-wise	35.84	16.82	14.98	14.13	16.48
Int8 zeropoint vector-wise	25.72	15.94	14.36	13.38	13.47
Int8 absmax row-wise + decomposition	30.76	16.19	14.65	13.25	12.46
Absmax LLM.int8() (vector-wise + decomp)	25.83	15.93	14.44	**13.24**	**12.45**
Zeropoint LLM.int8() (vector-wise + decomp)	**25.69**	**15.92**	**14.43**	**13.24**	**12.45**

LLM.int8() Experimental Results

Table 2: Different hardware setups and which methods can be run in 16-bit vs. 8-bit precision. We can see that our 8-bit method makes many models accessible that were not accessible before, in particular, OPT-175B/BLOOM.

			Largest Model that can be run	
Class	Hardware	GPU Memory	8-bit	16-bit
Enterprise	8x A100	80 GB	**OPT-175B / BLOOM**	**OPT-175B / BLOOM**
Enterprise	8x A100	40 GB	**OPT-175B / BLOOM**	OPT-66B
Academic server	8x RTX 3090	24 GB	**OPT-175B / BLOOM**	OPT-66B
Academic desktop	4x RTX 3090	24 GB	**OPT-66B**	OPT-30B
Paid Cloud	Colab Pro	15 GB	**OPT-13B**	GPT-J-6B
Free Cloud	Colab	12 GB	**T0/T5-11B**	GPT-2 1.3B

Hardware Support

Table 5: Inference speedups compared to 16-bit matrix multiplication for the first hidden layer in the feed-forward of differently sized GPT-3 transformers. The hidden dimension is 4x the model dimension. The 8-bit without overhead speedups assumes that no quantization or dequantization is performed. Numbers small than 1.0x represent slowdowns. Int8 matrix multiplication speeds up inference only for models with large model and hidden dimensions.

GPT-3 Size	Small	Medium	Large	XL	2.7B	6.7B	13B	175B
Model dimension	768	1024	1536	2048	2560	4096	5140	12288
FP16-bit baseline	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x
Int8 without overhead	0.99x	1.08x	1.43x	1.61x	1.63x	1.67x	2.13x	2.29x
Absmax PyTorch+NVIDIA	0.25x	0.24x	0.36x	0.45x	0.53x	0.70x	0.96x	1.50x
Vector-wise PyTorch+NVIDIA	0.21x	0.22x	0.33x	0.41x	0.50x	0.65x	0.91x	1.50x
Vector-wise	**0.43x**	**0.49x**	**0.74x**	**0.91x**	**0.94x**	**1.18x**	**1.59x**	**2.00x**
LLM.int8() (vector-wise+decomp)	0.14x	0.20x	0.36x	0.51x	0.64x	0.86x	1.22x	1.81x

Speedup Ratio

LLM.int8() Experimental Results

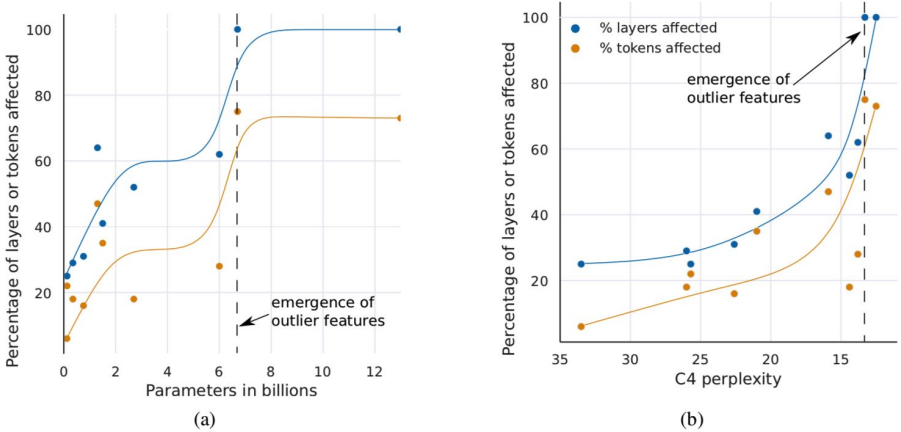


Figure 3: Percentage of layers and all sequence dimensions affected by large magnitude outlier features across the transformer by (a) model size or (b) C4 perplexity. Lines are B-spline interpolations of 4 and 9 linear segments for (a) and (b). Once the phase shift occurs, outliers are present in all layers and in about 75% of all sequence dimensions. While (a) suggest a sudden phase shift in parameter size, (b) suggests a gradual exponential phase shift as perplexity decreases. The stark shift in (a) co-occurs with the sudden degradation of performance in quantization methods.

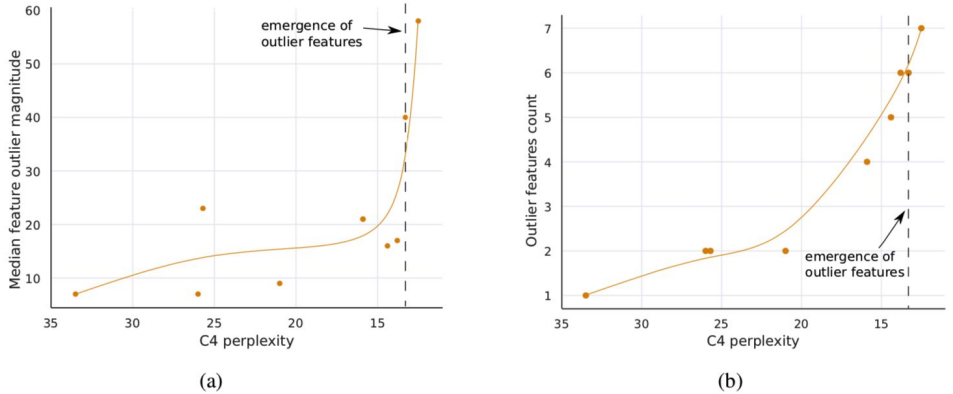


Figure 4: The median magnitude of the largest outlier feature in (a) indicates a sudden shift in outlier size. This appears to be the prime reason why quantization methods fail after emergence. While the number of outlier feature dimensions is only roughly proportional to model size, (b) shows that the number of outliers is *strictly monotonic* with respect to perplexity across all models analyzed. Lines are B-spline interpolations of 9 linear segments.

- The emergence of large magnitude features across all layers of a transformer occurs suddenly between 6B and 6.7B parameters
- Median outlier feature magnitude rapidly increases once outlier features occur in all layers of the transformer. The range of the quantization distribution is too large ==> most quantization bins are empty + small quantization values are quantized to zero ==> quantization fails appear around 6.7B

Memory Efficient training and inference

Outline

1. LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale
2. **GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers**
3. SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models

GPTQ Background

- Huge Memory Consumption
  GPT3-175B: fp16, 326GB GPU memory
- Model Compression (or Knowledge Distillation)
  Need re-training, highly expensive
- Post Training
  Compress the model in one shot without retraining

GPTQ Background

Prior Post-Quantization Method

- Layer-Wise Quantization

Quantization Objective:

$$\text{argmin}_{\widehat{\mathbf{W}}} ||\mathbf{WX} - \widehat{\mathbf{W}}\mathbf{X}||_2^2$$

W: the weights of a layer
$\widehat{\mathbf{W}}$: the quantized weights
X: the layer input

- Optimal Brain Quantization

Quantization Objective:

$$\text{argmin}_{\widehat{\mathbf{W}}} (\mathbf{WX} - \widehat{\mathbf{W}}\mathbf{X})^2$$

GPTQ Background

Prior Post-Quantization Method

➢ Optimal Brain Quantization

GPTQ adopts OBQ

W: the weights of a layer

$\widehat{\mathbf{W}}$: the quantized weights

X: the layer input

Quantization Objective:

$$\text{argmin}_{\widehat{\mathbf{W}}} \left(\mathbf{WX} - \widehat{\mathbf{W}}\mathbf{X}\right)^2$$

quadratic, convex optimization

GPTQ Background

Prior Post-Quantization Method

➢ Optimal Brain Quantization

Quantization Procedure:

Two quantization phases:

for q in range(num-weights):
    Greedy search the weight $\boldsymbol{w_q}$ in **W** that has minimum quantization error
    Update remaining weights w.r.t. each $\boldsymbol{w_q}$

GPTQ Background

Prior Post-Quantization Method

➢ Optimal Brain Quantization

Quantization Procedure:

Procedure 1: Greedy Search & Quantization

$$w_q = \text{argmin}_{w_q} \frac{(\text{quant}(w_q) - w_q)^2}{[\mathbf{H}_F^{-1}]_{qq}}$$

Since the OBQ objective is quadratic, the Hessian matrix is $\mathbf{H} = 2\mathbf{X}\mathbf{X}^T$

, F denotes the set of remaining full-precision weights

quant(w) rounds w to the nearest value on the quantization grid

GPTQ Background

Prior Post-Quantization Method

➢ Optimal Brain Quantization

Quantization Procedure:

Procedure 2: Update Remaining Weights

$$\delta_F = -\frac{w_q - \text{quant}(w_q)}{[\mathbf{H}_F^{-1}]_{qq}} \cdot (\mathbf{H}_F^{-1})_{:,q}$$

Update remaining weights for each selected weight

GPTQ Background

Prior Post-Quantization Method

➢ Optimal Brain Quantization

Quantization Procedure:

efficient update of the inverse matrix

$$\mathbf{H}_{-q}^{-1} = \left(\mathbf{H}^{-1} - \frac{1}{[\mathbf{H}^{-1}]_{qq}} \mathbf{H}_{:,q}^{-1} \mathbf{H}_{q,:}^{-1}\right)_{-p}$$

Hessian matrix for the weight tensor can be calculated once, and applied for both computing the optimal weight and optimization gradient of remaining weights.

GPTQ Methodology

Motivation

- GPTQ is based on OBQ
- Computational complexity of OBQ: $O(d_{\text{row}} \cdot d_{\text{col}}^3)$
- Low compute/memory-access ratio
- Numerical inaccuracy with the model size grows

Major Strategies

- Element-wise to column-wise quantization
- Lazy batch-updates
- Cholesky Reformulation

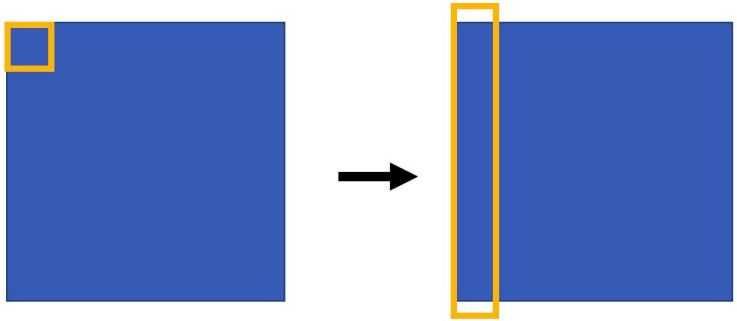
GPTQ Methodology

Implementation

➢ **Element-wise to column-wise quantization**

- ❑ $\mathbf{H}_F$ relies only on the layer input $\mathbf{X}_F$ that is the same for all rows. It does not depend on any weights. ⇒ processing weights in each column in parallel.
- ❑ Runtime reduction with parallel computation: for LLM $d_{col} >> d_{row}$

$$O(d_{\text{row}} \cdot d_{\text{col}}^3) \Rightarrow O(\max\{d_{\text{row}} \cdot d_{\text{col}}^2, d_{\text{col}}^3\})$$



GPTQ Methodology

Implementation

➢ **Lazy batch-updates**

$$\boldsymbol{\delta}_F = -(\mathbf{w}_Q - \text{quant}(\mathbf{w}_Q))([\mathbf{H}_F^{-1}]_{QQ})^{-1}(\mathbf{H}_F^{-1})_{:,Q}$$
$$\mathbf{H}_{-Q}^{-1} = \left(\mathbf{H}^{-1} - \mathbf{H}_{:,Q}^{-1}([\mathbf{H}^{-1}]_{QQ})^{-1}\mathbf{H}_{Q,:}^{-1}\right)_{-Q}$$

- ❏ Update weights (at the second phase) after quantizing *B* columns;
- ❏ since the inverse matrix of **H** can be computed in advance, we just calculate the inverse matrix once. The OBQ method needs to compute it every time updating remaining weights for the quantization of each single weight (high memory access & modification).

GPTQ Methodology

Implementation

Algorithm 1 Quantize $\mathbf{W}$ given inverse Hessian $\mathbf{H}^{-1} = (2\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I})^{-1}$ and blocksize B.

$\mathbf{Q} \leftarrow \mathbf{0}_{d_{\text{row}} \times d_{\text{col}}}$ // *quantized output*
$\mathbf{E} \leftarrow \mathbf{0}_{d_{\text{row}} \times B}$ // *block quantization errors*
$\mathbf{H}^{-1} \leftarrow \text{Cholesky}(\mathbf{H}^{-1})^\top$ // *Hessian inverse information*
for $i = 0, B, 2B, \ldots$ **do**
update remaining weights for each B columns
  for $j = i, \ldots, i + B - 1$ **do**
    $\mathbf{Q}_{:,j} \leftarrow \text{quant}(\mathbf{W}_{:,j})$ // *quantize column*
    $\mathbf{E}_{:,j-i} \leftarrow (\mathbf{W}_{:,j} - \mathbf{Q}_{:,j}) \,/\, [\mathbf{H}^{-1}]_{jj}$ // *quantization error*
    $\mathbf{W}_{:,j:(i+B)} \leftarrow \mathbf{W}_{:,j:(i+B)} - \mathbf{E}_{:,j-i} \cdot \mathbf{H}^{-1}_{j,j:(i+B)}$ // *update weights in block*
  end for
  $\mathbf{W}_{:,(i+B):} \leftarrow \mathbf{W}_{:,(i+B):} - \mathbf{E} \cdot \mathbf{H}^{-1}_{i:(i+B),(i+B):}$ // *update all remaining weights*
end for

GPTQ Experimental Results

Performance After Quantizing

➢ **Vision Model Quantization**

- ❏ Quantizing ResNet18 and ResNet50 in image classification task
- ❏ Competitive to prior SOTA quantization methods

Method	RN18 – 69.76 %		RN50 – 76.13%	
	4bit	3bit	4bit	3bit
AdaRound	69.34	68.37	75.84	75.14
AdaQuant	68.12	59.21	74.68	64.98
BRECQ	69.37	68.47	75.88	75.32
OBQ	69.56	68.69	75.72	75.24
GPTQ	69.37	67.88	75.71	74.87

Table 1: Comparison with state-of-the-art post-training methods for vision models.

GPTQ Experimental Results

Performance After Quantizing

➢ **Language Model Quantization**

- ❑ Quantizing OPT with varied sizes
- ❑ Achieving better perplexity result
- ❑ RTN: simple Round-To-Nearest quantization

OPT	Bits	125M	350M	1.3B	2.7B	6.7B	13B	30B	66B	175B
full	16	27.65	22.00	14.63	12.47	10.86	10.13	9.56	9.34	8.34
RTN	4	37.28	25.94	48.17	16.92	12.10	11.32	10.98	110	10.54
GPTQ	4	**31.12**	**24.24**	**15.47**	**12.87**	**11.39**	**10.31**	**9.63**	**9.55**	**8.37**
RTN	3	1.3e3	64.57	1.3e4	1.6e4	5.8e3	3.4e3	1.6e3	6.1e3	7.3e3
GPTQ	3	**53.85**	**33.79**	**20.97**	**16.88**	**14.86**	**11.61**	**10.27**	**14.16**	**8.68**

Table 3: OPT perplexity results on WikiText2.

GPTQ Experimental Results

Minor gap in performance

Performance After Quantizing

➢ Language Model Quantization

- ❏ Quantizing BLOOM with varied sizes
- ❏ Achieving better perplexity results

BLOOM	Bits	560M	1.1B	1.7B	3B	7.1B	176B
full	16	22.42	17.69	15.39	13.48	11.37	8.11
RTN	4	25.90	22.00	16.97	14.76	12.10	8.37
GPTQ	4	**24.03**	**19.05**	**16.48**	**14.20**	**11.73**	**8.21**
RTN	3	57.08	50.19	63.59	39.36	17.38	571
GPTQ	3	**32.31**	**25.08**	**21.11**	**17.40**	**13.47**	**8.64**

Table 4: BLOOM perplexity results for WikiText2.

Speedup & Memory-Reduce

GPU	FP16	3bit	Speedup	GPU reduction
A6000 – 48GB	589ms	130ms	4.53×	8 → 2
A100 – 80GB	230ms	71ms	3.24×	5 → 1

pile in single GPU

Table 6: Average per-token latency (batch size 1) when generating sequences of length 128.

GPTQ Experimental Results

Impact of Group-Size

Model	FP16	g128	g64	g32	3-bit
OPT-175B	8.34	9.58	9.18	8.94	8.68
BLOOM	8.11	9.55	9.17	8.83	8.64

Table 7: 2-bit GPTQ quantization results with varying group-sizes; perplexity on WikiText2.

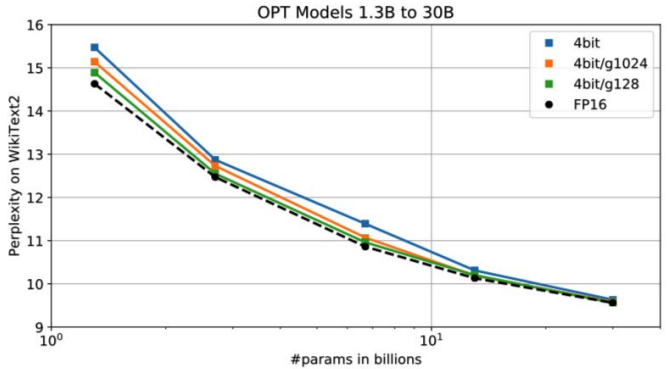


Figure 4: GPTQ at 4-bit with different group-sizes on medium sized OPT models.

- Grouping *g* consecutive weights, applying independent quantization to these weights. Performance drop is acceptable even in 2-bits quantization.
- For medium size models (1.3B to 30B), applying quantization to groups of weights seems helpful to reduce the performance drop.

Memory Efficient training and inference

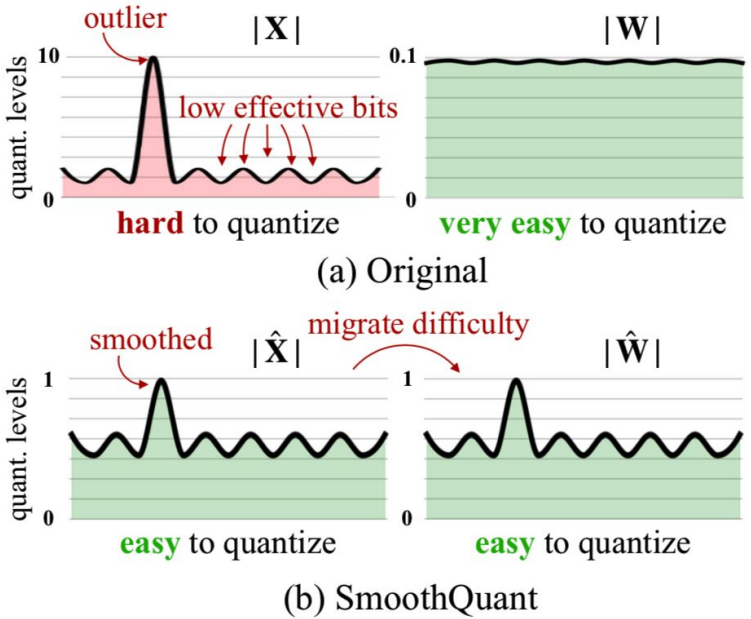
Outline

1. LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale
2. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers
3. **SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models**

SmoothQuant

Motivation

- Activations are harder to quantize than weights
- Activation quantization difficulties come from outliers. These outliers also concentrate within few channels
- Migrate the difficulty of quantizing activations into the weights quantization



SmoothQuant

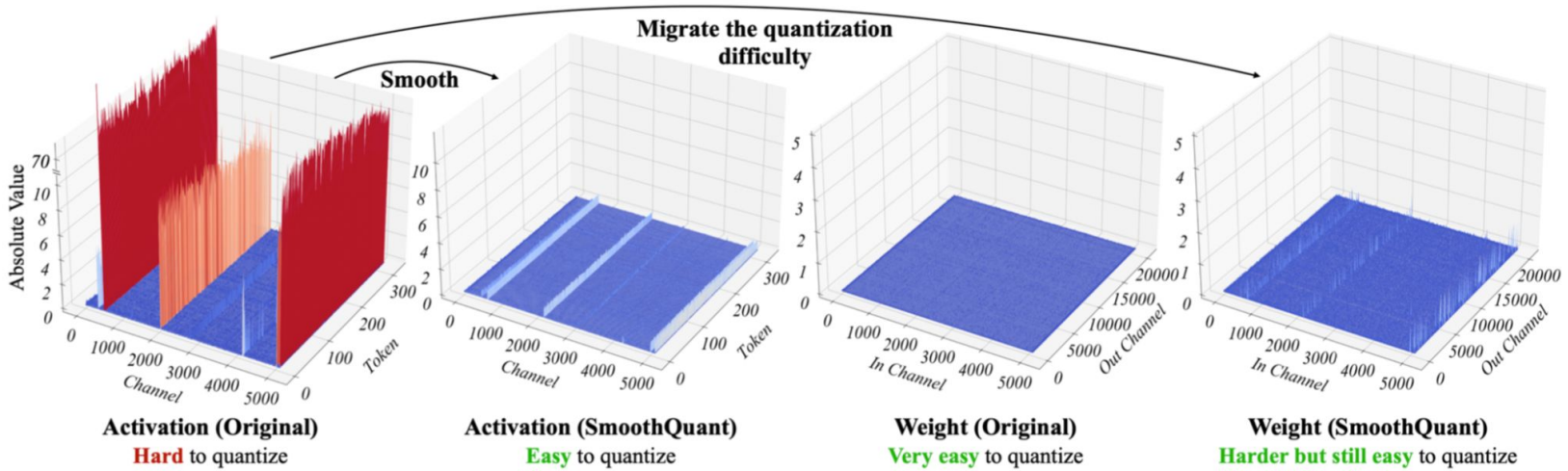


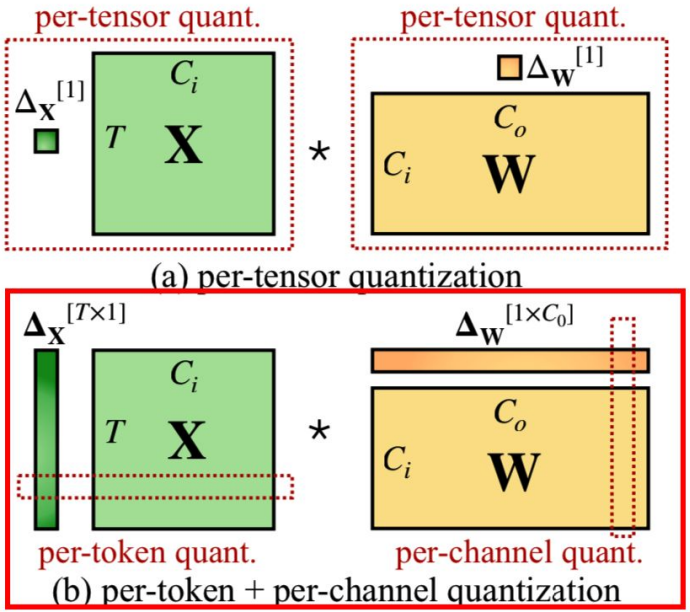
Figure: Magnitude of the input activations and weights of a linear layer in OPT-13B before and after SmoothQuant.

- SmoothQuant migrates the outlier channels from activation to weight
- There are a few channels in the original activation map whose magnitudes are very large (greater than 70)
- Generally, the variance in one activation channel is small
- The original weight distribution is flat and uniform
- Smoothing before migrating to reduce the variance of the quantized weights.

SmoothQuant

Implementation

- ➢ Per-Token & Per-Channel Quantization
  - ○ Finer-grained quantization than per-tensor quantization
  - ○ Manipulate in both feature tensor and weight tensor
  - ○ Effectively utilize the INT8 GEMM kernels



SmoothQuant

Implementation

- Smooth activation
  - dividing it by a per-channel smoothing factor s (diagonal matrix)
  - To keep the mathematical equivalence of a linear layer, scale the weights accordingly in the reversed direction:

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

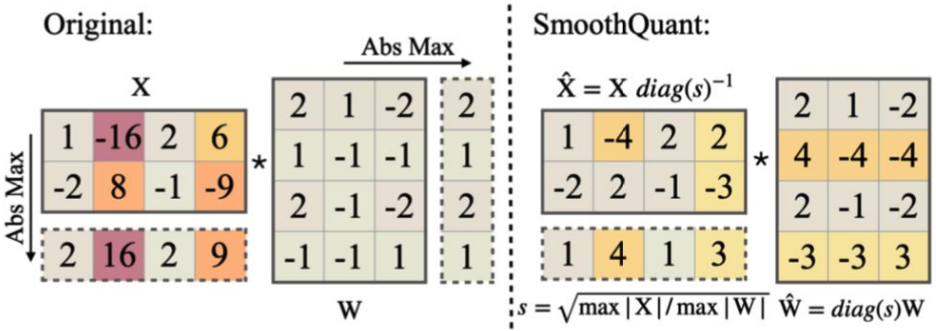
  - Migration strength is controlled with the hyper-parameter α

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

SmoothQuant

Implementation

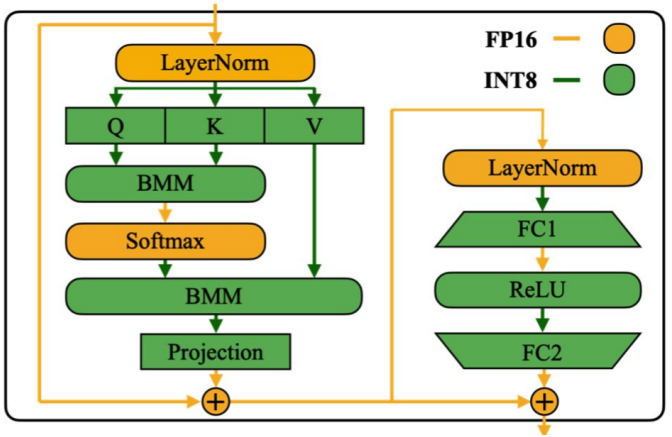
➢ Smooth the activation into low variance



α=0.5 here

Smoothing the variance of feature tensor, and increasing the variance of weight tensor

➢ Precision mapping for a Transformer block



The input, weights, and computation-heavy optations are quantized with INT8

SmoothQuant

Experimental Results

➢ Different activation quantization

Table 1: Among different activation quantization schemes, only per-channel quantization (Bondarenko et al., 2021) preserves the accuracy, but it is *not* compatible (marked in gray) with INT8 GEMM kernels. We report the average accuracy on WinoGrande, HellaSwag, PIQA, and LAMBADA.

Model size (OPT-)	6.7B	13B	30B	66B	175B
FP16	64.9%	65.6%	67.9%	69.5%	71.6%
INT8 per-tensor	39.9%	33.0%	32.8%	33.1%	32.3%
INT8 per-token	42.5%	33.0%	33.1%	32.9%	31.7%
INT8 per-channel	64.8%	65.6%	68.0%	69.4%	71.4%

coarse-grained

↓

fine-grained

Perform well in maintaining LLM performance

➢ Outlier control vs Varied model sizes

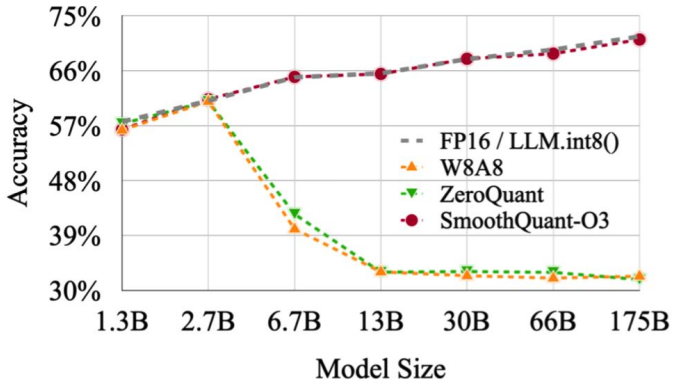


Figure 7: SmoothQuant-O3 (the most efficient setting, defined in Table 2) preserves the accuracy of OPT models across different scales when quantized to INT8. LLM.int8() requires mixed precision and suffers from slowing down.

SmoothQuant

Experimental Results

➢ Downstream tasks: 7 zero-shot benchmarks

Table 3: SmoothQuant maintains the accuracy of OPT-175B model after INT8 quantization, even with the most aggressive and most efficient O3 setting (Table 2). We extensively benchmark the performance on 7 zero-shot benchmarks (by reporting the average accuracy) and 1 language modeling benchmark (perplexity). *For ZeroQuant, we also tried leaving the input activation of self-attention in FP16 and quantizing the rest to INT8, which is their solution to the GPT-NeoX-20B. But this does not solve the accuracy degradation of OPT-175B.

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

No degradation in performance; competitive to LLM.int8(); Faster inference

SmoothQuant

Experimental Results

➢ Migration ratio study

- α is too small: the activation is hard to quantize, resulting to performance degradation
- α is too large: migrates large amount of outlier triggered by input features.

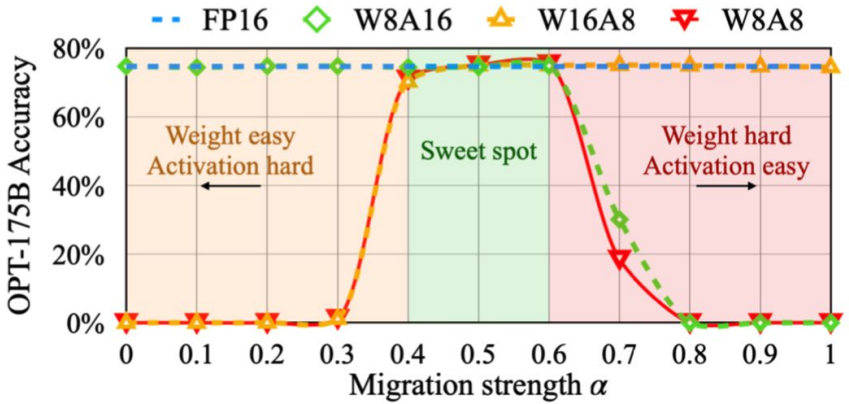


Figure 10: A suitable migration strength α (sweet spot) makes both activations and weights easy to quantize. If the α is too large, weights will be hard to quantize; if too small, activations will be hard to quantize.

SmoothQuant

Experimental Results

➢ Speedup and Memory Efficiency

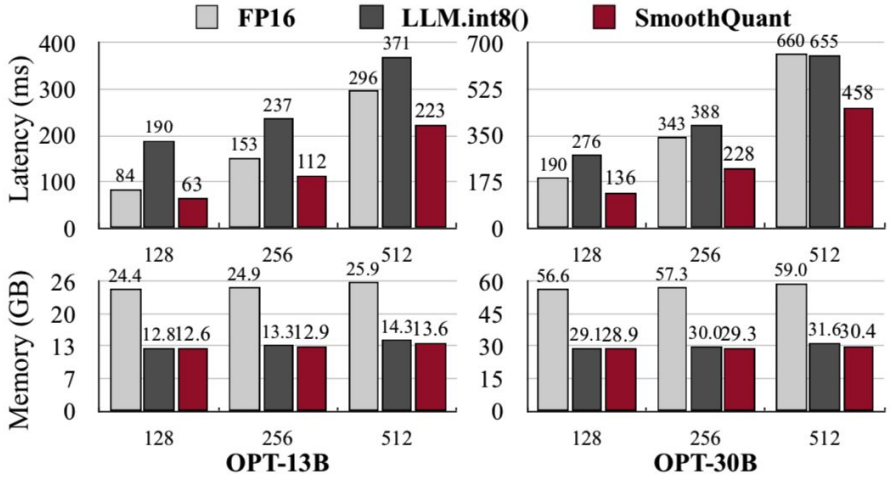


Table 7: SmoothQuant 's performance in the decoding stage.

BS	SeqLen	Latency (ms)			Memory (GB)		
		FP16	Ours	Speedup (↑)	FP16	Ours	Saving (↑)
		OPT-30B (1 GPU)					
1	512	422	314	1.35×	57	30	1.91×
1	1024	559	440	1.27×	58	31	1.87×
16	512	2488	1753	1.42×	69	44	1.59×
16	1024	OOM	3947	-	OOM	61	-
		OPT-175B (8 GPUs)					
1	512	426	359	1.19×	44	23	1.87×
1	1024	571	475	1.20×	44	24	1.85×
16	512	2212	1628	1.36×	50	30	1.67×
16	1024	4133	3231	1.28×	56	37	1.52×

competitive to LLM.int8(); Faster inference; ~1.8x memory saving

Summarization of this talk

- **Computation and Memory-friendly Training Methods**
  - Megatron-LM: Partition model parameters to save memory
  - ZeRO: Partition optimizer states to save memory, ZeRo1 ~ ZeRo3
  - FlashAttention: Efficient attention without sacrificing the performance

- **Computation and Memory-saving Inference Methods**
  - LLM.Int8(): Propose mixed-precision decomposition to remedy outliers
  - GPTQ: Increase the computation efficiency of OBQ and decrease the impact of outliers with vector-wise quantization
  - SmoothQuant: Quantize the activations and migrate the quantization difficulty into weights to solve the harder quantization problem.

Discussion

1. Can we have a language model modeling infinite context length and what are the challenges?
2. Can we perform CUDA operations fusion on other part of Transformers? and what are the drawbacks or limitations of optimizing the model from CUDA?
3. Can we carry out FP8 quantization instead of Int8 quantization? and what are the cons. and pros of it?