

# LLM Sys

## LLM Quantization - GPTQ

Lei Li



Language  
Technologies  
Institute

**Carnegie Mellon University**

School of Computer Science

Ack: Kath Choi, Aashiq Muhamed, Zhen Wu, Hongyi Jin, Wayne Wang

# Recap of Basic Quantization Methods

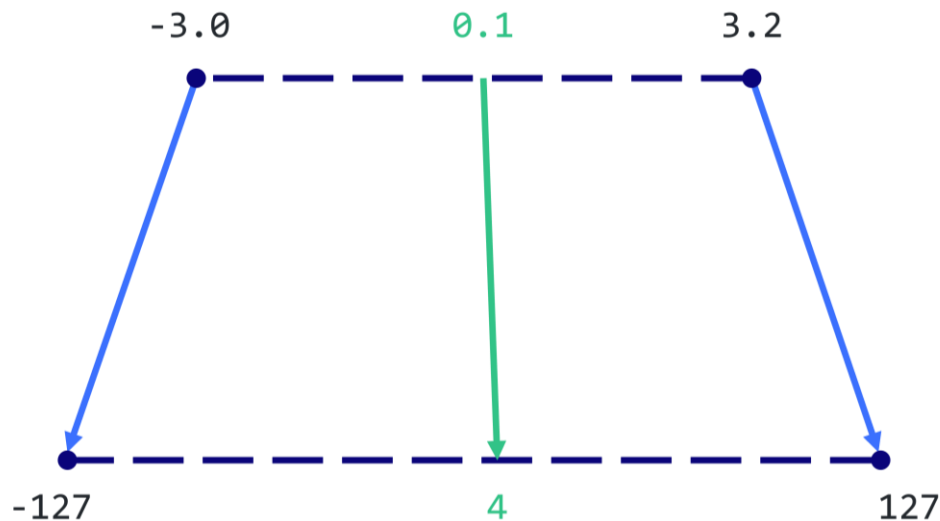
- low-bit number representation in computer
  - BF16: 16-bit half precision floating point numbers, better for ML tasks
  - int8
- Direct quantization of numbers
  - absmax: linearly scale according to max abs value
  - zero-point: finding zero-point and scale
- Layer-wise quantization approaches for NN weights
  - AdaQuant
  - KD: ZeroQuant
  - LLM.int8(): mixed strategy for 8-bit quantization and 16-bit (for outliers)

# Quantize a Number to Int8

- Absmax quant

$$\mathbf{X}_{\text{quant}} = \text{round}\left(\frac{127}{\max|\mathbf{X}|} \cdot \mathbf{X}\right)$$

$$\mathbf{X}_{\text{dequant}} = \frac{\max|\mathbf{X}|}{127} \cdot \mathbf{X}_{\text{quant}}$$



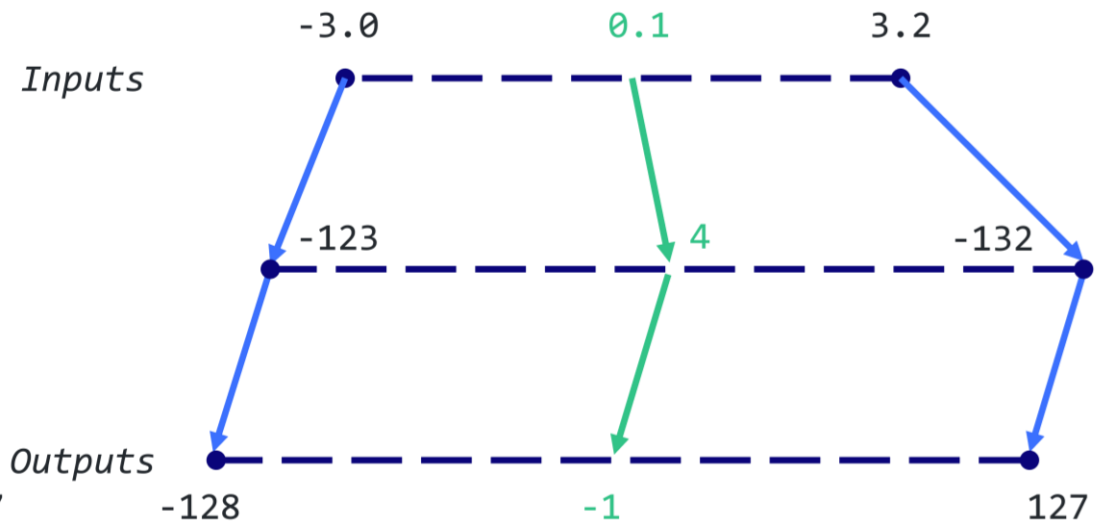
- Zero-point quant

$$\text{scale} = \frac{255}{\max(\mathbf{X}) - \min(\mathbf{X})}$$

$$\text{zeropoint} = -\text{round}(\text{scale} \cdot \min(\mathbf{X})) - 128$$

$$\mathbf{X}_{\text{quant}} = \text{round}\left(\text{scale} \cdot \mathbf{X} + \text{zeropoint}\right)$$

$$\mathbf{X}_{\text{dequant}} = \frac{\mathbf{X}_{\text{quant}} - \text{zeropoint}}{\text{scale}}$$



# GPTQ

- Goal: quantization for very large models while maintain accuracy

# Overall idea of GPTQ

- Revisit layer-wise quantization of weight matrices

$$\operatorname{argmin}_{\hat{W}} \|WX - \hat{W}X\|_2^2$$

X: input data to current layer (d x Len)      W: projection matrix

- Key ideas:
  1. Quantizes one column-block of weights at a time
  2. Updates all the not-yet-quantized weights, to **compensate for the error** incurred by quantizing a single weight

GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. Frantar et al. ICLR 2023.

Optimal Brain Surgeon and General Network Pruning (1993)

Optimal Brain Compression: A framework for accurate post-training quantization and pruning (2022)

# GPTQ algorithm: Overview

1. Pre-compute Cholesky decomposition of the Hessian inverse for input data  $X$  of current (Linear) layer
2. Iteratively handle one batch of columns of weight matrix  $W$ 
  1. quantize the weights using a rounding method
  2. calculates the rounding error
  3. updates the weights in the column block accordingly.
  4. After processing the batch, it updates all remaining weights based on the block's errors.

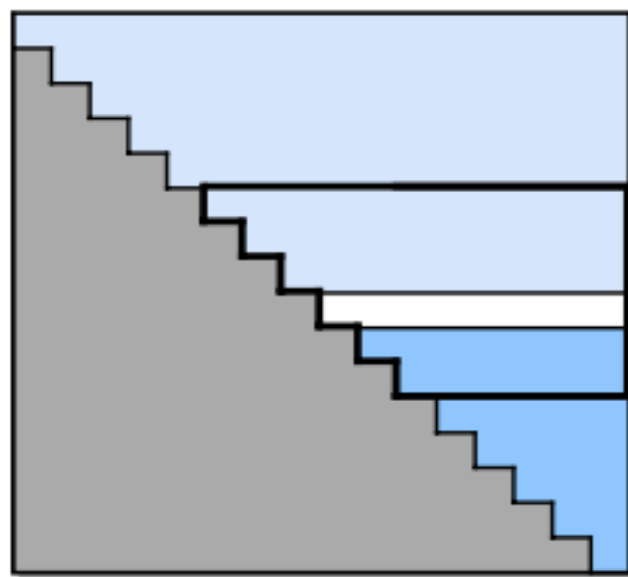
# Pre-compute Hessian Inverse and Decompose

$$G = \text{Cholesky}((2X \cdot X^T + \lambda I)^{-1})^T$$

X is batch\_size × 128k(len) × 7168(dim) in Deepseek-V3

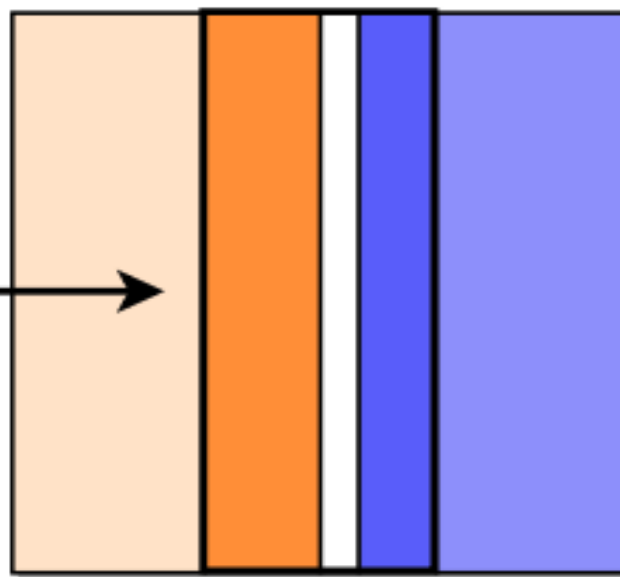
→ 7168 × (bs × 128k)

Inverse Layer Hessian  
(Cholesky Form)



computed initially

Weight Matrix / Block



block *i* quantized recursively  
column-by-column

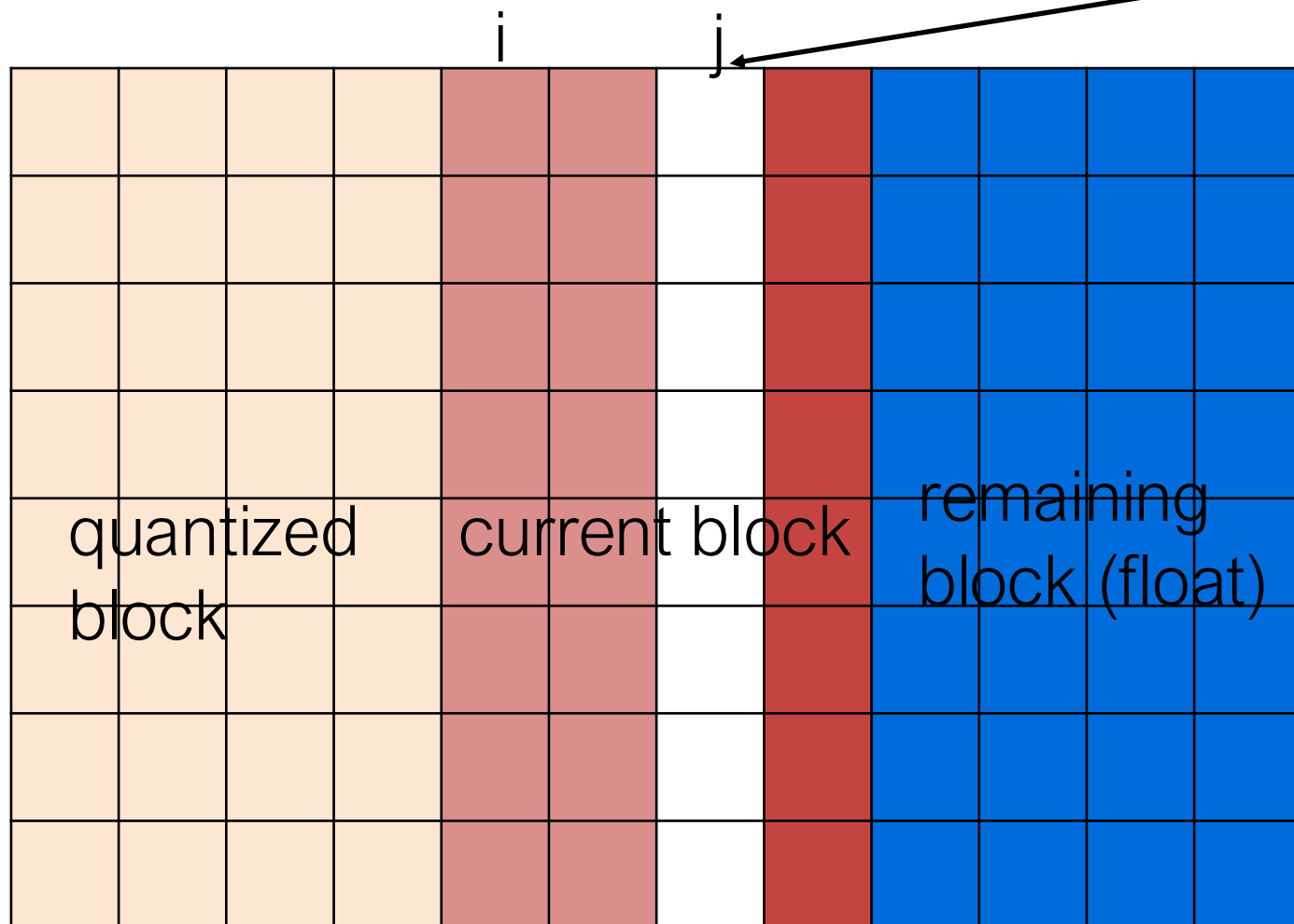


Cholesky decomposition, given a symmetric positive-definite matrix A

$$A = L \cdot L^T$$

# GPTQ: Block-wise Quantize and Update

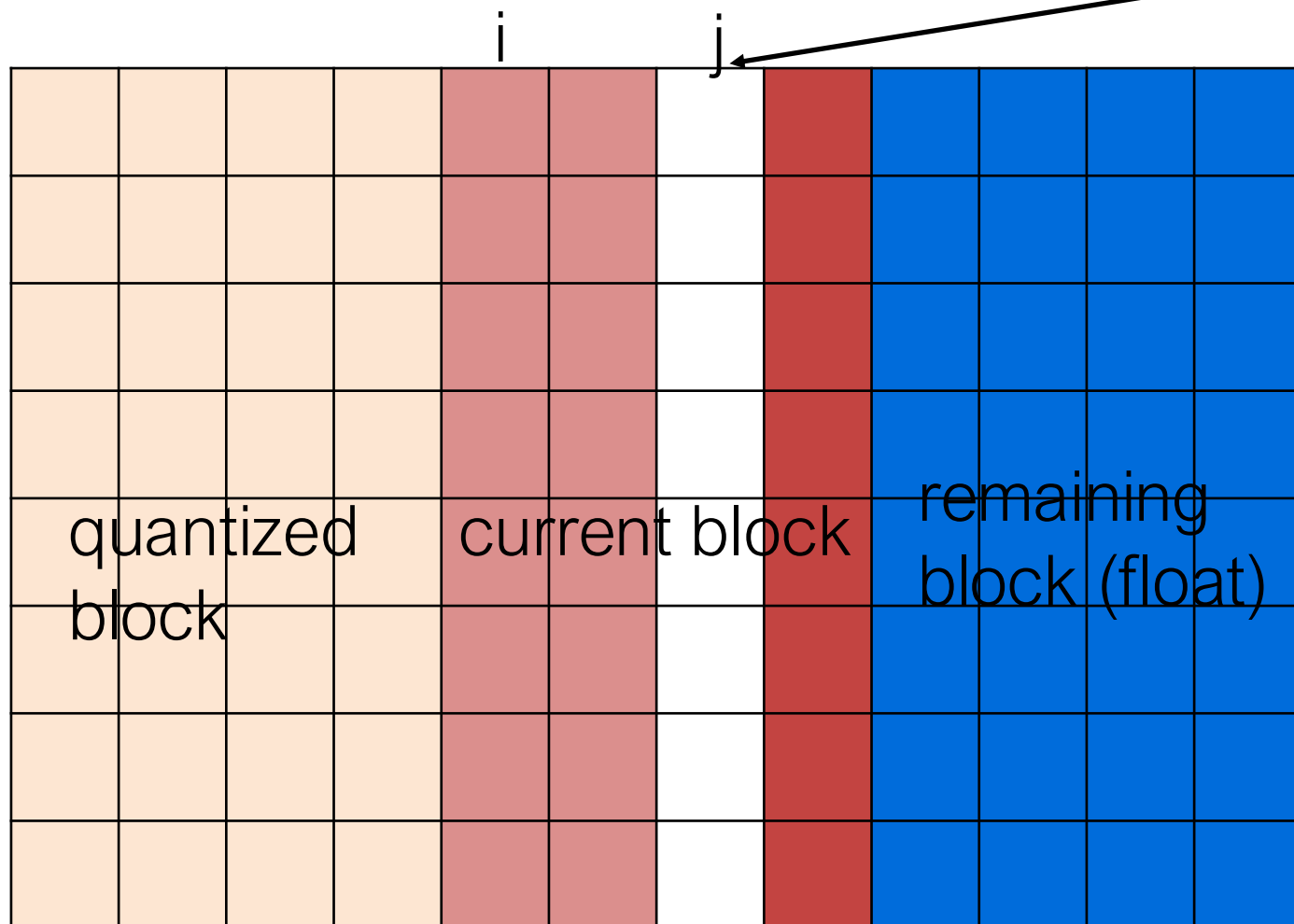
weight matrix  $W$ , block size  $B=4$  current column



1. quantize the column's weights (e.g. using int8 or int4)  
 $q_{:,j} = \text{quant}(W_{:,j})$

# GPTQ: Block-wise Quantize and Update

weight matrix  $W$ , block size  $B=4$  current column



1. quantize the column's weights (e.g. using int8 or int4)

$$q_{:,j} = \text{quant}(W_{:,j})$$

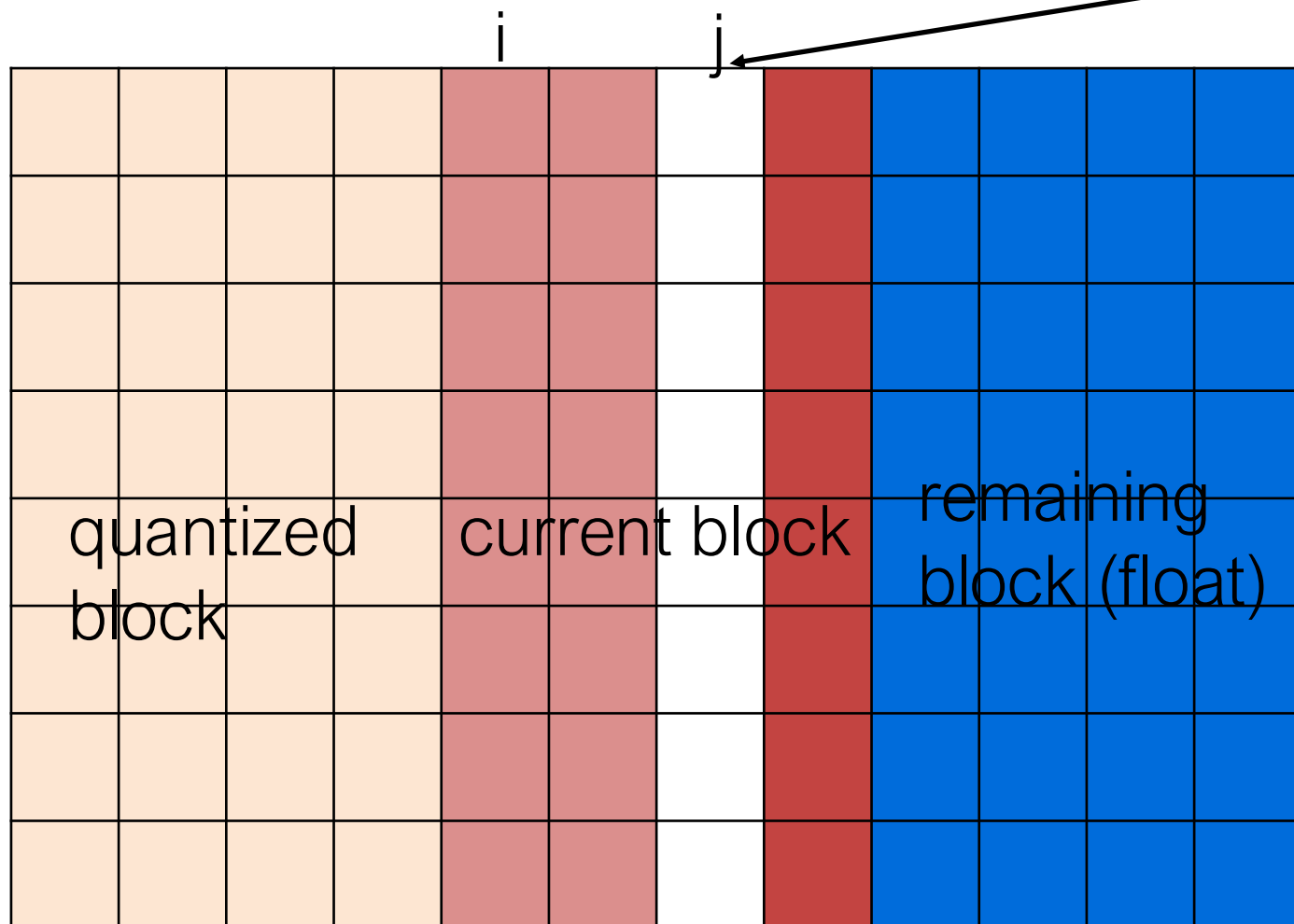
2. calculates the rounding error

$$E_{:,j-i} = (W_{:,j} - Q_{:,j})/G_{j,j}$$

$$G = \text{Cholesky}((2X \cdot X^T + \lambda I)^{-1})^T$$

# GPTQ: Block-wise Quantize and Update

weight matrix  $W$ , block size  $B=4$  current column



1. quantize the column's weights (e.g. using int8 or int4)

$$q_{:,j} = \text{quant}(W_{:,j})$$

2. calculates the rounding error

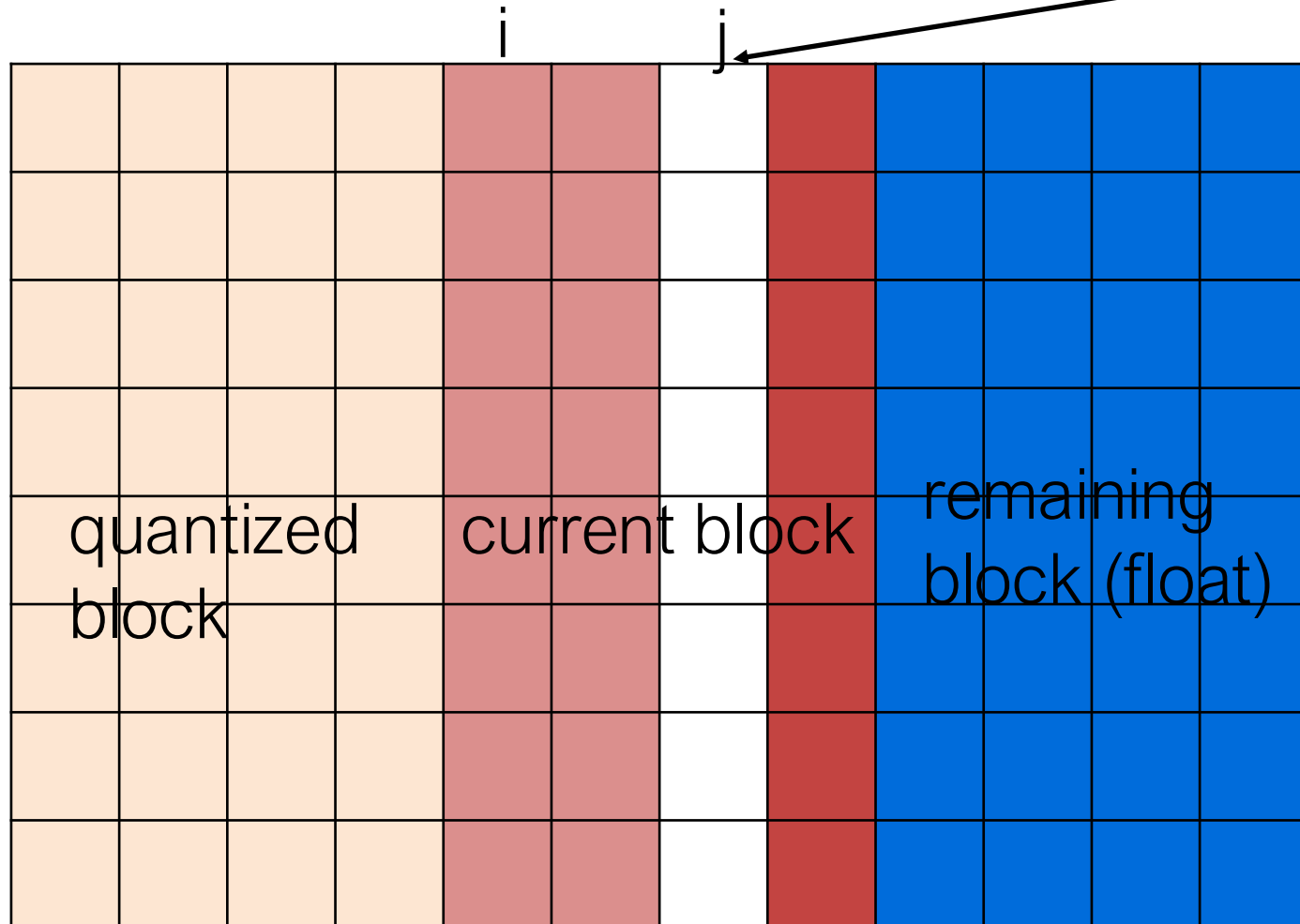
$$E_{:,j-i} = (W_{:,j} - Q_{:,j}) / G_{j,j}$$

3. updates the remaining weights in the column block

$$W_{:,j:(i+B)} = W_{:,j:(i+B)} - E_{:,j-i} \cdot G_{j,j:(i+B)}$$

# GPTQ: Lazy-update for rest weights

weight matrix  $W$ , block size  $B=4$  current column



1. quantize the column's weights  
(e.g. using int8 or int4)

$$q_{:,j} = \text{quant}(W_{:,j})$$

2. calculates the rounding error

$$E_{:,j-i} = (W_{:,j} - Q_{:,j}) / G_{j,j}$$

3. updates the remaining weights  
in the column block

$$W_{:,j:(i+B)} = W_{:,j:(i+B)} - E_{:,j-i} \cdot G_{j,j:(i+B)}$$

After compute the current block

4. update all remaining weights

$$W_{:,i:(i+B):} = W_{:,i:(i+B):} - E \cdot G_{i:(i+B),i:(i+B):}$$

# GPTQ Algorithm

---

**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 =G in previous slides  
**for**  $i = 0, B, 2B, \dots$  **do**  
    **for**  $j = i, \dots, i + B - 1$  **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  
         $\mathbf{W}_{:,j:(i+B)} \leftarrow \mathbf{W}_{:,j:(i+B)} - \mathbf{E}_{:,j-i} \cdot \mathbf{H}_{j,j:(i+B)}^{-1}$  // update weights in block  
    **end for**  
     $\mathbf{W}_{:, (i+B):} \leftarrow \mathbf{W}_{:, (i+B):} - \mathbf{E} \cdot \mathbf{H}_{i:(i+B), (i+B):}^{-1}$  // update all remaining weights  
**end for**

---

# Why GPTQ works?

- Quantize one weight in  $W$  can be solved using the Optimal Brain Surgeon method (OBS)
  - → We can update one column of weights
- Iteratively updating the inverse Hessian can be updated efficiently (rank-1 update using Optimal Brain Quantization, OBQ method)
  - → Using Cholesky to pre-compute
- Updating weights after calculating rounding errors can be done in batch and lazy-fashion

# Optimal Brain Surgeon

- Taylor approximation to find optimal single weight in  $W$  to remove, and optimal update of remaining weights to compensate.
- Weight to prune  $\omega_p$  which incurs the minimal increase in loss and the corresponding update of the remaining weights  $\delta_p$  is,

$$\omega_p = \underset{\omega_p}{\operatorname{argmin}} \omega_p \frac{\omega_p^2}{[H^{-1}]_{pp}}, \delta_p = -\frac{\omega_p}{[H^{-1}]_{pp}} \cdot H_{:,p}^{-1},$$

In transformers,  $\omega_p$  could potentially be LayerNorm/FFN/MHA weights

- $H$  is a  $d \times d$  Hessian matrix where  $d = d_{row} \cdot d_{col}$ , which is expensive to store and compute with.
- $H$  needs to be updated and inverted at  $O(d)$  pruning steps with a  $\Theta(d^3)$  complexity. Total runtime  $O(d^4)$  is too inefficient.

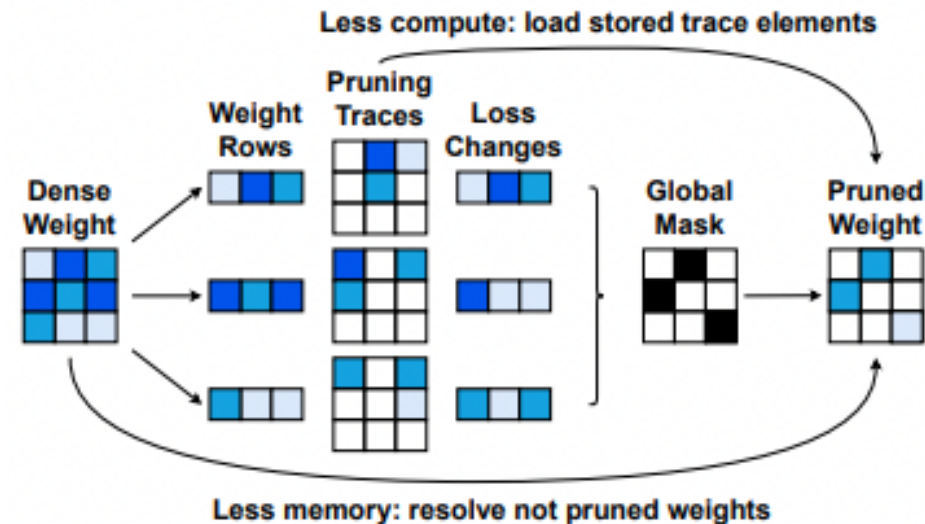
# Optimal Brain Quantization

- OBQ picks the greedy optimal weight  $w_q$  to quantize next, along with the update  $\delta_F$  to all remaining unquantized weights  $F$ .

$$w_q = \underset{w_q}{\operatorname{argmin}} \frac{(\operatorname{quant}(w_q) - w_q)^2}{[H_F^{-1}]_{pp}}, \delta_F = \underset{\text{update}}{\frac{w_q - \operatorname{quant}(w_q)}{[H_F^{-1}]_{pp}}} \cdot (H_F^{-1})_{:,q}.$$

full-precision

- quantizes weights iteratively until all weights are quantized.
- Hessian  $H = 2XX^T$



# Optimal Brain Quantization

1. Row wise decomposition: OBQ applies OBS per row of the weight matrix

$$\sum_{i=1}^{d_{row}} \|W_{i,:X} - \widehat{W}_{i,:X}\|_2^2$$

No Hessian interaction between different rows and so we can work with the individual  $d_{col} \times d_{col}$  H corresponding to each of the  $d_{row}$  rows.

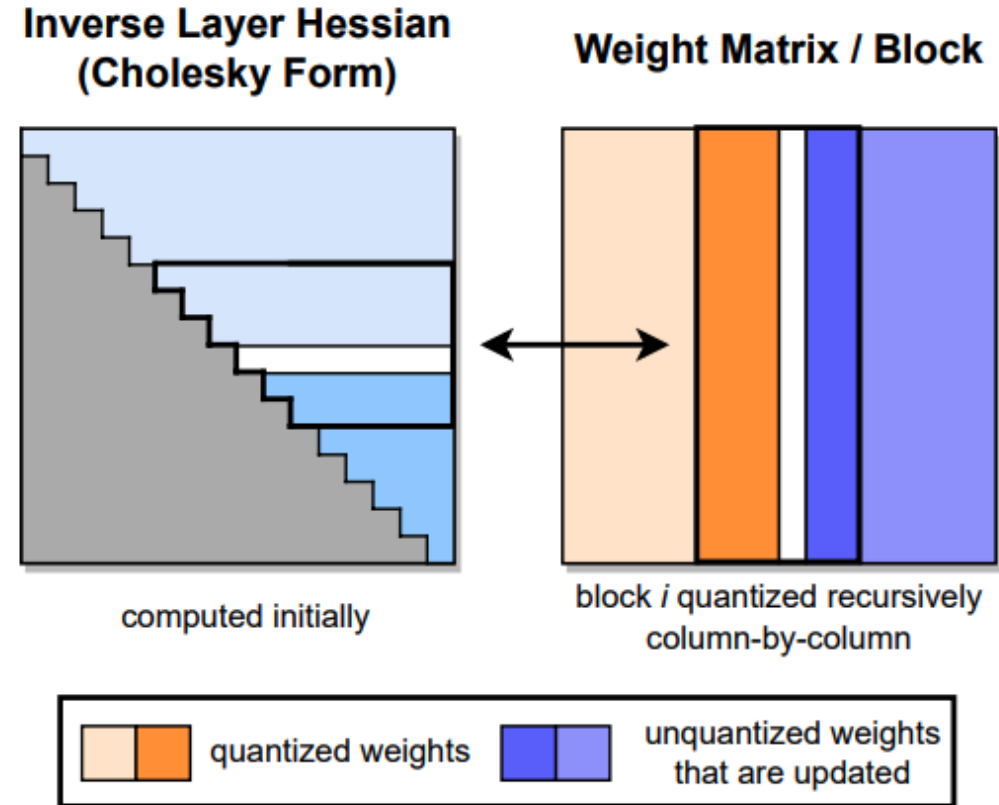
2. Efficient inverse

Reduces overall costs of this process to  $O(d_{row} \cdot d_{col}^3)$  time and  $O(d_{col}^2)$  memory.

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

# Column Update using OBQ is efficient

- Quantize all rows of **weights in same order**.
  - → column-wise update
- $F$  and  $H_F^{-1}$  are always the same for all rows as  $H_F$  depends only on the layer inputs  $X_F$ , which are the same for all rows.
- Perform update on  $H_F^{-1}$  only  $d_{col}$  times, once per column, rather than  $d_{row} \cdot d_{col}$  times, once per weight.
- This reduces the overall runtime from  $O(d_{row} \cdot d_{col}^3)$  to  $O(\max(d_{row} \cdot d_{col}^2, d_{col}^3))$ .

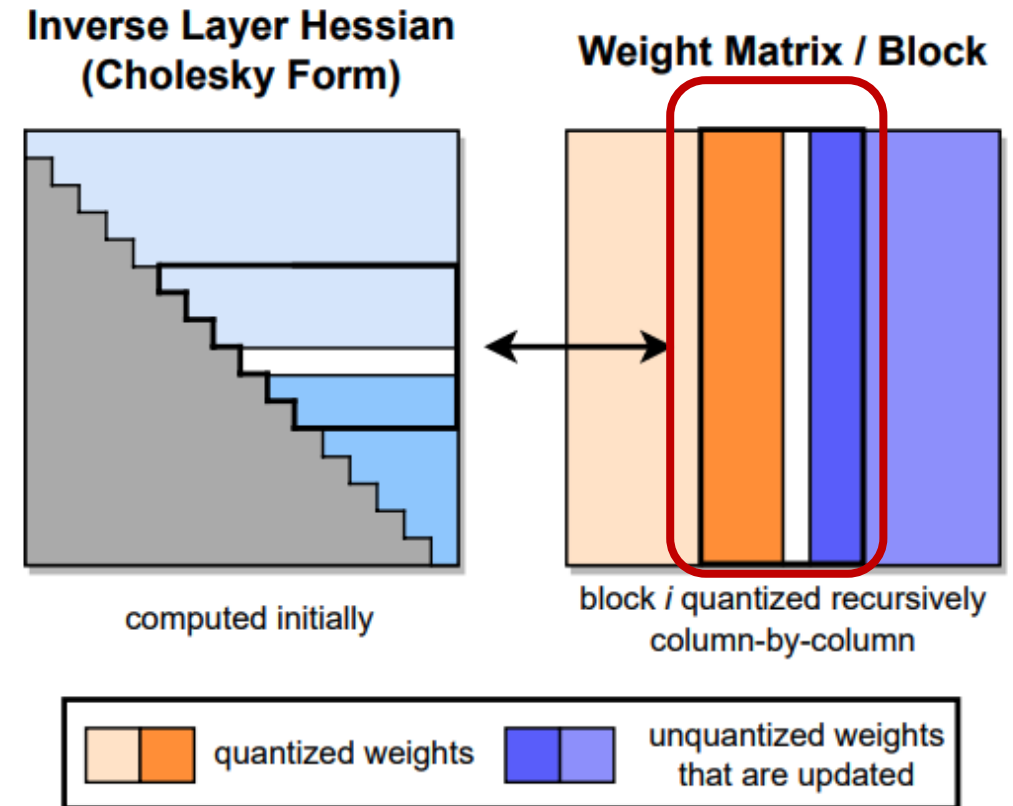


# Insight of Arbitrary Update Order for OBQ

- OBQ quantizes weights in a specific order defined by the corresponding errors.
- Improvement over quantizing the weights in arbitrary order is generally small
  - quantized weights with large individual error is balanced out by those weights towards the end of the process
  - few other unquantized weights can be adjusted for compensation

# Lazy Batch Updates

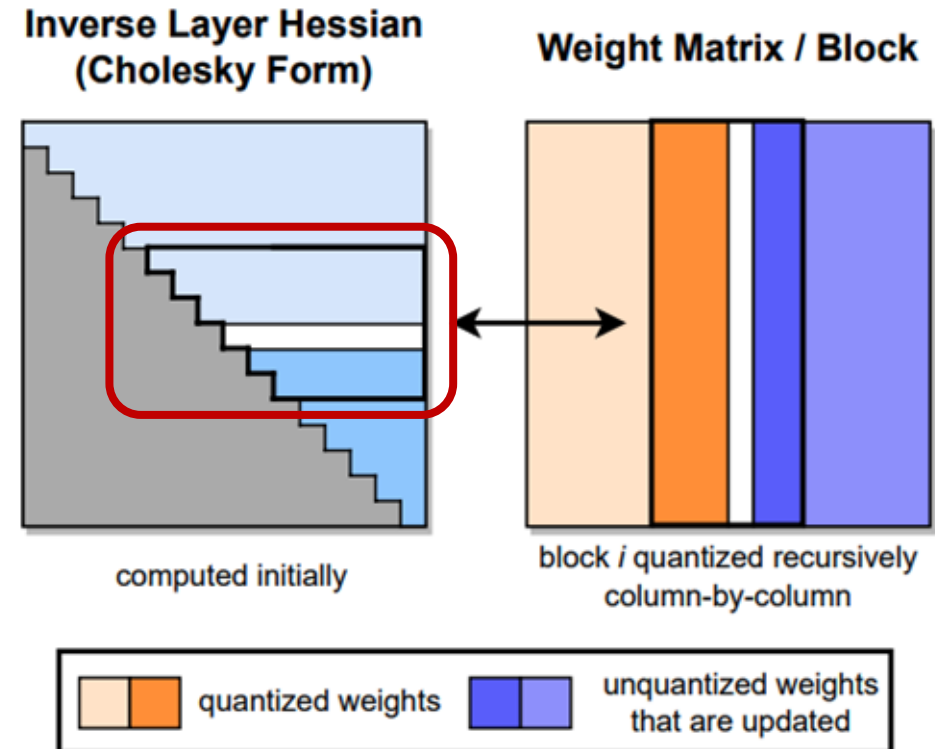
- Naïve column update is not fast in practice
  - **low compute-to-memory-access ratio**
  - cannot highly utilize GPUs compute.
- Observation:
  - Rounding decisions for col  $i$  only affected by updates on this col
  - Updates to later columns are irrelevant at this point in the process.
- Efficient update



$$W_{:, (i+B):} = W_{:, (i+B):} - E \cdot G_{i: (i+B), (i+B):}$$

# Cholesky Pre-computation

- **Numerical inaccuracies, can become a major problem** at the scale of LLMs,
- $H_F^{-1}$  can become indefinite
- Observation:
  - Only information required from  $H_{F_q}^{-1}$  when quantizing weight  $q$  from unquantized  $F_q$ , are the elements in row  $q$  starting with the diagonal.
- GPTQ leverages **Cholesky kernels** to precompute all information from  $H^{-1}$  without any significant increase in memory consumption.



# Examine Effectiveness of GPTQ

- How is GPT-Q's perf on small models compared with accurate-but-expensive methods?
- How does GPT-Q's quantization time scale with model size?
- How is GPT-Q's perf on large models compared with Round-to-nearest methods?
- How does GPT-Q speed up model inference in practical applications?
- Does GPT-Q even work for extreme 2-bit quantization?

# Quantization Experiment Setup

- Calibration data randomly sampled from C-4 dataset to ensure GPTQ is not task-aware.
- Standard uniform per-row asymmetric quantization on the min-max grid
- Quantize on each transformer block (6 layers), with input  $X$  from last quantized block output.

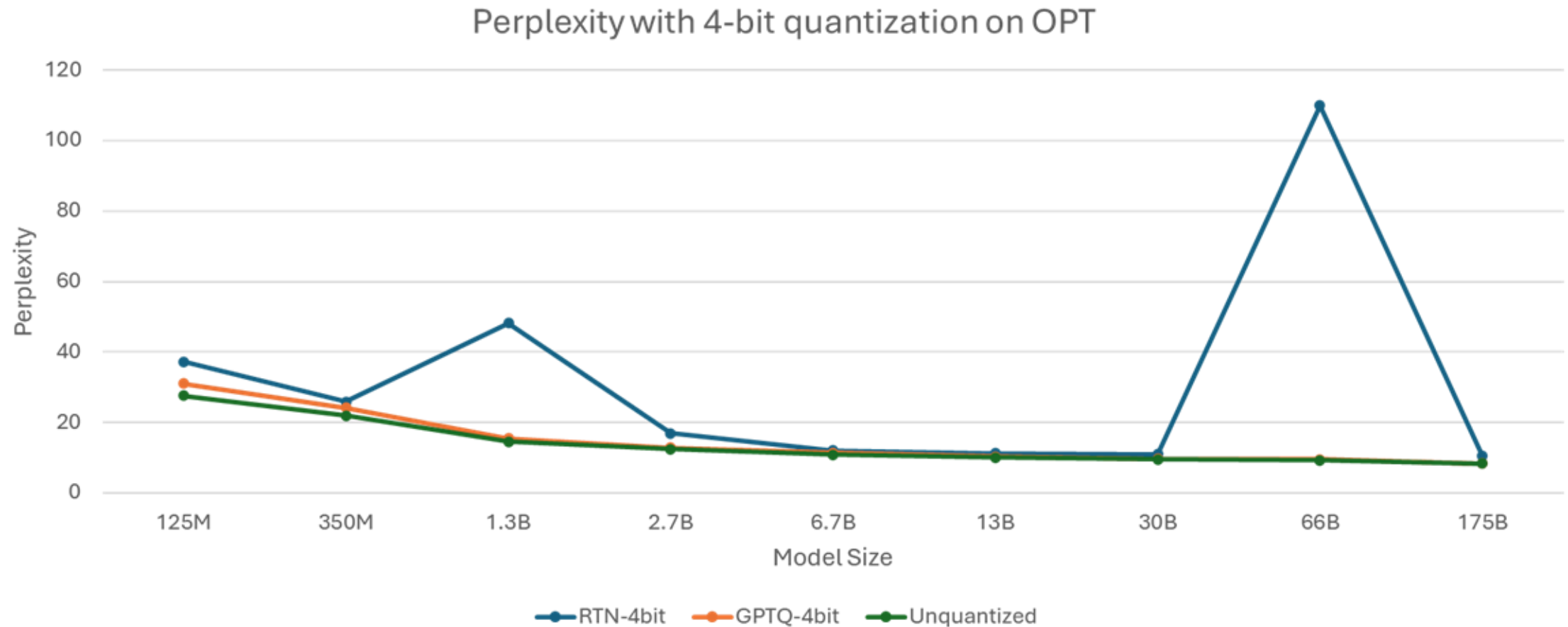
# How does GPT-Q speed up model inference in practical applications?

OPT-175B mode (Measured with pipeline parallelism)

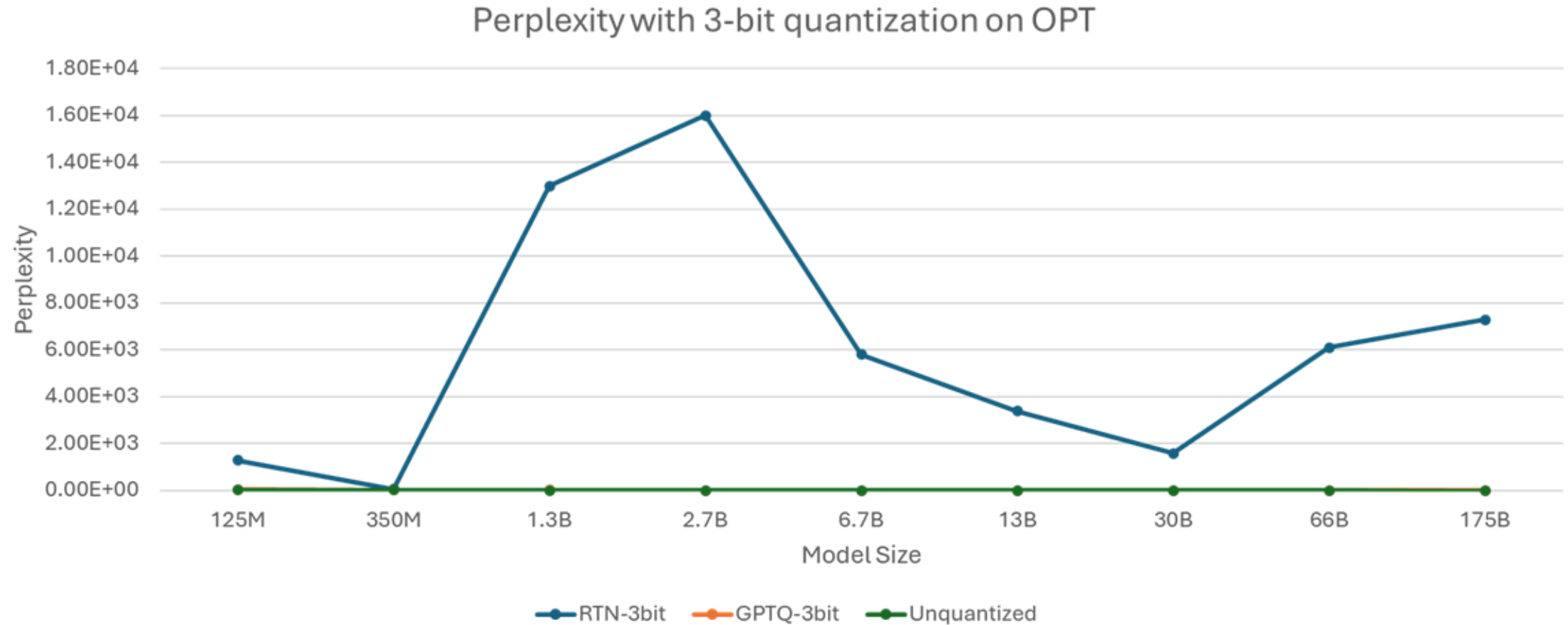
GPU	FP16	3bit	Speedup	GPU reduction
A6000 – 48GB	589ms	130ms	4.53×	8 → 2
A100 – 80GB	230ms	71ms	3.24×	5 → 1

- Single-batch inference is memory-bound because of GEMVs. Although dequantization consumes extra compute, the custom kernel reduces memory access and thus reduces e2e time.

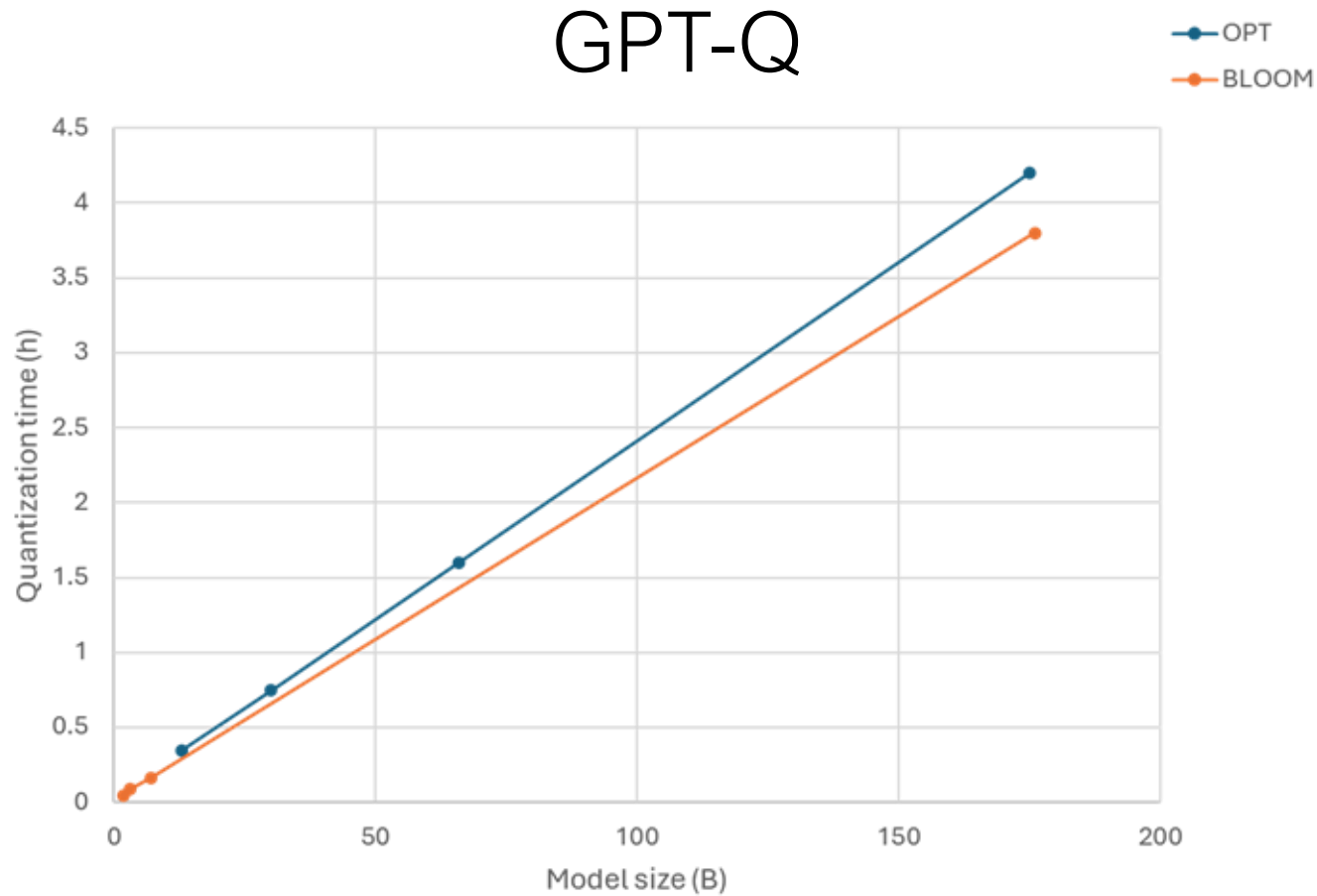
# How is GPT-Q's perf on large models compared with Round-to-nearest methods?



# How is GPT-Q's perf on large models compared with Round-to-nearest methods?



# How does GPT-Q's quantization time scale with model size?

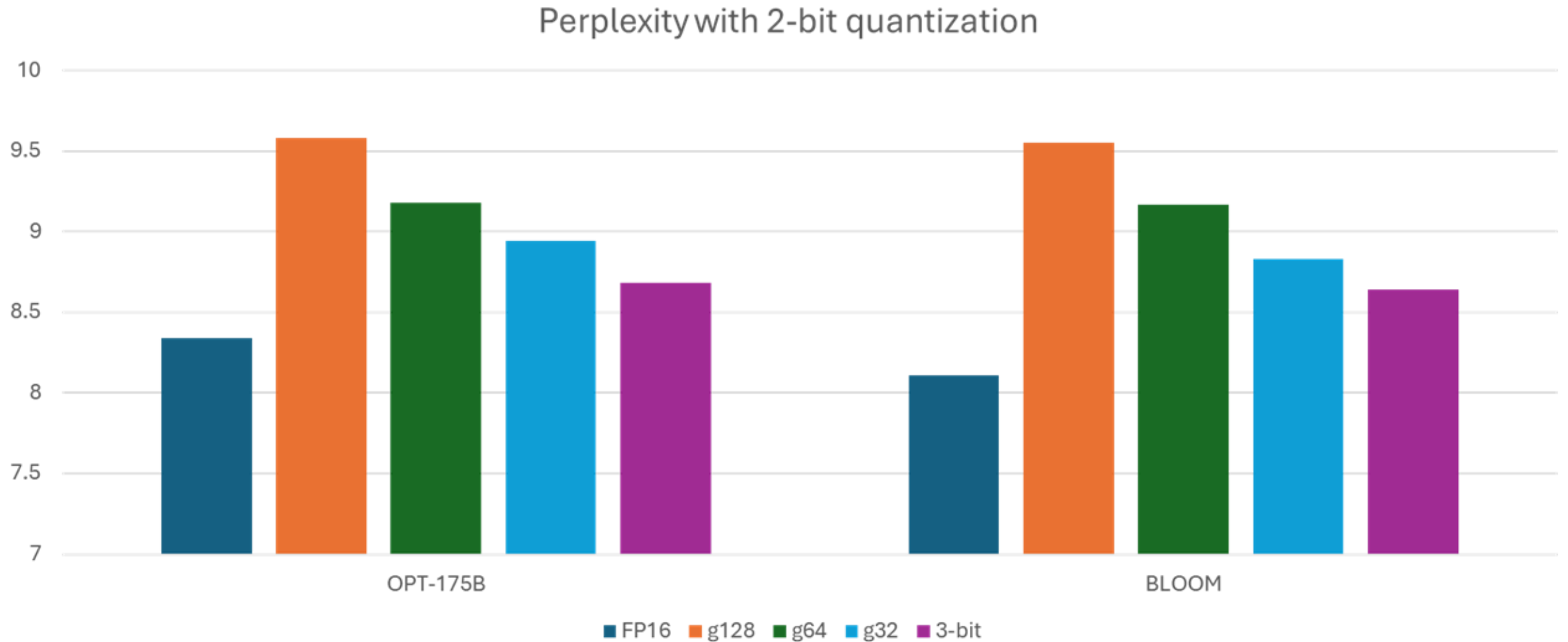


ZeroQuant-LKD

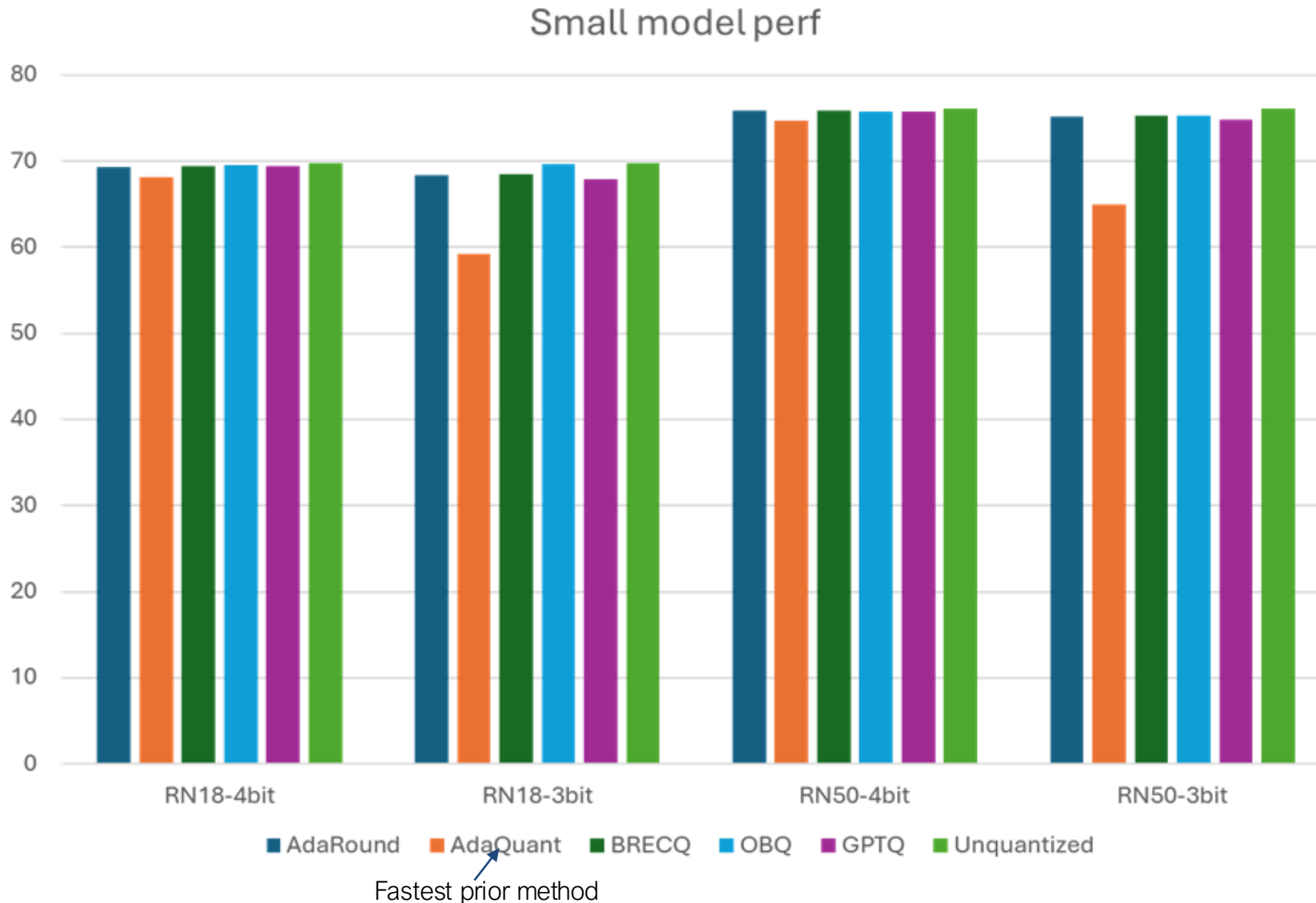
1.3B model - 3h

\* Measured on single A100

# Does GPT-Q even work for extreme 2-bit quantization?



# How is GPT-Q's perf on small models compared with accurate-but-expensive methods?



# GPTQ for LLaMA

- <https://github.com/qwopqwop200/GPTQ-for-LLaMa/>
- GPTQ in
  - <https://github.com/qwopqwop200/GPTQ-for-LLaMa/blob/triton/gptq.py>

# GPTQ: Initialization

```
def __init__(self, layer):
    self.layer = layer
    self.dev = self.layer.weight.device
    W = layer.weight.data.clone()
    if isinstance(self.layer, nn.Conv2d):
        W = W.flatten(1)
    if isinstance(self.layer, transformers.Conv1D):
        W = W.t()
    self.rows = W.shape[0]
    self.columns = W.shape[1]
    self.H = torch.zeros((self.columns, self.columns), device=self.dev)
    self.nsamples = 0
```

- Reshape weights from the input layer
- Initialize Hessian matrix

# GPTQ: Hessian Matrix Update

```
def add_batch(self, inp, out):
    if len(inp.shape) == 2:
        inp = inp.unsqueeze(0)
    tmp = inp.shape[0]
    if isinstance(self.layer, nn.Linear) or isinstance(self.layer, transformers.Conv1D):
        if len(inp.shape) == 3:
            inp = inp.reshape((-1, inp.shape[-1]))
            inp = inp.t()
    if isinstance(self.layer, nn.Conv2d):
        unfold = nn.Unfold(
            self.layer.kernel_size,
            dilation=self.layer.dilation,
            padding=self.layer.padding,
            stride=self.layer.stride
        )
        inp = unfold(inp)
        inp = inp.permute([1, 0, 2])
        inp = inp.flatten(1)
    self.H *= self.nsamples / (self.nsamples + tmp)
    self.nsamples += tmp
    # inp = inp.float()
    inp = math.sqrt(2 / self.nsamples) * inp.float()
    # self.H += 2 / self.nsamples * inp.matmul(inp.t())
    self.H += inp.matmul(inp.t())
```

- Update Hessian matrix with information from a new batch of the input and output pairs

# GPTQ: Lazy Batch-Update

```
for i1 in range(0, self.columns, blocksize):  
    i2 = min(i1 + blocksize, self.columns)  
    count = i2 - i1
```

```
W1 = W[:, i1:i2].clone()  
Q1 = torch.zeros_like(W1)  
Err1 = torch.zeros_like(W1)  
Losses1 = torch.zeros_like(W1)  
Hinv1 = Hinv[i1:i2, i1:i2]
```

```
for i in range(count):  
    w = W1[:, i]  
    d = Hinv1[i, i]
```

```
if groupsize != -1:  
    if not static_groups:  
        if (i1 + i) % groupsize == 0:  
            self.quantizer.find_params(W[:, (i1 + i):(i1 + i + groupsize)], weight=True)  
    else:  
        idx = i1 + i  
        if actorder:  
            idx = perm[idx]  
        self.quantizer = groups[idx // groupsize]
```

- Processes weight matrix W in blocks.
- Updates quantization parameters conditionally based on group size and static grouping settings.

# GPTQ: Lazy Batch-Update

```
q = quantize(  
    w.unsqueeze(1), self.quantizer.scale, self.quantizer.zero, self.quantizer.maxq  
) .flatten()
```

```
Q1[:, i] = q  
Losses1[:, i] = (w - q) ** 2 / d ** 2
```

```
err1 = (w - q) / d
```

```
W1[:, i:] -= err1.unsqueeze(1).matmul(Hinv1[i, i:].unsqueeze(0))
```

```
Err1[:, i] = err1
```

```
Q[:, i1:i2] = Q1  
Losses[:, i1:i2] = Losses1 / 2
```

```
W[:, i2:] -= Err1.matmul(Hinv[i1:i2, i2:])
```

- Applies quantization function `quantize` to weights and computes the loss due to quantization.
- Adjusts remaining block weights based on quantization error to minimize the overall error.

# GPTQ: Cholesky Reformulation

```
damp = percdamp * torch.mean(torch.diag(H))
diag = torch.arange(self.columns, device=self.dev)
H[diag, diag] += damp
H = torch.linalg.cholesky(H)
H = torch.cholesky_inverse(H)
H = torch.linalg.cholesky(H, upper=True)
Hinv = H
```

- Applies damping to the Hessian matrix diagonals
- Performs Cholesky decomposition and inversion
- Transforms the Hessian into its inverse.

# AutoGPTQ Tool

```
import random
from auto_gptq import AutoGPTQForCausalLM, BaseQuantizeConfig
from datasets import load_dataset
import torch
from transformers import AutoTokenizer

# Define base model and output directory
model_id = "gpt2" #modify to your model
out_dir = model_id + "-GPTQ"

# Load quantize config, model and tokenizer
quantize_config = BaseQuantizeConfig(bits=4, group_size=128, damp_percent=0.01, desc_act=False)
model = AutoGPTQForCausalLM.from_pretrained(model_id, quantize_config)
tokenizer = AutoTokenizer.from_pretrained(model_id)

# Load data and tokenize examples
n_samples = 1024
data = load_dataset("allenai/c4", data_files="en/c4-train.00001-of-01024.json.gz", split=f"train[:{n_samples*5}]")
tokenized_data = tokenizer("\n\n".join(data['text']), return_tensors='pt')

# Format tokenized examples
examples_ids = []
for _ in range(n_samples):
    i = random.randint(0, tokenized_data.input_ids.shape[1] - tokenizer.model_max_length - 1)
    j = i + tokenizer.model_max_length
    input_ids = tokenized_data.input_ids[:, i:j]
    attention_mask = torch.ones_like(input_ids)
    examples_ids.append({'input_ids': input_ids, 'attention_mask': attention_mask})

# Quantize with GPTQ
model.quantize(examples_ids, batch_size=1, use_triton=True)

# Save model and tokenizer
model.save_quantized(out_dir, use_safetensors=True)
tokenizer.save_pretrained(out_dir)
```

# Summary and Limitations

- GPTQ
  - approximate second-order of weights
  - layer-wise quantization + compensation for errors + precompute
  - accurately compress some of the largest publicly-available models down to 3 and 4 bits, and bring end-to-end speedups
- Limitations
  - Theoretical computation is the same
  - Focus on weight quantization, and does not consider activation quantization

# Quiz

- on canvas