



# TensorRT-LLM量化原理、实现与优化

by Yuxin



# Agenda

- 量化原理
  - 数值量化
  - 矩阵量化
  - 反量化
- 常用的量化方案
- 动手实践
  - 计算量化系数
  - 计算量化后权重
- 性能优化
  - 转置
  - 交织列
  - 重排行
  - 加偏置
  - 附录（代码导读）
- 小试牛刀



# 数值量化

INT8

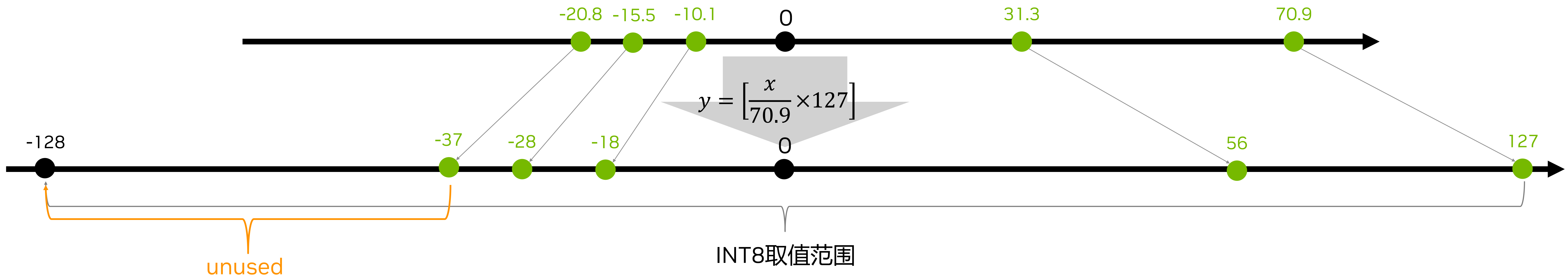
量化 (quantization) , 即“量子化”, 是将无限的、连续的值域映射到有限的、离散的值域的过程。

计算机中, N比特可以表达 $2^N$ 个数。给定一个集合S, 如何将S分为至多 $2^N$ 个类 (每个类代表一个数), 使得其尽可能精确地表达原有的数值分布? 这便是“N比特量化”研究的问题。根据这 $2^N$ 个数在数轴上的分布特征, 量化方法可以正交地分为线性 / 非线性, 以及对称 / 非对称。如果量化后的值域是等间距的, 则称为线性的, 否则为非线性的。如果量化后的值域关于原点对称, 则称为对称的, 否则为非对称的。

8位整数INT8是典型的线性量化, 其量化过程可以用如下公式表达:

$$y = \left\lfloor \frac{x}{\max(\text{abs}(S))} \times 127 \right\rfloor, \quad x \in S$$

假设给定集合 $S=\{-20.8, -15.5, -10.1, 31.3, 70.9\}$ , 量化过程如下:



$127 \div 70.9 \approx 1.791$ 即为量化的缩放系数 (scale)。



S中正数的最大值 (70.9) 比负数的绝对值的最大值 (20.8) 大很多, 导致INT8负半轴上一大段值域[-128, -37)被闲置了。

# 数值量化

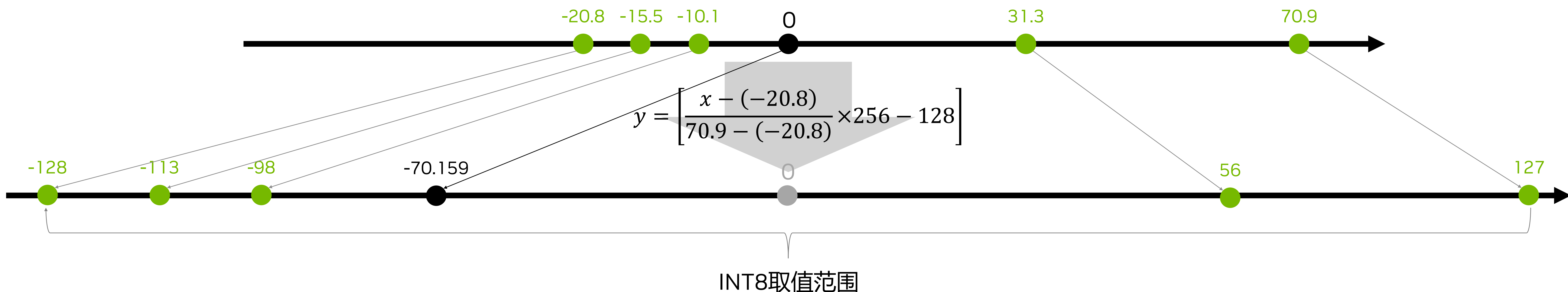
## INT8+BIAS

为量化过程增加偏置（bias），可以避免数据集S偏态分布而导致的量化空间浪费。改造后的INT8量化过程可以用如下公式表达：

$$y = \left\lfloor \frac{x - \min(S)}{\max(S) - \min(S)} \times 255 - 128 \right\rfloor, \quad x \in S$$

从公式可以看出，y仍然线性于x。另外，当 $x = \min(S)$ 时， $y = -128$ ；当 $x = \max(S)$ 时， $y = 127$ 。

仍以集合 $S = \{-20.8, -15.5, -10.1, 31.3, 70.9\}$ 为例，量化过程如下：



简化上述算式，得  $y = \left\lfloor \frac{255}{91.7}x + \left(\frac{20.8 \times 255}{91.7} - 128\right) \right\rfloor \approx [2.781x - 70.159]$ ，2.781即为缩放系数（scale），-70.159即为偏置（bias）。由于源数轴的零点被平移到了新数轴的bias上，因此有些文献或代码中将bias称为zero。

通过对源数轴的缩放与平移，S的边界与INT8值域边界对齐，从而充分利用了INT8的表达空间，提高了数值之间的区分度。与上文中的缩放系数1.791相比，现在的缩放系数明显增大，如同放大镜拥有了更高的倍率，能够保留更多的信息。

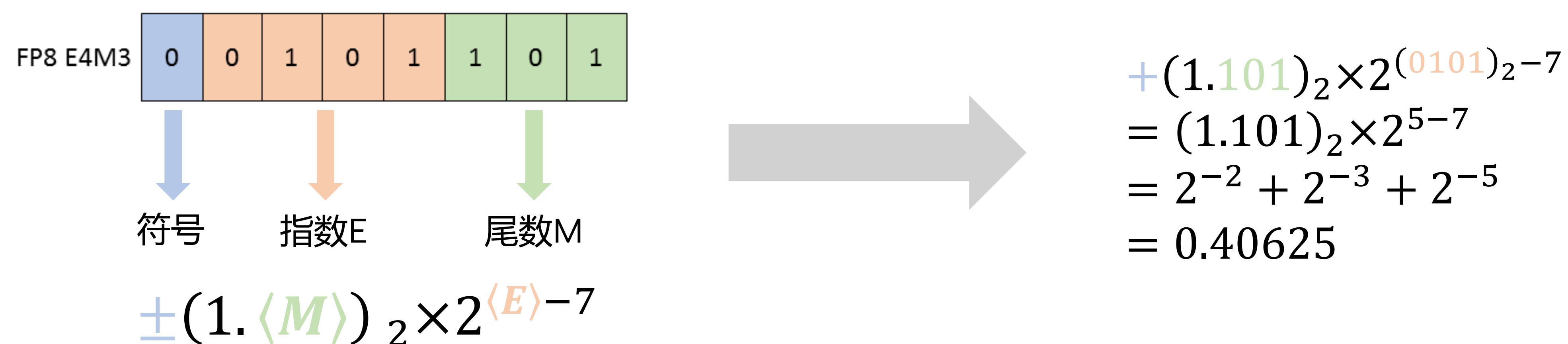
缺点在于增加了额外的计算量。

# 数值量化

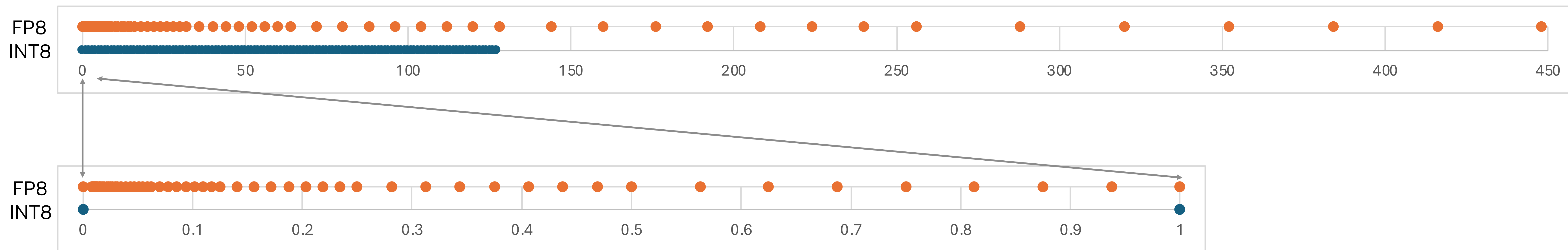
## FP8

线性量化对S的值域边界比较敏感。如果S中的数值尺度相差悬殊（比如差若干个数量级），那么小尺度的数值就会被量化到零点附近，甚至归零。对于这种情况，可以考虑FP8为代表的非线性量化方法。

FP8是8位浮点数，具体分为E4M3和E5M2两种格式，区别是指数位与尾数位的划分。以常用的E4M3格式为例，其8个比特的含义如下图所示：



将FP8 E4M3与INT8在正半轴上的数值分布画在数轴上，如下图所示：



向无穷大看，FP8的取值比INT8大得多；向零点看，FP8的取值比INT8精细得多。既“海纳百川”，又“明察秋毫”，这使得FP8在大模型上的表现通常比INT8更优。



# 数值量化

## FP8

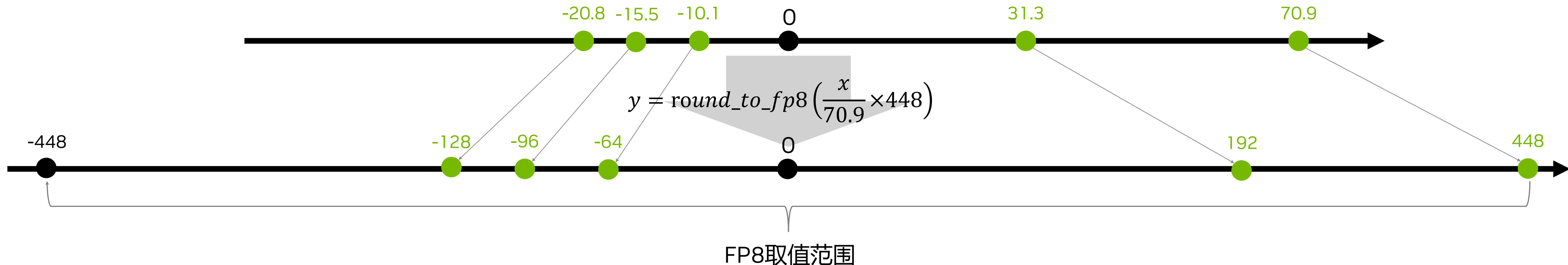
FP8量化过程与INT8类似，即先确定缩放比例，然后近似到最接近的取值点上。用公式表达如下：

$$y = \text{round\_to\_fp8} \left( \frac{x}{\max(\text{abs}(S))} \times 448 \right), \quad x \in S$$

其中448即为FP8 E4M3能表达的最大值。  $\text{round\_to\_fp8}(x)$ 的取值如下表所示（仅展示正半轴，负半轴对称）。当然，计算机中并不需要查表，有高效的实现算法。

0.00000	0.00781	0.00879	0.00977	0.01074	0.01172	0.01270	0.01367	0.01465	0.01563	0.01758	0.01953	0.02148	0.02344	0.02539	0.02734
0.02930	0.03125	0.03516	0.03906	0.04297	0.04688	0.05078	0.05469	0.05859	0.06250	0.07031	0.07813	0.08594	0.09375	0.10156	0.10938
0.11719	0.12500	0.14063	0.15625	0.17188	0.18750	0.20313	0.21875	0.23438	0.25000	0.28125	0.31250	0.34375	0.37500	0.40625	0.43750
0.46875	0.50000	0.56250	0.62500	0.68750	0.75000	0.81250	0.87500	0.93750	1.000	1.125	1.250	1.375	1.500	1.625	1.750
1.875	2.000	2.250	2.500	2.750	3.000	3.250	3.500	3.750	4.0	4.5	5.0	5.5	6.0	6.5	7.0
7.5	8.0	9.0	10.0	11.0	12.0	13.0	14.0	15.0	16	18	20	22	24	26	28
30	32	36	40	44	48	52	56	60	64	72	80	88	96	104	112
120	128	144	160	176	192	208	224	240	256	288	320	352	384	416	448

仍以集合  $S = \{-20.8, -15.5, -10.1, 31.3, 70.9\}$  为例，量化过程如下：



与INT8类似，FP8量化也可以加偏置以应对偏态分布，不再赘述。

# 矩阵量化

per tensor

上文讨论的都是实数集合S的量化方案。具体到矩阵的量化，又有多种划分方案，大致可以分为<sup>[1]</sup>：

- per tensor (亦称layerwise)
- per channel (亦称channelwise)
- groupwise

所谓per tensor，是将整个矩阵看作集合S，矩阵中的所有元素共享一个缩放系数。

假设我们有一个4行5列矩阵如下表所示。per tensor方法就是将这20个数看作一个集合S，寻找 $\max(abs(S))$ 以确定缩放系数，然后将所有元素量化到目标类型。以INT8量化为例，量化过程如图所示：

0.72	-1.37	9.31	1.57	-6.83
-5.53	1.59	3.63	8.91	-6.75
3.39	-8.95	0.89	2.47	5.18
2.46	4.27	-3.54	0.33	-0.91

$$y = \left\lfloor \frac{127}{9.31} x \right\rfloor$$

10	-19	127	21	-93
-75	22	50	122	-92
46	-122	12	34	71
34	58	-48	5	-12

Per tensor方案优点在于简单，全局只有一个缩放系数。

但是缺点也很明显。如前文所述，如果矩阵中数值尺度相差悬殊，那么大尺度的数值会导致一个很小的缩放系数，从而导致小尺度的数值趋于零甚至归零。而大语言模型常常具有这种数值分布特征，所以通常不使用per tensor，尤其是使用线形量化时。



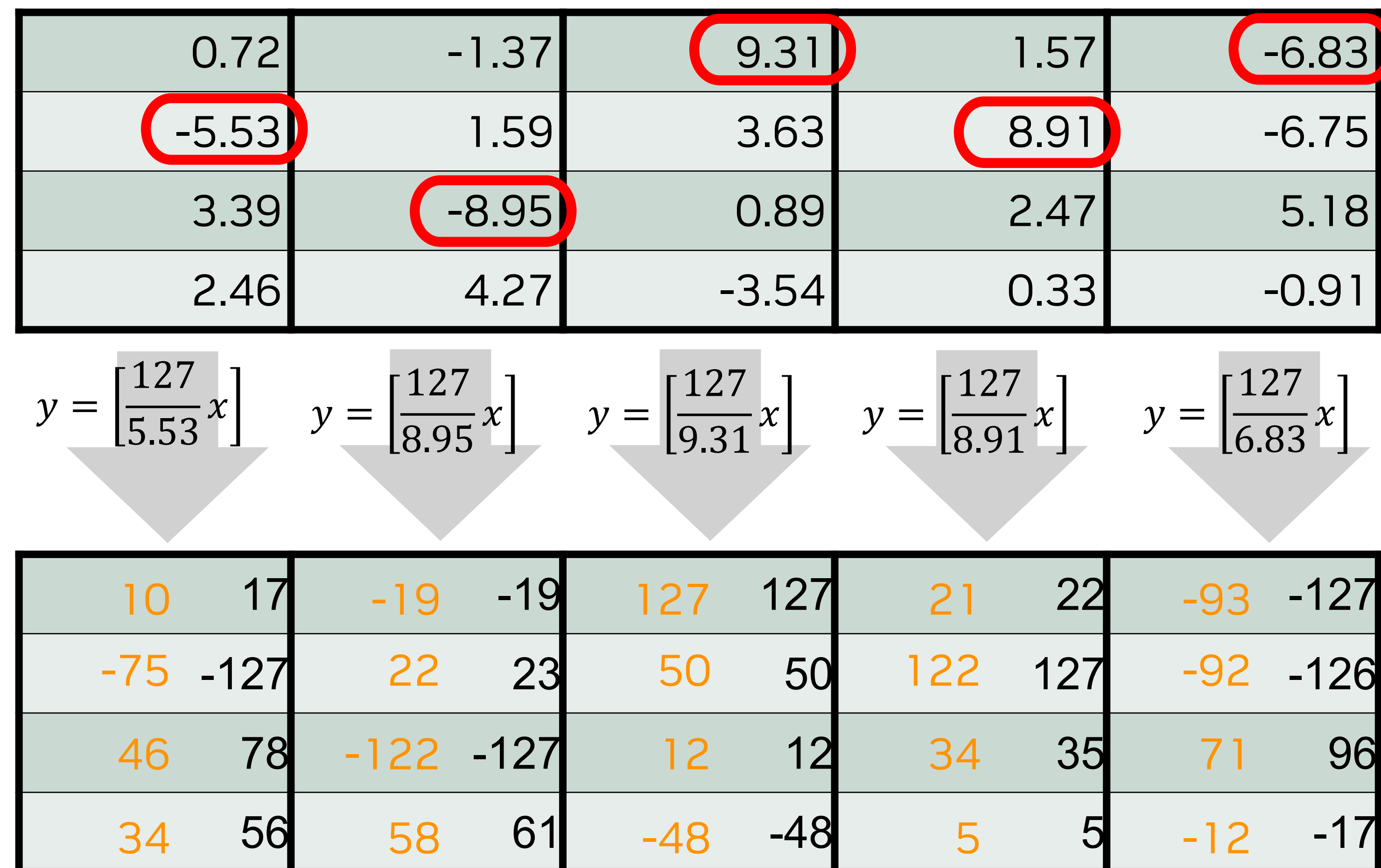
# 矩阵量化

per channel

如果集合S内的数值波动越小，那么缩放系数就能越大，量化精度就越高。另一方面，如果S内的个数越少，出现大幅波动的概率也就越小。于是自然想到，将矩阵分片量化能提高量化精度，且分片越小，精度就越高。

首先想到的是将矩阵每一列作为一个分片进行量化，每一列内的所有元素共享一个缩放系数，而列之间相互独立，这便是per channel量化。

仍以上文中的矩阵为例，我们将每一列的4个元素看作一个集合 $S_i$ ，寻找 $\max(\text{abs}(S_i))$ 以确定各自的缩放系数。以INT8量化为例，量化过程如图所示：



per channel量化结果的绝对值必大于  
(或至少等于) per tensor量化结果，  
能增加数值之间的区分度，保留更多的  
源分布信息。

当然，按行分片也是可行的。在矩阵乘法 $A \times B$ 中，我们通常将A按行量化，将B按列量化，即原则上是沿着内维度（inner dimension）量化。换言之，假设A为 $M \times K$ 矩阵，B为 $K \times N$ 矩阵，则沿K维度量化，A有M个行缩放系数，B有N个列缩放系数。

在LLM中，A通常为activation，其每一行是某一个token的embedding，因此我们把A的按行量化也称为per token量化。



# 矩阵量化


## groupwise

能不能进一步分片以提高量化精度呢？那便是groupwise量化。

Groupwise是将G个元素看作集合S进行数值量化，G称为group size。

仍以上文中的矩阵为例，我们取group size为2，按列优先的顺序分组，将每一组的2个元素看作一个集合 $S_i$ ，计算 $\max(\text{abs}(S_i))$ 以确定各自的缩放系数。以INT8量化为例，量化过程如图所示：

0.72	-1.37	9.31	1.57	-6.83
-5.53	1.59	3.63	8.91	-6.75
3.39	-8.95	0.89	2.47	5.18
2.46	4.27	-3.54	0.33	-0.91



17	17	-19	-109	127	127	22	22	-127	-127
-127	-127	23	127	50	50	127	127	-126	-126
78	127	-127	-127	12	127	35	127	96	127
56	92	61	61	-48	-50	5	17	-17	-22

不出意外地，相较于per channel，groupwise量化后的数值绝对值又有所放大，量化精度更高了。

但代价是所需的缩放系数从per channel时的 $N$ 几何膨胀至：

$$\frac{M \times N}{group\_size}$$

其中 $M$ 为矩阵行数， $N$ 为矩阵列数。

随着分片粒度变小，精度越来越高，而量化开销越大，这需要折衷。事实上，对于一个 $M \times N$ 的矩阵，per channel只是groupwise在 $group\_size = M$ 时的特例，per tensor则是 $group\_size = M \times N$ 时的特例。

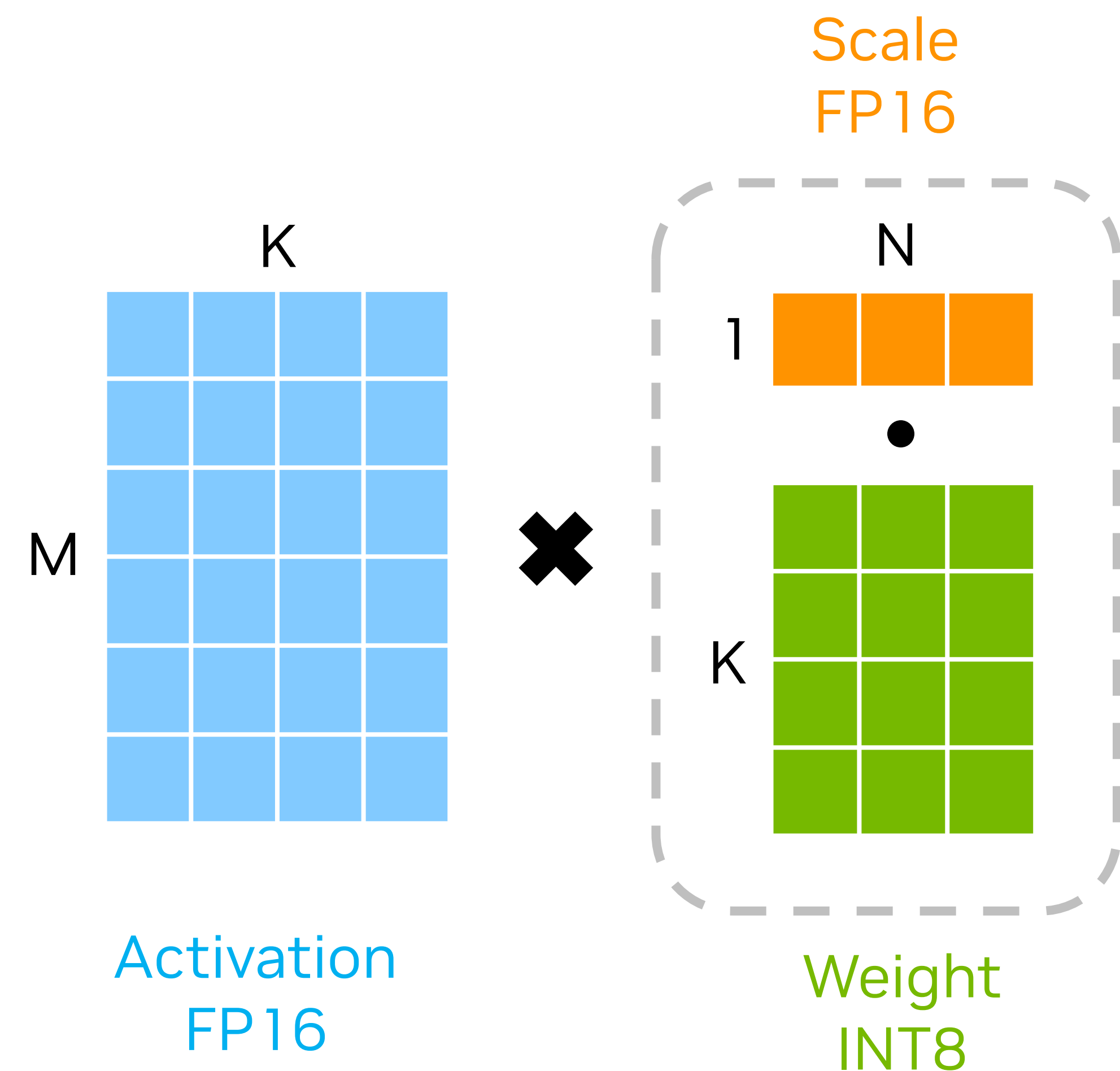
# 反量化

在构建模型时，各层的权重已经完成了量化。在推理过程中，根据activation是否使用量化数值类型，又可分为两大类：

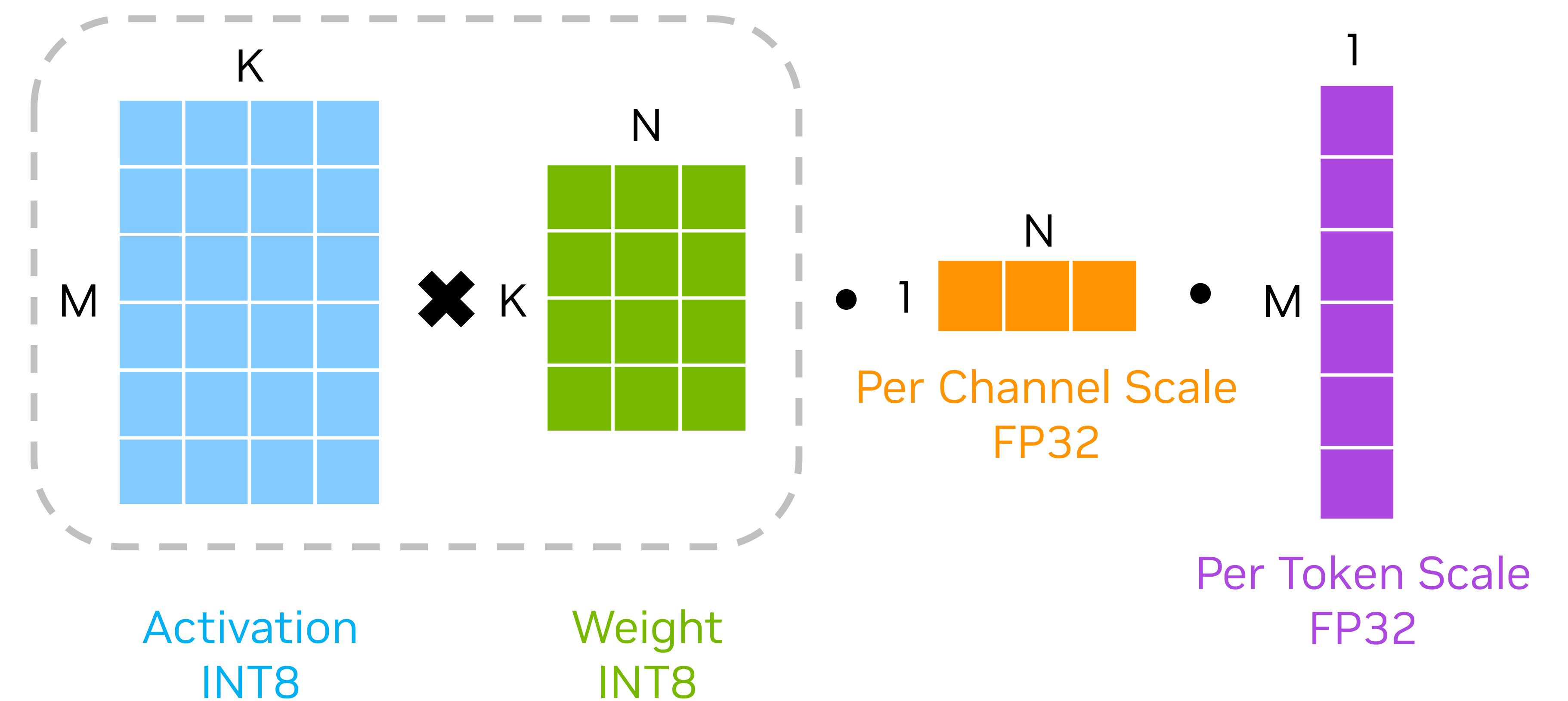
- weight-only
- weight & activation

它们决定了矩阵乘法过程中，反量化操作的执行时机。

以典型的INT8 weight-only量化为例，其weight是per channel量化的INT8，而activation使用FP16类型。在推理时，需要先按列乘以scale将weight反量化为FP16，再与activation相乘，如下图所示：



SmoothQuant方法是weight & activation量化，其weight和activation都使用INT8类型，所以有两组缩放系数，一组是用于weight的per channel scale，一组是用于activation的per token scale。先执行INT8的矩阵乘法，再分别按列、按行乘以两组缩放系数，如下图所示。





# 常用的量化方案

理论上，weight数值类型、 activation数值类型和矩阵量化粒度是互相正交的，可以任意组合。但是工程实现中需要综合考虑精度影响、性能收益、易用性，以及开发与维护难度等因素。TensorRT-LLM支持常用的几种量化方案：

名称	Weight类型	Activation类型	量化粒度	实现
FP8	FP8	FP8	Per Tensor	ModelOPT
INT8 Weight-Only	INT8	FP16	Per Channel	Built-in
INT8 SmoothQuant	INT8	INT8	Per Channel + Per Token	Built-in / ModelOPT
INT4 Weight-Only	INT4	FP16	Per Channel	Built-in
INT4 AWQ	INT4	FP16	Groupwise	ModelOPT
INT4 GPTQ	INT4	FP16	Groupwise	3 <sup>rd</sup> party software

目前，有些量化方案是由TensorRT-LLM自身实现的，有些是依赖NVIDIA量化工具ModelOPT<sup>[1]</sup>，有些依赖于第三方开源软件（比如AutoGPTQ<sup>[2]</sup>）。

[1] A Library to Quantize and Compress Deep Learning Models for Optimized Inference on GPUs. (<https://github.com/NVIDIA/TensorRT-Model-Optimizer>)  
[2] An easy-to-use LLMs quantization package with user-friendly apis, based on GPTQ algorithm. (<https://github.com/AutoGPTQ/AutoGPTQ>)

# 动手实践

## 查看Llama7B模型的权重

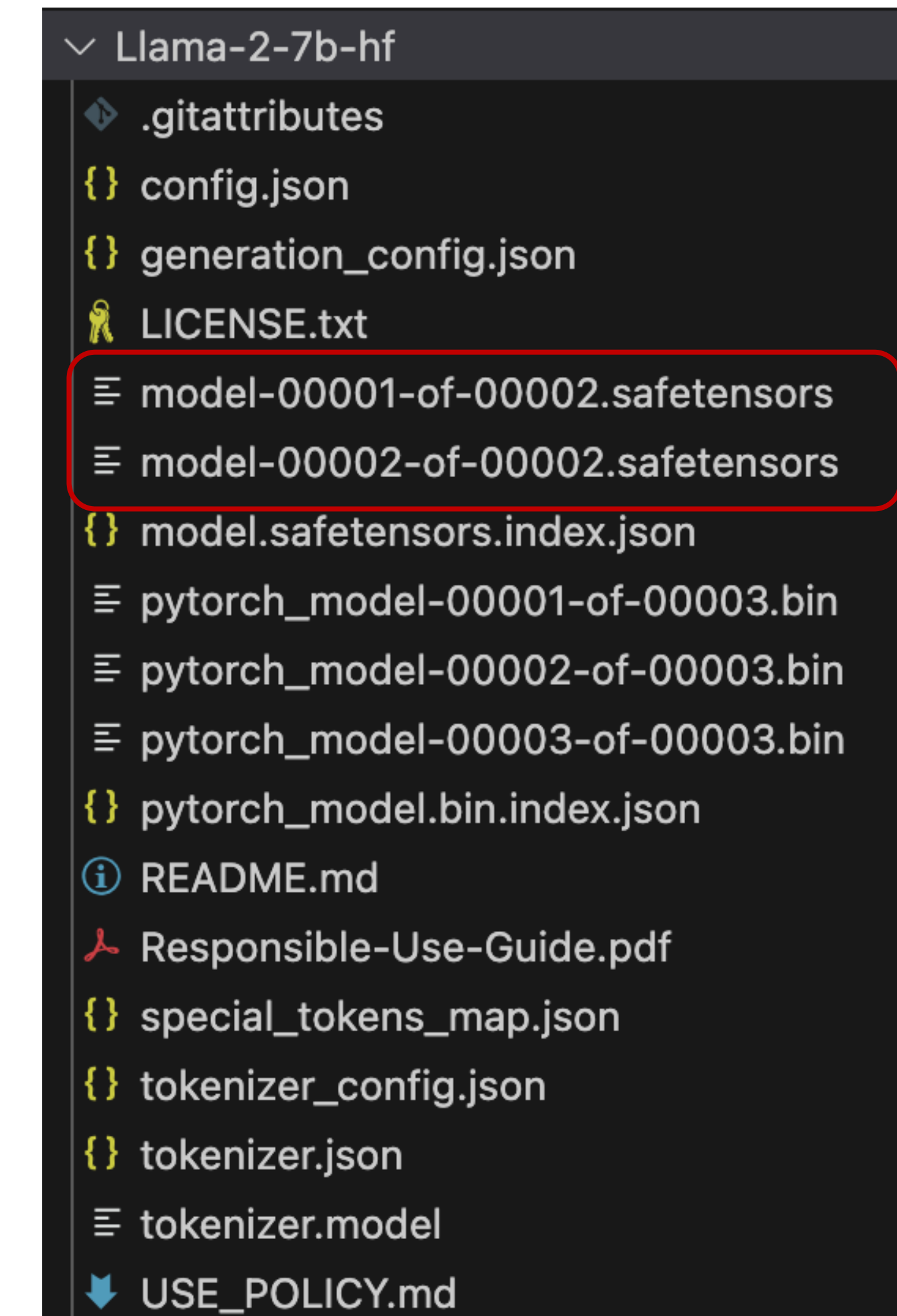
从HuggingFace上下载一个原始Llama 7B模型，数据类型是FP16：

```
git clone https://huggingface.co/NousResearch/Llama-2-7b-hf
```

它的目录结构如右图所示。可以看到有两个safetensors文件，这里面存储了模型中所有的权重。

写一段python代码，存为dump.py，用来查看指定safetensors文件里都有哪些tensor：

```
dump.py ×
dump.py > ...
1 import sys
2 import torch
3 from safetensors import safe_open
4
5 fname = sys.argv[1]
6
7 with safe_open(fname, framework = "pt") as f:
8     for k in f.keys():
9         tensor: torch.Tensor = f.get_tensor(k)
10        print(k, tensor.dtype, tensor.shape)
```



执行`python dump.py Llama-2-7b-hf/model-00001-of-00002.safetensors`，可以看到如下输出：

```
model.layers.0.input_layernorm.weight torch.float16 torch.Size([4096])
model.layers.0.mlp.down_proj.weight torch.float16 torch.Size([4096, 11008])
model.layers.0.mlp.gate_proj.weight torch.float16 torch.Size([11008, 4096])
model.layers.0.mlp.up_proj.weight torch.float16 torch.Size([11008, 4096])
model.layers.0.post_attention_layernorm.weight torch.float16 torch.Size([4096])
model.layers.0.self_attn.k_proj.weight torch.float16 torch.Size([4096, 4096])
model.layers.0.self_attn.o_proj.weight torch.float16 torch.Size([4096, 4096])
model.layers.0.self_attn.q_proj.weight torch.float16 torch.Size([4096, 4096])
model.layers.0.self_attn.rotary_emb.inv_freq torch.float32 torch.Size([16])
model.layers.0.self_attn.v_proj.weight torch.float16 torch.Size([4096, 4096])
```

这三个4096x4096的矩阵便是  
每一层self-attention中的Q、K  
和V矩阵。



# 动手实践

## 查看Llama7B模型的权重

再进一步看一下这三个矩阵的数值：

```
fname = "Llama-2-7b-hf/model-00001-of-00002.safetensors"
with safe_open(fname, framework = "pt") as f:
    Q: torch.Tensor = f.get_tensor("model.layers.0.self_attn.q_proj.weight")
    print(Q)
    K: torch.Tensor = f.get_tensor("model.layers.0.self_attn.k_proj.weight")
    print(K)
    V: torch.Tensor = f.get_tensor("model.layers.0.self_attn.v_proj.weight")
    print(V)
```

输出：

```
tensor([[ -0.0062, -0.0148, -0.0022, ...,  0.0045,  0.0017, -0.0036],
        [  0.0142, -0.0043,  0.0028, ..., -0.0093, -0.0114,  0.0076],
        [-0.0146,  0.0126,  0.0005, ...,  0.0063,  0.0188, -0.0031],
        ...,
        [  0.0013,  0.0109, -0.0003, ...,  0.0098, -0.0298,  0.0097],
        [  0.0256,  0.0102,  0.0032, ..., -0.0334, -0.0156, -0.0123],
        [-0.0134, -0.0066,  0.0018, ...,  0.0181,  0.0166, -0.0082]],
        dtype=torch.float16)
tensor([[ -0.0162,  0.0079, -0.0013, ...,  0.0166, -0.0099, -0.0135],
        [  0.0192,  0.0015,  0.0036, ..., -0.0211,  0.0152,  0.0234],
        [-0.0236, -0.0217,  0.0017, ...,  0.0150, -0.0165, -0.0118],
        ...,
        [  0.0128, -0.0007, -0.0008, ...,  0.0002,  0.0031,  0.0081],
        [-0.0056,  0.0173, -0.0032, ..., -0.0032,  0.0115, -0.0110],
        [  0.0037, -0.0021,  0.0013, ...,  0.0070, -0.0115,  0.0095]],
        dtype=torch.float16)
tensor([[ 0.0008, -0.0006,  0.0019, ...,  0.0059, -0.0006,  0.0103],
        [-0.0069, -0.0005, -0.0077, ..., -0.0106,  0.0126,  0.0048],
        [ 0.0018,  0.0096,  0.0010, ...,  0.0048, -0.0139, -0.0142],
        ...,
        [-0.0063, -0.0057,  0.0103, ...,  0.0031,  0.0040, -0.0022],
        [ 0.0031,  0.0048, -0.0010, ...,  0.0054,  0.0156,  0.0007],
        [ 0.0001,  0.0025,  0.0056, ..., -0.0007, -0.0007,  0.0015]],
        dtype=torch.float16)
```

Q

K

V



# 动手实践

## 转为TensorRT-LLM的FP16 checkpoint

执行如下命令，将HuggingFace的模型转为TensorRT-LLM的checkpoint。这里我们保留FP16，不做量化，单纯做格式转换：

```
cd examples/llama
python convert_checkpoint.py --model_dir ~/Llama-2-7b-hf --dtype float16 --output_dir ~/llama2_7b_fp16
```

TensorRT-LLM将所有的权重都放在了rank0.safetensors中。

我们用命令`python dump.py ~/llama2\_7b\_fp16/rank0.safetensors`查看其中存储的权重：

```
transformer.layers.0.attention.dense.weight torch.float16 torch.Size([4096, 4096])
transformer.layers.0.attention.qkv.weight torch.float16 torch.Size([12288, 4096])
transformer.layers.0.input_layernorm.weight torch.float16 torch.Size([4096])
transformer.layers.0.mlp.fc.weight torch.float16 torch.Size([11008, 4096])
transformer.layers.0.mlp.gate.weight torch.float16 torch.Size([11008, 4096])
transformer.layers.0.mlp.proj.weight torch.float16 torch.Size([4096, 11008])
transformer.layers.0.post_layernorm.weight torch.float16 torch.Size([4096])
```

```
▼ llama2_7b_fp16
  {} config.json
  ≡ rank0.safetensors
```

TensorRT-LLM好像将Q、K、V三个矩阵按行拼接在了一起，因为这个tensor的名字暗示了这一点，并且 $4096 \times 3 = 12288$ 。为了验证猜想，我们分三段打印一下这个12288x4096的矩阵：

```
import torch
from safetensors import safe_open

fname = "llama2_7b_fp16/rank0.safetensors"
with safe_open(fname, framework = "pt") as f:
    QKV: torch.Tensor = f.get_tensor("transformer.layers.0.attention.qkv.weight")
    print(QKV[0: 4096])
    print(QKV[4096: 8192])
    print(QKV[8192: 12288])
```

```
tensor([[ -0.0062, -0.0148, -0.0022, ...,  0.0045,  0.0017, -0.0036],
        [  0.0142, -0.0043,  0.0028, ..., -0.0093, -0.0114,  0.0076],
        [ -0.0146,  0.0126,  0.0005, ...,  0.0063,  0.0188, -0.0031],
        ...,
        [  0.0013,  0.0109, -0.0003, ...,  0.0098, -0.0298,  0.0097],
        [  0.0256,  0.0102,  0.0032, ..., -0.0334, -0.0156, -0.0123],
        [ -0.0134, -0.0066,  0.0018, ...,  0.0181,  0.0166, -0.0082]],
       dtype=torch.float16)
tensor([[ -0.0162,  0.0079, -0.0013, ...,  0.0166, -0.0099, -0.0135],
        [  0.0192,  0.0015,  0.0036, ..., -0.0211,  0.0152,  0.0234],
        [ -0.0236, -0.0217,  0.0017, ...,  0.0150, -0.0165, -0.0118],
        ...,
        [  0.0128, -0.0007, -0.0008, ...,  0.0002,  0.0031,  0.0081],
        [ -0.0056,  0.0173, -0.0032, ..., -0.0032,  0.0115, -0.0110],
        [  0.0037, -0.0021,  0.0013, ...,  0.0070, -0.0115,  0.0095]],
       dtype=torch.float16)
tensor([[  0.0008, -0.0006,  0.0019, ...,  0.0059, -0.0006,  0.0103],
        [ -0.0069, -0.0005, -0.0077, ..., -0.0106,  0.0126,  0.0048],
        [  0.0018,  0.0096,  0.0010, ...,  0.0048, -0.0139, -0.0142],
        ...,
        [ -0.0063, -0.0057,  0.0103, ...,  0.0031,  0.0040, -0.0022],
        [  0.0031,  0.0048, -0.0010, ...,  0.0054,  0.0156,  0.0007],
        [  0.0001,  0.0025,  0.0056, ..., -0.0007, -0.0007,  0.0015]],
       dtype=torch.float16)
```

与HuggingFace的原始模型一致！这说明TensorRT-LLM的FP16 checkpoint并不改变权重的数值，只是重新命名并做一些小小的优化。



# 动手实践

## 验证INT8 Weight-Only的系数

让我们更进一步！执行如下命令，将HuggingFace的模型量化为INT8 Weight-Only：

```
python convert_checkpoint.py --model_dir ~/Llama-2-7b-hf --dtype float16 --output_dir ~/llama2_7b_int8wo \
--use_weight_only --weight_only_precision int8
```

与刚才一样，`python dump.py ~/llama2\_7b\_int8wo/rank0.safetensors` 查看其中存储的权重：

```
transformer.layers.0.attention.dense.per_channel_scale torch.float16 torch.Size([4096])
transformer.layers.0.attention.dense.weight torch.int8 torch.Size([4096, 4096])
transformer.layers.0.attention.qkv.per_channel_scale torch.float16 torch.Size([12288])
transformer.layers.0.attention.qkv.weight torch.int8 torch.Size([4096, 12288])
transformer.layers.0.input_layernorm.weight torch.float16 torch.Size([4096])
transformer.layers.0.mlp.fc.per_channel_scale torch.float16 torch.Size([11008])
transformer.layers.0.mlp.fc.weight torch.int8 torch.Size([4096, 11008])
transformer.layers.0.mlp.gate.per_channel_scale torch.float16 torch.Size([11008])
transformer.layers.0.mlp.gate.weight torch.int8 torch.Size([4096, 11008])
transformer.layers.0.mlp.proj.per_channel_scale torch.float16 torch.Size([4096])
transformer.layers.0.mlp.proj.weight torch.int8 torch.Size([11008, 4096])
```

QKV依然是拼接在一起（但是发生了转置），所有的weight矩阵都变成了torch.int8类型，且多了对应的名为qkv.per\_channel\_scale的一维数组，数组长度与矩阵列数相同，这应该就是量化系数。

根据之前的知识，我们可以在FP16的QKV中计算出每一列的 $\max(\text{abs}(S_i))$ ，然后计算量化系数。再从INT8 Weight-Only的checkpoint中直接读取量化系数，两相比较，看看是否相同。代码如下：

```
import torch
from safetensors import safe_open

with safe_open("llama2_7b_fp16/rank0.safetensors", framework = "pt") as f:
    QKV: torch.Tensor = f.get_tensor("transformer.layers.0.attention.qkv.weight")
    my_scale = QKV.abs().max(dim = 1).values / 128.0

with safe_open("llama2_7b_int8wo/rank0.safetensors", framework = "pt") as f:
    scale: torch.Tensor = f.get_tensor("transformer.layers.0.attention.qkv.per_channel_scale")

print(f"my_scale: {my_scale}\nread_scale: {scale}\ncompare: {my_scale == scale}")
```

```
my_scale: tensor([0.0009, 0.0016, 0.0020, ..., 0.0002, 0.0002, 0.0002],
dtype=torch.float16)
read_scale: tensor([0.0009, 0.0016, 0.0020, ..., 0.0002, 0.0002, 0.0002],
dtype=torch.float16)
compare: tensor([True, True, True, ..., True, True, True])
```

完全一致！这说明了INT8 Weight-Only确实用了按列的per channel量化，我们正确地掌握了相关知识。



# 动手实践

## 验证INT8 Weight-Only的权重

既然能手动计算系数了，那么再手工量化QKV也不在话下了，代码如下：

```
import torch
from safetensors import safe_open

with safe_open("llama2_7b_fp16/rank0.safetensors", framework = "pt") as f:
    QKV: torch.Tensor = f.get_tensor("transformer.layers.0.attention.qkv.weight")
    my_scale = QKV.abs().max(dim = 1).values / 128.0
    quant_QKV = (QKV.t() / my_scale).round().to(dtype = torch.int8)
    print(f"{quant_QKV.shape}\n{quant_QKV}")

with safe_open("llama2_7b_int8wo/rank0.safetensors", framework = "pt") as f:
    QKV: torch.Tensor = f.get_tensor("transformer.layers.0.attention.qkv.weight")
    print(f"{QKV.shape}\n{QKV}")
```

```
torch.Size([4096, 12288])
tensor([[ -7,   9,  -7, ..., -29,  13,   1],
        [-17,  -3,   6, ..., -26,  21,  11],
        [ -2,   2,   0, ...,  47,  -4,  25],
        ...,
        [  5,  -6,   3, ...,  14,  23,  -3],
        [  2,  -7,  10, ...,  18,  68,  -3],
        [ -4,   5,  -2, ..., -10,   3,   7]], dtype=torch.int8)
torch.Size([4096, 12288])
tensor([[ 121, -127,  111, ..., -127, -117,  125],
        [ 126,  125, -125, ...,  120, -126, -128],
        [ 127,  127, -121, ..., -120,  125,  122],
        ...,
        [ -66, -106, -111, ...,  -80,   68,  123],
        [ -81,  125,  -79, ...,  -67,  116,   74],
        [ 123,   98,   73, ...,  125, -105, -121]], dtype=torch.int8)
```



诡异的事情发生了：虽然手工量化的矩阵尺寸与直接读取的一致，但是没有一个元素是能对上的！

一开始我百思不得其解，反复验证，以为是哪个环节算错了。最后在专家同事们的指引下，学习了相关资料、阅读了相关源码，恍然大悟，不禁拍案叫绝。

这一切要从性能优化说起.....



# 性能优化

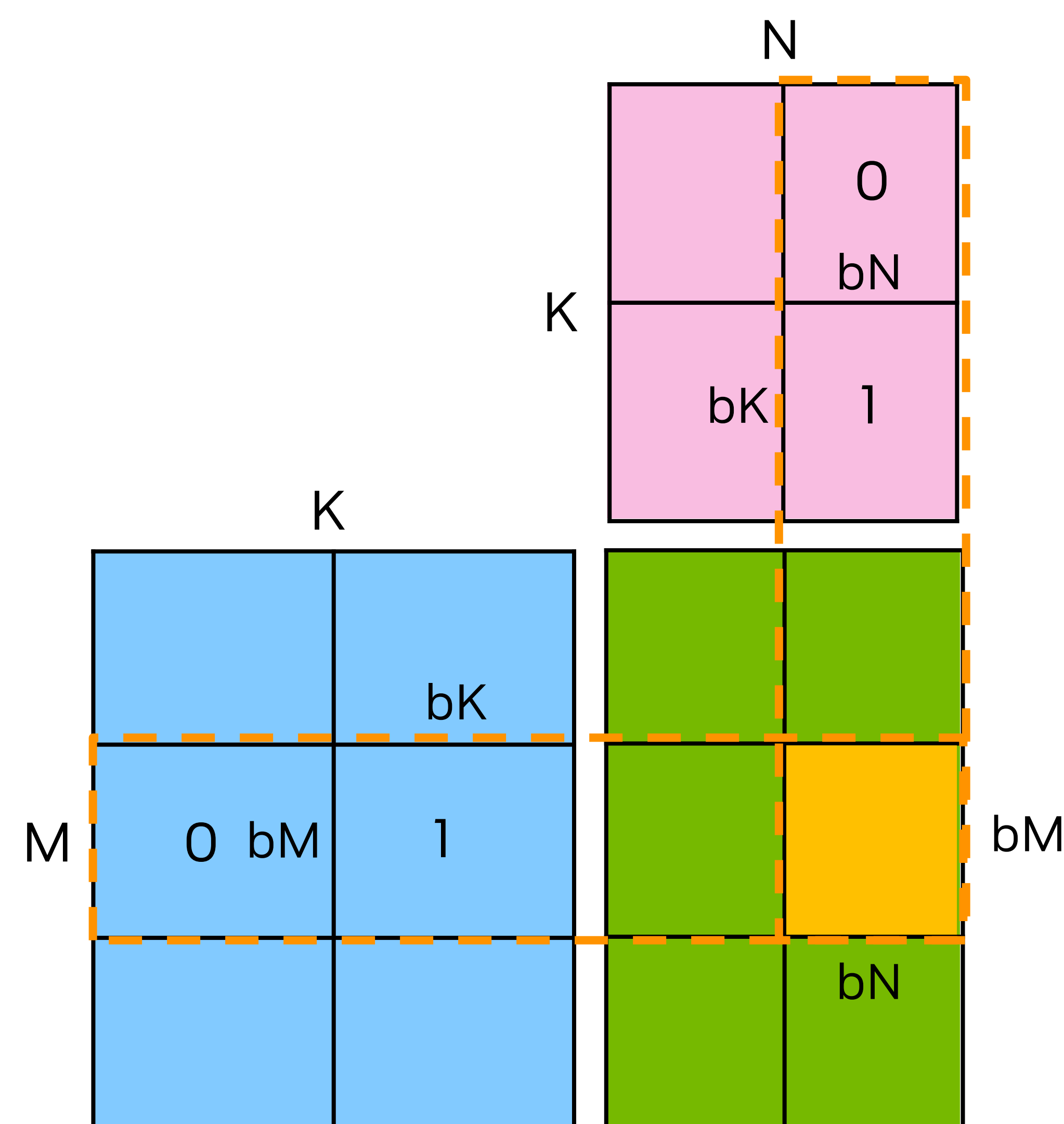
## 转置

其实我们计算的权重是正确的。但是TensorRT-LLM会在此基础上，对权重做一系列的预处理，包括：

- 转置 (transpose)
- 交织列 (interleave column)
- 重排行 (permute row)
- 加偏置 (add bias)

这些操作都是为了充分利用硬件特性而设计的。正是这些操作的存在，使得量化后的权重被加工地“面目全非”。

先让我们温习一下矩阵乘法  $C = A \times B$  在GPU上的典型实现（假设  $A$  是  $M \times K$  矩阵， $B$  是  $K \times N$  矩阵）。



将  $A$  分成  $bM \times bK$  的块，将  $B$  分成  $bK \times bN$  的块，将  $C$  分成  $bM \times bN$  的块。

一个CUDA block负责C中一分块的计算，共需要  $\frac{M}{bM} \times \frac{N}{bN}$  个CUDA blocks。每一个CUDA block 计算过程如下：

沿着K维度迭代A与B，共迭代  $\frac{K}{bK}$  轮：

第  $i$  轮迭代中计算第  $i$  个A分块与第  $i$  个B分块的矩阵乘积，累加到C分块中。

显然，A是按行读取的，而B是按列读取的。

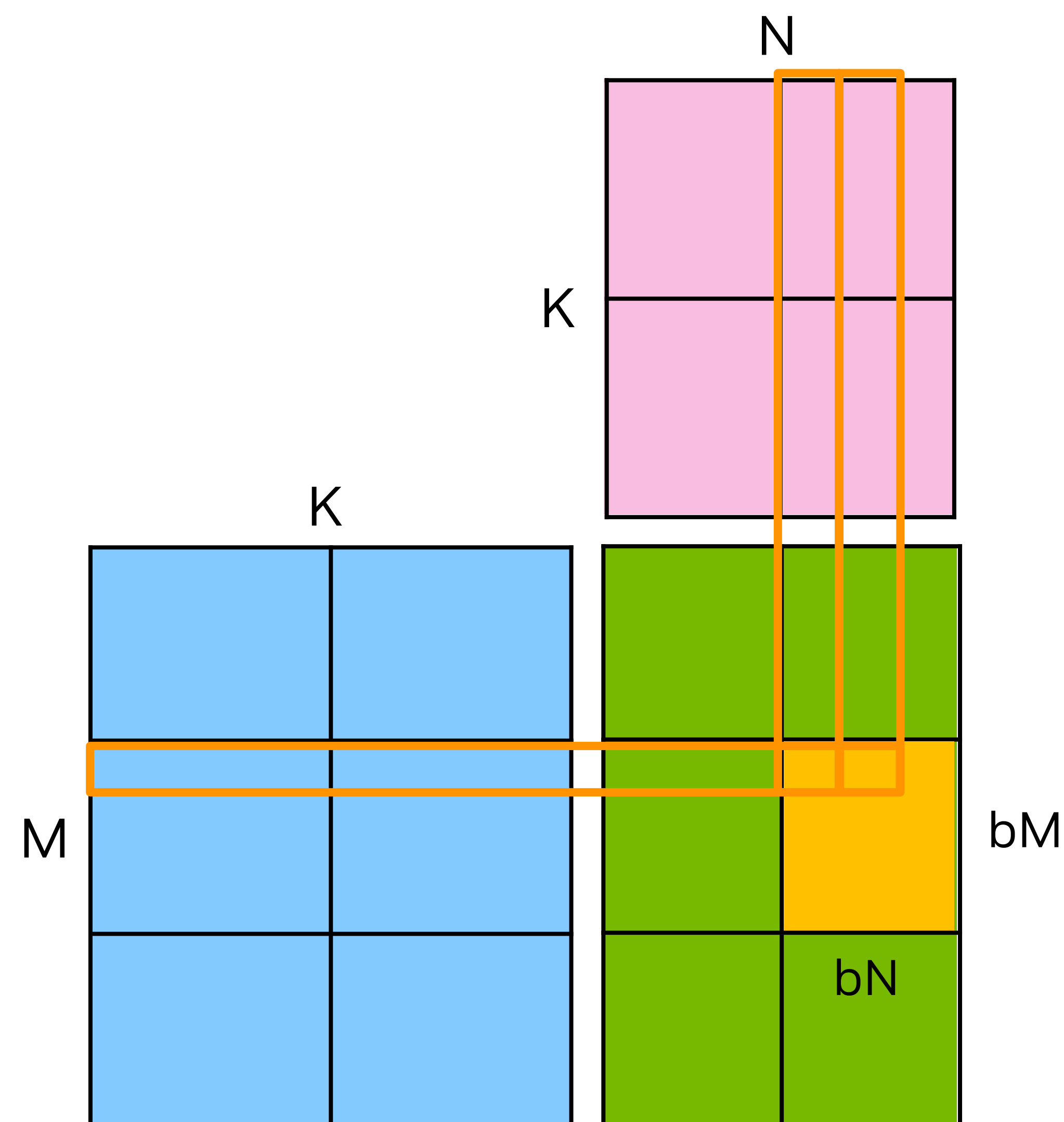
在之前的实验中，由QKV拼接构成的、大小为4096x12288的矩阵作为B矩阵，与作为A矩阵的activation相乘。因此，为了优化内存读取性能，预处理时做了转置操作，使得B按列存储，那么按列读取QKV是对地址空间的顺序访问。

# 性能优化

## 交织列

深入CUDA block内部，在每次迭代时，都需要先将A分块与B分块从全局内存（global memory）加载到共享内存（shared memory）中，这可以使用ldg指令完成，也就是“load global”的意思。ldg可一次加载32字节、64字节或128字节。为了用尽量少的指令加载尽量多的数据，我们当然使用ldg.128。

但是，当我们使用混合精度（即A矩阵与B矩阵使用不同的精度）时，就出现了问题。以INT8 Weight-Only为例，A是FP16类型，B是INT8类型。一条ldg.128指令可以加载8个FP16，或者说16个INT8，也就是说读取的B元素个数是A的2倍。但是我们知道，计算C中的一个元素时，所需的A元素与B元素是一样多的。那么填平这个差异的最简单的办法就是，B中并行地加载两列，如左图所示。



既然B是按列存储，为了让并行加载两列不发生地址空间的跳跃访问，预处理时将相邻两列交织在一起，如右图所示。如此，读取物理上的一列等于同时读取了逻辑上的两列。

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7

interleave column

0,0	0,2	0,4	0,6
0,1	0,3	0,5	0,7
1,0	1,2	1,4	1,6
1,1	1,3	1,5	1,7
2,0	2,2	2,4	2,6
2,1	2,3	2,5	2,7
3,0	3,2	3,4	3,6
3,1	3,3	3,5	3,7



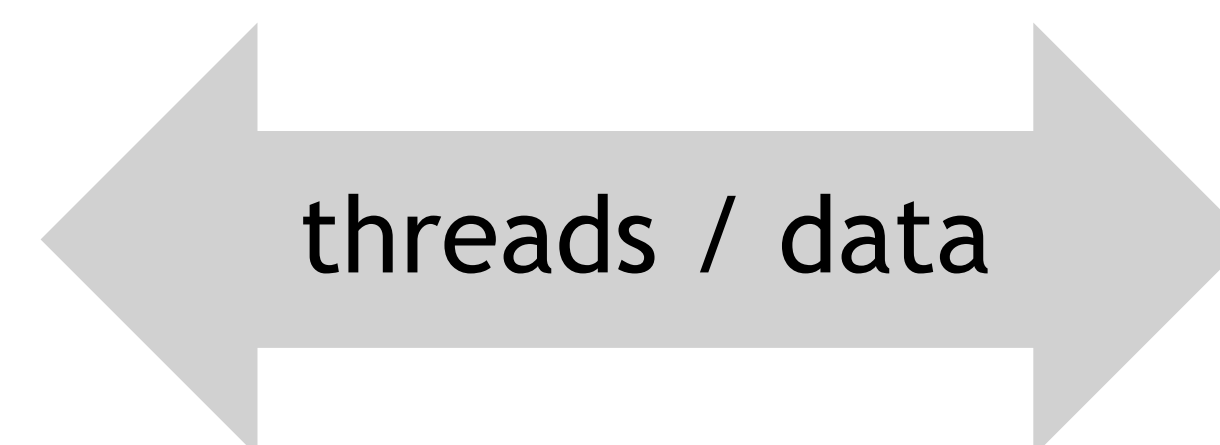
# 性能优化

## 重排行

权重已经从全局内存加载到共享内存中，我们再将它们读入寄存器，转为FP16，就可以使用MMA指令，利用Tensor Core高效计算分片矩阵乘积。从共享内存加载到寄存器的最高效的方案是使用ldmatrix指令，它会为线程束（warp）中的每一个线程加载连续的4字节数据。

如果数据类型是FP16，那么执行ldmatrix指令后，每个线程持有2个FP16，其排布如下图所示：

T0	T4	T8	T12	T16	T20	T24	T28
T1	T5	T9	T13	T17	T21	T25	T29
T2	T6	T10	T14	T18	T22	T26	T30
T3	T7	T11	T15	T19	T23	T27	T31



0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7
4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7
5,0	5,1	5,2	5,3	5,4	5,5	5,6	5,7
6,0	6,1	6,2	6,3	6,4	6,5	6,6	6,7
7,0	7,1	7,2	7,3	7,4	7,5	7,6	7,7
8,0	8,1	8,2	8,3	8,4	8,5	8,6	8,7
9,0	9,1	9,2	9,3	9,4	9,5	9,6	9,7
A,0	A,1	A,2	A,3	A,4	A,5	A,6	A,7
B,0	B,1	B,2	B,3	B,4	B,5	B,6	B,7
C,0	C,1	C,2	C,3	C,4	C,5	C,6	C,7
D,0	D,1	D,2	D,3	D,4	D,5	D,6	D,7
E,0	E,1	E,2	E,3	E,4	E,5	E,6	E,7
F,0	F,1	F,2	F,3	F,4	F,5	F,6	F,7

正如ldg指令有32字节、64字节和128字节等版本一样，ldmatrix指令也有x1、x2和x4多个版本。如果使用ldmatrix.x2指令，那么每个线程就会持有4个FP16，如图所示。比如线程0持有(0,0)、(1,0)、(8,0)、(9,0)。

而这恰好符合mma.m16n8k16指令对B分片在寄存器中的排布要求。因此当使用FP16时，ldmatrix指令可以与MMA指令完美搭配使用。

# 性能优化

## 重排行

但是，当权重是INT8时就出现了问题。由于ldmatrix指令为每一个线程加载连续的4字节数据，便会加载连续的4个，于是各线程所加载的权重如左图所示。而如上一页所说，mma.m16n8k16指令所需的数据排布如中图所示。

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7
4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7
5,0	5,1	5,2	5,3	5,4	5,5	5,6	5,7
6,0	6,1	6,2	6,3	6,4	6,5	6,6	6,7
7,0	7,1	7,2	7,3	7,4	7,5	7,6	7,7
8,0	8,1	8,2	8,3	8,4	8,5	8,6	8,7
9,0	9,1	9,2	9,3	9,4	9,5	9,6	9,7
A,0	A,1	A,2	A,3	A,4	A,5	A,6	A,7
B,0	B,1	B,2	B,3	B,4	B,5	B,6	B,7
C,0	C,1	C,2	C,3	C,4	C,5	C,6	C,7
D,0	D,1	D,2	D,3	D,4	D,5	D,6	D,7
E,0	E,1	E,2	E,3	E,4	E,5	E,6	E,7
F,0	F,1	F,2	F,3	F,4	F,5	F,6	F,7

≠

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7
4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7
5,0	5,1	5,2	5,3	5,4	5,5	5,6	5,7
6,0	6,1	6,2	6,3	6,4	6,5	6,6	6,7
7,0	7,1	7,2	7,3	7,4	7,5	7,6	7,7
8,0	8,1	8,2	8,3	8,4	8,5	8,6	8,7
9,0	9,1	9,2	9,3	9,4	9,5	9,6	9,7
A,0	A,1	A,2	A,3	A,4	A,5	A,6	A,7
B,0	B,1	B,2	B,3	B,4	B,5	B,6	B,7
C,0	C,1	C,2	C,3	C,4	C,5	C,6	C,7
D,0	D,1	D,2	D,3	D,4	D,5	D,6	D,7
E,0	E,1	E,2	E,3	E,4	E,5	E,6	E,7
F,0	F,1	F,2	F,3	F,4	F,5	F,6	F,7

=

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7
8,0	8,1	8,2	8,3	8,4	8,5	8,6	8,7
9,0	9,1	9,2	9,3	9,4	9,5	9,6	9,7
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7
A,0	A,1	A,2	A,3	A,4	A,5	A,6	A,7
B,0	B,1	B,2	B,3	B,4	B,5	B,6	B,7
4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7
5,0	5,1	4,2	5,3	5,4	5,5	5,6	5,7
C,0	C,1	C,2	C,3	C,4	C,5	C,6	C,7
D,0	D,1	D,2	D,3	D,4	D,5	D,6	D,7
6,0	6,1	6,2	6,3	6,4	6,5	6,6	6,7
7,0	7,1	7,2	7,3	7,4	7,5	7,6	7,7
E,0	E,1	E,2	E,3	E,4	E,5	E,6	E,7
F,0	F,1	F,2	F,3	F,4	F,5	F,6	F,7

为了填平这样的差异，就需要在预处理时将权重重排为右图所示。以线程0为例，重排之前读取到的是(0,0)、(1,0)、(2,0)、(3,0)，重排之后读到的是(0,0)、(1,0)、(8,0)、(9,0)，与MMA指令所需的数据排布匹配了。进一步观察发现，只需按行重排，即把原来连续的16行按照[0, 1, 8, 9, 2, 3, A, B, 4, 5, C, D, 6, 7, E, F]的顺序重排即可。如此便无需在运行时做额外处理。



# 性能优化

## 加偏置

现在，所需的所有数据都在寄存器中了。在执行MMA指令之前，只差将INT8转为FP16了。最简单的转换代码如下所示：

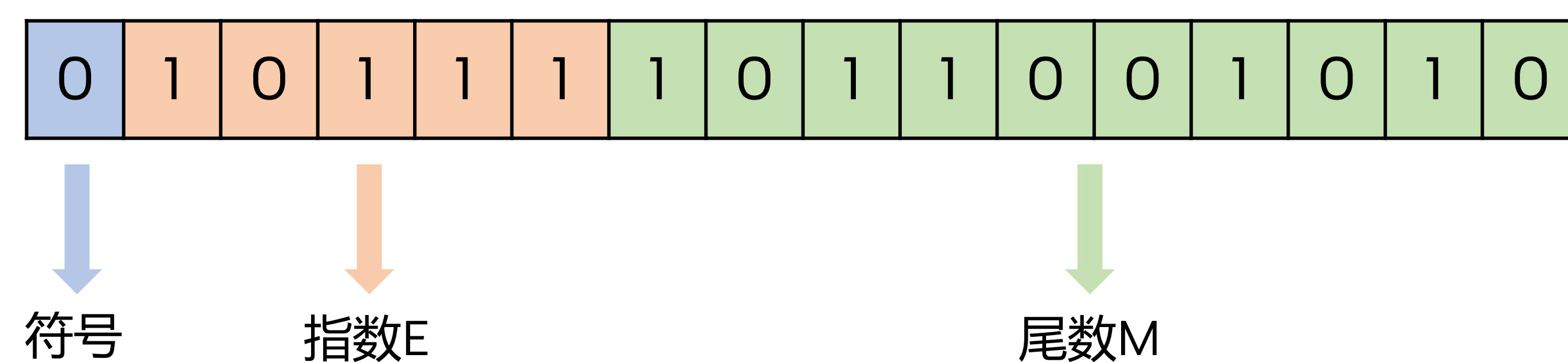
```
__global__ void func(int8_t* src, half* dst) {
    dst[0] = static_cast<half>(src[0]);
}
```

→

```
/*0000*/      LDC R1, c[0x0][0x28] ;      /* 0x00000a00ff017b82 */
/*0010*/      LDC.64 R2, c[0x0][0x210] ;  /* 0x0000ff00000000800 */
/*0020*/      ULDC.64 UR4, c[0x0][0x208] ; /* 0x00008400ff027b82 */
/*0030*/      LDG.E.S8 R2, desc[UR4][R2.64] ; /* 0x0000e620000000a00 */
/*0040*/      LDC.64 R4, c[0x0][0x218] ;  /* 0x00008200000047ab9 */
/*0050*/      I2F.F16 R7, R2 ;           /* 0x0000fe40000000a00 */
/*0060*/      STG.E.U16 desc[UR4][R4.64], R7 ; /* 0x00000000402027981 */
/*0070*/      EXIT ;                     /* 0x0002eaa000c1e1300 */
/*0080*/      /* 0x00008600ff047b82 */
/*0090*/      /* 0x0000e620000000a00 */
/*00a0*/      /* 0x00000000200077306 */
/*00b0*/      /* 0x0004e640000200c00 */
/*00c0*/      /* 0x00000000704007986 */
/*00d0*/      /* 0x0002fe2000c101504 */
/*00e0*/      /* 0x0000000000000794d */
/*00f0*/      /* 0x000fea0003800000 */
```

但是I2F指令的性能并不高，我们需要一个更加高效的实现，最好是SIMD执行的算法，因为一个线程有四个数值需要处理。

FP16的格式如下图所示：



$$\pm (1.\langle M \rangle)_2 \times 2^{\langle E \rangle - 15}$$

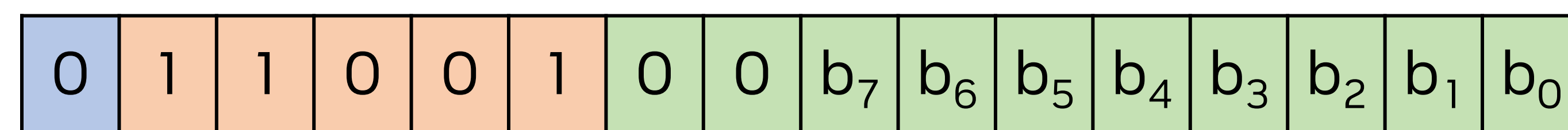
$$\begin{aligned}
 &+ (1.1011001010)_2 \times 2^{(1011)_2 - 15} \\
 &= (1.1011001010)_2 \times 2^{23 - 15} \\
 &= 2^8 + 2^7 + 2^5 + 2^4 + 2^1 + 2^{-1} \\
 &= 434.5
 \end{aligned}$$

# 性能优化

## 加偏置

假设有一个UINT8（8位无符号整数）的数据，其数值为X，并将其8比特记为 $b_7 \sim b_0$ 。我们构造这么一个FP16：

- 符号位为0；
- 指数位为11001；
- 尾数高2位为00；
- 尾数低8位为 $b_7 \sim b_0$ 。



$$\begin{aligned} & + (1.00b_7b_6b_5b_4b_3b_2b_1b_0)_2 \times 2^{(11001)_2 - 15} \\ & = (1.00b_7b_6b_5b_4b_3b_2b_1b_0)_2 \times 2^{25-15} \\ & = 2^{10} + (0.00b_7b_6b_5b_4b_3b_2b_1b_0)_2 \times 2^{10} \\ & = 1024 + (b_7b_6b_5b_4b_3b_2b_1b_0)_2 \times 2^0 \\ & = 1024 + X \end{aligned}$$

于是，UINT8转FP16的算法为：

1. 高8位置为01100100；
2. 低8位复制为UINT8；
3. 再将FP16减去1024（浮点运算）。

浮点数变更正负号只需要改变最高位，即正负数的表达是对称的。但是整数的负数采用补码表示，并不能简单地变换最高位。为了避免转换过程出现if...else分支语句而降低性能，将INT8转为FP16可以采用“INT8 -> UINT8 -> FP16”的路线：

1. **预处理时将INT8加128变为UINT8；**
2. 运行时：
  - ① 高8位置为01100100；
  - ② 低8位复制为UINT8；
  - ③ 再将FP16减去1152（=1024+128）。



# 附录

## 代码导读

以v0.9.0为例，INT8 Weight-Only量化权重的生成逻辑都在TensorRT-LLM源码的如下路径：

[cpp/tensorrt\\_llm/kernels/cutlass\\_kernels/cutlass\\_preprocessors.cpp](#)

的如下函数中：

```
621     template <typename ComputeType, typename WeightType>
622     void symmetric_quantize(int8_t* processed_quantized_weight, int8_t* unprocessed_quantized_weight,
623         ComputeType* scale_ptr, WeightType const* input_weight_ptr, std::vector<size_t> const& shape, QuantType quant_type,
624         bool force_interleave)
```

先求出每列的最大绝对值per\_col\_max[jj]：

```
663         for (int ii = 0; ii < num_rows; ++ii)
664         {
665             WeightType const* current_weight_row = current_weight + ii * num_cols;
666             for (int jj = 0; jj < num_cols; ++jj)
667             {
668                 per_col_max[jj] = std::max(per_col_max[jj], std::abs(float(current_weight_row[jj])));
669             }
670         }
```

将每列权重缩放到[-128, 127]区间：

```
690         float const col_scale = per_col_max[jj];
691         float const weight_elt = float(current_weight_row[jj]);
692         float const scaled_weight = round(weight_elt / col_scale);
693         const int8_t clipped_weight = int8_t(std::max(-128.f, std::min(127.f, scaled_weight)));
694         current_quantized_weight_row[jj] = clipped_weight;
```

这里的操作与我们之前的代码逻辑是一致的。



# 附录

## 代码导读

在函数最后，调用了如下函数进行上文所述的预处理操作：

```
727 preprocess_weights_for_mixed_gemm(  
728     processed_quantized_weight, unprocessed_quantized_weight, shape, quant_type, force_interleave);
```

其中unprocessed\_quantized\_weight便是按数学原理计算得到的量化权重，而processed\_quantized\_weight则是经过一系列预处理之后的权重。

为了证明我们用python计算的量化权重是正确的，可以在预处理之前添加如下代码，将权重最左上角16x16的子矩阵打印出来：

```
727 printf("quantized weight:\n");  
728 for(int row = 0; row < 16; row++) {  
729     int8_t* row_weight = unprocessed_quantized_weight + row * bytes_per_out_col;  
730     for(int col = 0; col < 16; col++) {  
731         printf("%04d ", row_weight[col]);  
732     }  
733     printf("\n");  
734 }
```

重新编译：

```
python3 ./scripts/build_wheel.py --trt_root /usr/local/tensorrt
```

用新编译的tensorrt\_llm执行INT8 Weight-Only量化：

```
PYTHONPATH=../.. \  
python convert_checkpoint.py --model_dir ~/Llama-2-7b-hf --dtype float16 --output_dir ~/llama2_7b_int8wo \  
--use_weight_only --weight_only_precision int8
```



# 附录

## 代码导读

可以看到如下输出：

```
Loading checkpoint shards: 100%|
quantized weight:
-007 0009 -007 -005 -005 0001 -001 -005 0001 -003 0003 0003 -001 0009 -002 -005
-017 -003 0006 0006 0010 -005 0007 0003 -006 0011 -016 0005 -002 -005 -001 0004
-002 0002 0000 0000 0001 -001 0000 -001 -001 0002 0000 0001 0000 -003 0002 0001
0002 0003 0001 -005 -003 -004 -001 0004 0003 -004 0005 -004 0000 -027 0007 -002
-011 -001 0004 0000 0005 -004 0003 0003 -003 0003 -002 0004 -005 -001 0002 0013
0012 -003 0003 0003 0001 0000 0000 -002 -001 0002 -008 -001 -002 -007 0005 -004
-002 -003 0002 0001 0004 -001 0000 -003 -003 0005 0001 -001 0000 -006 0001 0000
0000 0002 -002 0003 -003 0001 -002 -006 0001 -002 0003 -001 0002 0019 -004 -002
0001 0001 -005 -002 -005 0003 -001 -008 0003 -004 0006 -002 0004 0023 -005 -011
0013 0002 0001 0002 -007 -001 0003 -003 0003 -011 0013 -005 0003 -017 -001 0008
0004 -011 0006 0006 0011 0000 0003 0013 -004 0005 -011 0003 -005 -042 0007 0016
0000 0002 0000 0001 -001 -001 0000 -001 0001 -002 0003 -001 0000 -005 0002 0001
0005 0001 -004 -003 -004 0002 -002 -006 0002 -005 0007 -001 0001 -001 0001 -004
-010 0011 -010 0002 -009 0007 -007 -015 0007 -013 0019 -005 0009 0052 -014 -009
0002 0008 -012 -006 -012 0008 -009 -010 0007 -014 0016 -006 0007 0041 -013 -014
-001 0004 0001 -005 -001 -003 0001 -006 0000 0002 0003 0003 -002 0003 0003 0006
```

与前面用python计算的结果对比一下，前面3x3完全一致：

```
torch.Size([4096, 12288])
tensor([[ -7,   9,  -7, ..., -29,  13,   1],
        [-17,  -3,   6, ..., -26,  21,  11],
        [ -2,   2,   0, ...,  47,  -4,  25],
        ...,
        [  5,  -6,   3, ...,  14,  23,  -3],
        [  2,  -7,  10, ...,  18,  68,  -3],
        [ -4,   5,  -2, ..., -10,   3,   7]], dtype=torch.int8)
torch.Size([4096, 12288])
```



# 附录

## 代码导读

而在函数

```
535 void preprocess_weights_for_mixed_gemm(int8_t* preprocessed_quantized_weight, int8_t const* row_major_quantized_weight,
536 |   std::vector<size_t> const& shape, QuantType quant_type, bool force_interleave)
```

内部，可以看到预处理时为了优化性能而做的4种洗牌操作：

```
560 // Works on row major data, so issue this permutation first.
561 if (details.uses_imma_ldsm)
562 {
563     permute_B_rows_for_mixed_gemm(dst_buf.data(), src_buf.data(), shape, quant_type, arch);
564     src_buf.swap(dst_buf);
565 }
566
567 if (details.layoutB == LayoutDetails::Layout::COLUMN_MAJOR)
568 {
569     subbyte_transpose(dst_buf.data(), src_buf.data(), shape, quant_type);
570     src_buf.swap(dst_buf);
571 }
572
573 if (details.columns_interleaved > 1)
574 {
575     interleave_column_major_tensor(dst_buf.data(), src_buf.data(), shape, quant_type, details);
576     src_buf.swap(dst_buf);
577 }
578
579 if (arch >= 70 && arch < 90)
580 {
581     add_bias_and_interleave_quantized_tensor_inplace(src_buf.data(), num_elts, quant_type);
582 }
```

如果感兴趣的话，也可以深入其中的代码一探究竟，或者打印一下每一步之后的权重，直观地感受其效果。



# 附录

## FP8与INT4 GPTQ量化命令

将HuggingFace的模型量化为FP8可用如下命令：

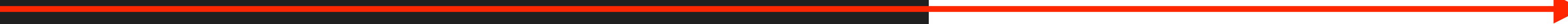
```
python ../quantization/quantize.py \
  --model_dir ~/Llama-2-7b-hf --dtype float16 --output_dir ~/llama2_7b_fp8 \
  --qformat fp8 --calib_size 512
```

将HuggingFace的模型量化为INT4 GPTQ需要使用外部工具，以AutoGPTQ为例，先安装之：

```
pip install auto-gptq --no-build-isolation
```

写一个python脚本convert.py，将~/Llama-2-7b-hf量化为INT4 GPTQ：

```
1  import logging
2  from transformers import AutoTokenizer
3  from auto_gptq import AutoGPTQForCausalLM, BaseQuantizeConfig
4
5  logging.basicConfig(
6      format="%(asctime)s %(levelname)s [%(name)s] %(message)s",
7      level=logging.INFO, datefmt="%Y-%m-%d %H:%M:%S"
8  )
9
10 pretrained_model_dir = "/root/Llama-2-7b-hf"
11 quantized_model_dir = "/root/llama2_7b_int4_gptq_tmp"
12
13 quantize_config = BaseQuantizeConfig(
14     bits=4,          # quantize model to 4-bit
15     group_size=128,   # it is recommended to set the value to 128
16     desc_act=False,   # set to False can significantly speed up inference but the perplexity may slightly bad
17 )
18 model = AutoGPTQForCausalLM.from_pretrained(pretrained_model_dir, quantize_config)
19 tokenizer = AutoTokenizer.from_pretrained(pretrained_model_dir, use_fast=True)
20 examples = [
21     tokenizer(
22         "auto-gptq is an easy-to-use model quantization library "
23         "with user-friendly apis, based on GPTQ algorithm."
24     ),
25     tokenizer(
26         "TensorRT-LLM provides users with an easy-to-use Python API to define Large Language Models (LLMs) "
27         "and build TensorRT engines that contain state-of-the-art optimizations to perform inference "
28         "efficiently on NVIDIA GPUs."
29     )
30 ]
31 model.quantize(examples)
32 model.save_quantized(quantized_model_dir)
```



```
✓ llama2_7b_int4_gptq_tmp
  {} config.json
  ≡ gptq_model-4bit-128g.safetensors
  {} quantize_config.json
```

# 附录

## FP8与INT4 GPTQ量化命令

之后即可导入TensorRT-LLM中：

```
python convert_checkpoint.py \  
  --model_dir ~/Llama-2-7b-hf --dtype float16 --output_dir ~/llama2_7b_int4_gptq \  
  --use_weight_only --weight_only_precision int4_gptq --per_group \  
  --ammo_quant_ckpt_path ~/llama2_7b_int4_gptq_tmp/gptq_model-4bit-128g.safetensors
```



# 小试牛刀

我给大家布置了几道题目作为课后作业。我将给大家的回答打分，并抽取十名优秀的同学参加7月12日英伟达Open AI Day线下活动。期待大家积极作答、踊跃报名~

背景：

目前TensorRT-LLM中的FP8量化只支持weight & activation一种。我们希望支持FP8 weight-only，即权重以FP8表达，在推理时从全局内存加载权重，在寄存器中转换为FP16，与FP16类型的activation做矩阵乘法。原理与INT8 weight-only相似，许多代码逻辑可以复用，改动主要在数据类型转换上。

问题：

1. 量化时需要将缩放后的权重近似为FP8。请用C/C++代码实现第6页中的`round_to_fp8(x)`函数， $x$ 是FP16类型的数值。
2. 运行时需要将FP8权重转换为FP16再缩放。请用C/C++代码实现`convert_to_fp16(x)`函数， $x$ 是FP8类型的数值。
3. 为了最佳的性能，我们希望每个线程能够一次处理多个数值转换。请用SIMD指令实现`convert_to_fp16s(x)`函数，即`convert_to_fp16(x)`函数的升级版，其中 $x$ 是有N个元素的FP8数组。（提示：可用您熟悉的任意硬件平台，包括但不限于x86 CPU的SSE/AVX2/AVX512，ARM CPU的NEON/SVE，以及NVIDIA GPU的PTX等。）





**Thanks!**